

CS231n课程笔记翻译

笔记本： CS231n

创建时间： 2018/1/15 11:18

更新时间： 2018/1/31 15:33

作者： eagleao

URL： <https://zhuanlan.zhihu.com/p/20894041?refer=intelligentunit>

CS231n课程笔记翻译

由知乎@杜客 等人翻译

David 整理合并

1. 图像分类笔记 (上)
2. 图像分类笔记 (下)
3. 线性分类笔记 (上)
4. 线性分类笔记 (中)
5. 线性分类笔记 (下)
6. 最优化笔记 (上)
7. 最优化笔记 (下)
8. 反向传播笔记
9. 神经网络笔记1 (上)
10. 神经网络笔记1 (下)
11. 神经网络笔记 2
12. 神经网络笔记3 (上)
13. 神经网络笔记3 (下)
14. 卷积神经网络笔记

1. 图像分类笔记 (上)

这是一篇介绍性教程，面向非计算机视觉领域的同学。教程将向同学们介绍图像分类问题和数据驱动方法。下面是**内容列表**：

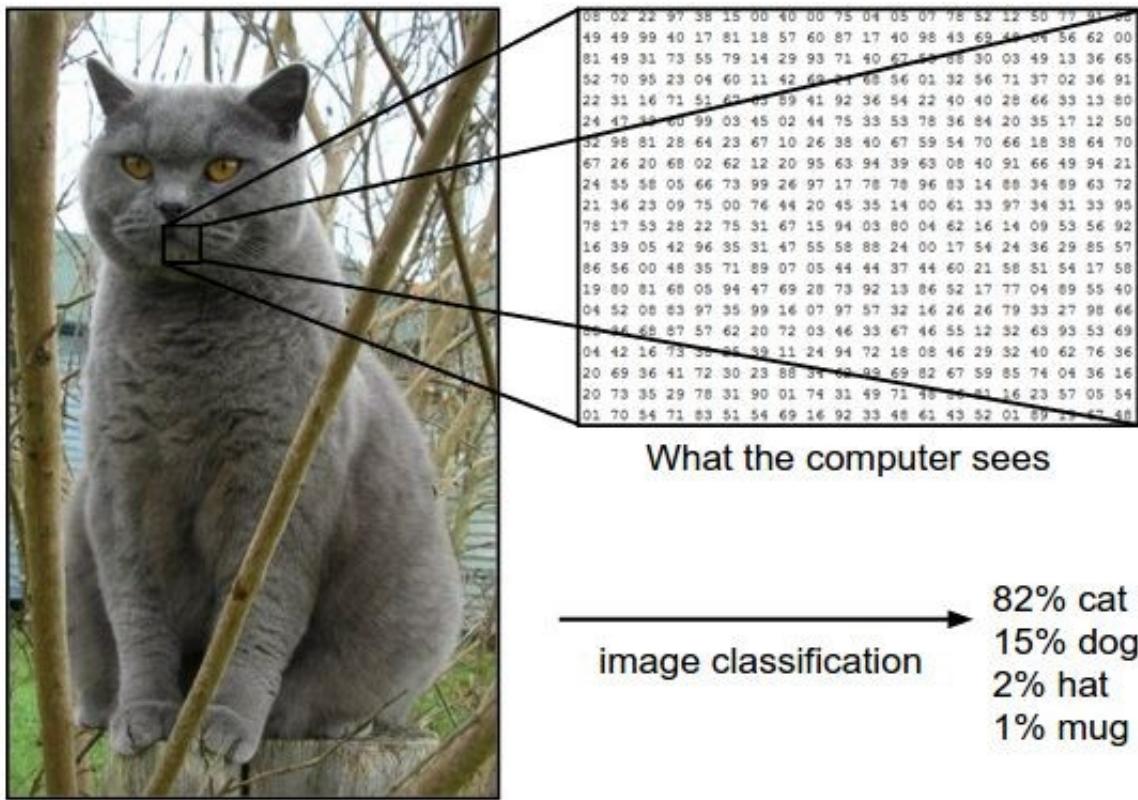
- 图像分类、数据驱动方法和流程
- Nearest Neighbor分类器

- k-Nearest Neighbor **译者注：上篇翻译截止处**
- 验证集、交叉验证集和超参数调参
- Nearest Neighbor的优劣
- 小结
- 小结：应用kNN实践
- 拓展阅读

图像分类

目标：这一节我们将介绍图像分类问题。所谓图像分类问题，就是已有固定的分类标签集合，然后对于输入的图像，从分类标签集合中找出一个分类标签，最后把分类标签分配给该输入图像。虽然看起来挺简单的，但这可是计算机视觉领域的核心问题之一，并且有着各种各样的实际应用。在后面的课程中，我们可以看到计算机视觉领域中很多看似不同的问题（比如物体检测和分割），都可以被归结为图像分类问题。

例子：以下图为例，图像分类模型读取该图片，并生成该图片属于集合 {cat, dog, hat, mua} 中各个标签的概率。需要注意的是，对于计算机来说，图像是一个由数字组成的巨大的3维数组。在这个例子中，猫的图像大小是宽248像素，高400像素，有3个颜色通道，分别是红、绿和蓝（简称RGB）。如此，该图像就包含了 $248 \times 400 \times 3 = 297600$ 个数字，每个数字都是在范围0-255之间的整型，其中0表示全黑，255表示全白。我们的任务就是把这些上百万的数字变成一个简单的标签，比如“猫”。



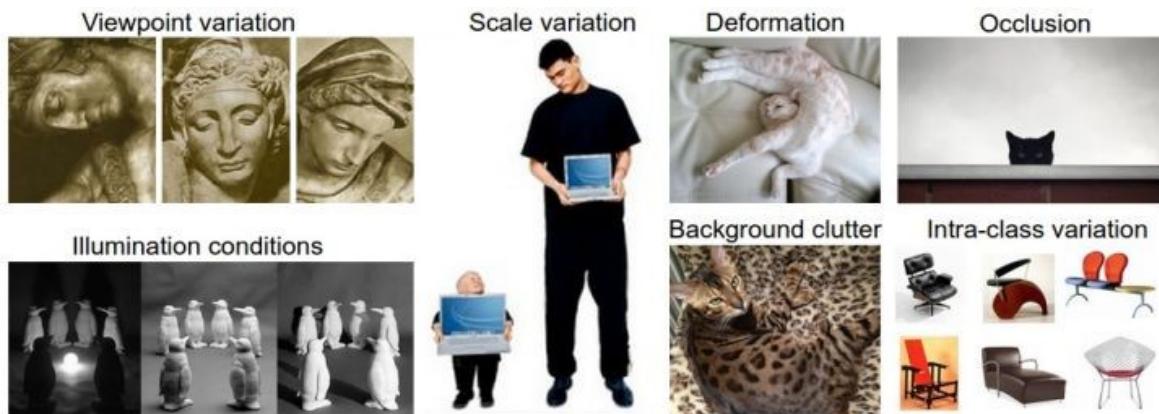
图像分类的任务，就是对于一个给定的图像，预测它属于的那个分类标签（或者给出属于一系列不同标签的可能性）。图像是3维数组，数组元素是取值范围从0到255的整数。数组的尺寸是宽度x高度x3，其中这个3代表的是红、绿和蓝3个颜色通道。

困难和挑战：对于人来说，识别出一个像“猫”一样视觉概念是简单至极的，然而从计算机视觉算法的角度来看就值得深思了。我们在下面列举了计算机视觉算法在图像识别方面遇到的一些困难，要记住图像是以3维数组来表示的，数组中的元素是亮度值。

- **视角变化 (Viewpoint variation)**：同一个物体，摄像机可以从多个角度来展现。
- **大小变化 (Scale variation)**：物体可视的大小通常会变化的（不仅是在图片中，在真实世界中大小也是变化的）。
- **形变 (Deformation)**：很多东西的形状并非一成不变，会有很大变化。
- **遮挡 (Occlusion)**：目标物体可能被挡住。有时候只有物体的一小部分（可以小到几个像素）是可见的。
- **光照条件 (Illumination conditions)**：在像素层面上，光照的影响非常大。

- **背景干扰 (Background clutter)** : 物体可能混入背景之中，使之难以被辨认。
- **类内差异 (Intra-class variation)** : 一类物体的个体之间的外形差异很大，比如椅子。这一类物体有许多不同的对象，每个都有自己的外形。

面对以上所有变化及其组合，好的图像分类模型能够在维持分类结论稳定的同时，保持对类间差异足够敏感。



数据驱动方法：如何写一个图像分类的算法呢？这和写个排序算法可是大不一样。怎么写一个从图像中认出猫的算法？搞不清楚。因此，与其在代码中直接写明各类物体到底看起来是什么样的，倒不如说我们采取的方法和教小孩儿看图识物类似：给计算机很多数据，然后实现学习算法，让计算机学习到每个类的外形。这种方法，就是**数据驱动方法**。既然该方法的第一步就是收集已经做好分类标注的图片来作为训练集，那么下面就看看数据库到底长什么样：



一个有4个视觉分类的训练集。在实际中，我们可能有上千的分类，每个分类都有成千上万的图像。

图像分类流程。在课程视频中已经学习过，**图像分类**就是输入一个元素为像素值的数组，然后给它分配一个分类标签。完整流程如下：

- **输入**：输入是包含N个图像的集合，每个图像的标签是K种分类标签中的一种。这个集合称为**训练集**。
- **学习**：这一步的任务是使用训练集来学习每个类到底长什么样。一般该步骤叫做**训练分类器**或者**学习一个模型**。
- **评价**：让分类器来预测它未曾见过的图像的分类标签，并以此来评价分类器的质量。我们会把分类器预测的标签和图像真正的分类标签对比。毫无疑问，分类器预测的分类标签和图像真正的分类标签如果一致，那就是好事，这样的情况越多越好。

Nearest Neighbor分类器

作为课程介绍的第一个方法，我们来实现一个**Nearest Neighbor分类器**。虽然这个分类器和卷积神经网络没有任何关系，实际中也极少使用，但通过实现它，可以让读者对于解决图像分类问题的方法有个基本的认识。

图像分类数据集：CIFAR-10。一个非常流行的图像分类数据集是**CIFAR-10**。这个数据集包含了60000张32X32的小图像。每张图像都有10种分类标签中的一种。这60000张图像被分为包含50000张图像的训练集和包含10000张图像的测试集。在下图中你可以看见10个类的10张随机图片。



左边：从**CIFAR-10**数据库来的样本图像。**右边**：第一列是测试图像，然后第一列的每个测试图像右边是使用**Nearest Neighbor**算法，根据像素差异，从训练集中选出的10张最类似的图片。

假设现在我们有**CIFAR-10**的50000张图片（每种分类5000张）作为训练集，我们希望将余下的10000作为测试集并给他们打上标签。**Nearest Neighbor**

算法将会拿着测试图片和训练集中每一张图片去比较，然后将它认为最相似的那个训练集图片的标签赋给这张测试图片。上面右边的图片就展示了这样的结果。请注意上面10个分类中，只有3个是准确的。比如第8行中，马头被分类为一个红色的跑车，原因在于红色跑车的黑色背景非常强烈，所以这匹马就被错误分类为跑车了。

那么具体如何比较两张图片呢？在本例中，就是比较 $32 \times 32 \times 3$ 的像素块。最简单的方法就是逐个像素比较，最后将差异值全部加起来。换句话说，就是将两张图片先转化为两个向量 I_1 和 I_2 ，然后计算他们的L1距离：

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

这里的求和是针对所有的像素。下面是整个比较流程的图例：

test image				training image				pixel-wise absolute value differences			
56	32	10	18	10	20	24	17	46	12	14	1
90	23	128	133	8	10	89	100	82	13	39	33
24	26	178	200	12	16	178	170	12	10	0	30
2	0	255	220	4	32	233	112	2	32	22	108

- = → 456

以图片中的一个颜色通道为例来进行说明。两张图片使用L1距离来进行比较。逐个像素求差值，然后将所有差值加起来得到一个数值。如果两张图片一模一样，那么L1距离为0，但是如果两张图片很是不同，那L1值将会非常大。

下面，让我们看看如何用代码来实现这个分类器。首先，我们将CIFAR-10的数据加载到内存中，并分成4个数组：训练数据和标签，测试数据和标签。在下面的代码中，**Xtr**（大小是 $50000 \times 32 \times 32 \times 3$ ）存有训练集中所有的图像，**Ytr**是对应的长度为50000的1维数组，存有图像对应的分类标签（从0到9）：

```
Xtr, Ytr, Xte, Yte = load CIFAR10('data/cifar10/') # a magic
function we provide# flatten out all images to be one-dimensional
Xtr_rows = Xtr.reshape(Xtr.shape[0], 32 * 32 * 3) # Xtr_rows becomes
50000 x 3072
Xte_rows = Xte.reshape(Xte.shape[0], 32 * 32 * 3) # Xte_rows becomes
10000 x 3072
```

现在我们得到所有的图像数据，并且把他们拉长成为行向量了。接下来展示如何训练并评价一个分类器：

```

nn = NearestNeighbor() # create a Nearest Neighbor classifier class
nn.train(Xtr_rows, Ytr) # train the classifier on the training
images and labels
Yte_predict = nn.predict(Xte_rows) # predict labels on the test
images
# and now print the classification accuracy, which is the average
number
# of examples that are correctly predicted (i.e. label matches)
print 'accuracy: %f' % ( np.mean(Yte_predict == Yte) )

```

作为评价标准，我们常常使用**准确率**，它描述了我们预测正确的得分。请注意以后我们实现的所有分类器都需要有这个API：**train(X, y)**函数。该函数使用训练集的数据和标签来进行训练。从其内部来看，类应该实现一些关于标签和标签如何被预测的模型。这里还有个**predict(X)**函数，它的作用是预测输入的新数据的分类标签。现在还没介绍分类器的实现，下面就是使用L1距离的Nearest Neighbor分类器的实现套路：

```

import numpy as np

class NearestNeighbor(object):
    def __init__(self):
        pass

    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-
dimension of size N """
        # the nearest neighbor classifier simply remembers all
        # the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """ X is N x D where each row is an example we wish to
predict label for """
        num_test = X.shape[0]
        # Lets make sure that the output type matches the input
        type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # Loop over all test rows
        for i in xrange(num_test):
            # find the nearest training image to the i'th test
            image
            # using the L1 distance (sum of absolute value
            differences)

```

```

    distances = np.sum(np.abs(self.Xtr - X[i, :]), axis = 1)
    min_index = np.argmin(distances) # get the index with
    smallest distance
    Ypred[i] = self.ytr[min_index] # predict the label of
    the nearest example

    return Ypred

```

如果你用这段代码跑CIFAR-10，你会发现准确率能达到**38.6%**。这比随机猜测的10%要好，但是比人类识别的水平（据研究推测是94%）和卷积神经网络能达到的95%还是差多了。点击查看基于CIFAR-10数据的[Kaggle算法竞赛排行榜](#)。

距离选择：计算向量间的距离有很多种方法，另一个常用的方法是**L2距离**，从几何学的角度，可以理解为它在计算两个向量间的欧式距离。L2距离的公式如下：

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

换句话说，我们依旧是在计算像素间的差值，只是先求其平方，然后把这些平方全部加起来，最后对这个和开方。在Numpy中，我们只需要替换上面代码中的1行代码就行：

```
distances = np.sqrt(np.sum(np.square(self.Xtr - X[i, :]), axis = 1))
```

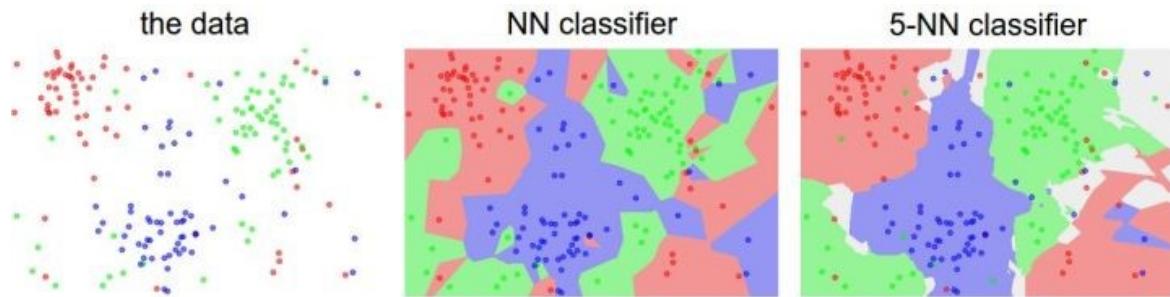
注意在这里使用了**np.sqrt**，但是在实际中可能不用。因为求平方根函数是一个**单调函数**，它对不同距离的绝对值求平方根虽然改变了数值大小，但依然保持了不同距离大小的顺序。所以用不用它，都能够对像素差异的大小进行正确比较。如果你在CIFAR-10上面跑这个模型，正确率是**35.4%**，比刚才低了一点。

L1和L2比较。比较这两个度量方式是挺有意思的。在面对两个向量之间的差异时，L2比L1更加不能容忍这些差异。也就是说，相对于1个巨大的差异，L2距离更倾向于接受多个中等程度的差异。L1和L2都是在**p-norm**常用的特殊形式。

k-Nearest Neighbor分类器

你可能注意到了，为什么只用最相似的1张图片的标签来作为测试图像的标签呢？这不是很奇怪吗！是的，使用**k-Nearest Neighbor分类器**就能做得更

好。它的思想很简单：与其只找最相近的那1个图片的标签，我们找最相似的k个图片的标签，然后让他们针对测试图片进行投票，最后把票数最高的标签作为对测试图片的预测。所以当k=1的时候，k-Nearest Neighbor分类器就是Nearest Neighbor分类器。从直观感受上就可以看到，更高的k值可以让分类的效果更平滑，使得分类器对于异常值更有抵抗力。



上面示例展示了Nearest Neighbor分类器和5-Nearest Neighbor分类器的区别。例子使用了2维的点来表示，分成3类（红、蓝和绿）。不同颜色区域代表的是使用L2距离的分类器的**决策边界**。白色的区域是分类模糊的例子（即图像与两个以上的分类标签绑定）。需要注意的是，在NN分类器中，异常的数据点（比如：在蓝色区域中的绿点）制造出一个不正确预测的孤岛。5-NN分类器将这些不规则都平滑了，使得它针对测试数据的**泛化 (generalization)**能力更好（例子中未展示）。注意，5-NN中也存在一些灰色区域，这些区域是因为近邻标签的最高票数相同导致的（比如：2个邻居是红色，2个邻居是蓝色，还有1个是绿色）。

在实际中，大多使用k-NN分类器。但是k值如何确定呢？接下来就讨论这个问题。

2. 图像分类笔记（下）

内容列表

- 图像分类、数据驱动方法和流程
- Nearest Neighbor分类器
 - k-Nearest Neighbor
- 验证集、交叉验证集和超参数调优 **译者注：下篇翻译起始处**
- Nearest Neighbor的优劣
- 小结

- 小结：应用kNN实践
- 拓展阅读

用于超参数调优的验证集

k-NN分类器需要设定k值，那么选择哪个k值最合适呢？我们可以选择不同的距离函数，比如L1范数和L2范数等，那么选哪个好？还有不少选择我们甚至连考虑都没有考虑到（比如：点积）。所有这些选择，被称为**超参数**（**hyperparameter**）。在基于数据进行学习的机器学习算法设计中，超参数是很常见的。一般说来，这些超参数具体怎么设置或取值并不是显而易见的。

你可能会建议尝试不同的值，看哪个值表现最好就选哪个。好主意！我们就是这么做的，但这样做的时候要非常细心。特别注意：**决不能使用测试集来进行调优**。当你在设计机器学习算法的时候，应该把测试集看做非常珍贵的资源，不到最后一步，绝不使用它。如果你使用测试集来调优，而且算法看起来效果不错，那么真正的危险在于：算法实际部署后，性能可能会远低于预期。这种情况，称之为算法对测试集**过拟合**。从另一个角度来说，如果使用测试集来调优，实际上就是把测试集当做训练集，由测试集训练出来的算法再跑测试集，自然性能看起来会很好。这其实是过于乐观了，实际部署起来效果就会差很多。所以，最终测试的时候再使用测试集，可以很好地近似度量你所设计的分类器的泛化性能（在接下来的课程中会有很多关于泛化性能的讨论）。

测试数据集只使用一次，即在训练完成后评价最终的模型时使用。

好在我们有不用测试集调优的方法。其思路是：从训练集中取出一部分数据用来调优，我们称之为**验证集**（**validation set**）。以CIFAR-10为例，我们可以用49000个图像作为训练集，用1000个图像作为验证集。验证集其实就是作为假的测试集来调优。下面就是代码：

```
# assume we have Xtr_rows, Ytr, Xte_rows, Yte as before
# recall
Xtr_rows is 50,000 x 3072 matrix
Xval_rows = Xtr_rows[:1000, :] # take first 1000 for validation
Yval = Ytr[:1000]
Xtr_rows = Xtr_rows[1000:, :] # keep last 49,000 for train
Ytr = Ytr[1000:]

# find hyperparameters that work best on the validation set
validation_accuracies = []
for k in [1, 3, 5, 10, 20, 50, 100]:
    # use a particular value of k and evaluation on validation data
```

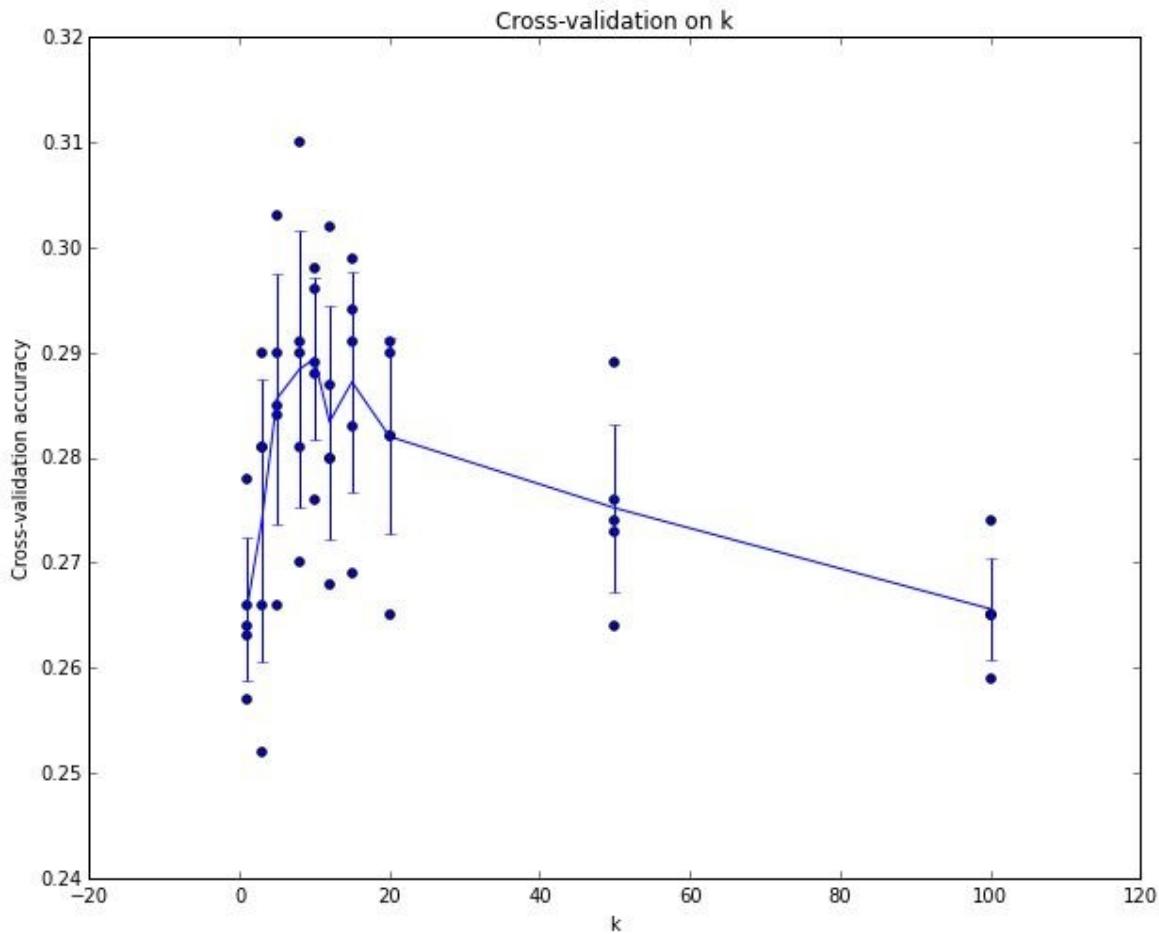
```
nn = NearestNeighbor()
nn.train(Xtr_rows, Ytr)
# here we assume a modified NearestNeighbor class that can take a
# k as input
Yval_predict = nn.predict(Xval_rows, k = k)
acc = np.mean(Yval_predict == Yval)
print 'accuracy: %f' % (acc,)

# keep track of what works on the validation set
validation_accuracies.append((k, acc))
```

程序结束后，我们会作图分析出哪个k值表现最好，然后用这个k值来跑真正的测试集，并作出对算法的评价。

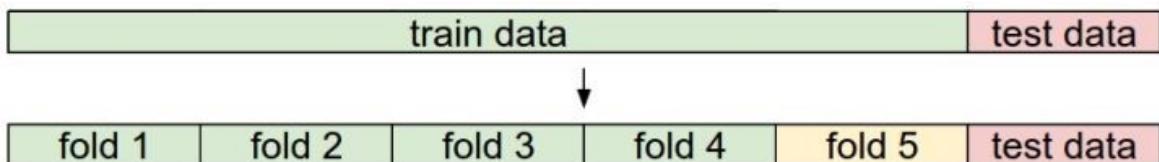
把训练集分成训练集和验证集。使用验证集来对所有超参数调优。最后只在测试集上跑一次并报告结果。

交叉验证。有时候，训练集数量较小（因此验证集的数量更小），人们会使用一种被称为**交叉验证**的方法，这种方法更加复杂些。还是用刚才的例子，如果是交叉验证集，我们就不是取1000个图像，而是将训练集平均分成5份，其中4份用来训练，1份用来验证。然后我们循环着取其中4份来训练，其中1份来验证，最后取所有5次验证结果的平均值作为算法验证结果。



这就是5份交叉验证对k值调优的例子。针对每个k值，得到5个准确率结果，取其平均值，然后对不同k值的平均表现画线连接。本例中，当k=7的时候算法表现最好（对应图中的准确率峰值）。如果我们将训练集分成更多份，直线一般会更加平滑（噪音更少）。

实际应用。在实际情况下，人们不是很喜欢用交叉验证，主要是因为它会耗费较多的计算资源。一般直接把训练集按照50%-90%的比例分成训练集和验证集。但这也是根据具体情况来定的：如果超参数数量多，你可能就想用更大的验证集，而验证集的数量不够，那么最好还是用交叉验证吧。至于分成几份比较好，一般都是分成3、5和10份。



常用的数据分割模式。给出训练集和测试集后，训练集一般会被均分。这里是分成5份。前面4份用来训练，黄色那份用作验证集调优。如果采取交叉验证，那就各份轮流作为验证集。最后模型训练完毕，超参数都定好了，让模型跑一

次（而且只跑一次）测试集，以此测试结果评价算法。

Nearest Neighbor分类器的优劣

现在对Nearest Neighbor分类器的优缺点进行思考。首先，Nearest Neighbor分类器易于理解，实现简单。其次，算法的训练不需要花时间，因为其训练过程只是将训练集数据存储起来。然而测试要花费大量时间计算，因为每个测试图像需要和所有存储的训练图像进行比较，这显然是一个缺点。在实际应用中，我们关注测试效率远远高于训练效率。其实，我们后续要学习的卷积神经网络在这个权衡上走到了另一个极端：虽然训练花费很多时间，但是一旦训练完成，对新的测试数据进行分类非常快。这样的模式就符合实际使用需求。

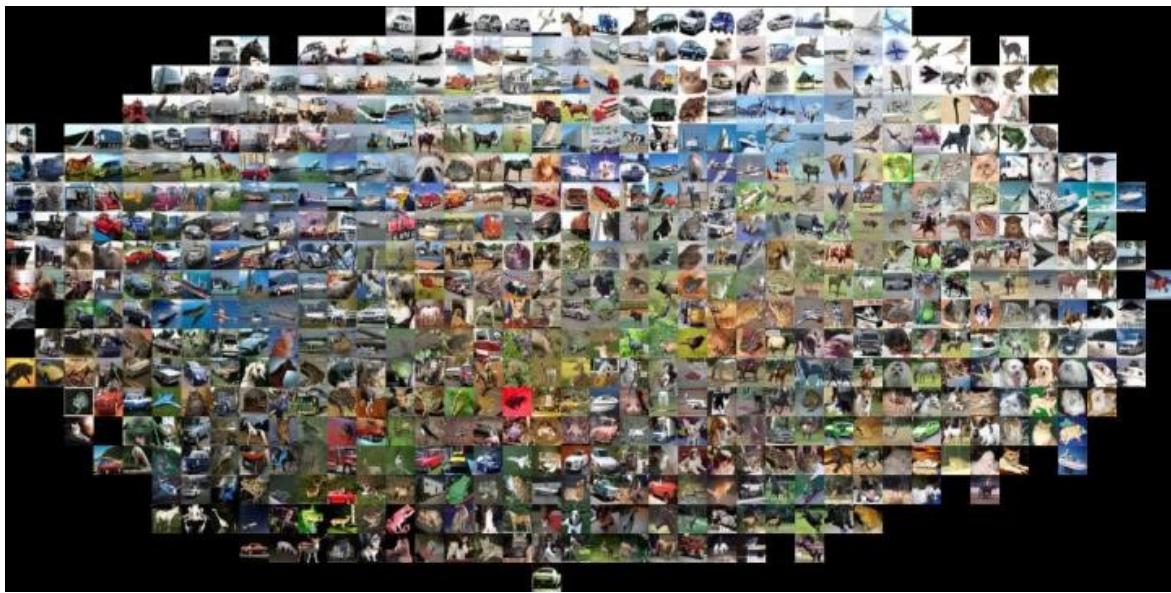
Nearest Neighbor分类器的计算复杂度研究是一个活跃的研究领域，若干**Approximate Nearest Neighbor** (ANN) 算法和库的使用可以提升 Nearest Neighbor分类器在数据上的计算速度（比如：[FLANN](#)）。这些算法可以在准确率和时空复杂度之间进行权衡，并通常依赖一个预处理/索引过程，这个过程中一般包含kd树的创建和k-means算法的运用。

Nearest Neighbor分类器在某些特定情况（比如数据维度较低）下，可能是不错的选择。但是在实际的图像分类工作中，很少使用。因为图像都是高维度数据（他们通常包含很多像素），而高维度向量之间的距离通常是反直觉的。下面的图片展示了基于像素的相似和基于感官的相似是有很大不同的：



在高维度数据上，基于像素的距离和感官上的非常不同。上图中，右边3张图片和左边第1张原始图片的L2距离是一样的。很显然，基于像素比较的相似和感官上以及语义上的相似是不同的。

这里还有个可视化证据，可以证明使用像素差异来比较图像是不够的。这是一个叫做t-SNE的可视化技术，它将CIFAR-10中的图片按照二维方式排布，这样能很好展示图片之间的像素差异值。在这张图片中，排列相邻的图片L2距离就小。



上图使用t-SNE的可视化技术将CIFAR-10的图片进行了二维排列。排列相近的图片L2距离小。可以看出，图片的排列是被背景主导而不是图片语义内容本身主导。

具体说来，这些图片的排布更像是一种颜色分布函数，或者说是基于背景的，而不是图片的语义主体。比如，狗的图片可能和青蛙的图片非常接近，这是因为两张图片都是白色背景。从理想效果上来说，我们肯定是希望同类的图片能够聚集在一起，而不被背景或其他不相关因素干扰。为了达到这个目的，我们不能止步于原始像素比较，得继续前进。

小结

简要说来：

- 介绍了**图像分类**问题。在该问题中，给出一个由被标注了分类标签的图像组成的集合，要求算法能预测没有标签的图像的分类标签，并根据算法预测准确率进行评价。
- 介绍了一个简单的图像分类器：**最近邻分类器(Nearest Neighbor classifier)**。分类器中存在不同的超参数(比如k值或距离类型的选取)，要想选取好的超参数不是一件轻而易举的事。
- 选取超参数的正确方法是：将原始训练集分为训练集和**验证集**，我们在验证集上尝试不同的超参数，最后保留表现最好那个。
- 如果训练数据量不够，使用**交叉验证**方法，它能帮助我们在选取最优超参数的时候减少噪音。
- 一旦找到最优的超参数，就让算法以该参数在测试集跑且只跑一次，并根据测试结果评价算法。

- 最近邻分类器能够在CIFAR-10上得到将近40%的准确率。该算法简单易实现，但需要存储所有训练数据，并且在测试的时候过于耗费计算能力。
- 最后，我们知道了仅仅使用L1和L2范数来进行像素比较是不够的，图像更多的是按照背景和颜色被分类，而不是语义主体分身。

在接下来的课程中，我们将专注于解决这些问题和挑战，并最终能够得到超过90%准确率的解决方案。该方案能够在完成学习就丢掉训练集，并在一毫秒之内就完成一张图片的分类。

小结：实际应用k-NN

如果你希望将k-NN分类器用到实处（最好别用到图像上，若是仅仅作为练手还可以接受），那么可以按照以下流程：

1. 预处理你的数据：对你数据中的特征进行归一化（normalize），让其具有零平均值（zero mean）和单位方差（unit variance）。在后面的小节我们会讨论这些细节。本小节不讨论，是因为图像中的像素都是同质的，不会表现出较大的差异分布，也就不需要标准化处理了。
2. 如果数据是高维数据，考虑使用降维方法，比如PCA([wiki ref](#), [CS229ref](#), [blog ref](#))或[随机投影](#)。
3. 将数据随机分入训练集和验证集。按照一般规律，70%-90%数据作为训练集。这个比例根据算法中有多少超参数，以及这些超参数对于算法的预期影响来决定。如果需要预测的超参数很多，那么就应该使用更大的验证集来有效地估计它们。如果担心验证集数量不够，那么就尝试交叉验证方法。如果计算资源足够，使用交叉验证总是更加安全的（份数越多，效果越好，也更耗费计算资源）。
4. 在验证集上调优，尝试足够多的k值，尝试L1和L2两种范数计算方式。
5. 如果分类器跑得太慢，尝试使用Approximate Nearest Neighbor库（比如[FLANN](#)）来加速这个过程，其代价是降低一些准确率。
6. 对最优的超参数做记录。记录最优参数后，是否应该让使用最优参数的算法在完整的训练集上运行并再次训练呢？因为如果把验证集重新放回到训练集中（自然训练集的数据量就又变大了），有可能最优参数又会有所变化。在实践中，**不要这样做**。千万不要在最终的分类器中使用验证集数据，这样做会破坏对于最优参数的估计。**直接使用测试集来测试用最优参数设置好的最优模型**，得到测试集数据的分类准确率，并以此作为你的kNN分类器在该数据上的性能表现。

拓展阅读

下面是一些你可能感兴趣的拓展阅读链接：

- [A Few Useful Things to Know about Machine Learning](#)，文中第6节与本节相关，但是整篇文章都强烈推荐。
- [Recognizing and Learning Object Categories](#)，ICCV 2005上的一节关于物体分类的课程。

3. 线性分类笔记（上）

内容列表：

- 线性分类器简介
- 线性评分函数
- 阐明线性分类器 **译者注：上篇翻译截止处**
- 损失函数
 - 多类SVM
 - Softmax分类器
 - SVM和Softmax的比较
- 基于Web的可交互线性分类器原型
- 小结

线性分类

上一篇笔记介绍了图像分类问题。图像分类的任务，就是从已有的固定分类标签集合中选择一个并分配给一张图像。我们还介绍了k-Nearest Neighbor (k-NN) 分类器，该分类器的基本思想是通过将测试图像与训练集带标签的图像进行比较，来给测试图像打上分类标签。k-Nearest Neighbor分类器存在以下不足：

- 分类器必须记住所有训练数据并将其存储起来，以便于未来测试数据用于比较。这在存储空间上是低效的，数据集的大小很容易就以GB计。
- 对一个测试图像进行分类需要和所有训练图像作比较，算法计算资源耗

费高。

概述：我们将要实现一种更强大的方法来解决图像分类问题，该方法可以自然地延伸到神经网络和卷积神经网络上。这种方法主要有两部分组成：一个是评分函数 (score function)，它是原始图像数据到类别分值的映射。另一个是损失函数 (loss function)，它是用来量化预测分类标签的得分与真实标签之间一致性的。该方法可转化为一个最优化问题，在最优化过程中，将通过更新评分函数的参数来最小化损失函数值。

从图像到标签分值的参数化映射

该方法的第一部分就是定义一个评分函数，这个函数将图像的像素值映射为各个分类类别的得分，得分高低代表图像属于该类别的可能性高低。下面会利用一个具体例子来展示该方法。现在假设有一个包含很多图像的训练集 $x_i \in R^D$ ，每个图像都有一个对应的分类标签 y_i 。这里 $i = 1, 2, \dots, N$ 并且 $y_i \in 1, \dots, K$ 。这就是说，我们有 N 个图像样例，每个图像的维度是 D ，共有 K 种不同的分类。

举例来说，在 CIFAR-10 中，我们有一个 $N=50000$ 的训练集，每个图像有 $D=32 \times 32 \times 3 = 3072$ 个像素，而 $K=10$ ，这是因为图片被分为 10 个不同的类别（狗，猫，汽车等）。我们现在定义评分函数为： $f: R^D \rightarrow R^K$ ，该函数是原始图像像素到分类分值的映射。

线性分类器：在本模型中，我们从最简单的概率函数开始，一个线性映射：

$$f(x_i, W, b) = Wx_i + b$$

在上面的公式中，假设每个图像数据都被拉长为一个长度为 D 的列向量，大小为 $[D \times 1]$ 。其中大小为 $[K \times D]$ 的矩阵 W 和大小为 $[K \times 1]$ 列向量 b 为该函数的参数 (parameters)。还是以 CIFAR-10 为例， x_i 就包含了第 i 个图像的所有像素信息，这些信息被拉成为一个 $[3072 \times 1]$ 的列向量， W 大小为 $[10 \times 3072]$ ， b 的大小为 $[10 \times 1]$ 。因此，3072 个数字（原始像素数值）输入函数，函数输出 10 个数字（不同分类得到的分值）。参数 W 被称为权重 (weights)。 b 被称为偏差向量 (bias vector)，这是因为它影响输出数值，但是并不和原始数据 x_i 产生关联。在实际情况中，人们常常混用权重和参数这两个术语。

需要注意的几点：

- 首先，一个单独的矩阵乘法 Wx_i 就高效地并行评估 10 个不同的分类器（每个分类器针对一个分类），其中每个类的分类器就是 W 的一个行向

量。

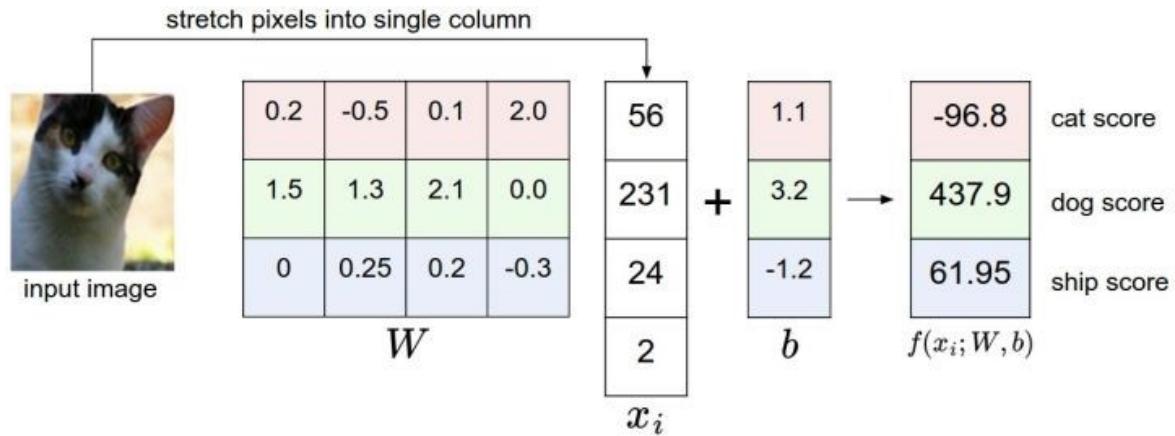
- 注意我们认为输入数据 (x_i, y_i) 是给定且不可改变的，但参数 W 和 b 是可控制改变的。我们的目标就是通过设置这些参数，使得计算出来的分类分值情况和训练集中图像数据的真实类别标签相符。在接下来的课程中，我们将详细介绍如何做到这一点，但是目前只需要直观地让正确分类的分值比错误分类的分值高即可。
- 该方法的一个优势是训练数据是用来学习到参数 W 和 b 的，一旦训练完成，训练数据就可以丢弃，留下学习到的参数即可。这是因为一个测试图像可以简单地输入函数，并基于计算出的分类分值来进行分类。
- 最后，注意只需要做一个矩阵乘法和一个矩阵加法就能对一个测试数据分类，这比 k-NN 中将测试图像和所有训练数据做比较的方法快多了。

预告：卷积神经网络映射图像像素值到分类分值的方法和上面一样，但是映射(f)就要复杂多了，其包含的参数也更多。

理解线性分类器

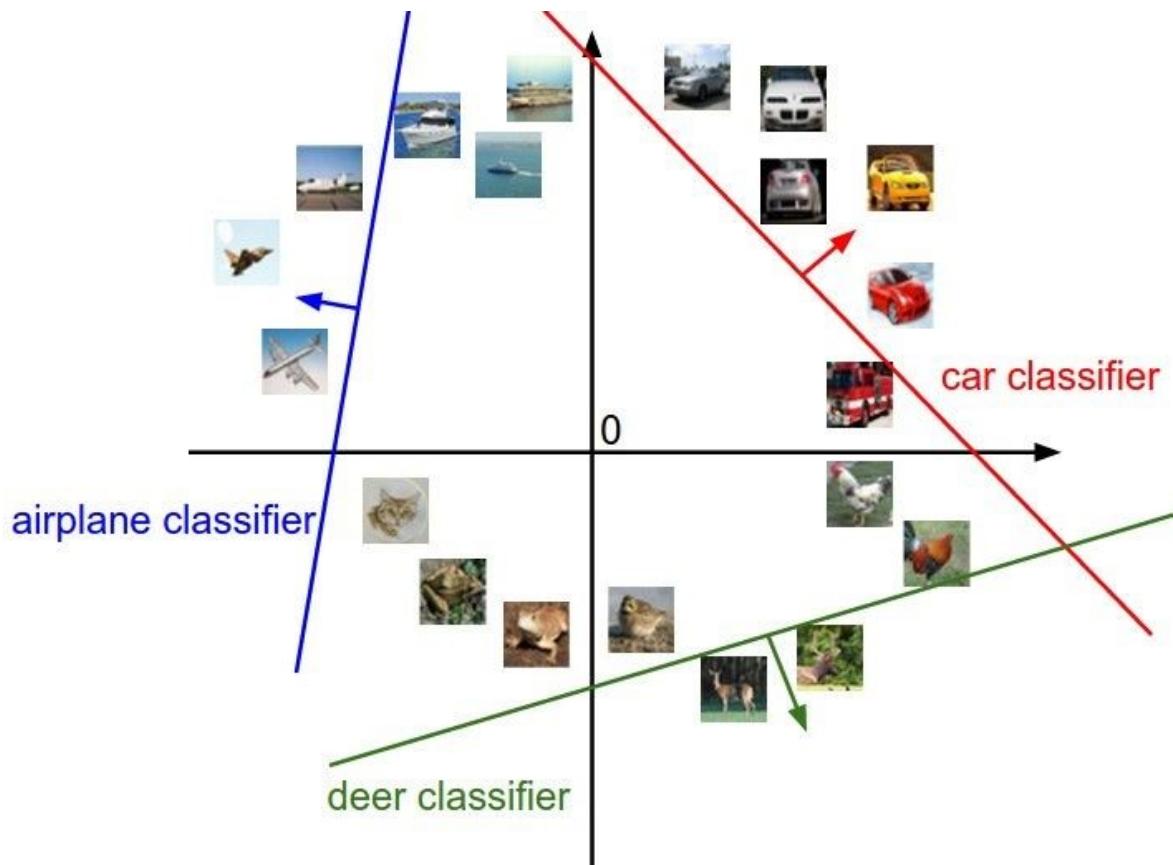
线性分类器计算图像中3个颜色通道中所有像素的值与权重的矩阵乘，从而得到分类分值。根据我们对权重设置的值，对于图像中的某些位置的某些颜色，函数表现出喜好或者厌恶（根据每个权重的符号而定）。举个例子，可以想象“船”分类就是被大量的蓝色所包围（对应的就是水）。那么“船”分类器在蓝色通道上的权重就有很多的正权重（它们的出现提高了“船”分类的分值），而在绿色和红色通道上的权重为负的就比较多（它们的出现降低了“船”分类的分值）。

一个将图像映射到分类分值的例子。为了便于可视化，假设图像只有4个像素（都是黑白像素，这里不考虑RGB通道），有3个分类（红色代表猫，绿色代表狗，蓝色代表船，注意，这里的红、绿和蓝3种颜色仅代表分类，和RGB通道没有关系）。首先将图像像素拉伸为一个列向量，与 W 进行矩阵乘，然后得到各个分类的分值。需要注意的是，这个 W 一点也不好：猫分类的分值非常低。从上图来看，算法倒是觉得这个图像是一只狗。



将图像看做高维度的点：既然图像被伸展成为了一个高维度的列向量，那么我们可以把图像看做这个高维度空间中的一个点（即每张图像是3072维空间中的一个点）。整个数据集就是一个点的集合，每个点都带有1个分类标签。

既然定义每个分类类别的分值是权重和图像的矩阵乘，那么每个分类类别的分数就是这个空间中的一个线性函数的函数值。我们没办法可视化3072维空间中的线性函数，但假设把这些维度挤压到二维，那么就可以看看这些分类器在做什么了：



图像空间的示意图。其中每个图像是一个点，有3个分类器。以红色的汽车分

类器为例，红线表示空间中汽车分类分数为0的点的集合，红色的箭头表示分值上升的方向。所有红线右边的点的分数值均为正，且线性升高。红线左边的点分值为负，且线性降低。

从上面可以看到， \mathbf{W} 的每一行都是一个分类类别的分类器。对于这些数字的几何解释是：如果改变其中一行的数字，会看见分类器在空间中对应的直线开始向着不同方向旋转。而偏差 \mathbf{b} ，则允许分类器对应的直线平移。需要注意的是，如果没有偏差，无论权重如何，在 $x_i = 0$ 时分类分值始终为0。这样所有分类器的线都不得不穿过原点。

将线性分类器看做模板匹配：关于权重 \mathbf{W} 的另一个解释是它的每一行对应着一个分类的模板（有时候也叫作原型）。一张图像对应不同分类的得分，是通过使用内积（也叫点积）来比较图像和模板，然后找到和哪个模板最相似。从这个角度来看，线性分类器就是在利用学习到的模板，针对图像做模板匹配。从另一个角度来看，可以认为还是在高效地使用k-NN，不同的是我们没有使用所有的训练集的图像来比较，而是每个类别只用了一张图片（这张图片是我们学习到的，而不是训练集中的某一张），而且我们会使用（负）内积来计算向量间的距离，而不是使用L1或者L2距离。



将课程进度快进一点。这里展示的是以CIFAR-10为训练集，学习结束后的权重的例子。注意，船的模板如期望的那样有很多蓝色像素。如果图像是一艘船行驶在大海上，那么这个模板利用内积计算图像将给出很高的分数。

可以看到马的模板看起来似乎是两个头的马，这是因为训练集中的马的图像中马头朝向各有左右造成的。线性分类器将这两种情况融合到一起了。类似的，汽车的模板看起来也是将几个不同的模型融合到了一个模板中，并以此来分辨不同方向不同颜色的汽车。这个模板上的车是红色的，这是因为CIFAR-10中训练集的车大多是红色的。线性分类器对于不同颜色的车的分类能力是很弱的，但是后面可以看到神经网络是可以完成这一任务的。神经网络可以在它的隐藏层中实现中间神经元来探测不同种类的车（比如绿色车头向左，蓝色车头向前等）。而下一层的神经元通过计算不同的汽车探测器的权重和，将这些合并为一个更精确的汽车分类分值。

偏差和权重的合并技巧：在进一步学习前，要提一下这个经常使用的技巧。它能够将我们常用的参数 \mathbf{W} 和 \mathbf{b} 合二为一。回忆一下，分类评分函数定义为：

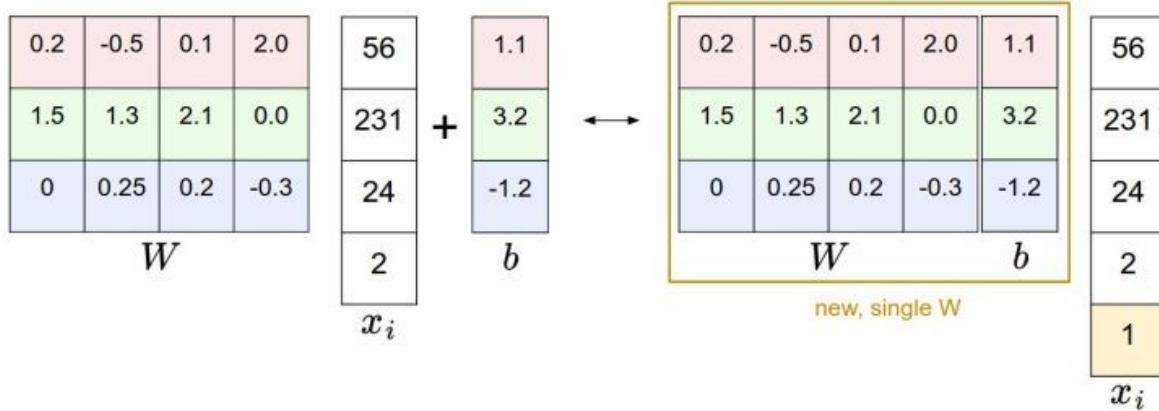
$$f(x_i, \mathbf{W}, \mathbf{b}) = \mathbf{W}x_i + \mathbf{b}$$

分开处理这两个参数（权重参数 \mathbf{W} 和偏差参数 \mathbf{b} ）有点笨拙，一般常用的方

法是把两个参数放到同一个矩阵中，同时 x_i 向量就要增加一个维度，这个维度的数值是常量1，这就是默认的偏差维度。这样新的公式就简化成下面这样：

$$f(x_i, W) = Wx_i$$

还是以CIFAR-10为例，那么 x_i 的大小就变成[3073x1]，而不是[3072x1]了，多出了包含常量1的1个维度）。W大小就是[10x3073]了。W中多出来的这一列对应的就是偏差值 b ，具体见下图：



偏差技巧的示意图。左边是先做矩阵乘法然后做加法，右边是将所有输入向量的维度增加1个含常量1的维度，并且在权重矩阵中增加一个偏差列，最后做一个矩阵乘法即可。左右是等价的。通过右边这样做，我们就只需要学习一个权重矩阵，而不用去学习两个分别装着权重和偏差的矩阵了。

图像数据预处理：在上面的例子中，所有图像都是使用的原始像素值（从0到255）。在机器学习中，对于输入的特征做归一化（normalization）处理是常见的套路。而在图像分类的例子中，图像上的每个像素可以看做一个特征。在实践中，对每个特征减去平均值来**中心化**数据是非常重要的。在这些图片的例子中，该步骤意味着根据训练集中所有的图像计算出一个平均图像值，然后每个图像都减去这个平均值，这样图像的像素值就大约分布在[-127, 127]之间了。下一个常见步骤是，让所有数值分布的区间变为[-1, 1]。零均值的**中心化**是很重要的，等我们理解了梯度下降后再来详细解释。

4.线性分类笔记（中）

内容列表：

- 线性分类器简介

- 线性评分函数
- 阐明线性分类器
- 损失函数
 - 多类SVM **译者注：中篇翻译截止处**
 - Softmax分类器
 - SVM和Softmax的比较
- 基于Web的可交互线性分类器原型
- 小结

损失函数 Loss function

在上一节定义了从图像像素值到所属类别的评分函数 (score function)，该函数的参数是权重矩阵 W 。在函数中，数据 (x_i, y_i) 是给定的，不能修改。但是我们可以调整权重矩阵这个参数，使得评分函数的结果与训练数据集中图像的真实类别一致，即评分函数在正确的分类的位置应当得到最高的评分 (score)。

回到之前那张猫的图像分类例子，它有针对“猫”，“狗”，“船”三个类别的分数。我们看到例子中权重值非常差，因为猫分类的得分非常低 (-96.8)，而狗 (437.9) 和船 (61.95) 比较高。我们将使用**损失函数** (Loss Function) (有时也叫**代价函数**Cost Function或**目标函数**Objective) 来衡量我们对结果的不满意程度。直观地讲，当评分函数输出结果与真实结果之间差异越大，损失函数输出越大，反之越小。

多类支持向量机损失 Multiclass Support Vector Machine Loss

损失函数的具体形式多种多样。首先，介绍常用的多类支持向量机 (SVM) 损失函数。SVM的损失函数想要SVM在正确分类上的得分始终比不正确分类上的得分高出一个边界值 Δ 。我们可以把损失函数想象成一个人，这位SVM先生 (或者女士) 对于结果有自己的品位，如果某个结果能使得损失值更低，那么SVM就更加喜欢它。

让我们更精确一些。回忆一下，第*i*个数据中包含图像 x_i 的像素和代表正确类别的标签 y_i 。评分函数输入像素数据，然后通过公式 $f(x_i, W)$ 来计算不同分分类类别的分值。这里我们将分值简写为 s 。比如，针对第*j*个类别的得分就是第

j个元素： $s_j = f(x_i, W)_j$ 。针对第i个数据的多类SVM的损失函数定义如下：

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

举例：用一个例子演示公式是如何计算的。假设有3个分类，并且得到了分值 $s = [13, -7, 11]$ 。其中第一个类别是正确类别，即 $y_i = 0$ 。同时假设 Δ 是 10（后面会详细介绍该超参数）。上面的公式是将所有不正确分类 ($j \neq y_i$) 加起来，所以我们得到两个部分：

$$L_i = \max(0, -7 - 13 + 10) + \max(0, 11 - 13 + 10)$$

可以看到第一个部分结果是0。这是因为 $[-7 - 13 + 10]$ 得到的是负数，经过 $\max(0, -)$ 函数处理后得到0。这一对类别分数和标签的损失值是0，这是因为正确分类的得分13与错误分类的得分-7的差为20，大于边界值10。而SVM只关心差距至少要大于10，更大的差值还是算作损失值为0。第二个部分计算 $[11 - 13 + 10]$ 得到8。虽然正确分类的得分比不正确分类的得分要高 ($13 > 11$)，但是比10的边界值还是小了，分差只有2。这就是为什么损失值等于8。简而言之，SVM的损失函数想要正确分类类别 y_i 的分数比不正确类别分数高，而且至少要高 Δ 。如果不满足这点，就开始计算损失值。

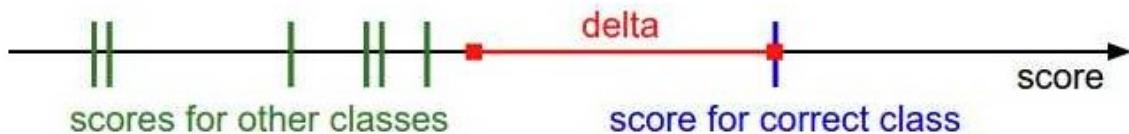
那么在这次的模型中，我们面对的是线性评分函数 ($f(x_i, W) = Wx_i$)，所以我们可以将损失函数的公式稍微改写一下：

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

其中 w_j 是权重 W 的第j行，被变形为列向量。然而，一旦开始考虑更复杂的评分函数 f 公式，这样做就不是必须的了。

在结束这一小节前，还必须提一下的是关于0的阀值： $\max(0, -)$ 函数，它常被称为**折叶损失 (hinge loss)**。有时候会听到人们使用平方折叶损失 SVM（即L2-SVM），它使用的是 $\max(0, -)^2$ ，将更强烈（平方地而不是线性地）地惩罚过界的边界值。不使用平方是更标准的版本，但是在某些数据集中，平方折叶损失会工作得更好。可以通过交叉验证来决定到底使用哪个。

我们对于预测训练集数据分类标签的情况总有一些不满意的，而损失函数就能将这些不满意的程度量化。



多类SVM“想要”正确类别的分类分数比其他不正确分类类别的分数要高，而且至少高出 δ 的边界值。如果其他分类分数进入了红色的区域，甚至更高，那么就开始计算损失。如果没有这些情况，损失值为0。我们的目标是找到一些权重，它们既能够让训练集中的数据样例满足这些限制，也能让总的损失值尽可能地低。

正则化 (Regularization)：上面损失函数有一个问题。假设有一个数据集和一个权重集 W 能够正确地分类每个数据（即所有的边界都满足，对于所有的 i 都有 $L_i = 0$ ）。问题在于这个 W 并不唯一：可能有很多相似的 W 都能正确地分类所有的数据。一个简单的例子：如果 W 能够正确分类所有数据，即对于每个数据，损失值都是0。那么当 $\lambda > 1$ 时，任何数乘 λW 都能使得损失值为0，因为这个变化将所有分值的大小都均等地扩大了，所以它们之间的绝对差值也扩大了。举个例子，如果一个正确分类的分值和举例它最近的错误分类的分值的差距是15，对 W 乘以2将使得差距变成30。

换句话说，我们希望能向某些特定的权重 W 添加一些偏好，对其他权重则不添加，以此来消除模糊性。这一点是能够实现的，方法是向损失函数增加一个**正则化惩罚 (regularization penalty)** $R(W)$ 部分。最常用的正则化惩罚是L2范式，L2范式通过对所有参数进行逐元素的平方惩罚来抑制大数值的权重：

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

上面的表达式中，将 W 中所有元素平方后求和。注意正则化函数不是数据的函数，仅基于权重。包含正则化惩罚后，就能够给出完整的多类SVM损失函数了，它由两个部分组成：**数据损失 (data loss)**，即所有样例的的平均损失 L_i ，以及**正则化损失 (regularization loss)**。完整公式如下所示：

$$L = \underbrace{\frac{1}{N} \sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

将其展开完整公式是：

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} [max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta)] + \lambda \sum_k \sum_l W_{k,l}^2$$

其中， N 是训练集的数据量。现在正则化惩罚添加到了损失函数里面，并用超参数 λ 来计算其权重。该超参数无法简单确定，需要通过交叉验证来获取。

除了上述理由外，引入正则化惩罚还带来很多良好的性质，这些性质大多会在后续章节介绍。比如引入了L2惩罚后，SVM们就有了**最大边界 (max margin)** 这一良好性质。（如果感兴趣，可以查看[CS229课程](#)）。

其中最好的性质就是对大数值权重进行惩罚，可以提升其泛化能力。因为这就意味着没有哪个维度能够独自对整体分值有过大的影响。举个例子，假设输入向量 $x = [1, 1, 1, 1]$ ，两个权重向量 $w_1 = [1, 0, 0, 0]$ ， $w_2 = [0.25, 0.25, 0.25, 0.25]$ 。那么 $w_1^T x = w_2^T x = 1$ ，两个权重向量都得到同样的内积，但是 w_1 的L2惩罚是1.0，而 w_2 的L2惩罚是0.25。因此，根据L2惩罚来看， w_2 更好，因为它的正则化损失更小。从直观上来看，这是因为 w_2 的权重值更小且更分散。既然L2惩罚倾向于更小更分散的权重向量，这就会鼓励分类器最终将所有维度上的特征都用起来，而不是强烈依赖其中少数几个维度。在后面的课程中可以看到，这一效果将会提升分类器的泛化能力，并避免过拟合。

需要注意的是，和权重不同，偏差没有这样的效果，因为它们并不控制输入维度上的影响强度。因此通常只对权重 W 正则化，而不正则化偏差 b 。在实际操作中，可发现这一操作的影响可忽略不计。最后，因为正则化惩罚的存在，不可能在所有的例子中得到0的损失值，这是因为只有当 $W = 0$ 的特殊情况下，才能得到损失值为0。

代码：下面是一个无正则化部分的损失函数的Python实现，有非向量化和半向量化两个形式：

```
def L_i(x, y, W):
    """ unvectorized version. Compute the multiclass svm loss for
    a single example (x,y) - x is a column vector representing
    an image (e.g. 3073 x 1 in CIFAR-10) with an appended bias
    dimension in the 3073-rd position (i.e. bias trick) - y is an
    integer giving index of correct class (e.g. between 0 and 9 in
    CIFAR-10) - W is the weight matrix (e.g. 10 x 3073 in CIFAR-
    10) """
    delta = 1.0 # see notes about delta later in this section
    scores = W.dot(x) # scores becomes of size 10 x 1, the scores
    for each class
    correct_class_score = scores[y]
```

```

D = W.shape[0] # number of classes, e.g. 10
loss_i = 0.0
for j in xrange(D): # iterate over all wrong classes
    if j == y:
        # skip for the true class to only loop over incorrect
        # classes
        continue
    # accumulate loss for the i-th example
    loss_i += max(0, scores[j] - correct_class_score + delta)
return loss_i

def L_i_vectorized(x, y, W):
    """ A faster half-vectorized implementation. half-vectorized
    refers to the fact that for a single example the implementation
    contains no for loops, but there is still one loop over the
    examples (outside this function) """
    delta = 1.0
    scores = W.dot(x)
    # compute the margins for all classes in one vector operation
    margins = np.maximum(0, scores - scores[y] + delta)
    # on y-th position scores[y] - scores[y] canceled and gave
    delta. We want
    # to ignore the y-th position and only consider margin on max
    # wrong class
    margins[y] = 0
    loss_i = np.sum(margins)
    return loss_i

def L(X, y, W):
    """ fully-vectorized implementation : - X holds all the
    training examples as columns (e.g. 3073 x 50,000 in CIFAR-10) - y
    is array of integers specifying correct class (e.g. 50,000-D
    array) - W are weights (e.g. 10 x 3073) """
    # evaluate loss over all examples in X without using any for
    # loops
    # left as exercise to reader in the assignment

```

在本小节的学习中，一定要记得SVM损失采取了一种特殊的方法，使得能够衡量对于训练数据预测分类和实际分类标签的一致性。还有，对训练集中数据做出准确分类预测和让损失值最小化这两件事是等价的。

接下来要做的，就是找到能够使损失值最小化的权重了。

实际考虑

设置Delta：你可能注意到上面的内容对超参数 Δ 及其设置是一笔带过，那么它应该被设置成什么值？需要通过交叉验证来求得吗？现在看来，该超参数在绝大多数情况下设为 $\Delta = 1.0$ 都是安全的。超参数 Δ 和 λ 看起来是两个不同的超参数，但实际上他们一起控制同一个权衡：即损失函数中的数据损失和正则化损失之间的权衡。理解这一点的关键是要知道，权重 W 的大小对于分类分值有直接影响（当然对他们的差异也有直接影响）：当我们缩小 W 中值，分类分值之间的差异也变小，反之亦然。因此，不同分类分值之间的边界的具体值（比如 $\Delta = 1$ 或 $\Delta = 100$ ）从某些角度来看是没意义的，因为权重自己就可以控制差异变大和缩小。也就是说，真正的权衡是我们允许权重能够变大到何种程度（通过正则化强度 λ 来控制）。

与二元支持向量机 (Binary Support Vector Machine) 的关系：在学习本课程前，你可能对于二元支持向量机有些经验，它对于第 i 个数据的损失计算公式是：

$$L_i = C \max(0, 1 - y_i w^T x_i) + R(W)$$

其中， C 是一个超参数，并且 $y_i \in \{-1, 1\}$ 。可以认为本章节介绍的SVM公式包含了上述公式，上述公式是多类支持向量机公式只有两个分类类别的特例。也就是说，如果我们要分类的类别只有两个，那么公式就化为二元SVM公式。这个公式中的 C 和多类SVM公式中的 λ 都控制着同样的权衡，而且它们之间的关系是 $C \propto \frac{1}{\lambda}$

备注：在初始形式中进行最优化。如果在本课程之前学习过SVM，那么对 kernels，duals，SMO算法等将有所耳闻。在本课程（主要是神经网络相关）中，损失函数的最优化的始终在非限制初始形式下进行。很多这些损失函数从技术上来说是不可微的（比如当 $x = y$ 时， $\max(x, y)$ 函数就不可微分），但是在实际操作中并不存在问题，因为通常可以使用次梯度。

备注：其他多类SVM公式。需要指出的是，本课中展示的多类SVM只是多种SVM公式中的一种。另一种常用的公式是 $C_{ne}-Vs-A_{ll}$ (OVA) SVM，它针对每个类和其他类训练一个独立的二元分类器。还有另一种更少用的叫做 $A_{ll}-Vs-A_{ll}$ (AVA) 策略。我们的公式是按照 [Weston and Watkins 1999 \(pdf\)](#) 版本，比OVA性能更强（在构建有一个多类数据集的情况下，这个版本可以在损失值上取到0，而OVA就不行。感兴趣的话在论文中查阅细节）。最后一个需要知道的公式是 Structured SVM，它将正确分类的分类分值和非正确分类中的最高分值的边界最大化。理解这些公式的差异超出了本课程的范围。本课程笔记介绍的版本可以在实践中安全使用，而被论评为最简单的OVA策略在实践中看起来也能工作的同样出色（在 Rikin等人2004年的论文 [In Defense of On-e-Vs-All Classification \(pdf\)](#) 中可查）。

5. 线性分类笔记 (下)

内容列表：

- 线性分类器简介
- 线性评分函数
- 阐明线性分类器
- 损失函数
 - 多类SVM
 - Softmax分类器 **译者注：下篇翻译起始处**
 - SVM和Softmax的比较
- 基于Web的可交互线性分类器原型
- 小结

Softmax分类器

SVM是最常用的两个分类器之一，而另一个就是**Softmax分类器**，它的损失函数与SVM的损失函数不同。对于学习过二元逻辑回归分类器的读者来说，Softmax分类器就可以理解为逻辑回归分类器面对多个分类的一般化归纳。SVM将输出 $f(x_i, W)$ 作为每个分类的评分（因为无定标，所以难以直接解释）。与SVM不同，Softmax的输出（归一化的分类概率）更加直观，并且从概率上可以解释，这一点后文会讨论。在Softmax分类器中，函数映射 $f(x_i; W) = Wx_i$ 保持不变，但将这些评分值视为每个分类的未归一化的对数概率，并且将折叶损失 (hinge loss) 替换为交叉熵损失 (cross-entropy loss)。公式如下：

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \text{ 或等价的 } L_i = -f_{y_i} + \log\left(\sum_j e^{f_j}\right)$$

在上式中，使用 f_j 来表示分类评分向量 f 中的第 j 个元素。和之前一样，整个数据集的损失值是数据集中所有样本数据的损失值 L_i 的均值与正则化损失 $R(W)$ 之和。其中函数 $f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$ 被称作**softmax 函数**：其输入值是

一个向量，向量中元素为任意实数的评分值（ z 中的），函数对其进行压缩，输出一个向量，其中每个元素值在0到1之间，且所有元素之和为1。所以，包含softmax函数的完整交叉熵损失看起唬人，实际上还是比较容易理解的。

信息理论视角：在“真实”分布 p 和估计分布 q 之间的交叉熵定义如下：

$$H(p, q) = - \sum_x p(x) \log q(x)$$

因此，Softmax分类器所做的就是最小化在估计分类概率（就是上面的 $e^{f_{y_i}} / \sum_j e^{f_j}$ ）和“真实”分布之间的交叉熵，在这个解释中，“真实”分布就是所有概率密度都分布在正确的类别上（比如： $p = [0, \dots, 1, \dots, 0]$ 中在 y_i 的位置就有一个单独的1）。还有，既然交叉熵可以写成熵和相对熵（Kullback-Leibler divergence） $H(p, q) = H(p) + D_{KL}(p||q)$ ，并且 delta 函数 p 的熵是0，那么就能等价的看做是对两个分布之间的相对熵做最小化操作。换句话说，交叉熵损失函数“想要”预测分布的所有概率密度都在正确分类上。

译者注：Kullback-Leibler 差异 (Kullback-Leibler Divergence) 也叫做相对熵 (Relative Entropy)，它衡量的是相同事件空间里的两个概率分布的差异情况。

概率论解释：先看下面的公式：

$$P(y_i | x_i, W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

可以解释为是给定图像数据 x_i ，以 W 为参数，分配给正确分类标签 y_i 的归一化概率。为了理解这点，请回忆一下Softmax分类器将输出向量 f 中的评分值解释为没有归一化的对数概率。那么以这些数值做指数函数的幂就得到了没有归一化的概率，而除法操作则对数据进行了归一化处理，使得这些概率的和为1。从概率论的角度来理解，我们就是在最小化正确分类的负对数概率，这可以看做是在进行最大似然估计 (MLE)。该解释的另一个好处是，损失函数中的正则化部分 $R(W)$ 可以被看做是权重矩阵 W 的高斯先验，这里进行的是最大后验估计 (MAP) 而不是最大似然估计。提及这些解释只是为了让读者形成直观的印象，具体细节就超过本课程范围了。

实操事项：数值稳定。 编程实现softmax函数计算的时候，中间项 $e^{f_{y_i}}$ 和 $\sum_j e^{f_j}$ 因为存在指数函数，所以数值可能非常大。除以大数值可能导致数值

计算的不稳定，所以学会使用归一化技巧非常重要。如果在分式的分子和分母都乘以一个常数 C ，并把它变换到求和之中，就能得到一个从数学上等价的公式：

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{Ce^{f_{y_i}}}{C \sum_j e^{f_j}} = \frac{e^{f_{y_i} + \log C}}{\sum_j e^{f_j + \log C}}$$

C 的值可自由选择，不会影响计算结果，通过使用这个技巧可以提高计算中的数值稳定性。通常将 C 设为 $\log C = -\max_j f_j$ 。该技巧简单地说，就是应该将向量 f 中的数值进行平移，使得最大值为0。代码实现如下：

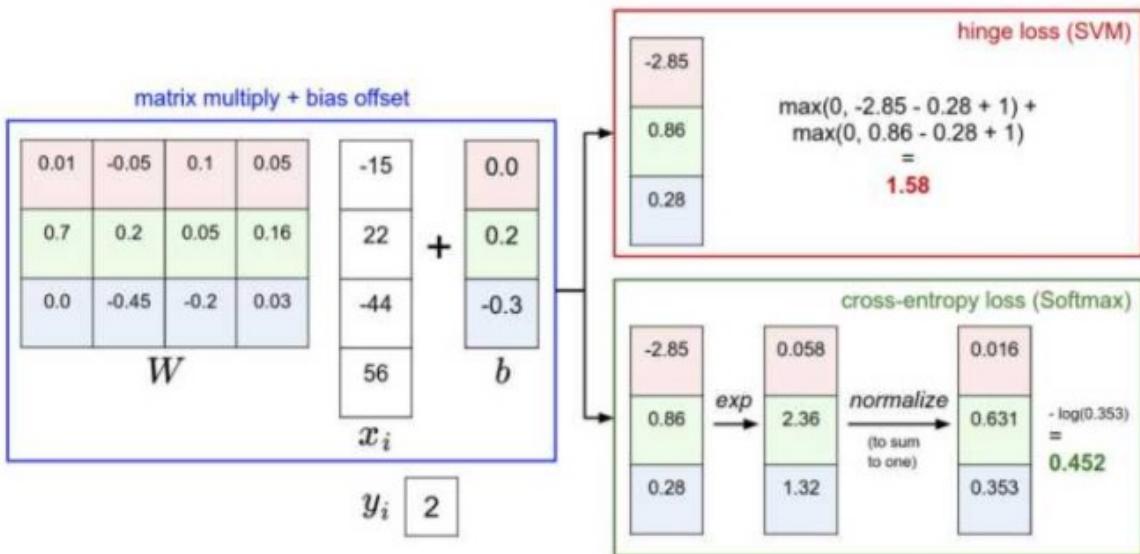
```
f = np.array([123, 456, 789]) # 例子中有3个分类，每个评分的数值都很大
p = np.exp(f) / np.sum(np.exp(f)) # 不妙：数值问题，可能导致数值爆炸

# 那么将f中的值平移到最大值为0：
f -= np.max(f) # f becomes [-666, -333, 0]
p = np.exp(f) / np.sum(np.exp(f)) # 现在OK了，将给出正确结果
```

让人迷惑的命名规则：精确地说，SVM分类器使用的是折叶损失 (*hinge loss*)，有时候又被称为最大边界损失 (*max-margin loss*)。Softmax分类器使用的是交叉熵损失 (*cross-entropy loss*)。Softmax分类器的命名是从 *softmax* 函数那里得来的，softmax函数将原始分类评分变成正的归一化数值，所有数值和为1，这样处理后交叉熵损失才能应用。注意从技术上说“softmax损失 (*softmax loss*)”是没有意义的，因为softmax只是一个压缩数值的函数。但是在这个说法常常被用来做简称。

SVM和Softmax的比较

下图有助于区分这 Softmax和SVM这两种分类器：



针对一个数据点，SVM和Softmax分类器的不同处理方式的例子。两个分类器都计算了同样的分值向量 f (本节中是通过矩阵乘来实现)。不同之处在于对 f 中分值的解释：SVM分类器将它们看做是分类评分，它的损失函数鼓励正确的分类 (本例中是蓝色的类别2) 的分值比其他分类的分值高出至少一个边界值。Softmax分类器将这些数值看做是每个分类没有归一化的对数概率，鼓励正确分类的归一化的对数概率变高，其余的变低。SVM的最终的损失值是1.58，Softmax的最终的损失值是0.452，但要注意这两个数值没有可比性。只在给定同样数据，在同样的分类器的损失值计算中，它们才有意义。

Softmax分类器为每个分类提供了“可能性”：SVM的计算是无标定的，而且难以针对所有分类的评分值给出直观解释。Softmax分类器则不同，它允许我们计算出对于所有分类标签的可能性。举个例子，针对给出的图像，SVM分类器可能给你的是一个[12.5, 0.6, -23.0]对应分类“猫”，“狗”，“船”。而softmax分类器可以计算出这三个标签的“可能性”是[0.9, 0.09, 0.01]，这就让你能看出对于不同分类准确性的把握。为什么我们要在“可能性”上面打引号呢？这是因为可能性分布的集中或离散程度是由正则化参数 λ 直接决定的， λ 是你能直接控制的一个输入参数。举个例子，假设3个分类的原始分数是[1, -2, 0]，那么softmax函数就会计算：

$$[1, -2, 0] \rightarrow [e^1, e^{-2}, e^0] = [2.71, 0.14, 1] \rightarrow [0.7, 0.04, 0.26]$$

现在，如果正则化参数 λ 更大，那么权重 W 就会被惩罚的更多，然后他的权重数值就会更小。这样算出来的分数也会更小，假设小了一半吧[0.5, -1, 0]，那么softmax函数的计算就是：

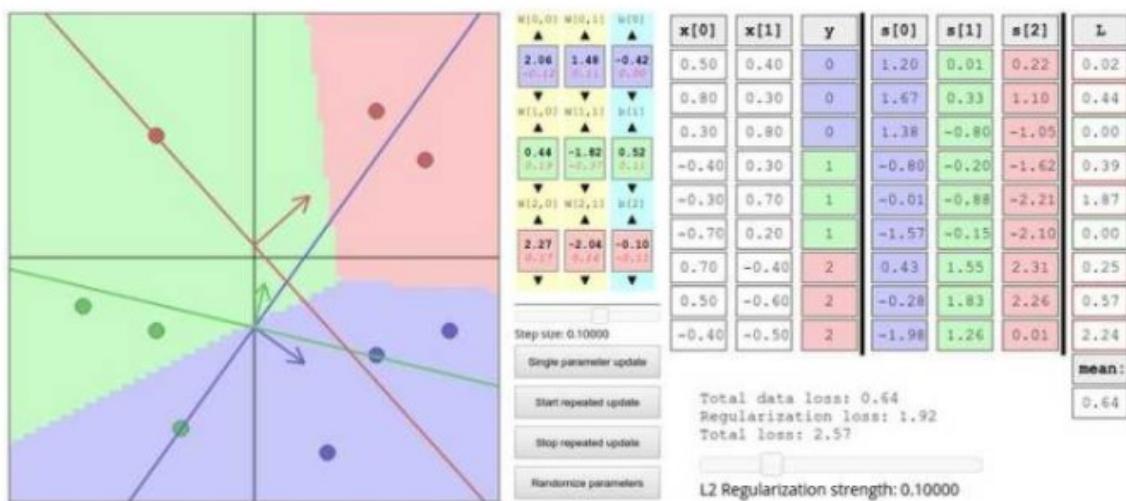
$$[0.5, -1, 0] \rightarrow [e^{0.5}, e^{-1}, e^0] = [1.65, 0.73, 1] \rightarrow [0.55, 0.12, 0.33]$$

现在看起来，概率的分布就更加分散了。还有，随着正则化参数 λ 不断增强，权重数值会越来越小，最后输出的概率会接近于均匀分布。这就是说，softmax分类器算出来的概率最好是看成一种对于分类正确性的自信。和SVM一样，数字间相互比较得出的大小顺序是可以解释的，但其绝对值则难以直观解释。

在实际使用中，SVM和Softmax经常是相似的：通常说来，两种分类器的表现差别很小，不同的人对于哪个分类器更好有不同的看法。相对于Softmax分类器，SVM更加“局部目标化 (local objective)”，这既可以看做是一个特性，也可以看做是一个劣势。考虑一个评分是[10, -2, 3]的数据，其中第一个分类是正确的。那么一个SVM ($\Delta = 1$) 会看到正确分类相较于不正确分类，已经得到了比边界值还要高的分数，它就会认为损失值是0。SVM对于数字个体的细节是不关心的：如果分数是[10, -100, -100]或者[10, 9, 9]，对于SVM来说没设么不同，只要满足超过边界值等于1，那么损失值就等于0。

对于softmax分类器，情况则不同。对于[10, 9, 9]来说，计算出的损失值就远远高于[10, -100, -100]的。换句话来说，softmax分类器对于分数是永远不会满意的：正确分类总能得到更高的可能性，错误分类总能得到更低的可能性，损失值总是能够更小。但是，SVM只要边界值被满足了就满意了，不会超过限制去细微地操作具体分数。这可以被看做是SVM的一种特性。举例说来，一个汽车的分类器应该把他的大量精力放在如何分辨小轿车和大卡车上，而不应该纠结于如何与青蛙进行区分，因为区分青蛙得到的评分已经足够低了。

交互式的网页Demo



我们实现了一个交互式的网页原型，来帮助读者直观地理解线性分类器。原型将损失函数进行可视化，画面表现的是对于2维数据的3种类别的分类。原型在课程进度上稍微超前，展现了最优化的内容，最优化将在下一节课讨论。

小结

总结如下：

- 定义了从图像像素映射到不同类别的分类评分的评分函数。在本节中，评分函数是一个基于权重 W 和偏差 b 的线性函数。
- 与kNN分类器不同，**参数方法**的优势在于一旦通过训练学习到了参数，就可以将训练数据丢弃了。同时该方法对于新的测试数据的预测非常快，因为只需要与权重 W 进行一个矩阵乘法运算。
- 介绍了偏差技巧，让我们能够将偏差向量和权重矩阵合二为一，然后就可以只跟踪一个矩阵。
- 定义了损失函数（介绍了SVM和Softmax线性分类器最常用的2个损失函数）。损失函数能够衡量给出的参数集与训练集数据真实类别情况之间的一致性。在损失函数的定义中可以看到，对训练集数据做出良好预测与得到一个足够低的损失值这两件事是等价的。

现在我们知道了如何基于参数，将数据集中的图像映射成为分类的评分，也知道了两种不同的损失函数，它们都能用来衡量算法分类预测的质量。但是，如何高效地得到能够使损失值最小的参数呢？这个求得最优参数的过程被称为最优化，将在下节课中进行介绍。

6. 最优化笔记（上）

内容列表：

- 简介
- 损失函数可视化
- 最优化
 - 策略#1：随机搜索
 - 策略#2：随机局部搜索
 - 策略#3：跟随梯度 **译者注：上篇截止处**
- 梯度计算
 - 使用有限差值进行数值计算

- 微分计算梯度
- 梯度下降
- 小结

简介

在上一节中，我们介绍了图像分类任务中的两个关键部分：

1. 基于参数的**评分函数**。该函数将原始图像像素映射为分类评分值（例如：一个线性函数）。
2. **损失函数**。该函数能够根据分类评分和训练集图像数据实际分类的一致性，衡量某个具体参数集的质量好坏。损失函数有多种版本和不同的实现方式（例如：Softmax或SVM）。

上节中，线性函数的形式是 $f(x_i, W) = Wx_i$ ，而SVM实现的公式是：

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} [max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1)] + \alpha R(W)$$

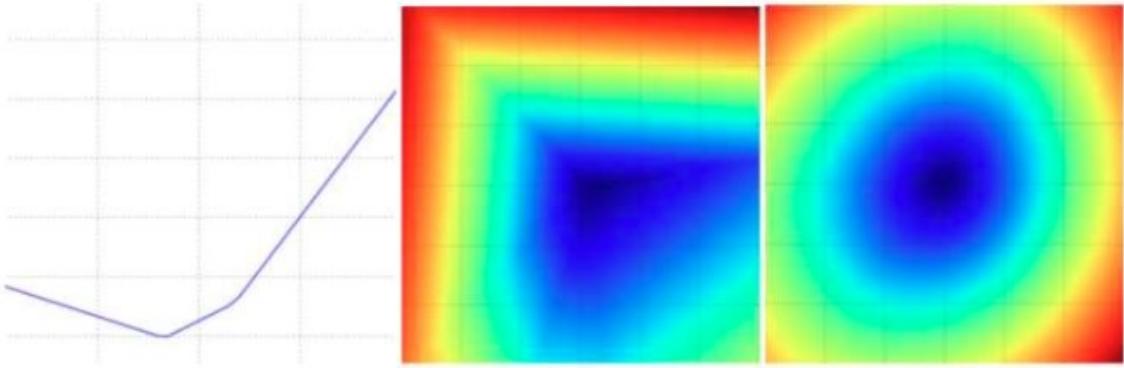
对于图像数据 x_i ，如果基于参数集 W 做出的分类预测与真实情况比较一致，那么计算出来的损失值 L 就很低。现在介绍第三个，也是最后一个关键部分：**最优化Optimization**。最优化是寻找能使得损失函数值最小化的参数 W 的过程。

铺垫：一旦理解了这三个部分是如何相互运作的，我们将会回到第一个部分（基于参数的函数映射），然后将其拓展为一个远比线性函数复杂的函数：首先是神经网络，然后是卷积神经网络。而损失函数和最优化过程这两个部分将会保持相对稳定。

损失函数可视化

本课中讨论的损失函数一般都是定义在高维度的空间中（比如，在CIFAR-10中一个线性分类器的权重矩阵大小是[10x3073]，就有30730个参数），这样要将其可视化就很困难。然而办法还是有的，在1个维度或者2个维度的方向上对高维空间进行切片，就能得到一些直观感受。例如，随机生成一个权重矩阵 W ，该矩阵就与高维空间中的一个点对应。然后沿着某个维度方向前进的同时记录损失函数值的变化。换句话说，就是生成一个随机的方向 W_1 并且沿着此方向计算损失值，计算方法是根据不同的 a 值来计算 $L(W + aW_1)$ 。这个过程将生成一个图表，其x轴是 a 值，y轴是损失函数值。同样的方法还可以用

在两个维度上，通过改变 a, b 来计算损失值 $L(W + aW_1 + bW_2)$ ，从而给出二维的图像。在图像中， a, b 可以分别用x和y轴表示，而损失函数的值可以用颜色变化表示：



一个无正则化的多类SVM的损失函数的图示。左边和中间只有一个样本数据，右边是CIFAR-10中的100个数据。左： a 值变化在某个维度方向上对应的损失值变化。中和右：两个维度方向上的损失值切片图，蓝色部分是低损失值区域，红色部分是高损失值区域。注意损失函数的分段线性结构。多个样本的损失值是总体的平均值，所以右边的碗状结构是很多的分段线性结构的平均（比如中间这个就是其中之一）。

我们可以通过数学公式来解释损失函数的分段线性结构。对于一个单独的数据，有损失函数的计算公式如下：

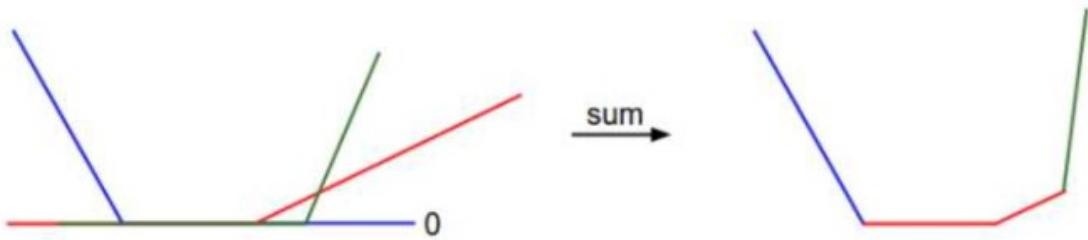
$$L_i = \sum_{j \neq y_i} [\max(0, w_j^T x_i - w_{y_i}^T x_i + 1)]$$

通过公式可见，每个样本的数据损失值是以 W 为参数的线性函数的总和（零阈值来源于 $\max(0, -)$ 函数）。 W 的每一行（即 w_j ），有时候它前面是一个正号（比如当它对应错误分类的时候），有时候它前面是一个负号（比如当它是正确分类的时候）。为进一步阐明，假设有一个简单的数据集，其中包含有3个只有1个维度的点，数据集数据点有3个类别。那么完整的无正则化SVM的损失值计算如下：

$$\begin{aligned} L_0 &= \max(0, w_1^T x_0 - w_0^T x_0 + 1) + \max(0, w_2^T x_0 - w_0^T x_0 + 1) \\ L_1 &= \max(0, w_0^T x_1 - w_1^T x_1 + 1) + \max(0, w_2^T x_1 - w_1^T x_1 + 1) \\ L_2 &= \max(0, w_0^T x_2 - w_2^T x_2 + 1) + \max(0, w_1^T x_2 - w_2^T x_2 + 1) \\ L &= (L_0 + L_1 + L_2)/3 \end{aligned}$$

因为这些例子都是一维的，所以数据 x_i 和权重 w_j 都是数字。观察 w_0 ，可以看到上面的式子中一些项是 w_0 的线性函数，且每一项都会与0比较，取两者

的最大值。可作图如下：



从一个维度方向上对数据损失值的展示。x轴方向就是一个权重，y轴就是损失值。数据损失是多个部分组合而成。其中每个部分要么是某个权重的独立部分，要么是该权重的线性函数与0阈值的比较。完整的SVM数据损失就是这个形状的30730维版本。

需要多说一句的是，你可能根据SVM的损失函数的碗状外观猜出它是一个**凸函数**。关于如何高效地最小化凸函数的论文有很多，你也可以学习斯坦福大学关于(**凸函数最优化**)的课程。但是一旦我们将 f 函数扩展到神经网络，目标函数就不再是凸函数了，图像也不会像上面那样是个碗状，而是凹凸不平的复杂地形形状。

不可导的损失函数。作为一个技术笔记，你要注意到：由于 \max 操作，损失函数中存在一些**不可导点 (kinks)**，这些点使得损失函数不可微，因为在这些不可导点，梯度是没有定义的。但是**次梯度 (subgradient)**依然存在且常常被使用。在本课中，我们将交换使用**次梯度**和**梯度**两个术语。

最优化 Optimization

重申一下：损失函数可以量化某个具体权重集 \mathbf{W} 的质量。而最优化的目标就是找到能够最小化损失函数值的 \mathbf{W} 。我们现在就朝着这个目标前进，实现一个能够最优化损失函数的方法。对于有一些经验的同学，这节课看起来有点奇怪，因为使用的例子(SVM 损失函数)是一个凸函数问题。但是要记得，最终的目标是不仅仅对凸函数做最优化，而是能够最优化一个神经网络，而对于神经网络是不能简单的使用凸函数的最优化技巧的。

策略#1：一个差劲的初始方案：随机搜索

既然确认参数集 \mathbf{W} 的好坏蛮简单的，那第一个想到的(差劲)方法，就是可以随机尝试很多不同的权重，然后看其中哪个最好。过程如下：

```
# 假设x_train的每一列都是一个数据样本（比如3073 x 50000）# 假设y_train  
是数据样本的类别标签（比如一个长50000的一维数组）# 假设函数L对损失函数进行
```

评价

```
bestloss = float("inf") # Python assigns the highest possible float
value
for num in xrange(1000):
    W = np.random.randn(10, 3073) * 0.0001 # generate random
parameters
    loss = L(X_train, Y_train, W) # get the loss over the entire
training set
    if loss < bestloss: # keep track of the best solution
        bestloss = loss
        bestW = W
print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# 输出:# in attempt 0 the loss was 9.401632, best 9.401632# in
attempt 1 the loss was 8.959668, best 8.959668# in attempt 2 the
loss was 9.044034, best 8.959668# in attempt 3 the loss was
9.278948, best 8.959668# in attempt 4 the loss was 8.857370, best
8.857370# in attempt 5 the loss was 8.943151, best 8.857370# in
attempt 6 the loss was 8.605604, best 8.605604# ... (truncated:
continues for 1000 lines)
```

在上面的代码中，我们尝试了若干随机生成的权重矩阵W，其中某些的损失值较小，而另一些的损失值大些。我们可以把这次随机搜索中找到的最好的权重W取出，然后去跑测试集：

```
# 假设X_test尺寸是[3073 x 10000], Y_test尺寸是[10000 x 1]
scores = Wbest.dot(Xte[:, :]) # 10 x 10000, the class scores for all
test examples# 找到在每列中评分值最大的索引(即预测的分类)
Yte_predict = np.argmax(scores, axis = 0)# 以及计算准确率
np.mean(Yte_predict == Yte)# 返回 0.1555
```

验证集上表现最好的权重W跑测试集的准确率是15.5%，而完全随机猜的准确率是10%，如此看来，这个准确率对于这样一个不经过大脑的策略来说，还算不错嘛！

核心思路：迭代优化。当然，我们肯定能做得更好些。核心思路是：虽然找到最优的权重W非常困难，甚至是不可能的（尤其当W中存的是整个神经网络的权重的时候），但如果问题转化为：对一个权重矩阵集W取优，使其损失值稍微减少。那么问题的难度就大大降低了。换句话说，我们的方法从一个随机的W开始，然后对其迭代取优，每次都让它的损失值变得更小一点。

我们的策略是从随机权重开始，然后迭代取优，从而获得更低的损失值。

蒙眼徒步者的比喻：一个助于理解的比喻是把你自己的眼睛蒙住，正走在山地地形上，目标是要慢慢走到山底。在CIFAR-10的例子中，这山是30730维的（因为 W 是3073x10）。我们在山上踩的每一点都对应一个的损失值，该损失值可以看做该点的海拔高度。

策略#2：随机本地搜索

第一个策略可以看做是每走一步都尝试几个随机方向，如果某个方向是向山下的，就向该方向走一步。这次我们从一个随机 W 开始，然后生成一个随机的扰动 δW ，只有当 $W + \delta W$ 的损失值变低，我们才会更新。这个过程的具体代码如下：

```
W = np.random.randn(10, 3073) * 0.001 # 生成随机初始W
bestloss = float("inf")
for i in xrange(1000):
    step_size = 0.0001
    Wtry = W + np.random.randn(10, 3073) * step_size
    loss = L(Xtr_cols, Ytr, Wtry)
    if loss < bestloss:
        W = Wtry
        bestloss = loss
    print 'iter %d loss is %f' % (i, bestloss)
```

使用同样的数据（1000），这个方法可以得到**21.4%**的分类准确率。这个比策略一好，但是依然过于浪费计算资源。

策略#3：跟随梯度

前两个策略中，我们是尝试在权重空间中找到一个方向，沿着该方向能降低损失函数的损失值。其实不需要随机寻找方向，因为可以直接计算出最好的方向，这就是从数学上计算出最陡峭的方向。这个方向就是损失函数的**梯度**（**gradient**）。在蒙眼徒步者的比喻中，这个方法就好比是感受我们脚下山体的倾斜程度，然后向着最陡峭的下降方向下山。

在一维函数中，斜率是函数在某一点的瞬时变化率。梯度是函数的斜率的一般化表达，它不是一个值，而是一个向量。在输入空间中，梯度是各个维度的斜率组成的向量（或者称为导数**derivatives**）。对一维函数的求导公式如下：

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

当函数有多个参数的时候，我们称导数为偏导数。而梯度就是在每个维度上偏

导数所形成的向量。

7. 最优化笔记 (下)

内容列表：

- 简介
- 损失函数可视化
- 最优化
 - 策略#1：随机搜索
 - 策略#2：随机局部搜索
 - 策略#3：跟随梯度
- 梯度计算 **译者注：下篇起始处**
 - 使用有限差值进行数值计算
 - 微分分析计算梯度
- 梯度下降
- 小结

梯度计算

计算梯度有两种方法：一个是缓慢的近似方法（**数值梯度法**），但实现相对简单。另一个方法（**分析梯度法**）计算迅速，结果精确，但是实现时容易出错，且需要使用微分。现在对两种方法进行介绍：

利用有限差值计算梯度

上节中的公式已经给出数值计算梯度的方法。下面代码是一个输入为函数f和向量x，计算f的梯度的通用函数，它返回函数f在点x处的梯度：

```
def eval_numerical_gradient(f, x):  
    """ 一个f在x处的数值梯度法的简单实现 - f是只有一个参数的函数 - x是计算梯度  
    的点 """
```

```

fx = f(x) # 在原点计算函数值
grad = np.zeros(x.shape)
h = 0.00001

# 对x中所有的索引进行迭代
it = np.nditer(x, flags=[ 'multi_index' ], op_flags=[ 'readwrite' ])
while not it.finished:

    # 计算x+h处的函数值
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # 增加h
    fxh = f(x) # 计算f(x + h)
    x[ix] = old_value # 存到前一个值中 (非常 important)

    # 计算偏导数
    grad[ix] = (fxh - fx) / h # 坡度
    it.iternext() # 到下个维度

return grad

```

根据上面的梯度公式，代码对所有维度进行迭代，在每个维度上产生一个很小的变化h，通过观察函数值变化，计算函数在该维度上的偏导数。最后，所有的梯度存储在变量grad中。

实践考量：注意在数学公式中，h的取值是趋近于0的，然而在实际中，用一个很小的数值（比如例子中的1e-5）就足够了。在不产生数值计算出错的理想前提下，你会使用尽可能小的h。还有，实际中用**中心差值公式 (centered difference formula)** $[f(x + h) - f(x - h)]/2h$ 效果较好。细节可查看[Wikipedia](#)。

可以使用上面这个公式来计算任意函数在任意点上的梯度。下面计算权重空间中的某些随机点上，CIFAR-10损失函数的梯度：

```

# 要使用上面的代码我们需要一个只有一个参数的函数# (在这里参数就是权重)所以
# 也包含了X_train和Y_train
def CIFAR10_loss_fun(W):
    return L(X_train, Y_train, W)

W = np.random.rand(10, 3073) * 0.001 # 随机权重向量
df = eval_numerical_gradient(CIFAR10_loss_fun, W) # 得到梯度

```

梯度告诉我们损失函数在每个维度上的斜率，以此来进行更新：

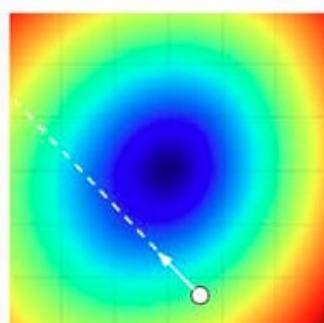
```
loss_original = CIFAR10_loss_fun(W) # 初始损失值
print 'original loss: %f' %(loss_original, )

# 查看不同步长的效果
for step_size_log in [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]:
    step_size = 10 ** step_size_log
    W_new = W - step_size * df # 权重空间中的新位置
    loss_new = CIFAR10_loss_fun(W_new)
    print 'for step size %f new loss: %f' %(step_size, loss_new)

# 输出:# original loss: 2.200718# for step size 1.000000e-10 new
# loss: 2.200652# for step size 1.000000e-09 new loss: 2.200057#
# for step size 1.000000e-08 new loss: 2.194116# for step size
1.000000e-07 new loss: 2.135493# for step size 1.000000e-06 new
loss: 1.647802# for step size 1.000000e-05 new loss: 2.844355#
# for step size 1.000000e-04 new loss: 25.558142# for step size
1.000000e-03 new loss: 254.086573# for step size 1.000000e-02 new
loss: 2539.370888# for step size 1.000000e-01 new loss:
25392.214036
```

在梯度负方向上更新：在上面的代码中，为了计算W_new，要注意我们是向着梯度df的负方向去更新，这是因为我们希望损失函数值是降低而不是升高。

步长的影响：梯度指明了函数在哪个方向是变化率最大的，但是没有指明在这个方向上应该走多远。在后续的课程中可以看到，选择步长（也叫作学习率）将会是神经网络训练中最重要（也是最头痛）的超参数设定之一。还是用蒙眼徒步者下山的比喻，这就好比我们可以感觉到脚朝向的不同方向上，地形的倾斜程度不同。但是该跨出多长的步长呢？不确定。如果谨慎地小步走，情况可能比较稳定但是进展较慢（这就是步长较小的情况）。相反，如果想尽快下山，那就大步走吧，但结果也不一定尽如人意。在上面的代码中就能看见反例，在某些点如果步长过大，反而可能越过最低点导致更高的损失值。



将步长效果视觉化的图例。从某个具体的点W开始计算梯度（白箭头方向是负梯度方向），梯度告诉了我们损失函数下降最陡峭的方向。小步长下降稳定但进度慢，大步长进展快但是风险更大。采取大步长可能导致错过最优点，让损失值上升。步长（后面会称其为学习率）将会是我们在调参中最重要的超参数之一。

效率问题：你可能已经注意到，计算数值梯度的复杂性和参数的量线性相关。在本例中有30730个参数，所以损失函数每走一步就需要计算30731次损失函数的梯度。现代神经网络很容易就有上千万的参数，因此这个问题只会越发严峻。显然这个策略不适合大规模数据，我们需要更好的策略。

微分分析计算梯度

使用有限差值近似计算梯度比较简单，但缺点在于终究只是近似（因为我们对于 h 值是选取了一个很小的数值，但真正的梯度定义中 h 趋向0的极限），且耗费计算资源太多。第二个梯度计算方法是利用微分来分析，能得到计算梯度的公式（不是近似），用公式计算梯度速度很快，唯一不好的就是实现的时候容易出错。为了解决这个问题，在实际操作时常常将分析梯度法的结果和数值梯度法的结果作比较，以此来检查其实现的正确性，这个步骤叫做**梯度检查**。

用SVM的损失函数在某个数据点上的计算来举例：

$$L_i = \sum_{j \neq y_i} [\max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)]$$

可以对函数进行微分。比如，对 w_{y_i} 进行微分得到：

$$\nabla_{w_{y_i}} L_i = -\left(\sum_{j \neq y_i} 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0)\right) x_i$$

译者注：原公式中1为空心字体，尝试\mathbb{}等多种方法仍无法实现，请知友指点。

其中1是一个示性函数，如果括号中的条件为真，那么函数值为1，如果为假，则函数值为0。虽然上述公式看起来复杂，但在代码实现的时候比较简单：只需要计算没有满足边界值的分类的数量（因此对损失函数产生了贡献），然后乘以 x_i 就是梯度了。注意，这个梯度只是对应正确分类的W的行向量的梯度，那些 $j \neq y_i$ 行的梯度是：

$$\nabla_{w_j} L_i = 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

一旦将梯度的公式微分出来，代码实现公式并用于梯度更新就比较顺畅了。

梯度下降

现在可以计算损失函数的梯度了，程序重复地计算梯度然后对参数进行更新，这一过程称为梯度下降，他的普通版本是这样的：

```
# 普通的梯度下降

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad # 进行梯度更新
```

这个简单的循环在所有的神经网络核心库中都有。虽然也有其他实现最优化的方法（比如LBFGS），但是到目前为止，梯度下降是对神经网络的损失函数最优化中最常用的方法。课程中，我们会在它的循环细节增加一些新的东西（比如更新的具体公式），但是核心思想不变，那就是我们一直跟着梯度走，直到结果不再变化。

小批量数据梯度下降 (Mini-batch gradient descent)：在大规模的应用中（比如ILSVRC挑战赛），训练数据可以达到百万级量级。如果像这样计算整个训练集，来获得仅仅一个参数的更新就太浪费了。一个常用的方法是计算训练集中的**小批量 (batches)**数据。例如，在目前最高水平的卷积神经网络中，一个典型的小批量包含256个例子，而整个训练集是多少呢？一百二十万个。这个小批量数据就用来实现一个参数更新：

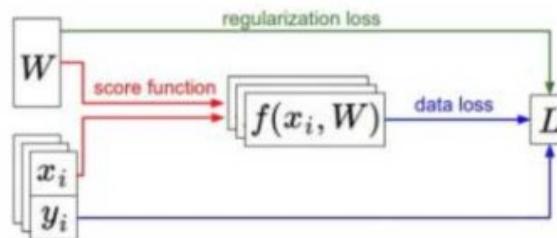
```
# 普通的小批量数据梯度下降

while True:
    data_batch = sample_training_data(data, 256) # 256个数据
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += - step_size * weights_grad # 参数更新
```

这个方法之所以效果不错，是因为训练集中的数据都是相关的。要理解这一点，可以想象一个极端情况：在ILSVRC中的120万个图像是1000张不同图片的复制（每个类别1张图片，每张图片有1200张复制）。那么显然计算这1200张复制图像的梯度就应该是一样的。对比120万张图片的数据损失的均值与只计算1000张的子集的数据损失均值时，结果应该是一样的。实际情况中，数据集肯定不会包含重复图像，那么小批量数据的梯度就是对整个数据集梯度的一个近似。因此，在实践中通过计算小批量数据的梯度可以实现更快速地收敛，并以此来进行更频繁的参数更新。

小批量数据策略有个极端情况，那就是每个批量中只有1个数据样本，这种策略被称为**随机梯度下降 (Stochastic Gradient Descent 简称SGD)**，有时候也被称为**在线梯度下降**。这种策略在实际情况中相对少见，因为向量化操作的代码一次计算100个数据 比100次计算1个数据要高效很多。即使SGD在技术上是指每次使用1个数据来计算梯度，你还是会听到人们使用SGD来指代小批量数据梯度下降（或者用MGD来指代小批量数据梯度下降，而BGD来指代则相对少见）。小批量数据的大小是一个超参数，但是一般并不需要通过交叉验证来调参。它一般由存储器的限制来决定的，或者干脆设置为同样大小，比如32, 64, 128等。之所以使用2的指数，是因为在实际中许多向量化操作实现的时候，如果输入数据量是2的倍数，那么运算更快。

小结



信息流的总结图例。数据集中的 (x, y) 是给定的。权重从一个随机数字开始，且可以改变。在前向传播时，评分函数计算出类别的分类评分并存储在向量 f 中。损失函数包含两个部分：数据损失和正则化损失。其中，数据损失计算的是分类评分 f 和实际标签 y 之间的差异，正则化损失只是一个关于权重的函数。在梯度下降过程中，我们计算权重的梯度（如果愿意的话，也可以计算数据上的梯度），然后使用它们来实现参数的更新。

在本节课中：

- 将损失函数比作了一个**高维度的最优化地形**，并尝试到达它的最底部。最优化的工作过程可以看做一个蒙着眼睛的徒步者希望摸索着走到山的底部。在例子中，可见SVM的损失函数是分段线性的，并且是碗状的。
- 提出了迭代优化的思想，从一个随机的权重开始，然后一步步地让损失值变小，直到最小。
- 函数的**梯度**给出了该函数最陡峭的上升方向。介绍了利用有限的差值来近似计算梯度的方法，该方法实现简单但是效率较低（有限差值就是 h ，用来计算数值梯度）。
- 参数更新需要有技巧地设置**步长**。也叫学习率。如果步长太小，进度稳

定但是缓慢，如果步长太大，进度快但是可能有风险。

- 讨论权衡了数值梯度法和分析梯度法。数值梯度法计算简单，但结果只是近似且耗费计算资源。分析梯度法计算准确迅速但是实现容易出错，而且需要对梯度公式进行推导的数学基本功。因此，在实际中使用分析梯度法，然后使用**梯度检查**来检查其实现正确与否，其本质就是将分析梯度法的结果与数值梯度法的计算结果对比。
- 介绍了**梯度下降算法**，它在循环中迭代地计算梯度并更新参数。

预告：这节课的核心内容是：理解并能计算损失函数关于权重的梯度，是设计、训练和理解神经网络的核心能力。下节中，将介绍如何使用链式法则来高效地计算梯度，也就是通常所说的**反向传播（backpropagation）机制**。该机制能够对包含卷积神经网络在内的几乎所有类型的神经网络的损失函数进行高效的最优化。

8. 反向传播笔记

内容列表：

- 简介
- 简单表达式和理解梯度
- 复合表达式，链式法则，反向传播
- 直观理解反向传播
- 模块：Sigmoid例子
- 反向传播实践：分段计算
- 回传流中的模式
- 用户向量化操作的梯度
- 小结

简介

目标：本节将帮助读者对**反向传播**形成直观而专业的理解。反向传播是利用**链式法则**递归计算表达式的梯度的方法。理解反向传播过程及其精妙之处，对于理解、实现、设计和调试神经网络非常**关键**。

问题陈述：这节的核心问题是：给定函数 $f(\mathbf{x})$ ，其中 \mathbf{x} 是输入数据的向量，需要计算函数 f 关于 \mathbf{x} 的梯度，也就是 $\nabla f(\mathbf{x})$ 。

目标：之所以关注上述问题，是因为在神经网络中 f 对应的是损失函数（ L ），输入 \mathbf{x} 里面包含训练数据和神经网络的权重。举个例子，损失函数可以是 SVM 的损失函数，输入则包含了训练数据 $(\mathbf{x}_i, y_i), i = 1 \dots N$ 、权重 \mathbf{W} 和偏差 b 。注意训练集是给定的（在机器学习中通常都是这样），而权重是可以控制的变量。因此，即使能用反向传播计算输入数据 \mathbf{x}_i 上的梯度，但在实践为了进行参数更新，通常也只计算参数（比如 \mathbf{W}, b ）的梯度。然而 \mathbf{x}_i 的梯度有时仍然是有用的：比如将神经网络所做的事情可视化便于直观理解的时候，就能用上。

如果读者之前对于利用链式法则计算偏微分已经很熟练，仍然建议浏览本篇笔记。因为它呈现了一个相对成熟的反向传播视角，在该视角中能看见基于实数值回路的反向传播过程，而对其细节的理解和收获将帮助读者更好地通过本课程。

简单表达式和理解梯度

从简单表达式入手可以为复杂表达式打好符号和规则基础。先考虑一个简单的二元乘法函数 $f(x, y) = xy$ 。对两个输入变量分别求偏导数还是很简单的：

$$f(x, y) = xy \rightarrow \frac{df}{dx} = y \quad \frac{df}{dy} = x$$

解释：牢记这些导数的意义：函数变量在某个点周围的极小区域内变化，而导数就是变量变化导致的函数在该方向上的变化率。

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

注意等号左边的分号和等号右边的分号不同，不是代表分数。相反，这个符号表示操作符 $\frac{d}{dx}$ 被应用于函数 f ，并返回一个不同的函数（导数）。对于上述公式，可以认为 h 值非常小，函数可以被一条直线近似，而导数就是这条直线的斜率。换句话说，每个变量的导数指明了整个表达式对于该变量的值的敏感程度。比如，若 $x = 4, y = -3$ ，则 $f(x, y) = -12$ ， x 的导数 $\frac{\partial f}{\partial x} = -3$

。这就说明如果将变量 x 的值变大一点，整个表达式的值就会变小（原因在于负号），而且变小的量是 x 变大的量的三倍。通过重新排列公式可以看到这一

点 ($f(x + h) = f(x) + h \frac{df(x)}{dx}$)。同样，因为 $\frac{\partial f}{\partial y} = 4$ ，可以知道如果将 y 的值增加 h ，那么函数的输出也将增加（原因在于正号），且增加量是 $4h$ 。

函数关于每个变量的导数指明了整个表达式对于该变量的敏感程度。

如上所述，梯度 ∇f 是偏导数的向量，所以有 $\nabla f(x) = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}] = [y, x]$ 。即使是梯度实际上是一个向量，仍然通常使用类似“ x 上的梯度”的术语，而不是使用如“ x 的偏导数”的正确说法，原因是因为前者说起来简单。

我们也可以对加法操作求导：

$$f(x, y) = x + y \rightarrow \frac{df}{dx} = 1 \quad \frac{df}{dy} = 1$$

这就是说，无论其值如何， x, y 的导数均为1。这是有道理的，因为无论增加 x, y 中任一个的值，函数 f 的值都会增加，并且增加的变化率独立于 x, y 的具体值（情况和乘法操作不同）。取最大值操作也是常常使用的：

$$f(x, y) = \max(x, y) \rightarrow \frac{df}{dx} = 1(x \geq y) \quad \frac{df}{dy} = 1(y \geq x)$$

上式是说，如果该变量比另一个变量大，那么梯度是1，反之为0。例如，若 $x = 4, y = 2$ ，那么 \max 是4，所以函数对于 y 就不敏感。也就是说，在 y 上增加 h ，函数还是输出为4，所以梯度是0：因为对于函数输出是没有效果的。当然，如果给 y 增加一个很大的量，比如大于2，那么函数 f 的值就变化了，但是导数并没有指明输入量有巨大变化情况对于函数的效果，他们只适用于输入量变化极小时的情况，因为定义已经指明： $\lim_{h \rightarrow 0}$ 。

使用链式法则计算复合表达式

现在考虑更复杂的包含多个函数的复合函数，比如 $f(x, y, z) = (x + y)z$ 。虽然这个表达足够简单，可以直接微分，但是在此使用一种有助于读者直观理解反向传播的方法。将公式分成两部分： $q = x + y$ 和 $f = qz$ 。在前面已经介绍过如何对这分开的两个公式进行计算，因为 f 是 q 和 z 相乘，所以 $\frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$ ，又因为 q 是 x 加 y ，所以 $\frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$ 。然而，并

不需要关心中间量 q 的梯度，因为 $\frac{\partial f}{\partial q}$ 没有用。相反，函数 f 关于 x, y, z 的

梯度才是需要关注的。链式法则指出将这些梯度表达式链接起来的正确方式是相乘，比如 $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$ 。在实际操作中，这只是简单地将两个梯度数值相乘，示例代码如下：

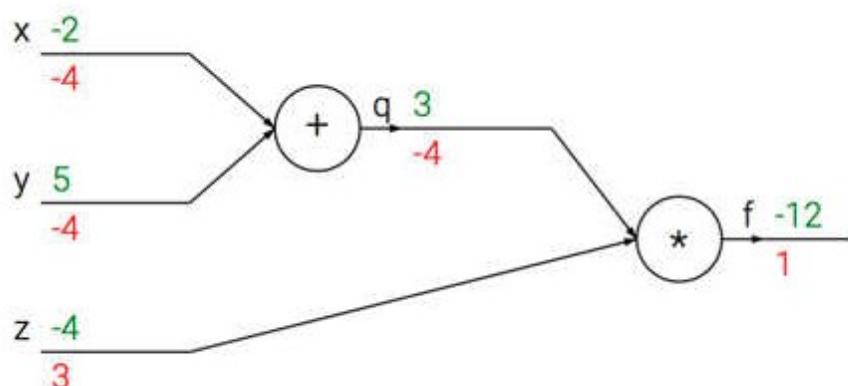
```
# 设置输入值
x = -2; y = 5; z = -4

# 进行前向传播
q = x + y # q becomes 3
f = q * z # f becomes -12

# 进行反向传播:# 首先回传到 f = q * z
dfdz = q # df/dz = q, 所以关于z的梯度是3
dfdq = z # df/dq = z, 所以关于q的梯度是-4# 现在回传到q = x + y
dfdx = 1.0 * dfdq # dq/dx = 1. 这里的乘法是因为链式法则
dfdy = 1.0 * dfdq # dq/dy = 1
```

最后得到变量的梯度[**dfdx, dfdy, dfdz**]，它们告诉我们函数f对于变量[x, y, z]的敏感程度。这是一个最简单的反向传播。一般会使用一个更简洁的表达符号，这样就不用写df了。这就是说，用dq来代替dfdq，且总是假设梯度是关于最终输出的。

这次计算可以被可视化为如下计算线路图像：



上图的真实值计算线路展示了计算的视觉化过程。前向传播从输入计算到输出（绿色），反向传播从尾部开始，根据链式法则递归地向前计算梯度（显示为红色），一直到网络的输入端。可以认为，梯度是从计算链路中回流。

反向传播的直观理解

反向传播是一个优美的局部过程。在整个计算线路图中，每个门单元都会得到一些输入并立即计算两个东西：1. 这个门的输出值，和2.其输出值关于输入值的局部梯度。门单元完成这两件事是完全独立的，它不需要知道计算线路中的其他细节。然而，一旦前向传播完毕，在反向传播的过程中，门单元将最终获得整个网络的最终输出值在自己的输出值上的梯度。链式法则指出，门单元应该将回传的梯度乘以它对其的输入的局部梯度，从而得到整个网络的输出对该门单元的每个输入值的梯度。

这里对于每个输入的乘法操作是基于链式法则的。该操作让一个相对独立的门单元变成复杂计算线路中不可或缺的一部分，这个复杂计算线路可以是神经网络等。

下面通过例子来对这一过程进行理解。加法门收到了输入[-2, 5]，计算输出是3。既然这个门是加法操作，那么对于两个输入的局部梯度都是+1。网络的其余部分计算出最终值为-12。在反向传播时将递归地使用链式法则，算到加法门（是乘法门的输入）的时候，知道加法门的输出的梯度是-4。如果网络如果想要输出值更高，那么可以认为它会想要加法门的输出更小一点（因为负号），而且还有一个4的倍数。继续递归并对梯度使用链式法则，加法门拿到梯度，然后把这个梯度分别乘到每个输入值的局部梯度（就是让-4乘以x和y的局部梯度，x和y的局部梯度都是1，所以最终都是-4）。可以看到得到了想要的效果：如果x, y减小（它们的梯度为负），那么加法门的输出值减小，这会让乘法门的输出值增大。

因此，反向传播可以看做是门单元之间在通过梯度信号相互通信，只要让它们的输入沿着梯度方向变化，无论它们自己的输出值在何种程度上升或降低，都是为了让整个网络的输出值更高。

模块化：Sigmoid例子

上面介绍的门是相对随意的。任何可微分的函数都可以看做门。可以将多个门组合成一个门，也可以根据需要将一个函数分拆成多个门。现在看看一个表达式：

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

在后面的课程中可以看到，这个表达式描述了一个含输入x和权重w的2维的神经元，该神经元使用了*sigmoid*激活函数。但是现在只是看做是一个简单的输入为x和w，输出为一个数字的函数。这个函数是由多个门组成的。除了上文

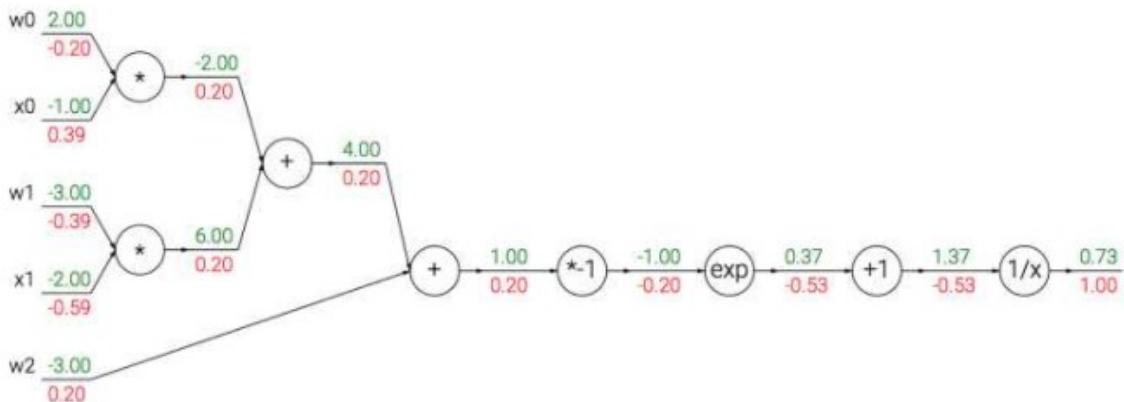
介绍的加法门，乘法门，取最大值门，还有下面这4种：

$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1 \quad f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

其中，函数 f_c 使用对输入值进行了常量 c 的平移， f_a 将输入值扩大了常量 a 倍。它们是加法和乘法的特例，但是这里将其看做一元门单元，因为确实需要计算常量 c, a 的梯度。整个计算线路如下：



使用sigmoid激活函数的2维神经元的例子。输入是 $[x_0, x_1]$ ，可学习的权重是 $[w_0, w_1, w_2]$ 。一会儿会看见，这个神经元对输入数据做点积运算，然后其激活数据被sigmoid函数挤压到0到1之间。

在上面的例子中可以看见一个函数操作的长链条，链条上的门都对w和x的点积结果进行操作。该函数被称为sigmoid函数 $\sigma(x)$ 。sigmoid函数关于其输入的求导是可以简化的(使用了在分子上先加后减1的技巧)：

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\rightarrow \frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}}\right)\left(\frac{1}{1 + e^{-x}}\right) = (1 - \sigma(x))\sigma(x)$$

可以看到梯度计算简单了很多。举个例子，sigmoid表达式输入为1.0，则在前向传播中计算出输出为0.73。根据上面的公式，局部梯度为 $(1 - 0.73) * 0.73 \approx 0.2$ ，和之前的计算流程比起来，现在的计算使用一个单独的简单表达式即可。因此，在实际的应用中将这些操作装进一个单独的门单元中将会非常有用。该神经元反向传播的代码实现如下：

```

w = [2, -3, -3] # 假设一些随机数据和权重
x = [-1, -2]

# 前向传播
dot = w[0]*x[0] + w[1]*x[1] + w[2]
f = 1.0 / (1 + math.exp(-dot)) # sigmoid函数

# 对神经元反向传播
ddot = (1 - f) * f # 点积变量的梯度，使用sigmoid函数求导
dx = [w[0] * ddot, w[1] * ddot] # 传回到x
dw = [x[0] * ddot, x[1] * ddot, 1.0 * ddot] # 传回到w# 完成！得到输入的梯度

```

实现提示：分段反向传播。上面的代码展示了在实际操作中，为了使反向传播过程更加简洁，把向前传播分成不同的阶段将是很有帮助的。比如我们创建了一个中间变量dot，它装着w和x的点乘结果。在反向传播的时，就可以（反向地）计算出装着w和x等的梯度的对应的变量（比如ddot，dx和dw）。

本节的要点就是展示反向传播的细节过程，以及前向传播过程中，哪些函数可以被组合成门，从而可以进行简化。知道表达式中哪部分的局部梯度计算比较简洁非常有用，这样他们可以“链”在一起，让代码量更少，效率更高。

反向传播实践：分段计算

看另一个例子。假设有如下函数：

$$f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

首先要说的是，这个函数完全没用，读者是不会用到它来进行梯度计算的，这里只是用来作为实践反向传播的一个例子，需要强调的是，如果对 x 或 y 进行微分运算，运算结束后会得到一个巨大而复杂的表达式。然而做如此复杂的运算实际上并无必要，因为我们不需要一个明确的函数来计算梯度，只需知道如何使用反向传播计算梯度即可。下面是构建前向传播的代码模式：

```

x = 3 # 例子数值
y = -4

# 前向传播
sigy = 1.0 / (1 + math.exp(-y)) # 分子中的sigmoid #(1)
num = x + sigy # 分子 #(2)
sigx = 1.0 / (1 + math.exp(-x)) # 分母中的sigmoid #(3)

```

```

xpy = x + y #(4)
xpysqr = xpy**2 #(5)
den = sigx + xpysqr # 分母 #(6)
invden = 1.0 / den #(7)
f = num * invden # 搞定! #(8)

```

嗷~~，到了表达式的最后，就完成了前向传播。注意在构建代码时创建了多个中间变量，每个都是比较简单的表达式，它们计算局部梯度的方法是已知的。这样计算反向传播就简单了：我们对前向传播时产生每个变量(**sigy, num, sigx, xpy, xpysqr, den, invden**)进行回传。我们会有同样数量的变量，但是都以d开头，用来存储对应变量的梯度。注意在反向传播的每一小块中都将包含了表达式的局部梯度，然后根据使用链式法则乘以上游梯度。对于每行代码，我们将指明其对应的是前向传播的哪部分。

```

# 回传 f = num * invden
dnum = invden # 分子的梯度 #(8)
dinvden = num #(8) # 回传 invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden #(7) # 回传 den = sigx + xpysqr
dsigx = (1) * dden #(6) dxpysqr = (1) * dden #(6) # 回传 xpysqr = xpy**2
dxpy = (2 * xpy) * dxpysqr #(5) # 回传 xpy = x + y
dx = (1) * dxpy #(4)
dy = (1) * dxpy #(4) # 回传 sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below #(3) #
# 回传 num = x + sigy
dx += (1) * dnum #(2)
dsigy = (1) * dnum #(2) # 回传 sigy = 1.0 / (1 + math.exp(-y))
dy += ((1 - sigy) * sigy) * dsigy #(1) # 完成! 哟~~

```

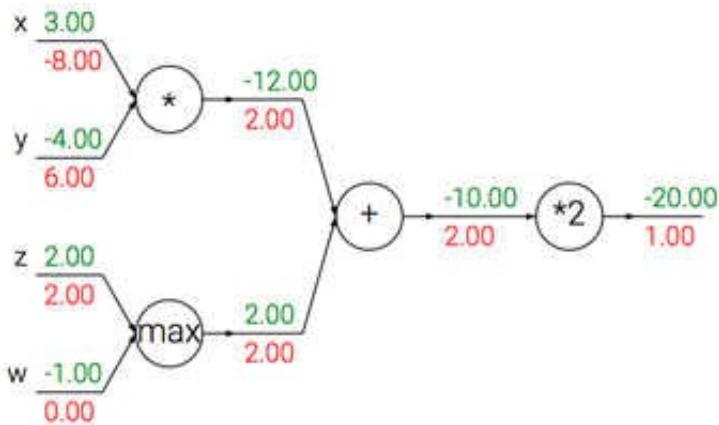
需要注意的一些东西：

对前向传播变量进行缓存：在计算反向传播时，前向传播过程中得到的一些中间变量非常有用。在实际操作中，最好代码实现对于这些中间变量的缓存，这样在反向传播的时候也能用上它们。如果这样做过于困难，也可以（但是浪费计算资源）重新计算它们。

在不同分支的梯度要相加：如果变量x, y在前向传播的表达式中出现多次，那么进行反向传播的时候就要非常小心，使用`+=`而不是`=`来累计这些变量的梯度（不然就会造成覆写）。这是遵循了在微积分中的多元链式法则，该法则指出如果变量在线路中分支走向不同的部分，那么梯度在回传的时候，就应该进行累加。

回传流中的模式

一个有趣的现象是在多数情况下，反向传播中的梯度可以被很直观地解释。例如神经网络中最常用的加法、乘法和取最大值这三个门单元，它们在反向传播过程中的行为都有非常简单的解释。先看下面这个例子：



一个展示反向传播的例子。加法操作将梯度相等地分发给它的输入。取最大操作将梯度路由给更大的输入。乘法门拿取输入激活数据，对它们进行交换，然后乘以梯度。

从上例可知：

加法门单元把输出的梯度相等地分发给它所有的输入，这一行为与输入值在前向传播时的值无关。这是因为加法操作的局部梯度都是简单的+1，所以所有输入的梯度实际上就等于输出的梯度，因为乘以1.0保持不变。上例中，加法门把梯度2.00不变且相等地路由给了两个输入。

取最大值门单元对梯度做路由。和加法门不同，取最大值门将梯度转给其中一个输入，这个输入是在前向传播中值最大的那个输入。这是因为在取最大值门中，最高值的局部梯度是1.0，其余的是0。上例中，取最大值门将梯度2.00转给了z变量，因为z的值比w高，于是w的梯度保持为0。

乘法门单元相对不容易解释。它的局部梯度就是输入值，但是是相互交换之后的，然后根据链式法则乘以输出值的梯度。上例中，x的梯度是 $-4.00 \times 2.00 = -8.00$ 。

非直观影响及真结果。注意一种比较特殊的情况，如果乘法门单元的其中一个输入非常小，而另一个输入非常大，那么乘法门的操作将会不是那么直观：它将会把大的梯度分配给小的输入，把小的梯度分配给大的输入。在线性分类器中，权重和输入是进行点积 $w^T x_i$ ，这说明输入数据的大小对于权重梯度的大

小有影响。例如，在计算过程中对所有输入数据样本 x_i 乘以 1000，那么权重的梯度将会增大 1000 倍，这样就必须降低学习率来弥补。这就是为什么数据预处理关系重大，它即使只是有微小变化，也会产生巨大影响。对于梯度在计算线路中是如何流动的有一个直观的理解，可以帮助读者调试网络。

用向量化操作计算梯度

上述内容考虑的都是单个变量情况，但是所有概念都适用于矩阵和向量操作。然而，在操作的时候要注意关注维度和转置操作。

矩阵相乘的梯度：可能最有技巧的操作是矩阵相乘（也适用于矩阵和向量，向量和向量相乘）的乘法操作：

```
# 前向传播
W = np.random.randn(5, 10)
X = np.random.randn(10, 3)
D = W.dot(X)

# 假设我们得到了D的梯度
dD = np.random.randn(*D.shape) # 和D一样的尺寸
dW = dD.dot(X.T) # .T就是对矩阵进行转置
dX = W.T.dot(dD)
```

提示：要分析维度！注意不需要去记忆 dW 和 dX 的表达，因为它们很容易通过维度推导出来。例如，权重的梯度 dW 的尺寸肯定和权重矩阵 W 的尺寸是一样的，而这又是由 X 和 dD 的矩阵乘法决定的（在上面的例子中 X 和 W 都是数字不是矩阵）。总有一个方式是能够让维度之间能够对上的。例如， X 的尺寸是 $[10 \times 3]$ ， dD 的尺寸是 $[5 \times 3]$ ，如果你想要 dW 和 W 的尺寸是 $[5 \times 10]$ ，那就需要 $dD.dot(X.T)$ 。

使用小而具体的例子：有些读者可能觉得向量化操作的梯度计算比较困难，建议是写出一个很小很明确的向量化例子，在纸上演算梯度，然后对其一般化，得到一个高效的向量化操作形式。

小结

- 对梯度的含义有了直观理解，知道了梯度是如何在网络中反向传播的，知道了它们是如何与网络的不同部分通信并控制其升高或者降低，并使得最终输出值更高的。
- 讨论了**分段计算**在反向传播的实现中的重要性。应该将函数分成不同的模块，这样计算局部梯度相对容易，然后基于链式法则将其“链”起来。

来。重要的是，不需要把这些表达式写在纸上然后演算它的完整求导公式，因为实际上并不需要关于输入变量的梯度的数学公式。只需要将表达式分成不同的可以求导的模块（模块可以是矩阵向量的乘法操作，或者取最大值操作，或者加法操作等），然后在反向传播中一步一步地计算梯度。

在下节课中，将会开始定义神经网络，而反向传播使我们能高效计算神经网络各个节点关于损失函数的梯度。换句话说，我们现在已经准备好训练神经网络了，本课程最困难的部分已经过去了！ConvNets相比只是向前走了一小步。

参考文献

- [Automatic differentiation in machine learning: a survey](#)

9. 神经网络笔记1（上）

内容列表：

- 不用大脑做类比的快速简介
- 单个神经元建模
 - 生物动机和连接
 - 作为线性分类器的单个神经元
 - 常用的激活函数 **译者注：上篇翻译截止处**
- 神经网络结构
 - 层组织
 - 前向传播计算例子
 - 表达能力
 - 设置层的数量和尺寸
- 小节
- 参考文献

快速简介

在不诉诸大脑的类比的情况下，依然可以对神经网络算法进行介绍的。在线性分类一节中，在给出图像的情况下，是使用 $s = Wx$ 来计算不同视觉类别的评分，其中 W 是一个矩阵， x 是一个输入列向量，它包含了图像的全部像素数据。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 $s = W_2 \max(0, W_1 x)$ 。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。这个非线性函数有多种选择，后续将会学到。但这个形式是一个最常用的选择，它就是简单地设置阈值，将所有小于0的值变成0。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。注意非线性函数在计算上是至关重要的，如果略去这一步，那么两个矩阵将会合二为一，对于分类的评分计算将重新变成关于输入的线性函数。这个非线性函数就是改变的关键点。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 $s = W_3 \max(0, W_2 \max(0, W_1 x))$ ，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，后续将学习如何设置它们。现在让我们先从神经元或者网络的角度理解上述计算。

单个神经元建模

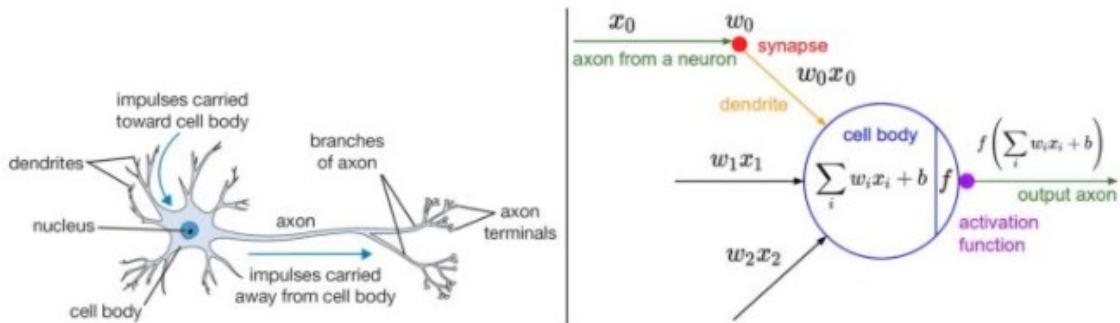
神经网络算法领域最初是被对生物神经系统建模这一目标启发，但随后与其分道扬镳，成为一个工程问题，并在机器学习领域取得良好效果。然而，讨论将还是从对生物系统的一个高层次的简略描述开始，因为神经网络毕竟是从这里得到了启发。

生物动机与连接

大脑的基本计算单位是神经元（neuron）。人类的神经系统中大约有860亿个神经元，它们被大约 10^{14} - 10^{15} 个突触（synapses）连接起来。下面图表的左边展示了一个生物学的神经元，右边展示了一个常用的数学模型。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。

在神经元的计算模型中，沿着轴突传播的信号（比如 x_0 ）将基于突触的突触强度（比如 w_0 ），与其他神经元的树突进行乘法交互（比如 $w_0 x_0$ ）。其观点是，突触的强度（也就是权重 w ），是可学习的且可以控制一个神经元对

于另一个神经元的影响强度（还可以控制影响方向：使其兴奋（正权重）或使其抑制（负权重））。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） f ，它表达了轴突上激活信号的频率。由于历史原因，激活函数常常选择使用sigmoid函数 σ ，该函数输入实数值（求和后的信号强度），然后将输入值压缩到0-1之间。在本节后面部分会看到这些激活函数的各种细节。



左边是生物神经元，右边是数学模型。
一个神经元前向传播的实例代码如下：

```
class Neuron(object):
    # ...
    def forward(self, inputs):
        """ 假设输入和权重是1-D的numpy数组，偏差是一个数字 """
        cell_body_sum = np.sum(inputs * self.weights) + self.bias
        firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) #
        sigmoid 激活函数
        return firing_rate
```

换句话说，每个神经元都对它的输入和权重进行点积，然后加上偏差，最后使用非线性函数（或称为激活函数）。本例中使用的是sigmoid函数 $\sigma(x) = 1/(1 + e^{-x})$ 。在本节的末尾部分将介绍不同激活函数的细节。

粗糙模型：要注意这个对于生物神经元的建模是非常粗糙的：在实际中，有很多不同类型的神经元，每种都有不同的属性。生物神经元的树突可以进行复杂的非线性计算。突触并不就是一个简单的权重，它们是复杂的非线性动态系统。很多系统中，输出的峰值信号的精确时间点非常重要，说明速率编码的近似是不够全面的。鉴于所有这些已经介绍和更多未介绍的简化，如果你画出人

类大脑和神经网络之间的类比，有神经科学背景的人对你的板书起哄也是非常自然的。如果你对此感兴趣，可以看看这份[评论](#)或者最新的[另一份](#)。

作为线性分类器的单个神经元

神经元模型的前向计算数学公式看起来可能比较眼熟。就像在线性分类器中看到的那样，神经元有能力“喜欢”（激活函数值接近1），或者不喜欢（激活函数值接近0）输入空间中的某些线性区域。因此，只要在神经元的输出端有一个合适的损失函数，就能让单个神经元变成一个线性分类器。

二分类Softmax分类器。举例来说，可以把 $\sigma(\sum_i w_i x_i + b)$ 看做其中一个分类的概率 $P(y_i = 1|x_i; w)$ ，其他分类的概率为

$P(y_i = 0|x_i; w) = 1 - P(y_i = 1|x_i; w)$ ，因为它们加起来必须为1。根据这种理解，可以得到交叉熵损失，这个在线性分一节中已经介绍。然后将它最优化为二分类的Softmax分类器（也就是逻辑回归）。因为sigmoid函数输出限定在0-1之间，所以分类器做出预测的基准是神经元的输出是否大于0.5。

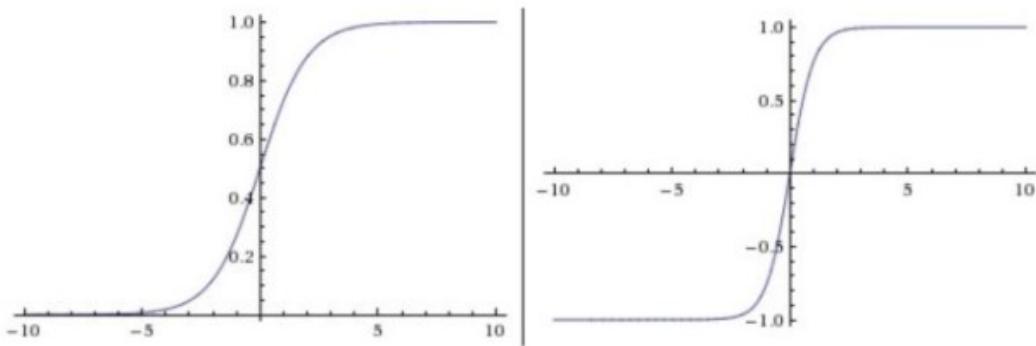
二分类SVM分类器。或者可以在神经元的输出外增加一个最大边界折叶损失（max-margin hinge loss）函数，将其训练成一个二分类的支持向量机。

理解正则化。在SVM/Softmax的例子中，正则化损失从生物学角度可以看做逐渐遗忘，因为它的效果是让所有突触权重 w 在参数更新过程中逐渐向着0变化。

一个单独的神经元可以用来实现一个二分类分类器，比如二分类的Softmax或者SVM分类器。

常用激活函数

每个激活函数（或非线性函数）的输入都是一个数字，然后对其进行某种固定的数学操作。下面是在实践中可能遇到的几种激活函数：



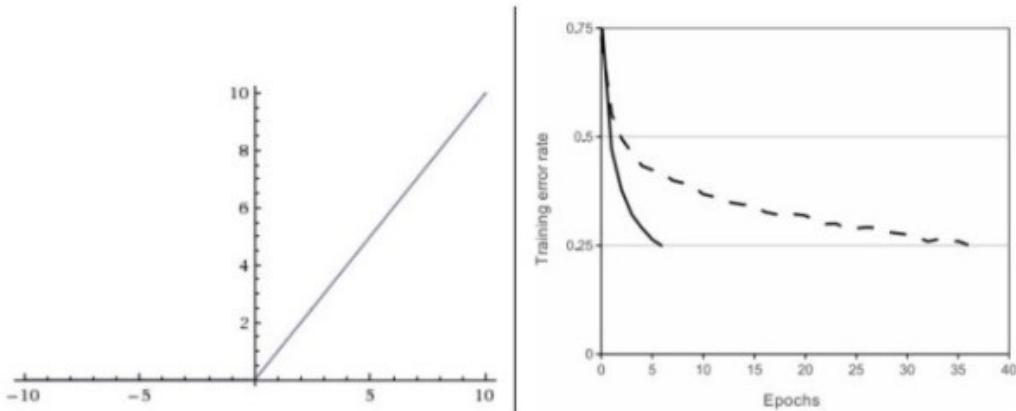
左边是Sigmoid非线性函数，将实数压缩到[0,1]之间。右边是tanh函数，将实数压缩到[-1,1]。

Sigmoid。 sigmoid非线性函数的数学公式是 $\sigma(x) = 1/(1 + e^{-x})$ ，函数图像如上图的左边所示。在前一节中已经提到过，它输入实数值并将其“挤压”到0到1范围内。更具体地说，很大的负数变成0，很大的正数变成1。在历史上，sigmoid函数非常常用，这是因为它对于神经元的激活频率有良好的解释：从完全不激活（0）到在求和后的最大频率处的完全饱和（**saturated**）的激活（1）。然而现在sigmoid函数已经不太受欢迎，实际很少使用了，这是因为它有两个主要缺点：

- *Sigmoid*函数饱和使梯度消失。sigmoid神经元有一个不好的特性，就是当神经元的激活在接近0或1处时会饱和：在这些区域，梯度几乎为0。回忆一下，在反向传播的时候，这个（局部）梯度将会与整个损失函数关于该门单元输出的梯度相乘。因此，如果局部梯度非常小，那么相乘的结果也会接近零，这会有效地“杀死”梯度，几乎就没有信号通过神经元传到权重再到数据了。还有，为了防止饱和，必须对于权重矩阵初始化特别留意。比如，如果初始化权重过大，那么大多数神经元将会饱和，导致网络就几乎不学习了。
- *Sigmoid*函数的输出不是零中心的。这个性质并不是我们想要的，因为在神经网络后面层中的神经元得到的数据将不是零中心的。这一情况将影响梯度下降的运作，因为如果输入神经元的数据总是正数（比如在 $f = \mathbf{w}^T \mathbf{x} + b$ 中每个元素都 $x > 0$ ），那么关于 \mathbf{w} 的梯度在反向传播的过程中，将会要么全部是正数，要么全部是负数（具体依整个表达式 f 而定）。这将会导致梯度下降权重更新时出现Z字型的下降。然而，可以看到整个批量的数据的梯度被加起来后，对于权重的最终更新将会有不同的正负，这样就从一定程度上减轻了这个问题。因此，该问题相对于上面的神经元饱和问题来说只是个小麻烦，没有那么严重。

Tanh。 tanh非线性函数图像如上图右边所示。它将实数值压缩到[-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。因此，在实际操作中， $tanh$ 非线性函数比sigmoid非线性函数更受欢迎。注意tanh神经元是一个简单放大的sigmoid神经元，具体说来就是： $tanh(x) = 2\sigma(2x) - 1$ 。



左边是ReLU（校正线性单元：Rectified Linear Unit）激活函数，当 $x = 0$ 时函数值为0。当 $x > 0$ 函数的斜率为1。右边是从Krizhevsky等的论文中截取的图表，指明使用ReLU比使用tanh的收敛快6倍。

ReLU。在近些年ReLU变得非常流行。它的函数公式是 $f(x) = \max(0, x)$ 。换句话说，这个激活函数就是一个关于0的阈值（如上图左侧）。使用ReLU有以下一些优缺点：

- 优点：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（Krizhevsky等的论文指出有6倍之多）。据称这是由它的线性，非饱和的公式导致的。
- 优点：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，那么从此以后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。例如，如果学习率设置得太高，可能会发现网络中40%的神经元都会死掉（在整个训练集中这些神经元都不会被激活）。通过合理设置学习率，这种情况的发生概率会降低。

Leaky ReLU。Leaky ReLU是为解决“ReLU死亡”问题的尝试。ReLU中当 $x < 0$ 时，函数值为0。而Leaky ReLU则是给出一个很小的负数梯度值，比如

0.01。所以其函数公式为 $f(x) = 1(x < 0)(\alpha x) + 1(x \geq 0)(x)$ 其中 α 是一个小的常量。有些研究者的论文指出这个激活函数表现很不错，但是其效果并不是很稳定。Kaiming He等人在2015年发布的论文[Delving Deep into Rectifiers](#)中介绍了一种新方法PReLU，把负区间上的斜率当做每个神经元中的一个参数。然而该激活函数在在不同任务中均有益处的一致性并没有特别清晰。

Maxout。 一些其他类型的单元被提了出来，它们对于权重和数据的内积结果不再使用 $f(w^T x + b)$ 函数形式。一个相关的流行选择是Maxout（最近由[Goodfellow](#)等发布）神经元。Maxout是对ReLU和leaky ReLU的一般化归纳，它的函数是： $\max(w_1^T x + b_1, w_2^T x + b_2)$ 。ReLU和Leaky ReLU都是这个公式的特殊情况（比如ReLU就是当 $w_1, b_1 = 0$ 的时候）。这样Maxout神经元就拥有ReLU单元的所有优点（线性操作和不饱和），而没有它的缺点（死亡的ReLU单元）。然而和ReLU对比，它每个神经元的参数数量增加了一倍，这就导致整体参数的数量激增。

以上就是一些常用的神经元及其激活函数。最后需要注意一点：在同一个网络中混合使用不同类型的神经元是非常少见的，虽然没有什么根本性问题来禁止这样做。

一句话：“那么该用那种呢？”用ReLU非线性函数。注意设置好学习率，或许可以监控你的网络中死亡的神经元占的比例。如果单元死亡问题困扰你，就试试Leaky ReLU或者Maxout，不要再用sigmoid了。也可以试试tanh，但是其效果应该不如ReLU或者Maxout。

10. 神经网络笔记1（下）

内容列表：

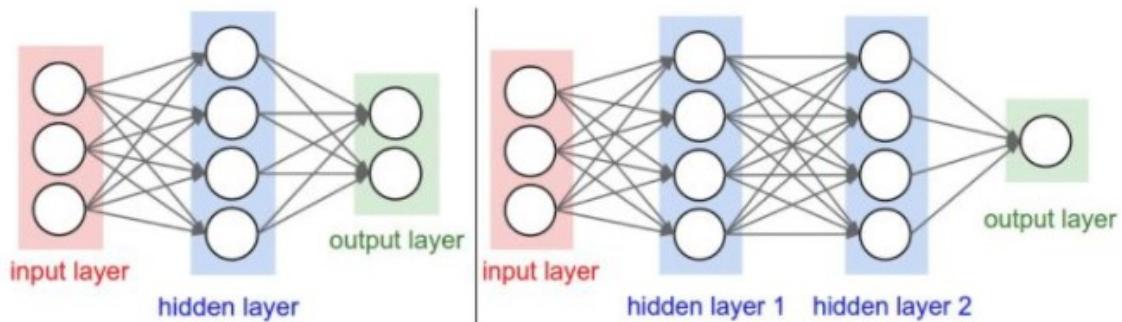
- 不用大脑做类比的快速简介
- 单个神经元建模
 - 生物动机和连接
 - 作为线性分类器的单个神经元
 - 常用的激活函数
- 神经网络结构 **译者注：下篇翻译起始处**
 - 层组织

- 前向传播计算例子
- 表达能力
- 设置层的数量和尺寸
- 小节
- 参考文献

神经网络结构

灵活地组织层

将神经网络算法以神经元的形式图形化。 神经网络被建模成神经元的集合，神经元之间以无环图的形式进行连接。也就是说，一些神经元的输出是另一些神经元的输入。在网络中是不允许循环的，因为这样会导致前向传播的无限循环。通常神经网络模型中神经元是分层的，而不是像生物神经元一样聚合成大小不一的团状。对于普通神经网络，最普通的层的类型是**全连接层 (fully-connected layer)**。全连接层中的神经元与其前后两层的神经元是完全成对连接的，但是在同一个全连接层内的神经元之间没有连接。下面是两个神经网络的图例，都使用的全连接层：



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。

命名规则。 当我们说N层神经网络的时候，我们没有把输入层算入。因此，单层的神经网络就是没有隐层的（输入直接映射到输出）。因此，有的研究者会说逻辑回归或者SVM只是单层神经网络的一个特例。研究者们也会使用人工神经网络（Artificial Neural Networks 缩写ANN）或者多层感知器（Multi-Layer Perceptrons 缩写MLP）来指代神经网络。很多研究者并不喜欢神经网

络算法和人类大脑之间的类比，他们更倾向于用单元 (*unit*) 而不是神经元作为术语。

输出层。 和神经网络中其他层不同，输出层的神经元一般是没有激活函数的（或者也可以认为它们有一个线性相等的激活函数）。这是因为最后的输出层大多用于表示分类评分值，因此是任意值的实数，或者某种实数值的目标数（比如在回归中）。

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）， $[3 \times 4] + [4 \times 2] = 20$ 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， $[3 \times 4] + [4 \times 4] + [4 \times 1] = 32$ 个权重， $4+4+1=9$ 个偏置，共41个可学习的参数。

为了方便对比，现代卷积神经网络能包含约1亿个参数，可由10-20层构成（这就是深度学习）。然而，有效 (*effective*) 连接的个数因为参数共享的缘故大大增多。在后面的卷积神经网络内容中我们将学习更多。

前向传播计算举例

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。用上面那个3层神经网络举例，输入是 $[3 \times 1]$ 的向量。一个层所有连接的强度可以存在一个单独的矩阵中。比如第一个隐层的权重 **W1** 是 $[4 \times 3]$ ，所有单元的偏置储存在 **b1** 中，尺寸 $[4 \times 1]$ 。这样，每个神经元的权重都在 **W1** 的一个行中，于是矩阵乘法 **np.dot(W1, x)** 就能计算该层中所有神经元的激活数据。类似的，**W2** 将会是 $[4 \times 4]$ 矩阵，存储着第二个隐层的连接，**W3** 是 $[1 \times 4]$ 的矩阵，用于输出层。完整的3层神经网络的前向传播就是简单的3次矩阵乘法，其中交织着激活函数的应用。

```
# 一个3层神经网络的前向传播:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # 激活函数(用的sigmoid)
x = np.random.randn(3, 1) # 含3个数字的随机输入向量(3x1)
h1 = f(np.dot(W1, x) + b1) # 计算第一个隐层的激活数据(4x1)
h2 = f(np.dot(W2, h1) + b2) # 计算第二个隐层的激活数据(4x1)
out = np.dot(W3, h2) + b3 # 神经元输出(1x1)
```

在上面的代码中，**W1**，**W2**，**W3**，**b1**，**b2**，**b3**都是网络中可以学习的参数。注意 **x** 并不是一个单独的列向量，而可以是一个批量的训练数据（其中每

个输入样本将会是 \mathbf{x} 中的一列），所有的样本将会被并行化的高效计算出来。注意神经网络最后一层通常是没有激活函数的（例如，在分类任务中它给出一个实数值的分类评分）。

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。

表达能力

理解具有全连接层的神经网络的一个方式是：可以认为它们定义了一个由一系列函数组成的函数族，网络的权重就是每个函数的参数。如此产生的问题是：该函数族的表达能力如何？存在不能被神经网络表达的函数吗？

现在看来，拥有至少一个隐层的神经网络是一个通用的近似器。在研究（例如1989年的论文[Approximation by Superpositions of Sigmoidal Function](#)，或者[Michael Nielsen](#)的这个直观解释。）中已经证明，给出任意连续函数 $f(x)$ 和任意 $\epsilon > 0$ ，均存在一个至少含1个隐层的神经网络 $g(x)$ （并且网络中有合理选择的非线性激活函数，比如sigmoid），对于 $\forall x$ ，使得 $|f(x) - g(x)| < \epsilon$ 。换句话说，神经网络可以近似任何连续函数。

既然一个隐层就能近似任何函数，那为什么还要构建更多层来将网络做得更深？答案是：虽然一个2层网络在数学理论上能完美地近似所有连续函数，但在实际操作中效果相对较差。在一个维度上，虽然以 $\mathbf{a}, \mathbf{b}, \mathbf{c}$ 为参数向量“指示块之和”函数 $g(x) = \sum_i c_i 1(a_i < x < b_i)$ 也是通用的近似器，但是谁也不会建议在机器学习中使用这个函数公式。神经网络在实践中非常好用，是因为它们表达出的函数不仅平滑，而且对于数据的统计特性有很好的拟合。同时，网络通过最优化算法（例如梯度下降）能比较容易地学习到这个函数。类似的，虽然在理论上深层网络（使用了多个隐层）和单层网络的表达能力是一样的，但是就实践经验而言，深度网络效果比单层网络好。

另外，在实践中3层的神经网络会比2层的表现好，然而继续加深（做到4，5，6层）很少有太大帮助。卷积神经网络的情况却不同，在卷积神经网络中，对于一个良好的识别系统来说，深度是一个极端重要的因素（比如数十（以10为量级）个可学习的层）。对于该现象的一种解释观点是：因为图像拥有层次化结构（比如脸是由眼睛等组成，眼睛又是由边缘组成），所以多层处理对于这种数据就有直观意义。

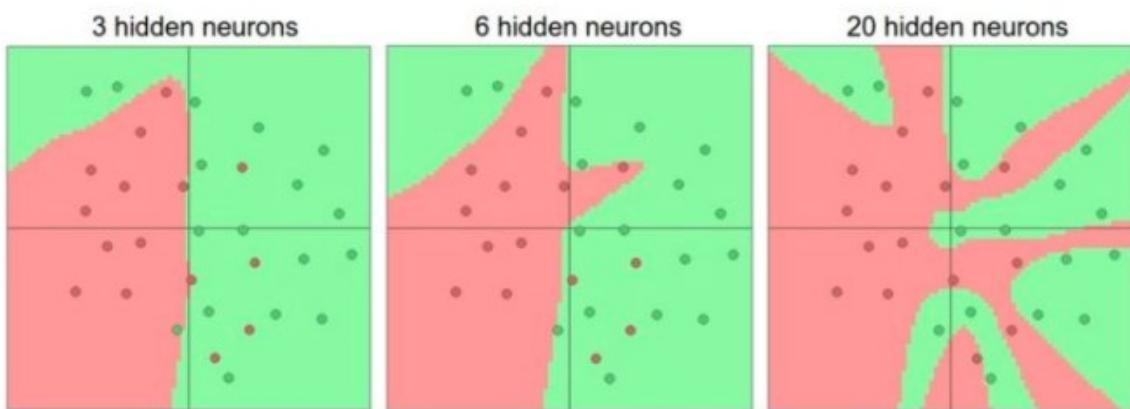
全面的研究内容还很多，近期研究的进展也很多。如果你对此感兴趣，我么推荐你阅读下面文献：

- Deep Learning的Chapter6.4，作者是Bengio等。
- Do Deep Nets Really Need to be Deep?
- FitNets: Hints for Thin Deep Nets

设置层的数量和尺寸

在面对一个具体问题的时候该确定网络结构呢？到底是不用隐层呢？还是一个隐层？两个隐层或更多？每个层的尺寸该多大？

首先，要知道当我们增加层的数量和尺寸时，网络的容量上升了。即神经元们可以合作表达许多复杂函数，所以表达函数的空间增加。例如，如果有一个在二维平面上的二分类问题。我们可以训练3个不同的神经网络，每个网络都只有一个隐层，但是每层的神经元数目不同：



更大的神经网络可以表达更复杂的函数。数据是用不同颜色的圆点表示他们的不同类别，决策边界是由训练过的神经网络做出的。你可以在[ConvNetsJS demo](#)上练练手。

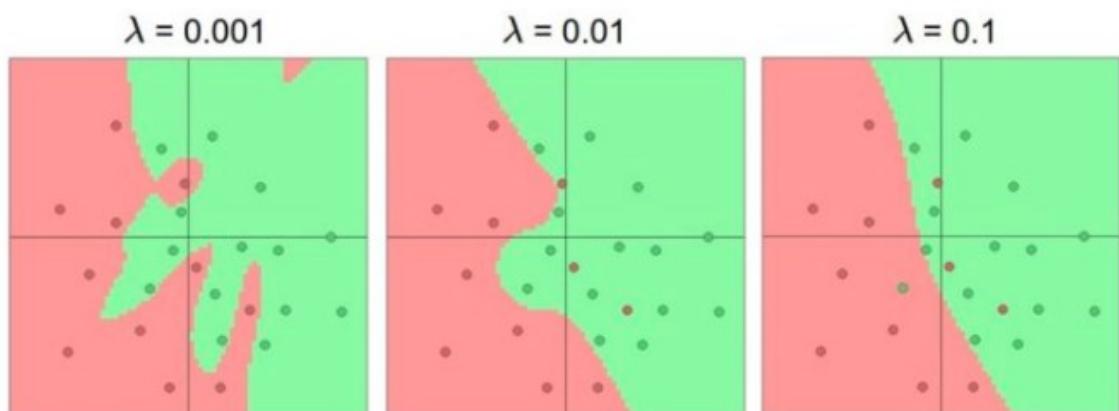
在上图中，可以看见有更多神经元的神经网络可以表达更复杂的函数。然而这既是优势也是不足，优势是可以分类更复杂的数据，不足是可能造成对训练数据的过拟合。**过拟合 (Overfitting)** 是网络对数据中的噪声有很强的拟合能力，而没有重视数据间（假设）的潜在基本关系。举例来说，有20个神经元隐层的网络拟合了所有的训练数据，但是其代价是把决策边界变成了许多不相连的红绿区域。而有3个神经元的模型的表达能力只能用比较宽泛的方式去分类数据。它将数据看做是两个大块，并把个别在绿色区域内的红色点看做噪声。在实际中，这样可以在测试数据中获得更好的**泛化 (generalization)** 能力。

基于上面的讨论，看起来如果数据不是足够复杂，则似乎小一点的网络更好，因为可以防止过拟合。然而并非如此，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），后面会详细讨论。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。

不要减少网络神经元数目的主要原因在于小网络更难使用梯度下降等局部方法来进行训练：虽然小型网络的损失函数的局部极小值更少，也比较容易收敛到这些局部极小值，但是这些最小值一般都很差，损失值很高。相反，大网络拥有更多的局部极小值，但就实际损失值来看，这些局部极小值表现更好，损失更小。因为神经网络是非凸的，就很难从数学上研究这些特性。即便如此，还是有一些文章尝试对这些目标函数进行理解，例如[The Loss Surfaces of Multilayer Networks](#)这篇论文。在实际中，你将发现如果训练的是一个小网络，那么最终的损失值将展现出多变性：某些情况下运气好会收敛到一个好的地方，某些情况下就收敛到一个不好的极值。从另一方面来说，如果你训练一个大的网络，你将发现许多不同的解决方法，但是最终损失值的差异将会小很多。这就是说，所有的解决办法都差不多，而且对于随机初始化参数好坏的依赖也会小很多。

重申一下，正则化强度是控制神经网络过拟合的好方法。看下图结果：



不同正则化强度的效果：每个神经网络都有20个隐层神经元，但是随着正则化强度增加，它的决策边界变得更加平滑。你可以在[ConvNetsJS demo](#)上练练手。

需要记住的是：不应该因为害怕出现过拟合而使用小网络。相反，应该尽可能使用大网络，然后使用正则化技巧来控制过拟合。

小结

小结如下：

- 介绍了生物神经元的粗略模型；
- 讨论了几种不同类型的激活函数，其中ReLU是最佳推荐；

- 介绍了**神经网络**，神经元通过**全连接层**连接，层间神经元两两相连，但是层内神经元不连接；
- 理解了分层的结构能够让神经网络高效地进行矩阵乘法和激活函数运算；
- 理解了神经网络是一个**通用函数近似器**，但是该性质与其广泛使用无太大关系。之所以使用神经网络，是因为它们对于实际问题中的函数的公式能够某种程度上做出“正确”假设。
- 讨论了更大网络总是更好的这一事实。然而更大容量的模型一定要和更强的正则化（比如更高的权重衰减）配合，否则它们就会过拟合。在后续章节中我们讲学习更多正则化的方法，尤其是dropout。

参考资料

- 使用Theano的[deeplearning.net tutorial](#)
- [ConvNetJS](#)
- [Michael Nielsen's tutorials](#)

11. 神经网络笔记 2

内容列表：

- 设置数据和模型
 - 数据预处理
 - 权重初始化
 - 批量归一化 (Batch Normalization)
 - 正则化 (L2/L1/Maxnorm/Dropout)
- 损失函数
- 小结

设置数据和模型

在上一节中介绍了神经元的模型，它在计算内积后进行非线性激活函数计算，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，该形式是从前面线性分类章节中的简单线性映射发展而

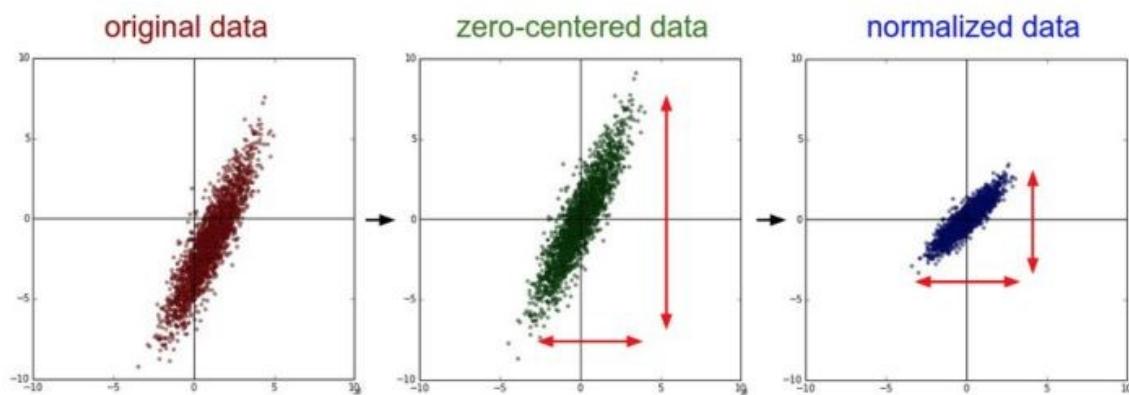
来的。具体来说，神经网络就是进行了一系列的线性映射与非线性激活函数交织的运算。本节将讨论更多的算法设计选项，比如数据预处理，权重初始化和损失函数。

数据预处理

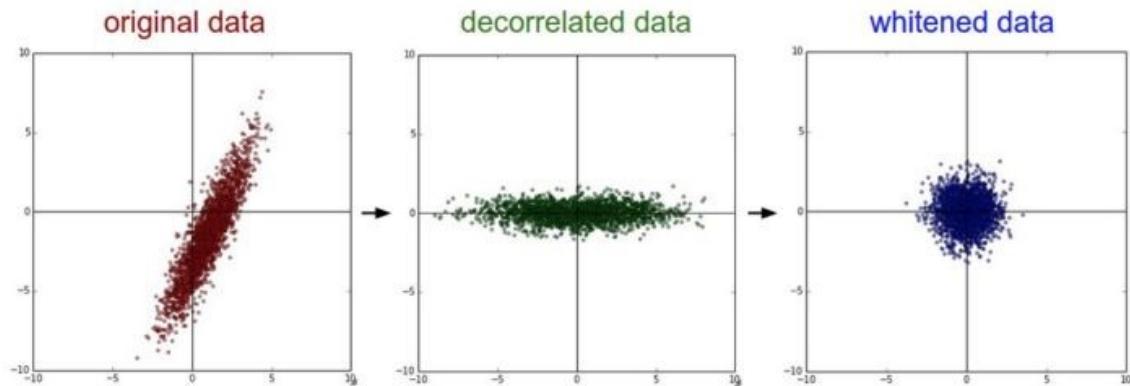
关于数据预处理我们有3个常用的符号，数据矩阵 \mathbf{X} ，假设其尺寸是 $[N \times D]$ (N 是数据样本的数量， D 是数据的维度)。

均值减法 (Mean subtraction) 是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。在numpy中，该操作可以通过代码 $\mathbf{X} -= \text{np.mean}(\mathbf{X}, \text{axis}=0)$ 实现。而对于图像，更常用的是对所有像素都减去一个值，可以用 $\mathbf{X} -= \text{np.mean}(\mathbf{X})$ 实现，也可以在3个颜色通道上分别操作。

归一化 (Normalization) 是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 $\mathbf{X} /= \text{np.std}(\mathbf{X}, \text{axis}=0)$ 。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。这个预处理操作只有在确信不同的输入特征有不同的数值范围 (或计量单位) 时才有意义，但要注意预处理操作的重要性几乎等同于学习算法本身。在图像处理中，由于像素的数值范围几乎是一致的 (都在0-255之间)，所以进行这个额外的预处理步骤并不是很必要。



一般数据预处理流程：**左边**：原始的2维输入数据。**中间**：在每个维度上都减去平均值后得到零中心化数据，现在数据云是以原点为中心的。**右边**：每个维度都除以其标准差来调整其数值范围。红色的线指出了数据各维度的数值范围，在中间的零中心化数据的数值范围不同，但在右边归一化数据中数值范围相同。



PCA和白化 (Whitening) 是另一种预处理形式。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。

```
# 假设输入数据矩阵X的尺寸为[N x D]
X -= np.mean(X, axis = 0) # 对数据进行零中心化(重要)
cov = np.dot(X.T, X) / X.shape[0] # 得到数据的协方差矩阵
```

数据协方差矩阵的第(i, j)个元素是数据第i个和第j个维度的协方差。具体来说，该矩阵的对角线上的元素是方差。还有，协方差矩阵是对称和半正定的。我们可以对数据协方差矩阵进行SVD (奇异值分解) 运算。

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

U的列是特征向量，S是装有奇异值的1维数组 (因为cov是对称且半正定的，所以S中元素是特征值的平方)。为了去除数据相关性，将已经零中心化处理过的原始数据投影到特征基准上：

```
Xrot = np.dot(X, U) # 对数据去相关性
```

注意U的列是标准正交向量的集合 (范式为1，列之间标准正交)，所以可以把它们看做标准正交基向量。因此，投影对应X中的数据的一个旋转，旋转产生的结果就是新的特征向量。如果计算Xrot的协方差矩阵，将会看到它是对角对称的。np.linalg.svd的一个良好性质是在它的返回值U中，特征向量是按照特征值的大小排列的。我们可以利用这个性质来对数据降维，只要使用前面的小部分特征向量，丢弃掉那些包含的数据没有方差的维度。这个操作也被称为主成分分析 (Principal Component Analysis 简称PCA) 降维：

```
Xrot_reduced = np.dot(X, U[:, :100]) # Xrot_reduced 变成 [N x 100]
```

经过上面的操作，将原始的数据集的大小由 $[N \times D]$ 降到了 $[N \times 100]$ ，留下了数据中包含最大方差的100个维度。通常使用PCA降维过的数据训练线性分类器和神经网络会达到非常好的性能效果，同时还能节省时间和存储器空间。

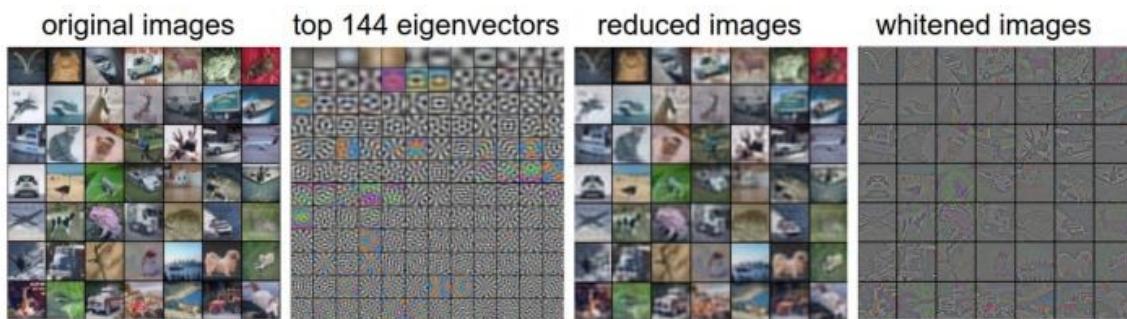
最后一个在实践中会看见的变换是**白化 (whitening)**。白化操作的输入是特征基准上的数据，然后对每个维度除以其特征值来对数值范围进行归一化。该变换的几何解释是：如果数据服从多变量的高斯分布，那么经过白化后，数据的分布将会是一个均值为零，且协方差相等的矩阵。该操作的代码如下：

```
# 对数据进行白化操作:# 除以特征值
Xwhite = Xrot / np.sqrt(S + 1e-5)
```

警告：夸大的噪声。注意分母中添加了 $1e-5$ （或一个更小的常量）来防止分母为0。该变换的一个缺陷是在变换的过程中可能会夸大数据中的噪声，这是因为将所有维度都拉伸到相同的数值范围，这些维度中也包含了那些只有极少差异性(方差小)而大多是噪声的维度。在实际操作中，这个问题可以用更强的平滑来解决（例如：采用比 $1e-5$ 更大的值）。

PCA/白化。**左边**是二维的原始数据。**中间**：经过PCA操作的数据。可以看出数据首先是零中心的，然后变换到了数据协方差矩阵的基准轴上。这样就对数据进行了解相关（协方差矩阵变成对角阵）。**右边**：每个维度都被特征值调整数值范围，将数据协方差矩阵变为单位矩阵。从几何上看，就是对数据在各个方向上拉伸压缩，使之变成服从高斯分布的一个数据点分布。

我们可以使用CIFAR-10数据将这些变化可视化出来。CIFAR-10训练集的大小是 50000×3072 ，其中每张图片都可以拉伸为3072维的行向量。我们可以计算 $[3072 \times 3072]$ 的协方差矩阵然后进行奇异值分解（比较耗费计算性能），那么经过计算的特征向量看起来是什么样子呢？



最左：一个用于演示的集合，含49张图片。**左二**：3072个特征值向量中的前144个。靠前面的特征向量解释了数据中大部分的方差，可以看见它们与图像中较低的频率相关。**第三张**是49张经过了PCA降维处理的图片，展示了144个特征向量。这就是说，展示原始图像是每个图像用3072维的向量，向量中的

元素是图片上某个位置的像素在某个颜色通道中的亮度值。而现在每张图片只使用了一个144维的向量，其中每个元素表示了特征向量对于组成这张图片的贡献度。为了让图片能够正常显示，需要将144维度重新变成基于像素基准的3072个数值。因为U是一个旋转，可以通过乘以`U.transpose()[:144,:]`来实现，然后将得到的3072个数值可视化。可以看见图像变得有点模糊了，这正好说明前面的特征向量获取了较低的频率。然而，大多数信息还是保留了下来。**最右**：将“白化”后的数据进行显示。其中144个维度中的方差都被压缩到了相同的数值范围。然后144个白化后的数值通过乘以`U.transpose()[:144,:]`转换到图像像素基准上。现在较低的频率（代表了大多数方差）可以忽略不计了，较高的频率（代表相对少的方差）就被夸大了。

实践操作。在这个笔记中提到PCA和白化主要是为了介绍的完整性，实际上在卷积神经网络中并不会采用这些变换。然而对数据进行零中心化操作还是非常重要的，对每个像素进行归一化也很常见。

常见错误。进行预处理很重要的一点是：任何预处理策略（比如数据均值）都只能在训练集数据上进行计算，算法训练完毕后再应用到验证集或者测试集上。例如，如果先计算整个数据集图像的平均值然后每张图片都减去平均值，最后将整个数据集分成训练/验证/测试集，那么这个做法是错误的。**应该怎么做呢？应该先分成训练/验证/测试集，只是从训练集中求图片平均值，然后各个集（训练/验证/测试集）中的图像再减去这个平均值。**

译者注：此处确为初学者常见错误，请务必注意！

权重初始化

我们已经看到如何构建一个神经网络的结构并对数据进行预处理，但是在开始训练网络之前，还需要初始化网络的参数。

错误：全零初始化。让我们从应该避免的错误开始。在训练完毕后，虽然不知道网络中每个权重的最终值应该是多少，但如果数据经过了恰当的归一化的话，就可以假设所有权重数值中大约一半为正数，一半为负数。这样，一个听起来蛮合理的想法就是把这些权重的初始值都设为0吧，因为在期望上来说0是最合理的猜测。这个做法错误的！因为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： **$W = 0.01 * np.random.randn(D, H)$** 。其中**randn**函数是基于零均值和标准差的一个高斯分布（**译者注：国内教程一般习惯称均值参数为期望 μ** ）来生成随机数的。根据这个式子，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。也可以使用均匀分布生成的随机数，但是从实践结果来看，对于算法的结果影响极小。

警告。并不是小数值一定会得到好的结果。例如，一个神经网络的层中的权重值很小，那么在反向传播的时候就会计算出非常小的梯度（因为梯度与权重值是成比例的）。这就会很大程度上减小反向传播中的“梯度信号”，在深度网络中，就会出现问题。

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。实践经验证明，这样做可以提高收敛的速度。

上述结论的推导过程如下：假设权重 w 和输入 x 之间的内积为 $s = \sum_i^n w_i x_i$

，这是还没有进行非线性激活函数运算之前的原始数值。我们可以检查 s 的方差：

$$\begin{aligned}
 Var(s) &= Var\left(\sum_i^n w_i x_i\right) \\
 &= \sum_i^n Var(w_i x_i) \\
 &= \sum_i^n [E(w_i)]^2 Var(x_i) + E[(x_i)]^2 Var(w_i) + Var(x_i) Var(w_i) \\
 &= \sum_i^n Var(x_i) Var(w_i) \\
 &= (n Var(w)) Var(x)
 \end{aligned}$$

在前两步，使用了方差的性质。在第三步，因为假设输入和权重的平均值都是0，所以 $E[x_i] = E[w_i] = 0$ 。注意这并不是一般化情况，比如在ReLU单元中均值就为正。在最后一步，我们假设所有的 w_i, x_i 都服从同样的分布。从这个推导过程我们可以看见，如果想要 s 有和输入 x 一样的方差，那么在初始化

的时候必须保证每个权重 w 的方差是 $1/n$ 。又因为对于一个随机变量 X 和标量 a ，有 $Var(aX) = a^2 Var(X)$ ，这就说明可以基于一个标准高斯分布，然后乘以 $a = \sqrt{1/n}$ ，使其方差为 $1/n$ ，于是得出： $w = np.random.randn(n) / sqrt(n)$ 。

Glorot等在论文[Understanding the difficulty of training deep feedforward neural networks](#)中作出了类似的分析。在论文中，作者推荐初始化公式为 $Var(w) = 2/(n_{in} + n_{out})$ ，其中 n_{in}, n_{out} 是在前一层和后一层中单元的个数。这是基于妥协和对反向传播中梯度的分析得出的结论。该主题下最新的一篇论文是：[Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](#)，作者是He等人。文中给出了一种针对ReLU神经元的特殊初始化，并给出结论：网络中神经元的方差应该是 $2.0/n$ 。代码为 $w = np.random.randn(n) * sqrt(2.0/n)$ 。这个形式是神经网络算法使用ReLU神经元时的当前最佳推荐。

稀疏初始化 (Sparse initialization)。另一个处理非标定方差的方法是将所有权重矩阵设为0，但是为了打破对称性，每个神经元都同下一层固定数目的神经元随机连接（其权重数值由一个小的高斯分布生成）。一个比较典型的连接数目是10个。

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。对于ReLU非线性激活函数，有研究人员喜欢使用如0.01这样的小数值常量作为所有偏置的初始值，这是因为他们认为这样做能让所有的ReLU单元一开始就激活，这样就能保存并传播一些梯度。然而，这样做是不是总是能提高算法性能并不清楚（有时候实验结果反而显示性能更差），所以通常还是使用0来初始化偏置参数。

实践。当前的推荐是使用ReLU激活函数，并且使用 $w = np.random.randn(n) * sqrt(2.0/n)$ 来进行权重初始化，关于这一点，[这篇文章](#)有讨论。

批量归一化 (Batch Normalization)。[批量归一化](#)是Ioffe和Szegedy最近才提出的方法，该方法减轻了如何合理初始化神经网络这个棘手问题带来的头痛：），其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。因为归一化是一个简单可求导的操作，所以上述思路是可行的。在实现层面，应用这个技巧通常意味着全连接层（或者是卷积层，后续会讲）与激活函数之间添加一个BatchNorm层。对于这个技巧本节不会展开讲，因为上面的参考文献中已经讲得很清楚了，需要知道的是在神经网络中使用批量归一化已经变得非常常见。在实践中，使用了批量归一化的网络对于不好的初始值有更强的鲁棒性。最后一句话总结：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。搞定！

正则化 Regularization

有不少方法是通过控制神经网络的容量来防止其过拟合的：

L2正则化可能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 $\frac{1}{2} \lambda w^2$

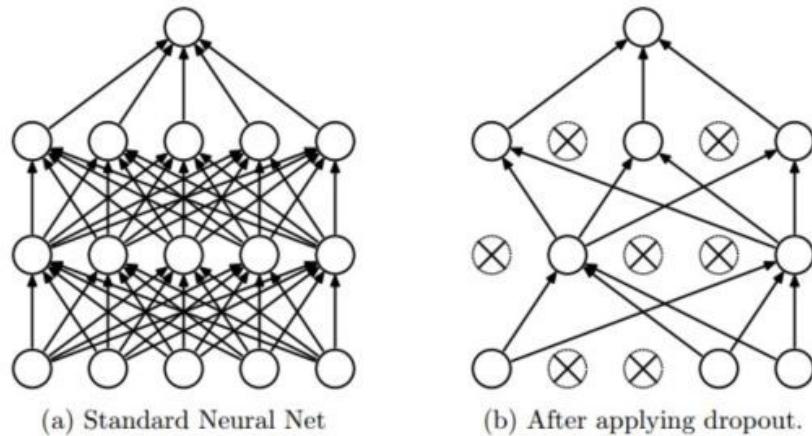
，其中 λ 是正则化强度。前面这个 $\frac{1}{2}$ 很常见，是因为加上 $\frac{1}{2}$ 后，该式子关于 w 梯度就是 λw 而不是 $2\lambda w$ 了。L2正则化可以直观理解为它对于大数值的权重向量进行严厉惩罚，倾向于更加分散的权重向量。在线性分类章节中讨论过，由于输入和权重之间的乘法操作，这样就有了一个优良的特性：使网络更倾向于使用所有输入特征，而不是严重依赖输入特征中某些小部分特征。最后需要注意在梯度下降和参数更新的时候，使用L2正则化意味着所有的权重都以 $w += -\lambda * W$ 向着0线性下降。

L1正则化是另一个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 $\lambda |w|$ 。L1和L2正则化也可以进行组合： $\lambda_1 |w| + \lambda_2 w^2$ ，这也被称作 [Elastic net regularization](#)。L1正则化有一个有趣的性质，它会让权重向量在最优化的过程中变得稀疏（即非常接近0）。也就是说，使用L1正则化的神经元最后使用的是它们最重要的输入数据的稀疏子集，同时对于噪音输入则几乎是不变的了。相较L1正则化，L2正则化中的权重向量大多是分散的小数字。在实践中，如果不是特别关注某些明确的特征选择，一般说来L2正则化都会比L1正则化效果好。

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。在实践中，与之对应的是参数更新方式不变，然后要求神经元中的权重向量 \vec{w} 必须满足 $\|\vec{w}\|_2 < c$ 这一条件，一般 c 值为3或者4。有研究者发文称在使用这种正则化方法时效果更好。这种正则化还有一个良好的性质，即使在学习率设置过高的时候，网络中也不会出现数值“爆炸”，这是因为它的参数更新始终是被限制着的。

随机失活 (Dropout)是一个简单又极其有效的正则化方法。该方法由 Srivastava在论文[Dropout: A Simple Way to Prevent Neural Networks from Overfitting](#)中提出的，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。



图片来自[论文](#)，展示其核心思路。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。

一个3层神经网络的普通版随机失活可以用下面代码实现：

""" 普通版随机失活：不推荐实现（看下面笔记） """

$p = 0.5$ # 激活神经元的概率。 p 值更高 = 随机失活更弱

```
def train_step(X):
    """ X中是输入数据 """
    # 3层neural network的前向传播
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # 第一个随机失活遮罩
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # 第二个随机失活遮罩
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3
    # 反向传播：计算梯度... (略)
    # 进行参数更新... (略)

def predict(X):
    # 前向传播时模型集成
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # 注意：激活数据要乘以p
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # 注意：激活数据要乘以p
    out = np.dot(W3, H2) + b3
```

在上面的代码中，**train_step**函数在第一个隐层和第二个隐层上进行了两次随机失活。在输入层上面进行随机失活也是可以的，为此需要为输入数据X创建一个二值的遮罩。反向传播保持不变，但是肯定需要将遮罩U1和U2加入进去。

注意：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。这一点非常重要，因为在测试时所有的神经元都能看见它们的输入，因此我们想要神经元的输出与训练时的预期输出是一致的。以 $p = 0.5$ 为例，在测试时神经元必须把它们的输出减半，这是因为在训练的时候它们的输出只有一半。为了理解这点，先假设有一个神经元 x 的输出，那么进行随机失活的时候，该神经元的输出就是 $px + (1 - p)0$ ，这是有 $1 - p$ 的概率神经元的输出为0。在测试时神经元总是激活的，就必须调整 $x \rightarrow px$ 来保持同样的预期输出。在测试时会在所有可能的二值遮罩（也就是数量庞大的所有子网络）中迭代并计算它们的协作预测，进行这种减弱的操作也可以认为是与之相关的。

上述操作不好的性质是必须在测试时对激活数据要按照 p 进行数值范围调整。既然测试性能如此关键，实际更倾向使用**反向随机失活 (inverted dropout)**，它是在训练时就进行数值范围调整，从而让前向传播在测试时保持不变。这样做还有一个好处，无论你决定是否使用随机失活，预测方法的代码可以保持不变。反向随机失活的代码如下：

""" 反向随机失活：推荐实现方式。在训练的时候**drop**和调整数值范围，测试时不做任何事。 """

```
p = 0.5 # 激活神经元的概率。p值更高 = 随机失活更弱

def train_step(X):
    # 3层neural network的前向传播
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # 第一个随机失活遮罩。注意/p！
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # 第二个随机失活遮罩。注意/p！
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # 反向传播：计算梯度... (略)
    # 进行参数更新... (略)
```

```

def predict(X):
    # 前向传播时模型集成
    H1 = np.maximum(0, np.dot(W1, X) + b1) # 不用数值范围调整了
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3

```

在随机失活发布后，很快有大量研究为什么它的实践效果如此之好，以及它和其他正则化方法之间的关系。如果你感兴趣，可以看看这些文献：

- [Dropout paper](#) by Srivastava et al. 2014.
- [Dropout Training as Adaptive Regularization](#)：“我们认为：在使用费希尔信息矩阵（fisher information matrix）的对角逆矩阵的期望对特征进行数值范围调整后，再进行L2正则化这一操作，与随机失活正则化是一阶相等的。”

前向传播中的噪音。在更一般化的分类上，随机失活属于网络在前向传播中有随机行为的方法。测试时，通过分析法（在使用随机失活的本例中就是乘以 p ）或数值法（例如通过抽样出很多子网络，随机选择不同子网络进行前向传播，最后对它们取平均）将噪音边缘化。在这个方向上的另一个研究是[DropConnect](#)，它在前向传播的时候，一系列权重被随机设置为0。提前说一下，卷积神经网络同样会吸取这类方法的优点，比如随机汇合（stochastic pooling），分级汇合（fractional pooling），数据增长（data augmentation）。我们在后面会详细介绍。

偏置正则化。在线性分类器的章节中介绍过，对于偏置参数的正则化并不常见，因为它们在矩阵乘法中和输入数据并不产生互动，所以并不需要控制其在数据维度上的效果。然而在实际应用中（使用了合理数据预处理的情况下），对偏置进行正则化也很少会导致算法性能变差。这可能是因为相较于权重参数，偏置参数实在太少，所以分类器需要它们来获得一个很好的数据损失，那么还是能够承受的。

每层正则化。对于不同的层进行不同强度的正则化很少见（可能除了输出层以外），关于这个思路的相关文献也很少。

实践：通过交叉验证获得一个全局使用的L2正则化强度是比较常见的。在使用L2正则化的同时在所有层后面使用随机失活也很常见。 p 值一般默认设为0.5，也可能在验证集上调参。

损失函数

我们已经讨论过损失函数的正则化损失部分，它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。也就是说， $L = \frac{1}{N} \sum_i L_i$ 中， N

是训练集数据的样本数。让我们把神经网络中输出层的激活函数简写为 $f = f(x_i; W)$ ，在实际中你可能需要解决以下几类问题：

分类问题是我们一直讨论的。在该问题中，假设有一个装满样本的数据集，每个样本都有一个唯一的正确标签（是固定分类标签之一）。在这类问题中，一个最常见的损失函数就是SVM（是Weston Watkins 公式）：

$$L_i = \sum_{j \neq y_i} \max(0, f_j - f_{y_i} + 1)$$

之前简要提起过，有些学者的论文中指出平方折叶损失（即使用 $\max(0, f_j - f_{y_i} + 1)^2$ 算法的结果会更好。第二个常用的损失函数是 Softmax 分类器，它使用交叉熵损失：

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

问题：类别数目巨大。当标签集非常庞大（例如字典中的所有英语单词，或者 ImageNet 中的 22000 种分类），就需要使用分层 Softmax (*Hierarchical Softmax*) 了（[参考文献](#)）。分层 softmax 将标签分解成一个树。每个标签都表示成这个树上的一个路径，这个树的每个节点处都训练一个 Softmax 分类器来在左和右分枝之间做决策。树的结构对于算法的最终结果影响很大，而且一般需要具体问题具体分析。

属性 (Attribute) 分类。上面两个损失公式的前提，都是假设每个样本只有一个正确的标签 y_i 。但是如果 y_i 是一个二值向量，每个样本可能有，也可能没有某个属性，而且属性之间并不相互排斥呢？比如在 Instagram 上的图片，就可以看成是被一个巨大的标签集合中的某个子集打上标签，一张图片上可能有多个标签。在这种情况下，一个明智的方法是为每个属性创建一个独立的二分类的分类器。例如，针对每个分类的二分类器会采用下面的公式：

$$L_i = \sum_j \max(0, 1 - y_{ij} f_j)$$

上式中，求和是对所有分类 j ， y_{ij} 的值为 1 或者 -1，具体根据第 i 个样本是否被第 j 个属性打标签而定，当该类别被正确预测并展示的时候，分值向量 f_j 为正，其余情况为负。可以发现，当一个正样本的得分小于 +1，或者一个负样

本得分大于-1的时候，算法就会累计损失值。

另一种方法是对每种属性训练一个独立的逻辑回归分类器。二分类的逻辑回归分类器只有两个分类（0，1），其中对于分类1的概率计算为：

$$P(y=1|x; w, b) = \frac{1}{1 + e^{-(w^T x + b)}} = \sigma(w^T x + b)$$

因为类别0和类别1的概率和为1，所以类别0的概率为：

$$P(y=0|x; w, b) = 1 - P(y=1|x; w, b)$$

$\sigma(w^T x + b) > 0.5$ 或者 $w^T x + b > 0$ ，那么样本就要被分类成为正样本（ $y=1$ ）。然后损失函数最大化这个对数似然函数，问题可以简化为：

$$L_i = \sum_j y_{ij} \log(\sigma(f_j)) + (1 - y_{ij}) \log(1 - \sigma(f_j))$$

上式中，假设标签 y_{ij} 非0即1， $\sigma(\cdot)$ 就是sigmoid函数。上面的公式看起来吓人，但是 f 的梯度实际上非常简单： $\frac{\partial L_i}{\partial f_j} = y_{ij} - \sigma(f_j)$ （你可以自己求导来验证）。

回归问题是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。对于某个样本，L2范式计算如下：

$$L_i = \|f - y_i\|_2^2$$

之所以在目标函数中要进行平方，是因为梯度算起来更加简单。因为平方是一个单调运算，所以不用改变最优参数。L1范式则是要将每个维度上的绝对值加起来：

$$L_i = \|f - y_i\|_1 = \sum_j |f_j - (y_i)_j|$$

在上式中，如果有多个数量被预测了，就要对预测的所有维度的预测求和，即 \sum_j 。观察第i个样本的第j维，用 δ_{ij} 表示预测值与真实值之间的差异。关于该维度的梯度（也就是 $\partial L_i / \partial f_j$ ）能够轻松地通过被求导为L2范式的 δ_{ij} 或 $sign(\delta_{ij})$ 。这就是说，评分值的梯度要么与误差中的差值直接成比例，要么是固定的并从差值中继承sign。

注意：L2损失比起较为稳定的Softmax损失来，其最优化过程要困难很多。直观而言，它需要网络具备一个特别的性质，即对于每个输入（和增量）都要输

出一个确切的正确值。而在Softmax中就不是这样，每个评分的准确值并不是那么重要：只有当它们量级适当的时候，才有意义。还有，L2损失鲁棒性不好，因为异常值可以导致很大的梯度。所以在面对一个回归问题时，先考虑将输出变成二值化是否真的不够用。例如，如果对一个产品的星级进行预测，使用5个独立的分类器来对1-5星进行打分的效果一般比使用一个回归损失要好很多。分类还有一个额外优点，就是能给出关于回归的输出的分布，而不是一个简单的毫无把握的输出值。如果确信分类不适用，那么使用L2损失吧，但是一定要谨慎：L2非常脆弱，在网络中使用随机失活（尤其是在L2损失层的上一层）不是好主意。

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。

结构化预测 (structured prediction)。结构化损失是指标签可以是任意的结构，例如图表、树或者其他复杂物体的情况。通常这种情况还会假设结构空间非常巨大，不容易进行遍历。结构化SVM背后的基本思想就是在正确的结构 y_i 和得分最高的非正确结构之间画出一个边界。解决这类问题，并不是像解决一个简单无限制的最优化问题那样使用梯度下降就可以了，而是需要设计一些特殊的解决方案，这样可以有效利用对于结构空间的特殊简化假设。我们简要地提一下这个问题，但是详细内容就超出本课程范围。

小结

小结如下：

- 推荐的预处理操作是对数据的每个特征都进行零中心化，然后将其数值范围都归一化到[-1,1]范围之内。
- 使用标准差为 $\sqrt{2/n}$ 的高斯分布来初始化权重，其中 n 是输入的神经元数。例如用numpy可以写作：`w = np.random.randn(n) * sqrt(2.0/n)`。
- 使用L2正则化和随机失活的倒置版本。
- 使用批量归一化。
- 讨论了在实践中可能要面对的不同任务，以及每个任务对应的常用损失函数。

现在，我们预处理了数据，初始化了模型。在下一节中，我们将讨论算法的学习过程及其运作特性。

12. 神经网络笔记3 (上)

内容列表：

- 梯度检查
- 合理性 (Sanity) 检查
- 检查学习过程
 - 损失函数
 - 训练集与验证集准确率
 - 权重 : 更新比例
 - 每层的激活数据与梯度分布
 - 可视化 *译者注：上篇翻译截止处*
- 参数更新
 - 一阶 (随机梯度下降) 方法 , 动量方法 , Nesterov动量方法
 - 学习率退火
 - 二阶方法
 - 逐参数适应学习率方法 (Adagrad , RMSProp)
- 超参数调优
- 评价
 - 模型集成
- 总结
- 拓展引用

学习过程

在前面章节中，我们讨论了神经网络的静态部分：如何创建网络的连接、数据和损失函数。本节将致力于讲解神经网络的动态部分，即神经网络学习参数和搜索最优超参数的过程。

梯度检查

理论上将进行梯度检查很简单，就是简单地把解析梯度和数值计算梯度进行比较。然而从实际操作层面上来说，这个过程更加复杂且容易出错。下面是一些提示、技巧和需要仔细注意的事情：

使用中心化公式。在使用有限差值近似来计算数值梯度的时候，常见的公式是：

$$\frac{df(x)}{dx} = \frac{f(x + h) - f(x)}{h} \text{ (bad, do not use)}$$

其中 h 是一个很小的数字，在实践中近似为 $1e-5$ 。在实践中证明，使用中心化公式效果更好：

$$\frac{df(x)}{dx} = \frac{f(x + h) - f(x - h)}{2h} \text{ (use instead)}$$

该公式在检查梯度的每个维度的时候，会要求计算两次损失函数（所以计算资源的耗费也是两倍），但是梯度的近似值会准确很多。要理解这一点，对 $f(x + h)$ 和 $f(x - h)$ 使用泰勒展开，可以看到第一个公式的误差近似 $O(h)$ ，第二个公式的误差近似 $O(h^2)$ （是个二阶近似）。（**译者注：泰勒展开相关内容可阅读《高等数学》第十二章第四节：函数展开成幂级数。**）

使用相对误差来比较。比较数值梯度 f'_n 和解析梯度 f'_a 的细节有哪些？如何得知此两者不匹配？你可能会倾向于监测它们的差的绝对值 $|f'_a - f'_n|$ 或者差的平方值，然后定义该值如果超过某个规定阈值，就判断梯度实现失败。然而该思路是有问题的。想想，假设这个差值是 $1e-4$ ，如果两个梯度值在 1.0 左右，这个差值看起来就很合适，可以认为两个梯度是匹配的。然而如果梯度值是

1e-5或者更低，那么1e-4就是非常大的差距，梯度实现肯定就是失败的了。

因此，使用相对误差总是更合适一些：

$$\frac{|f'_a - f'_n|}{\max(|f'_a|, |f'_n|)}$$

上式考虑了差值占两个梯度绝对值的比例。注意通常相对误差公式只包含两个式子中的一个（任意一个均可），但是我更倾向取两个式子的最大值或者取两个式子的和。这样做是为了防止在其中一个式子为0时，公式分母为0（这种情况，在ReLU中是经常发生的）。然而，还必须注意两个式子都为零且通过梯度检查的情况。在实践中：

- 相对误差>1e-2：通常就意味着梯度可能出错。
- 1e-2>相对误差>1e-4：要对这个值感到不舒服才行。
- 1e-4>相对误差：这个值的相对误差对于有不可导点的目标函数是OK的。但如果目标函数中没有kink（使用tanh和softmax），那么相对误差值还是太高。
- 1e-7或者更小：好结果，可以高兴一把了。

要知道的是网络的深度越深，相对误差就越高。所以如果你是在对一个10层网络的输入数据做梯度检查，那么1e-2的相对误差值可能就OK了，因为误差一直在累积。相反，如果一个可微函数的相对误差值是1e-2，那么通常说明梯度实现不正确。

使用双精度。一个常见的错误是使用单精度浮点数来进行梯度检查。这样会导致即使梯度实现正确，相对误差值也会很高（比如1e-2）。在我的经验而言，出现过使用单精度浮点数时相对误差为1e-2，换成双精度浮点数时就降低为1e-8的情况。

保持在浮点数的有效范围。建议通读《[What Every Computer Scientist Should Know About Floating-Point Arithmetic](#)》一文，该文将阐明你可能犯的错误，促使你写下更加细心的代码。例如，在神经网络中，在一个批量的数据上对损失函数进行归一化是很常见的。但是，如果每个数据点的梯度很小，

然后又用数据点的数量去除，就使得数值更小，这反过来会导致更多的数值问题。这就是我为什么总是会把原始的解析梯度和数值梯度数据打印出来，确保用来比较的数字的值不是过小（通常绝对值小于 $1e-10$ 就绝对让人担心）。如果确实过小，可以使用一个常数暂时将损失函数的数值范围扩展到一个更“好”的范围，在这个范围内浮点数变得更加致密。比较理想的是1.0的数量级上，即当浮点数指数为0时。

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。考虑当 $x = -1e6$ 的时，对ReLU函数进行梯度检查。因为 $x < 0$ ，所以解析梯度在该点的梯度为0。然而，在这里数值梯度会突然计算出一个非零的梯度值，因为 $f(x + h)$ 可能越过了不可导点(例如：如果 $h > 1e-6$)，导致了一个非零的结果。你可能会认为这是一个极端的案例，但实际上这种情况很常见。例如，一个用CIFAR-10训练的SVM中，因为有50,000个样本，且根据目标函数每个样本产生9个式子，所以包含有450,000个 $\max(0, x)$ 式子。而一个用SVM进行分类的神经网络因为采用了ReLU，还会有更多的不可导点。

注意，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。在具有 $\max(x, y)$ 形式的函数中持续跟踪所有“赢家”的身份，就可以实现这一点。其实就是看在前向传播时，到底x和y谁更大。如果在计算 $f(x + h)$ 和 $f(x - h)$ 的时候，至少有一个“赢家”的身份变了，那就说明不可导点被越过了，数值梯度会不准确。

使用少量数据点。解决上面的不可导点问题的一个办法是使用更少的数据点。因为含有不可导点的损失函数(例如：因为使用了ReLU或者边缘损失等函数)的数据点越少，不可导点就越少，所以在计算有限差值近似时越过不可导点的几率就越小。还有，如果你的梯度检查对2-3个数据点都有效，那么基本上对整个批量数据进行梯度检查也是没问题的。所以使用很少量的数据点，能让梯度检查更迅速高效。

谨慎设置步长 h 。在实践中 h 并不是越小越好，因为当 h 特别小的时候，就可能就会遇到数值精度问题。有时候如果梯度检查无法进行，可以试试将 h 调到

1e-4或者1e-6，然后突然梯度检查可能就恢复正常。这篇[维基百科文章](#)中有一个图表，其x轴为 h 值，y轴为数值梯度误差。

在操作的特性模式中梯度检查。有一点必须要认识到：梯度检查是在参数空间中的一个特定（往往还是随机的）的单独点进行的。即使是在该点上梯度检查成功了，也不能马上确保全局上梯度的实现都是正确的。还有，一个随机的初始化可能不是参数空间最优代表性的点，这可能导致进入某种病态的情况，即梯度看起来是正确实现了，实际上并没有。例如，SVM使用小数值权重初始化，就会把一些接近于0的得分分配给所有的数据点，而梯度将会在所有的数据点上展现出某种模式。一个不正确实现的梯度也许依然能够产生出这种模式，但是不能泛化到更具代表性的操作模式，比如在一些的得分比另一些得分更大的情况下就不行。因此为了安全起见，最好让网络学习（“预热”）一小段时间，等到损失函数开始下降的之后再进行梯度检查。在第一次迭代就进行梯度检查的危险就在于，此时可能正处在不正常的边界情况，从而掩盖了梯度没有正确实现的事实。

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，在这种情况下梯度主要来源于正则化部分（正则化部分的梯度表达式通常简单很多）。这样就会掩盖掉数据损失梯度的不正确实现。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。对于正则化的单独检查可以是修改代码，去掉其中数据损失的部分，也可以提高正则化强度，确认其效果在梯度检查中是无法忽略的，这样不正确的实现就会被观察到了。

记得关闭随机失活（dropout）和数据扩张（augmentation）。在进行梯度检查时，记得关闭网络中任何不确定的效果的操作，比如随机失活，随机数据扩展等。不然它们会在计算数值梯度的时候导致巨大误差。关闭这些操作不好的一点是无法对它们进行梯度检查（例如随机失活的反向传播实现可能有错误）。因此，一个更好的解决方案就是在计算 $f(x + h)$ 和 $f(x - h)$ 前强制增加一个特定的随机种子，在计算解析梯度时也同样如此。

检查少量的维度。在实际中，梯度可以有上百万的参数，在这种情况下只能检查其中一些维度然后假设其他维度是正确的。**注意：**确认在所有不同的参数中

都抽取一部分来梯度检查。在某些应用中，为了方便，人们将所有的参数放到一个巨大的参数向量中。在这种情况下，例如偏置就可能只占用整个向量中的很小一部分，所以不要随机地从向量中取维度，一定要把这种情况考虑到，确保所有参数都收到了正确的梯度。

学习之前：合理性检查的提示与技巧

在进行费时费力的最优化之前，最好进行一些合理性检查：

- **寻找特定情况的正确损失值。** 在使用小参数进行初始化时，确保得到的损失值与期望一致。最好先单独检查数据损失（让正则化强度为0）。例如，对于一个跑CIFAR-10的Softmax分类器，一般期望它的初始损失值是2.302，这是因为初始时预计每个类别的概率是0.1（因为有10个类别），然后Softmax损失值正确分类的负对数概率： $-\ln(0.1)=2.302$ 。对于Weston Watkins SVM，假设所有的边界都被越过（因为所有的分值都近似为零），所以损失值是9（因为对于每个错误分类，边界值是1）。如果没看到这些损失值，那么初始化中就可能有问题。
- 第二个合理性检查：提高正则化强度时导致损失值变大。
- **对小数据子集过拟合。** 最后也是最重要的一步，在整个数据集进行训练之前，尝试在一个很小的数据集上进行训练（比如20个数据），然后确保能到达0的损失值。进行这个实验的时候，最好让正则化强度为0，不然它会阻止得到0的损失。除非能通过这一个正常性检查，不然进行整个数据集训练是没有意义的。但是注意，能对小数据集进行过拟合并不代表万事大吉，依然有可能存在不正确的实现。比如，因为某些错误，数据点的特征是随机的，这样算法也可能对小数据进行过拟合，但是在整个数据集上跑算法的时候，就没有任何泛化能力。

检查整个学习过程

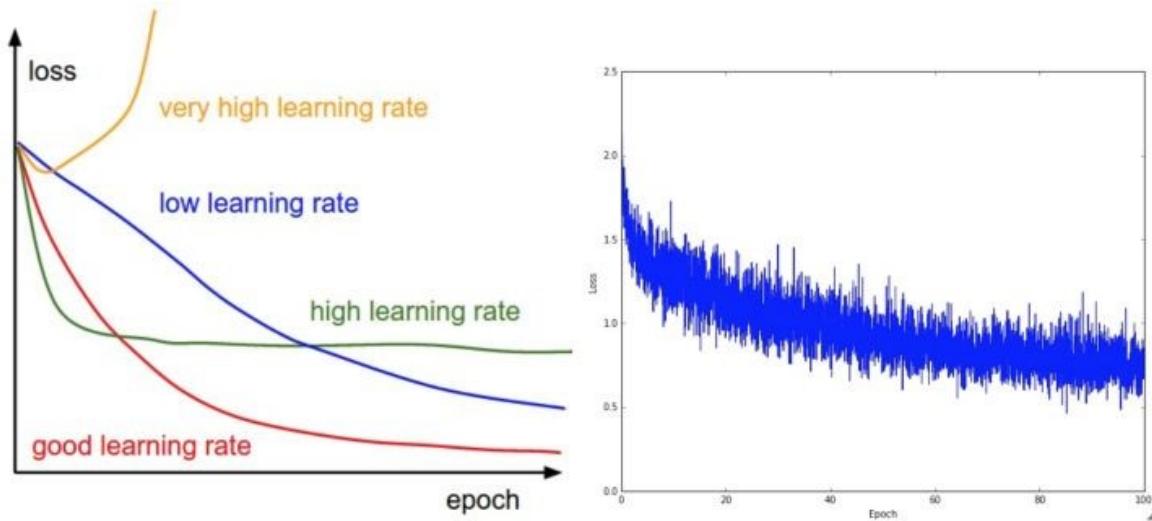
在训练神经网络的时候，应该跟踪多个重要数值。这些数值输出的图表是观察训练进程的一扇窗口，是直观理解不同的超参数设置效果的工具，从而知道如

何修改超参数以获得更高效的学习过程。

在下面的图表中，x轴通常都是表示**周期 (epochs)**单位，该单位衡量了在训练中每个样本数据都被观察过次数的期望（一个周期意味着每个样本数据都被观察过了一次）。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。

损失函数

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：



左图展示了不同的学习率的效果。过低的学习率导致算法的改善是线性的。高一些的学习率会看起来呈几何指数下降，更高的学习率会让损失值很快下降，但是接着就停在一个不好的损失值上（绿线）。这是因为最优化的“能量”太大，参数在混沌中随机震荡，不能最优化到一个很好的点上。**右图**显示了一个典型的随时间变化的损失函数值，在CIFAR-10数据集上面训练了一个小的网络，这个损失函数值曲线看起来比较合理（虽然可能学习率有点小，但是很难说），而且指出了批数据的数量可能有点太小（因为损失值的噪音很大）。

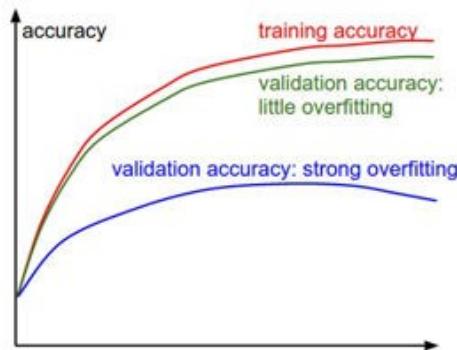
损失值的震荡程度和批尺寸 (batch size) 有关，当批尺寸为1，震荡会相对较大。当批尺寸就是整个数据集时震荡就会最小，因为每个梯度更新都是单调地优化损失函数（除非学习率设置得过高）。

有的研究者喜欢用对数域对损失函数值作图。因为学习过程一般都是采用指数型的形状，图表就会看起来更像是能够直观理解的直线，而不是呈曲棍球一样的曲线状。还有，如果多个交叉验证模型在一个图上同时输出图像，它们之间的差异就会比较明显。

有时候损失函数看起来很有意思：lossfunctions.tumblr.com。

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。

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。下面是具体例子：

```
# 假设参数向量为W, 其梯度向量为dWparam_scale =  
np.linalg.norm(W.ravel())update = -learning_rate*dW # 简单SGD更新  
update_scale = np.linalg.norm(update.ravel())W += update # 实际更新  
print update_scale / param_scale # 要得到1e-3左右
```

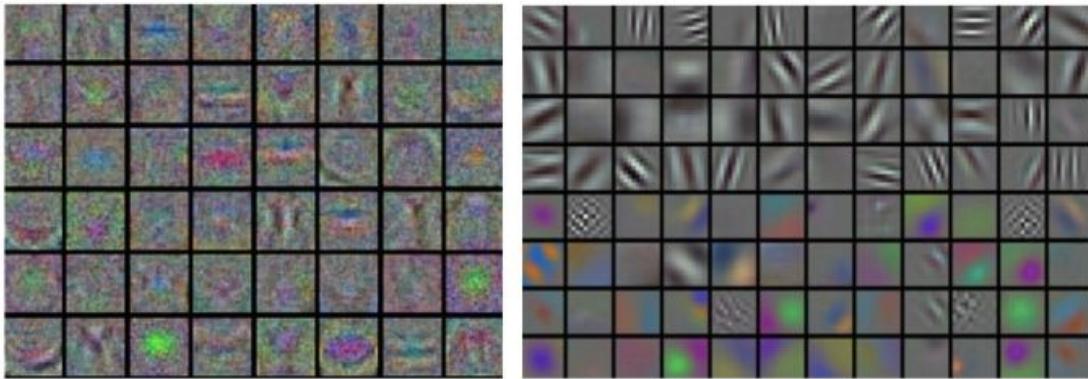
相较于跟踪最大和最小值，有研究者更喜欢计算和跟踪梯度的范式及其更新。这些矩阵通常是相关的，也能得到近似的结果。

每层的激活数据及梯度分布

一个不正确的初始化可能让学习过程变慢，甚至彻底停止。还好，这个问题可以比较简单地诊断出来。其中一个方法是输出网络中所有层的激活数据和梯度分布的柱状图。直观地说，就是如果看到任何奇怪的分布情况，那都不是好兆头。比如，对于使用tanh的神经元，我们应该看到激活数据的值在整个 $[-1, 1]$ 区间中都有分布。如果看到神经元的输出全部是0，或者全都饱和了往-1和1上跑，那肯定就是有问题了。

第一层可视化

最后，如果数据是图像像素数据，那么把第一层特征可视化会有帮助：



将神经网络第一层的权重可视化的例子。左图中的特征充满了噪音，这暗示了网络可能出现了问题：网络没有收敛，学习率设置不恰当，正则化惩罚的权重过低。右图的特征不错，平滑，干净而且种类繁多，说明训练过程进行良好。

13. 神经网络笔记3 (下)

内容列表：

- 梯度检查
- 合理性 (Sanity) 检查
- 检查学习过程
 - 损失函数
 - 训练与验证准确率
 - 权重 : 更新比例
 - 每层的激活数据与梯度分布
 - 可视化
- 参数更新 **译者注：下篇翻译起始处**
 - 一阶 (随机梯度下降) 方法，动量方法，Nesterov动量方法
 - 学习率退火
 - 二阶方法
 - 逐参数适应学习率方法 (Adagrad , RMSProp)

- 超参数调优
- 评价
 - 模型集成
- 总结
- 拓展引用

参数更新

一旦能使用反向传播计算解析梯度，梯度就能被用来进行参数更新了。进行参数更新有好几种方法，接下来都会进行讨论。

深度网络的最优化是现在非常活跃的研究领域。本节将重点介绍一些公认有效的常用的技巧，这些技巧都是在实践中会遇到的。我们将简要介绍这些技巧的直观概念，但不进行细节分析。对于细节感兴趣的读者，我们提供了一些拓展阅读。

随机梯度下降及各种更新方法

普通更新。最简单的更新形式是沿着负梯度方向改变参数（因为梯度指向的是上升方向，但是我们通常希望最小化损失函数）。假设有一个参数向量 \mathbf{x} 及其梯度 \mathbf{dx} ，那么最简单的更新的形式是：

```
# 普通更新
x += - learning_rate * dx
```

其中`learning_rate`是一个超参数，它是一个固定的常量。当在整个数据集上进行计算时，只要学习率足够低，总是能在损失函数上得到非负的进展。

动量 (Momentum) 更新是另一个方法，这个方法在深度网络上几乎总能得到更好的收敛速度。该方法可以看成是从物理角度上对于最优化问题得到的启发。损失值可以理解为是山的高度（因此高度势能是 $U = mgh$ ，所以有 $U \propto h$ ）。用随机数字初始化参数等同于在某个位置给质点设定初始速度为 0。这样最优化过程可以看做是模拟参数向量（即质点）在地形上滚动的过程。

因为作用于质点的力与梯度的潜在能量（ $F = -\nabla U$ ）有关，质点所受的力就是损失函数的（负）梯度。还有，因为 $F = ma$ ，所以在这个观点下（负）梯度与质点的加速度是成比例的。注意这个理解和上面的随机梯度下降

(SDG) 是不同的，在普通版本中，梯度直接影响位置。而在这个版本的更新中，物理观点建议梯度只是影响速度，然后速度再影响位置：

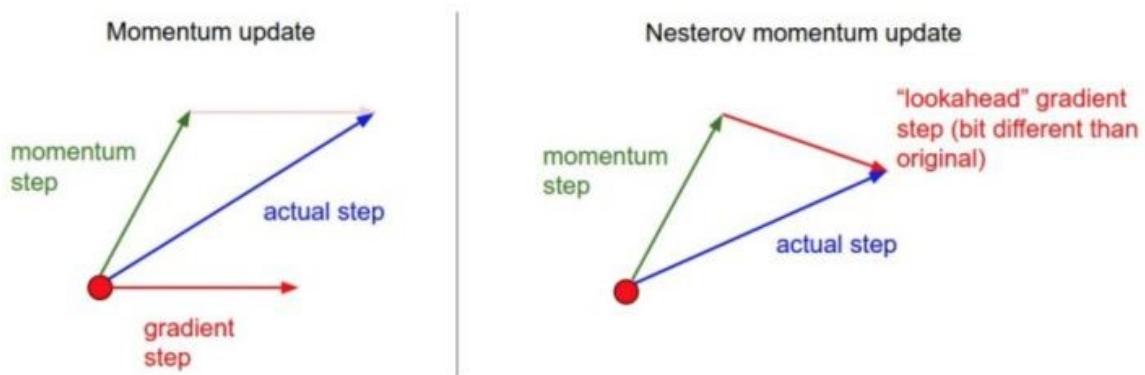
```
# 动量更新
v = mu * v - learning_rate * dx # 与速度融合
x += v # 与位置融合
```

在这里引入了一个初始化为0的变量 **v** 和一个超参数 **mu**。说得不恰当一点，这个变量 (mu) 在最优化的过程中被看做动量 (一般值设为0.9)，但其物理意义与摩擦系数更一致。这个变量有效地抑制了速度，降低了系统的动能，不然质点在山底永远不会停下来。通过交叉验证，这个参数通常设为 [0.5, 0.9, 0.95, 0.99] 中的一个。和学习率随着时间退火 (下文有讨论) 类似，动量随时间变化的设置有时能略微改善最优化的效果，其中动量在学习过程的后阶段会上升。一个典型的设置是刚开始将动量设为0.5而在后面的多个周期 (epoch) 中慢慢提升到0.99。

通过动量更新，参数向量会在任何有持续梯度的方向上增加速度。

Nesterov动量 与普通动量有些许不同，最近变得比较流行。在理论上对于凸函数它能得到更好的收敛，在实践中也确实比标准动量表现更好一些。

Nesterov动量的核心思路是，当参数向量位于某个位置 **x** 时，观察上面的动量更新公式可以发现，动量部分 (忽视带梯度的第二个部分) 会通过 **mu * v** 稍微改变参数向量。因此，如果要计算梯度，那么可以将未来的近似位置 **x + mu * v** 看做是“向前看”，这个点在我们一会儿要停止的位置附近。因此，计算 **x + mu * v** 的梯度而不是“旧”位置 **x** 的梯度就有意义了。



Nesterov动量。既然我们知道动量将会把我们带到绿色箭头指向的点，我们就不要在原点 (红色点) 那里计算梯度了。使用Nesterov动量，我们就在这个“向前看”的地方计算梯度。

也就是说，添加一些注释后，实现代码如下：

```
x_ahead = x + mu * v# 计算dx_ahead(在x_ahead处的梯度，而不是在x处的梯度)
v = mu * v - learning_rate * dx_ahead
x += v
```

然而在实践中，人们更喜欢和普通SGD或上面的动量方法一样简单的表达式。通过对 $x_{ahead} = x + mu * v$ 使用变量变换进行改写是可以做到的，然后用 x_{ahead} 而不是 x 来表示上面的更新。也就是说，实际存储的参数向量总是向前一步的那个版本。 x_{ahead} 的公式（将其重新命名为 x ）就变成了：

```
v_prev = v # 存储备份
v = mu * v - learning_rate * dx # 速度更新保持不变
x += -mu * v_prev + (1 + mu) * v # 位置更新变了形式
```

对于NAG (Nesterov's Accelerated Momentum) 的来源和数学公式推导，我们推荐以下的拓展阅读：

- Yoshua Bengio的[Advances in optimizing Recurrent Networks](#)，Section 3.5。
- [Ilya Sutskever's thesis](#) (pdf)在section 7.2对于这个主题有更详尽的阐述。

学习率退火

在训练深度网络的时候，让学习率随着时间退火通常是有帮助的。可以这样理解：如果学习率很高，系统的动能就过大，参数向量就会无规律地跳动，不能够稳定到损失函数更深更窄的部分去。知道什么时候开始衰减学习率是有技巧的：慢慢减小它，可能在很长时间内只能是浪费计算资源地看着它混沌地跳动，实际进展很少。但如果快速地减少它，系统可能过快地失去能量，不能到达原本可以到达的最好位置。通常，实现学习率退火有3种方式：

- **随步数衰减**：每进行几个周期就根据一些因素降低学习率。典型的值是每过5个周期就将学习率减少一半，或者每20个周期减少到之前的0.1。这些数值的设定是严重依赖具体问题和模型的选择的。在实践中可能看见这么一种经验做法：使用一个固定的学习率来进行训练的同时观察验证集错误率，每当验证集错误率停止下降，就乘以一个常数（比如0.5）来降低学习率。
- **指数衰减**。数学公式是 $\alpha = \alpha_0 e^{-kt}$ ，其中 α_0, k 是超参数， t 是迭代次数（也可以使用周期作为单位）。

- **1/t衰减**的数学公式是 $\alpha = \alpha_0 / (1 + kt)$ ，其中 α_0, k 是超参数，t是迭代次数。

在实践中，我们发现随步数衰减的随机失活（dropout）更受欢迎，因为它使用的超参数（衰减系数和以周期为时间单位的步数）比 k 更有解释性。最后，如果你有足够的计算资源，可以让衰减更加缓慢一些，让训练时间更长些。

二阶方法

在深度网络背景下，第二类常用的最优化方法是基于[牛顿法](#)的，其迭代如下：

$$x \leftarrow x - [Hf(x)]^{-1} \nabla f(x)$$

这里 $Hf(x)$ 是[Hessian矩阵](#)，它是函数的二阶偏导数的平方矩阵。 $\nabla f(x)$ 是梯度向量，这和梯度下降中一样。直观理解上，Hessian矩阵描述了损失函数的局部曲率，从而使得可以进行更高效的参数更新。具体来说，就是乘以Hessian转置矩阵可以让最优化过程在曲率小的时候大步前进，在曲率大的时候小步前进。需要重点注意的是，在这个公式中是没有学习率这个超参数的，这相较于一阶方法是一个巨大的优势。

然而上述更新方法很难运用到实际的深度学习应用中去，这是因为计算（以及求逆）Hessian矩阵操作非常耗费时间和空间。举例来说，假设一个有一百万个参数的神经网络，其Hessian矩阵大小就是 $[1,000,000 \times 1,000,000]$ ，将占用将近3,725GB的内存。这样，各种各样的拟-牛顿法就被发明出来用于近似转置Hessian矩阵。在这些方法中最流行的是[L-BFGS](#)，该方法使用随时间的梯度中的信息来隐式地近似（也就是说整个矩阵是从来没有被计算的）。

然而，即使解决了存储空间的问题，L-BFGS应用的一个巨大劣势是需要对整个训练集进行计算，而整个训练集一般包含几百万的样本。和小批量随机梯度下降（mini-batch SGD）不同，让L-BFGS在小批量上运行起来是很需要技巧，同时也是研究热点。

实践。在深度学习和卷积神经网络中，使用L-BFGS之类的二阶方法并不常见。相反，基于（Nesterov的）动量更新的各种随机梯度下降方法更加常用，因为它们更加简单且容易扩展。

参考资料：

- [Large Scale Distributed Deep Networks](#) 一文来自谷歌大脑团队，比较了在大规模数据情况下L-BFGS和SGD算法的表现。
- [SFO](#)算法想要把SGD和L-BFGS的优势结合起来。

逐参数适应学习率方法

前面讨论的所有方法都是对学习率进行全局地操作，并且对所有的参数都是一样的。学习率调参是很耗费计算资源的过程，所以很多工作投入到发明能够适应性地对学习率调参的方法，甚至是逐个参数适应学习率调参。很多这些方法依然需要其他的超参数设置，但是其观点是这些方法对于更广范围的超参数比原始的学习率方法有更良好的表现。在本小节我们会介绍一些在实践中可能会遇到的常用适应算法：

Adagrad是一个由[Duchi](#)等提出的适应性学习率算法

```
# 假设有梯度和参数向量x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

注意，变量**cache**的尺寸和梯度矩阵的尺寸是一样的，还跟踪了每个参数的梯度的平方和。这个一会儿将用来归一化参数更新步长，归一化是逐元素进行的。注意，接收到高梯度值的权重更新的效果被减弱，而接收到低梯度值的权重的更新效果将会增强。有趣的是平方根的操作非常重要，如果去掉，算法的表现将会糟糕很多。用于平滑的式子**eps**（一般设为1e-4到1e-8之间）是防止出现除以0的情况。Adagrad的一个缺点是，在深度学习中单调的学习率被证明通常过于激进且过早停止学习。

RMSprop。是一个非常高效，但没有公开发表的适应性学习率方法。有趣的是，每个使用这个方法的人在他们的论文中都引用自Geoff Hinton的Coursera课程的[第六课的第29页PPT](#)。这个方法用一种很简单的方式修改了Adagrad方法，让它不那么激进，单调地降低了学习率。具体说来，就是它使用了一个梯度平方的滑动平均：

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

在上面的代码中，**decay_rate**是一个超参数，常用的值是[0.9, 0.99, 0.999]。其中**x+=**和Adagrad中是一样的，但是**cache**变量是不同的。因此，RMSProp仍然是基于梯度的大小来对每个权重的学习率进行修改，这同样效果不错。但是和Adagrad不同，其更新不会让学习率单调变小。

Adam。[Adam](#)是最近才提出的一种更新方法，它看起来像是RMSProp的动量版。简化的代码是下面这样：

```

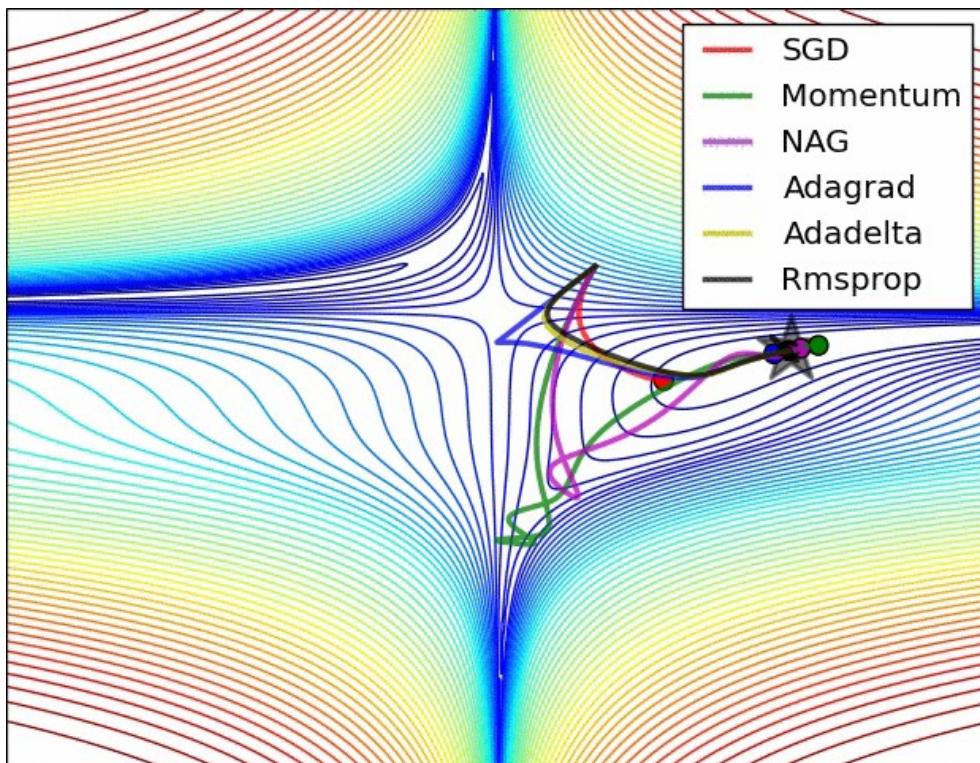
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)

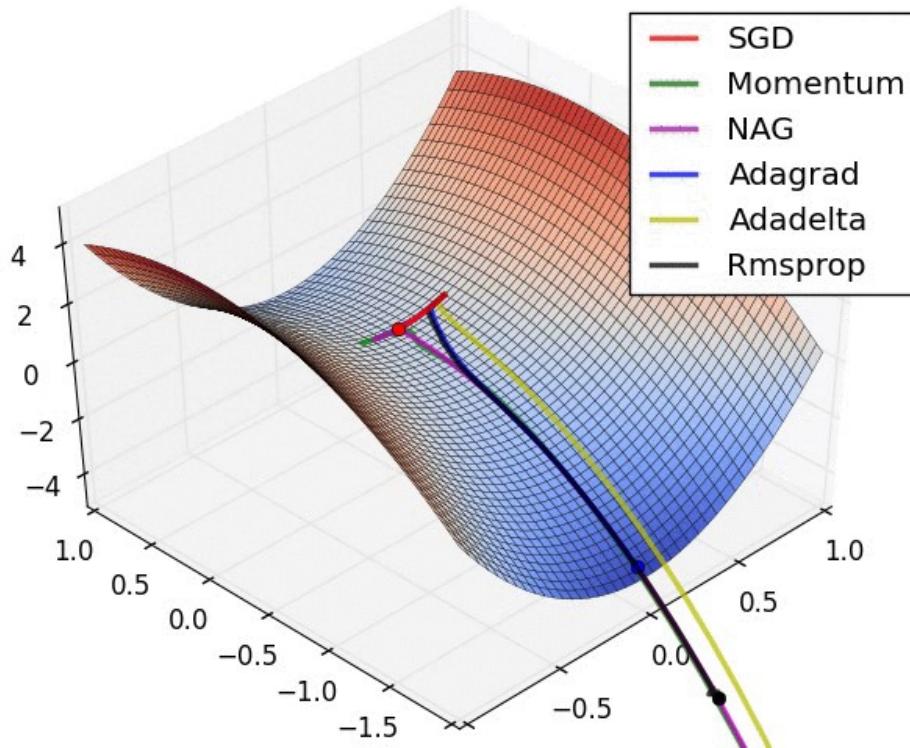
```

注意这个更新方法看起来真的和RMSProp很像，除了使用的是平滑版的梯度 \mathbf{m} ，而不是用的原始梯度向量 \mathbf{dx} 。论文中推荐的参数值 $\text{eps}=1e-8$, **beta1=0.9, beta2=0.999**。在实际操作中，我们推荐Adam作为默认的算法，一般而言跑起来比RMSProp要好一点。但是也可以试试SGD+Nesterov动量。完整的Adam更新算法也包含了一个偏置 (*bias*) 纠正机制，因为 \mathbf{m}, \mathbf{v} 两个矩阵初始为0，在没有完全热身之前存在偏差，需要采取一些补偿措施。建议读者可以阅读论文查看细节，或者课程的PPT。

拓展阅读：

- [Unit Tests for Stochastic Optimization](#)一文展示了对于随机最优化的测试。





译者注：上图原文中为动画，知乎专栏不支持动画，知友可点击[原文链接](#)查看。

上面的动画可以帮助你理解学习的动态过程。**左边**是一个损失函数的等高线图，上面跑的是不同的最优化算法。注意基于动量的方法出现了射偏了的情况，使得最优化过程看起来像是一个球滚下山的样子。**右边**展示了一个马鞍状的最优化地形，其中对于不同维度它的曲率不同（一个维度下降另一个维度上升）。注意SGD很难突破对称性，一直卡在顶部。而RMSProp之类的方法能够看到马鞍方向有很低的梯度。因为在RMSProp更新方法中的分母项，算法提高了在该方向的有效学习率，使得RMSProp能够继续前进。图片版权： [Alec Radford](#)。

超参数调优

我们已经看到，训练一个神经网络会遇到很多超参数设置。神经网络最常用的设置有：

- 初始学习率。
- 学习率衰减方式（例如一个衰减常量）。
- 正则化强度（L2惩罚，随机失活强度）。

但是也可以看到，还有很多相对不那么敏感的超参数。比如在逐参数适应学习

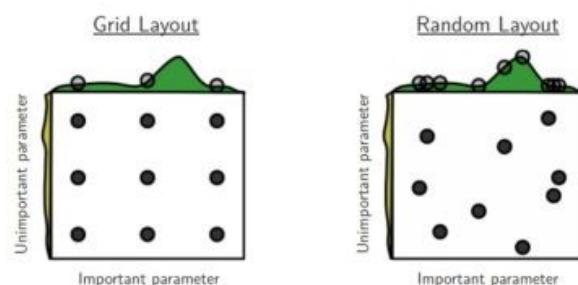
方法中，对于动量及其时间表的设置等。在本节中将介绍一些额外的调参要点和技巧：

实现。更大的神经网络需要更长的时间去训练，所以调参可能需要几天甚至几周。记住这一点很重要，因为这会影响你设计代码的思路。一个具体的设计是用**仆程序**持续地随机设置参数然后进行最优化。在训练过程中，**仆程序**会对每个周期后验证集的准确率进行监控，然后向文件系统写下一个模型的记录点（记录点中有各种各样的训练统计数据，比如随着时间的损失值变化等），这个文件系统最好是可共享的。在文件名中最好包含验证集的算法表现，这样就能方便地查找和排序了。然后还有一个**主程序**，它可以启动或者结束计算集群中的**仆程序**，有时候也可能根据条件查看**仆程序**写下的记录点，输出它们的训练统计数据等。

比起交叉验证最好使用一个验证集。在大多数情况下，一个尺寸合理的验证集可以让代码更简单，不需要用几个数据集来交叉验证。你可能会听到人们说他们“交叉验证”一个参数，但是大多数情况下，他们实际是使用的一个验证集。

超参数范围。在对数尺度上进行超参数搜索。例如，一个典型的学习率应该看起来是这样：`learning_rate = 10 ** uniform(-6, 1)`。也就是说，我们从标准分布中随机生成了一个数字，然后让它成为10的阶数。对于正则化强度，可以采用同样的策略。直观地说，这是因为学习率和正则化强度都对于训练的动态进程有乘的效果。例如：当学习率是0.001的时候，如果对其固定地增加0.01，那么对于学习进程会有很大影响。然而当学习率是10的时候，影响就微乎其微了。这就是因为学习率乘以了计算出的梯度。因此，比起加上或者减少某些值，思考学习率的范围是乘以或者除以某些值更加自然。但是有一些参数（比如随机失活）还是在原始尺度上进行搜索（例如：`dropout=uniform(0,1)`）。

随机搜索优于网格搜索。Bergstra和Bengio在文章[Random Search for Hyper-Parameter Optimization](#)中说“随机选择比网格化的选择更加有效”，而且在实践中也更容易实现。



在[Random Search for Hyper-Parameter Optimization](#)中的核心说明图。通常，有些超参数比其余的更重要，通过随机搜索，而不是网格化的搜索，可以让你更精确地发现那些比较重要的超参数的好数值。

对于边界上的最优值要小心。这种情况一般发生在你在一个不好的范围内搜索超参数（比如学习率）的时候。比如，假设我们使用`learning_rate = 10 ** uniform(-6, 1)`来进行搜索。一旦我们得到一个比较好的值，一定要确认你的值不是出于这个范围的边界上，不然你可能错过更好的其他搜索范围。

从粗到细地分阶段搜索。在实践中，先进行初略范围（比如`10 ** [-6, 1]`）搜索，然后根据好的结果出现的地方，缩小范围进行搜索。进行粗搜索的时候，让模型训练一个周期就可以了，因为很多超参数的设定会让模型没法学习，或者突然就爆出很大的损失值。第二个阶段就是对一个更小的范围进行搜索，这时可以让模型运行5个周期，而最后一个阶段就在最终的范围内进行仔细搜索，运行很多次周期。

贝叶斯超参数最优化是一个研究领域，主要是研究在超参数空间中更高效的导航算法。其核心的思路是在不同超参数设置下查看算法性能时，要在探索和使用中进行合理的权衡。基于这些模型，发展出很多的库，比较有名的有：[Scipy](#), [pearmint](#), [SMAC](#), 和[Hyperopt](#)。然而，在卷积神经网络的实际使用中，比起上面介绍的先认真挑选的一个范围，然后在该范围内随机搜索的方法，这个方法还是差一些。[这里有更详细的讨论。](#)

评价

模型集成

在实践的时候，有一个总是能提升神经网络几个百分点准确率的办法，就是在训练的时候训练几个独立的模型，然后在测试的时候平均它们预测结果。集成的模型数量增加，算法的结果也单调提升（但提升效果越来越少）。还有模型之间的差异度越大，提升效果可能越好。进行集成有以下几种方法：

- **同一个模型，不同的初始化。**使用交叉验证来得到最好的超参数，然后用最好的参数来训练不同初始化条件的模型。这种方法的风险在于多样性只来自于不同的初始化条件。
- **在交叉验证中发现最好的模型。**使用交叉验证来得到最好的超参数，然后取其中最好的几个（比如10个）模型来进行集成。这样就提高了集成的多样性，但风险在于可能会包含不够理想的模型。在实际操作中，这样操作起来比较简单，在交叉验证后就不需要额外的训练了。
- **一个模型设置多个记录点。**如果训练非常耗时，那就在不同的训练时间

对网络留下记录点（比如每个周期结束），然后用它们来进行模型集成。很显然，这样做多样性不足，但是在实践中效果还是不错的，这种方法的优势是代价比较小。

- **在训练的时候跑参数的平均值。** 和上面一点相关的，还有一个也能得到1-2个百分点的提升的小代价方法，这个方法就是在训练过程中，如果损失值相较于前一次权重出现指数下降时，就在内存中对网络的权重进行一个备份。这样你就对前几次循环中的网络状态进行了平均。你会发现这个“平滑”过的版本的权重总是能得到更少的误差。直观的理解就是目标函数是一个碗状的，你的网络在这个周围跳跃，所以对它们平均一下，就更可能跳到中心去。

模型集成的一个劣势就是在测试数据的时候会花费更多时间。最近Geoff Hinton在“[Dark Knowledge](#)”上的工作很有启发：其思路是通过将集成似然估计纳入到修改的目标函数中，从一个好的集成中抽出一个单独模型。

总结

训练一个神经网络需要：

- 利用小批量数据对实现进行梯度检查，还要注意各种错误。
- 进行合理性检查，确认初始损失值是合理的，在小数据集上能得到100%的准确率。
- 在训练时，跟踪损失函数值，训练集和验证集准确率，如果愿意，还可以跟踪更新的参数量相对于总参数量的比例（一般在 $1e-3$ 左右），然后如果是对于卷积神经网络，可以将第一层的权重可视化。
- 推荐的两个更新方法是SGD+Nesterov动量方法，或者Adam方法。
- 随着训练进行学习率衰减。比如，在固定多少个周期后让学习率减半，或者当验证集准确率下降的时候。
- 使用随机搜索（不要用网格搜索）来搜索最优的超参数。分阶段从粗（比较宽的超参数范围训练1-5个周期）到细（窄范围训练很多个周期）地来搜索。
- 进行模型集成来获得额外的性能提高。

拓展阅读

- [Leon Bottou的《SGD要点和技巧》。](#)

- Yann LeCun的《Efficient BackProp》。
- Yoshua Bengio的《Practical Recommendations for Gradient-Based Training of Deep Architectures》。

14. 卷积神经网络笔记

内容列表：

- **结构概述**
- **用来构建卷积神经网络的各种层**
 - 卷积层
 - 汇聚层
 - 归一化层
 - 全连接层
 - 将全连接层转化成卷积层
- **卷积神经网络的结构**
 - 层的排列规律
 - 层的尺寸设置规律
 - 案例学习 (LeNet / AlexNet / ZFNet / GoogLeNet / VGGNet)
 - 计算上的考量

卷积神经网络 (CNNs / ConvNets)

卷积神经网络和上一章讲的常规神经网络非常相似：它们都是由神经元组成，神经元中有具有学习能力的权重和偏差。每个神经元都得到一些输入数据，进行内积运算后再进行激活函数运算。整个网络依旧是一个可导的评分函数：该函数的输入是原始的图像像素，输出是不同类别的评分。在最后一层（往往是全连接层），网络依旧有一个损失函数（比如SVM或Softmax），并且在神经网络中我们实现的各种技巧和要点依旧适用于卷积神经网络。

那么有哪些地方变化了呢？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

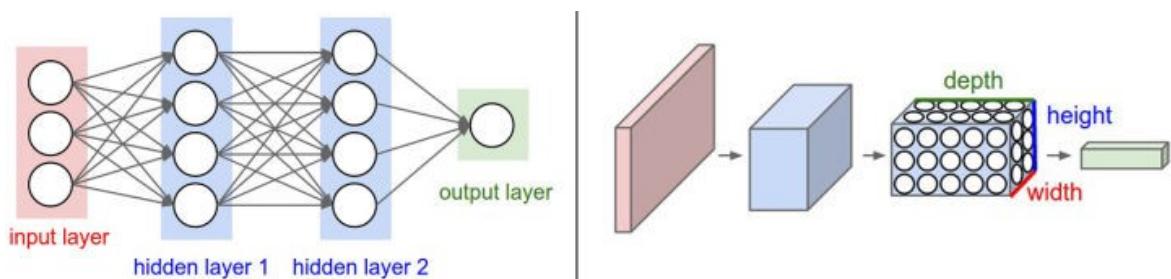
得前向传播函数实现起来更高效，并且大幅度降低了网络中参数的数量。

结构概述

回顾：常规神经网络。在上一章中，神经网络的输入是一个向量，然后在一系列的隐层中对它做变换。每个隐层都是由若干的神经元组成，每个神经元都与前一层中的所有神经元连接。但是在一个隐层中，神经元相互独立不进行任何连接。最后的全连接层被称为“输出层”，在分类问题中，它输出的值被看做是不同类别的评分值。

常规神经网络对于大尺寸图像效果不尽人意。在CIFAR-10中，图像的尺寸是 $32 \times 32 \times 3$ （宽高均为32像素，3个颜色通道），因此，对应的常规神经网络的第一个隐层中，每一个单独的全连接神经元就有 $32 \times 32 \times 3 = 3072$ 个权重。这个数量看起来还可以接受，但是很显然这个全连接的结构不适用于更大尺寸的图像。举例说来，一个尺寸为 $200 \times 200 \times 3$ 的图像，会让神经元包含 $200 \times 200 \times 3 = 120,000$ 个权重值。而网络中肯定不止一个神经元，那么参数的量就会快速增加！显而易见，这种全连接方式效率低下，大量的参数也很快会导致网络过拟合。

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）。举个例子，CIFAR-10中的图像是作为卷积神经网络的输入，该数据体的维度是 $32 \times 32 \times 3$ （宽度，高度和深度）。我们将看到，层中的神经元将只与前一层中的一小块区域连接，而不是采取全连接方式。对于用来分类CIFAR-10中的图像的卷积网络，其最后的输出层的维度是 $1 \times 1 \times 10$ ，因为在卷积神经网络结构的最后部分将会把全尺寸的图像压缩为包含分类评分的一个向量，向量是在深度方向排列的。下面是例子：



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。在这个例子中，红色的输入层装的是图像，所以它的宽度和高度就是图像的宽度和高度，它的深度是3（代表

了红、绿、蓝3种颜色通道)。

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。

用来构建卷积网络的各种层

一个简单的卷积神经网络是由各种层按照顺序排列组成，网络中的每个层使用一个可以微分的函数将激活数据从一个层传递到另一个层。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）。通过将这些层叠加起来，就可以构建一个完整的卷积神经网络。

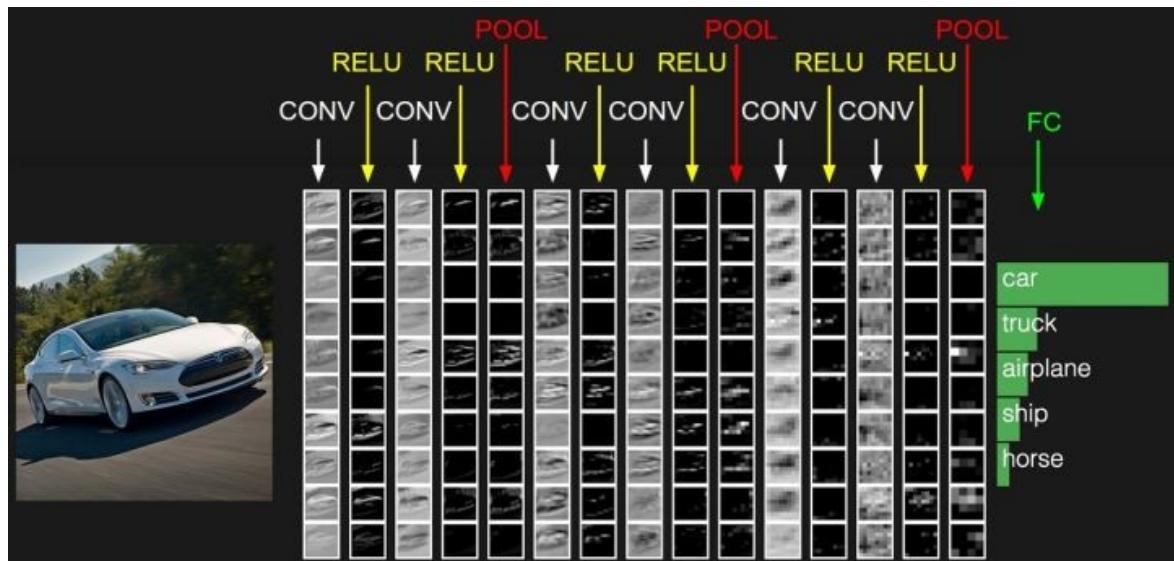
网络结构例子：这仅仅是个概述，下面会更详解的介绍细节。一个用于CIFAR-10图像数据分类的卷积神经网络的结构可以是[输入层-卷积层-ReLU层-汇聚层-全连接层]。细节如下：

- 输入[32x32x3]存有图像的原始像素值，本例中图像宽高均为32，有3个颜色通道。
- 卷积层中，神经元与输入层中的一个局部区域相连，每个神经元都计算自己与输入层相连的小区域与自己权重的内积。卷积层会计算所有神经元的输出。如果我们使用12个滤波器（也叫作核），得到的输出数据体的维度就是[32x32x12]。
- ReLU层将会逐个元素地进行激活函数操作，比如使用以0为阈值的 $\max(0, x)$ 作为激活函数。该层对数据尺寸没有改变，还是[32x32x12]。
- 汇聚层在在空间维度（宽度和高度）上进行降采样（downsampling）操作，数据尺寸变为[16x16x12]。
- 全连接层将会计算分类评分，数据尺寸变为[1x1x10]，其中10个数字对应的就是CIFAR-10中10个类别的分类评分值。正如其名，全连接层与常规神经网络一样，其中每个神经元都与前一层中所有神经元相连接。

由此看来，卷积神经网络一层一层地将图像从原始像素值转换成最终的分类评分值。其中有的层含有参数，有的没有。具体说来，卷积层和全连接层（CONV/FC）对输入执行变换操作的时候，不仅会用到激活函数，还会用到很多参数（神经元的突触权值和偏差）。而ReLU层和汇聚层则是进行一个固定不变的函数操作。卷积层和全连接层中的参数会随着梯度下降被训练，这样卷积神经网络计算出的分类评分就能和训练集中的每个图像的标签吻合了。

小结：

- 简单案例中卷积神经网络的结构，就是一系列的层将输入数据变换为输出数据（比如分类评分）。
- 卷积神经网络结构中有几种不同类型的层（目前最流行的有卷积层、全连接层、ReLU层和汇聚层）。
- 每个层的输入是3D数据，然后使用一个可导的函数将其变换为3D的输出数据。
- 有的层有参数，有的没有（卷积层和全连接层有，ReLU层和汇聚层没有）。
- 有的层有额外的超参数，有的没有（卷积层、全连接层和汇聚层有，ReLU层没有）。



一个卷积神经网络的激活输出例子。左边的输入层存有原始图像像素，右边的输出层存有类别分类评分。在处理流程中的每个激活数据体是铺成一列来展示的。因为对3D数据作图比较困难，我们就把每个数据体切成层，然后铺成一列显示。最后一层装的是针对不同类别的分类得分，这里只显示了得分最高的5个评分值和对应的类别。完整的[网页演示](#)在我们的课程主页。本例中的结构是一个小的VGG网络，VGG网络后面会有讨论。

现在讲解不同的层，层的超参数和连接情况的细节。

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。

概述和直观介绍：首先讨论的是，在没有大脑和生物意义上的神经元之类的比喻下，卷积层到底在计算什么。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。举例来说，卷积神经网络第一层的一个典型的滤波器的尺寸可以是 $5 \times 5 \times 3$ （宽高都是5像素，深度是3是因为图像应为颜色通道，所以有3的深度）。在前向传播的时候，让每个滤波器都在输入数据的宽度和高度上滑动（更精确地说是卷积），然后计算整个滤波器和输入数据任一处的内积。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，具体的视觉特征可能是某些方位上的边界，或者在第一层上某些颜色的斑点，甚至可以是网络更高层上的蜂巢状或者车轮状图案。

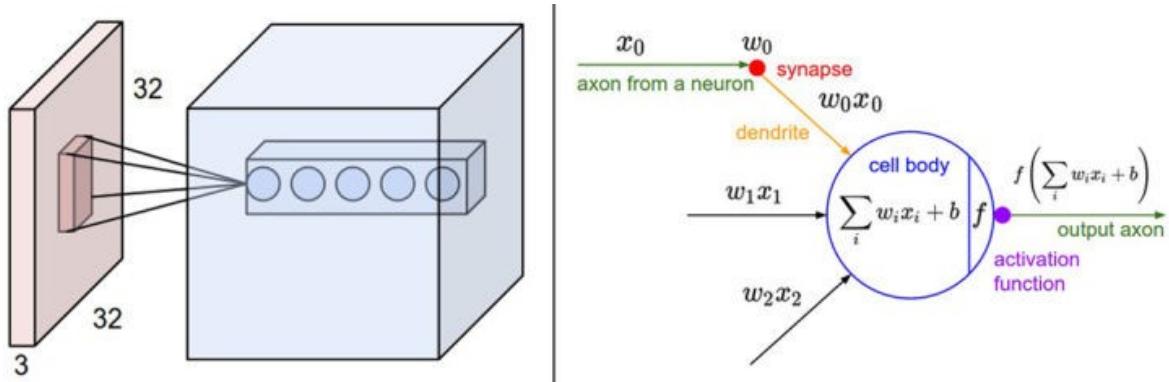
在每个卷积层上，我们会有一整个集合的滤波器（比如12个），每个都会生成一个不同的二维激活图。将这些激活映射在深度方向上层叠起来就生成了输出数据。

以大脑做比喻：如果你喜欢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（因为这些数字都是使用同一个滤波器得到的结果）。现在开始讨论神经元的连接，它们在空间中的排列，以及它们参数共享的模式。

局部连接：在处理图像这样的高维度输入时，让每个神经元都与前一层中的所有神经元进行全连接是不现实的。相反，我们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）。在深度方向上，这个连接的大小总是和输入量的深度相等。需要再次强调的是，我们对待空间维度（宽和高）与深度维度是不同的：连接在空间（宽高）上是局部的，但是在深度上总是和输入数据的深度一致。

例1：假设输入数据体尺寸为 $[32 \times 32 \times 3]$ （比如CIFAR-10的RGB图像），如果感受野（或滤波器尺寸）是 5×5 ，那么卷积层中的每个神经元会有输入数据体中 $[5 \times 5 \times 3]$ 区域的权重，共 $5 \times 5 \times 3 = 75$ 个权重（还要加一个偏差参数）。注意这个连接在深度维度上的大小必须为3，和输入数据体的深度一致。

例2：假设输入数据体的尺寸是 $[16 \times 16 \times 20]$ ，感受野尺寸是 3×3 ，那么卷积层中每个神经元和输入数据体就有 $3 \times 3 \times 20 = 180$ 个连接。再次提示：在空间上连接是局部的（ 3×3 ），但是在深度上是和输入数据体一致的（20）。



左边：红色的是输入数据体（比如CIFAR-10中的图像），蓝色的部分是第一个卷积层中的神经元。卷积层中的每个神经元都只是与输入数据体的一个局部在空间上相连，但是与输入数据体的所有深度维度全部相连（所有颜色通道）。在深度方向上有多个神经元（本例中5个），它们都接受输入数据的同一块区域（**感受野**相同）。至于深度列的讨论在下文中有。

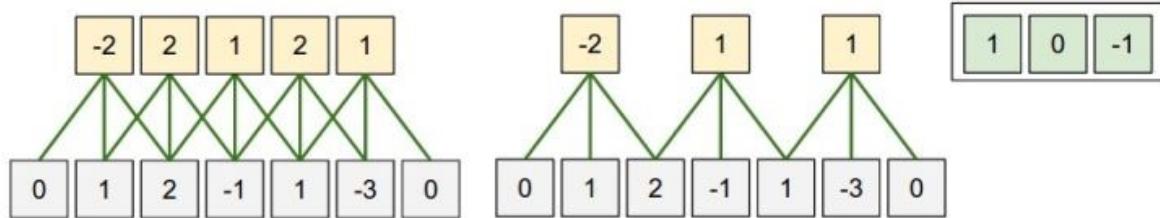
右边：神经网络章节中介绍的神经元保持不变，它们还是计算权重和输入的内积，然后进行激活函数运算，只是它们的连接被限制在一个局部空间。

空间排列：上文讲解了卷积层中每个神经元与输入数据体之间的连接方式，但是尚未讨论输出数据体中神经元的数量，以及它们的排列方式。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（或者不常用的3，或者更多，这些在实际中很少使用），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. 在下文可以看到，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（最常用的是用来保持输入数据体在空间上的尺寸，这样输入和输出的宽高都相等）。

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 $(W-F+2P)/S+1$ 。比如输入是 7×7 , 滤波器是 3×3 , 步长为 1 , 填充为 0 , 那么就能得到一个 5×5 的输出。如果步长为 2 , 输出就是 3×3 。下面是例子 :



空间排列的图示。在本例中只有一个空间维度 (x 轴) , 神经元的感受野尺寸 $F=3$, 输入尺寸 $W=5$, 零填充 $P=1$ 。左边 : 神经元使用的步长 $S=1$, 所以输出尺寸是 $(5-3+2)/1+1=5$ 。右边 : 神经元的步长 $S=2$, 则输出尺寸是 $(5-3+2)/2+1=3$ 。注意当步长 $S=3$ 时是无法使用的 , 因为它无法整齐地穿过数据体。从等式上来说 , 因为 $(5-3+2)=4$ 是不能被 3 整除的。

本例中 , 神经元的权重是 $[1, 0, -1]$, 显示在图的右上角 , 偏差值为 0。这些权重是被所有黄色的神经元共享的 (参数共享的内容看下文相关内容)。

使用零填充 : 在上面左边例子中 , 注意输入维度是 5 , 输出维度也是 5。之所以如此 , 是因为感受野是 3 并且使用了 1 的零填充。如果不使用零填充 , 则输出数据体的空间维度就只有 3 , 因为这就是滤波器整齐滑过并覆盖原始数据需要的数目。一般说来 , 当步长 $S=1$ 时 , 零填充的值是 $P=(F-1)/2$, 这样就能保证输入和输出数据体有相同的空间尺寸。这样做非常常见 , 在介绍卷积神经网络的结构的时候我们会详细讨论其原因。

步长的限制 : 注意这些空间排列的超参数之间是相互限制的。举例说来 , 当输入尺寸 $W=10$, 不使用零填充则 $P=0$, 滤波器尺寸 $F=3$, 这样步长 $S=2$ 就行不通 , 因为 $(W-F+2P)/S+1=(10-3+0)/2+1=4.5$, 结果不是整数 , 这就是说神经元不能整齐对称地滑过输入数据体。因此 , 这些超参数的设定就被认为是无效的 , 一个卷积神经网络库可能会报出一个错误 , 或者修改零填充值来让设置合理 , 或者修改输入数据体尺寸来让设置合理 , 或者其他什么措施。在后面的卷积神经网络结构小节中 , 读者可以看到合理地设置网络的尺寸让所有的维度都能正常工作 , 这件事可是相当让人头痛的。而使用零填充和遵守其他一些设计策略将会有效解决这个问题。

真实案例 : Krizhevsky 构架赢得了 2012 年的 ImageNet 挑战 , 其输入图像的尺寸是 $[227 \times 227 \times 3]$ 。在第一个卷积层 , 神经元使用的感受野尺寸 $F=11$, 步

长 $S = 4$ ，不使用零填充 $P = 0$ 。因为 $(227-11)/4+1=55$ ，卷积层的深度 $K = 96$ ，则卷积层的输出数据体尺寸为 $[55 \times 55 \times 96]$ 。55x55x96个神经元中，每个都和输入数据体中一个尺寸为 $[11 \times 11 \times 3]$ 的区域全连接。在深度列上的96个神经元都是与输入数据体中同一个 $[11 \times 11 \times 3]$ 区域连接，但是权重不同。有一个有趣的细节，在原论文中，说的输入图像尺寸是 224×224 ，这是肯定错误的，因为 $(224-11)/4+1$ 的结果不是整数。这件事在卷积神经网络的历史上让很多人迷惑，而这个错误到底是怎么发生的没人知道。我的猜测是 Alex 忘记在论文中指出自己使用了尺寸为 3 的额外的零填充。

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。就用上面的例子，在第一个卷积层就有 $55 \times 55 \times 96 = 290,400$ 个神经元，每个有 $11 \times 11 \times 3 = 364$ 个参数和 1 个偏差。将这些合起来就是 $290,400 \times 364 = 105,705,600$ 个参数。单单第一层就有这么多参数，显然这个数目是非常大的。

作一个合理的假设：如果一个特征在计算某个空间位置 (x, y) 的时候有用，那么它在计算另一个不同位置 (x_2, y_2) 的时候也有用。基于这个假设，可以显著地减少参数数量。换言之，就是将深度维度上一个单独的 2 维切片看做 **深度切片**（**depth slice**），比如一个数据体尺寸为 $[55 \times 55 \times 96]$ 的就有 96 个深度切片，每个尺寸为 $[55 \times 55]$ 。在每个深度切片上的神经元都使用同样的权重和偏差。在这样的参数共享下，例子中的第一个卷积层就只有 96 个不同的权重集，一个权重集对应一个深度切片，共有 $96 \times 11 \times 11 \times 3 = 34,848$ 个不同的权重，或 34,944 个参数（+96 个偏差）。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。

注意，如果在一个深度切片中的所有权重都使用同一个权重向量，那么卷积层的前向传播在每个深度切片中可以看做是在计算神经元权重和输入数据体的 **卷积**（这就是“卷积层”名字由来）。这也是为什么总是将这些权重集合称为 **滤波器**（**filter**）（或 **卷积核**（**kernel**）），因为它们和输入进行了卷积。



Krizhevsky等学习到的滤波器例子。这96个滤波器的尺寸都是 $[11 \times 11 \times 3]$ ，在一个深度切片中，每个滤波器都被 55×55 个神经元共享。注意参数共享的假设是有道理的：如果在图像某些地方探测到一个水平的边界是很重要的，那么在其他一些地方也会同样是有用的，这是因为图像结构具有平移不变性。所以在卷积层的输出数据体的 55×55 个不同位置中，就没有必要重新学习去探测一个水平边界了。

注意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。一个具体的例子就是输入图像是人脸，人脸一般都处于图片中心。你可能期望不同的特征，比如眼睛特征或者头发特征可能（也应该）会在图片的不同位置被学习。在这个例子中，通常就放松参数共享的限制，将层称为**局部连接层**（Locally-Connected Layer）。

Numpy例子：为了让讨论更加的具体，我们用代码来展示上述思路。假设输入数据体是numpy数组**X**。那么：

- 一个位于 (x, y) 的深度列（或纤维）将会是 $X[x, y, :]$ 。
- 在深度为 d 处的深度切片，或激活图应该是 $X[:, :, d]$ 。

卷积层例子：假设输入数据体**X**的尺寸**X.shape: (11, 11, 4)**，不使用零填充（ $P = 0$ ），滤波器的尺寸是 $F = 5$ ，步长 $S = 2$ 。那么输出数据体的空间尺寸就是 $(11-5)/2+1=4$ ，即输出数据体的宽度和高度都是4。那么在输出数据体中的激活映射（称其为**V**）看起来就是下面这样（在这个例子中，只有部分元素被计算）：

- $V[0, 0, 0] = \text{np.sum}(X[:5, :5, :] * W0) + b0$
- $V[1, 0, 0] = \text{np.sum}(X[2:7, :5, :] * W0) + b0$
- $V[2, 0, 0] = \text{np.sum}(X[4:9, :5, :] * W0) + b0$
- $V[3, 0, 0] = \text{np.sum}(X[6:11, :5, :] * W0) + b0$

在numpy中，*操作是进行数组间的逐元素相乘。权重向量**W0**是该神经元的权重，**b0**是其偏差。在这里，**W0**被假设尺寸是**W0.shape: (5, 5, 4)**，因为滤波器的宽高是5，输入数据量的深度是4。注意在每一个点，计算点积的方式和之前的常规神经网络是一样的。同时，计算内积的时候使用的是同一个权重和偏差（因为参数共享），在宽度方向的数字每次上升2（因为步长为2）。要构建输出数据体中的第二张激活图，代码应该是：

- $V[0, 0, 1] = \text{np.sum}(X[:5, :5, :] * W1) + b1$

- $V[1,0,1] = \text{np.sum}(X[2:7,:5,:]) * W1 + b1$
- $V[2,0,1] = \text{np.sum}(X[4:9,:5,:]) * W1 + b1$
- $V[3,0,1] = \text{np.sum}(X[6:11,:5,:]) * W1 + b1$
- $V[0,1,1] = \text{np.sum}(X[:5,2:7,:]) * W1 + b1$ (在y方向上)
- $V[2,3,1] = \text{np.sum}(X[4:9,6:11,:]) * W1 + b1$ (或两个方向上同时)

我们访问的是**V**的深度维度上的第二层 (即index1) , 因为是在计算第二个激活图 , 所以这次试用的参数集就是**W1**了。在上面的例子中 , 为了简洁略去了卷积层对于输出数组**V**中其他部分的操作。还有 , 要记得这些卷积操作通常后面接的是ReLU层 , 对激活图中的每个元素做激活函数运算 , 这里没有显示。

小结 : 我们总结一下卷积层的性质 :

- 输入数据体的尺寸为 $W_1 \times H_1 \times D_1$
- 4个超参数 :
 - 滤波器的数量 K
 - 滤波器的空间尺寸 F
 - 步长 S
 - 零填充数量 P
- 输出数据体的尺寸为 $W_2 \times H_2 \times D_2$, 其中 :

$$W_2 = (W_1 - F + 2P)/S + 1$$

$$D_2 = K$$

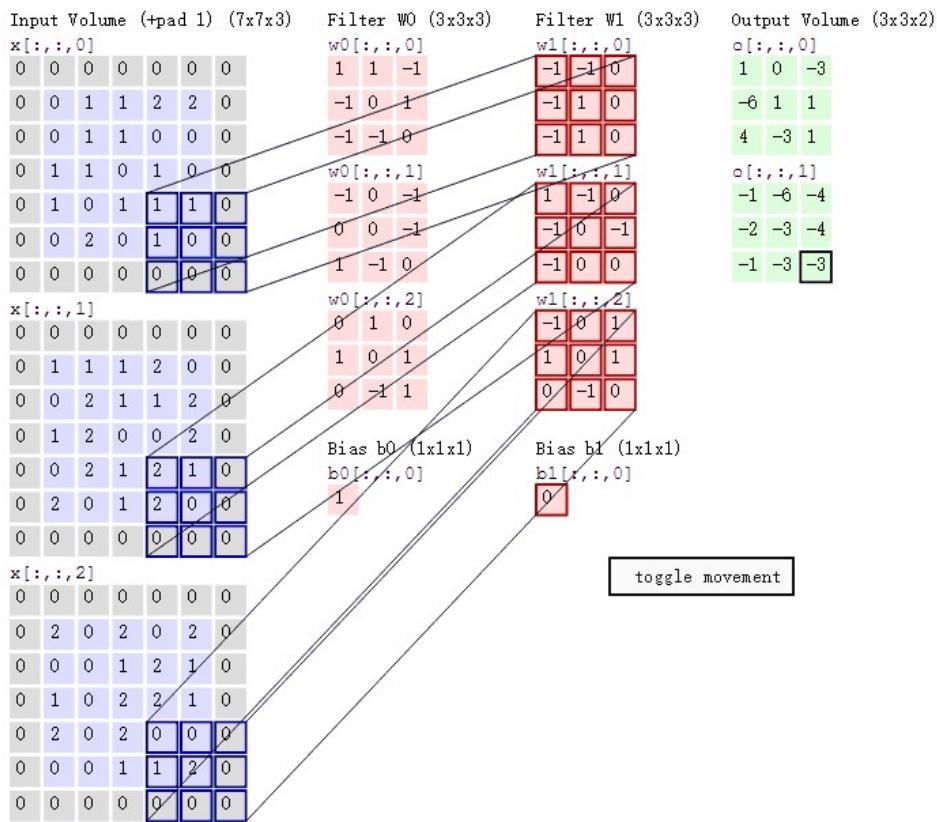
$$H_2 = (H_1 - F + 2P)/S + 1 \quad (\text{宽度和高度的计算方法相同})$$
- 由于参数共享 , 每个滤波器包含 $F \cdot F \cdot D_1$ 个权重 , 卷积层一共有 $F \cdot F \cdot D_1 \cdot K$ 个权重和 K 个偏置。
- 在输出数据体中 , 第 d 个深度切片 (空间尺寸是 $W_2 \times H_2$) , 用第 d 个滤波器和输入数据进行有效卷积运算的结果 (使用步长 S) , 最后在加上第 d 个偏差。

对这些超参数 , 常见的设置是 $F = 3$, $S = 1$, $P = 1$ 。同时设置这些超参数也有一些约定俗成的惯例和经验 , 可以在下面的卷积神经网络结构章节中查

看。

卷积层演示：下面是一个卷积层的运行演示。因为3D数据难以可视化，所以所有的数据（输入数据体是蓝色，权重数据体是红色，输出数据体是绿色）都采取将深度切片按照列的方式排列展现。输入数据体的尺寸是 $W_1 = 5, H_1 = 5, D_1 = 3$ ，卷积层参数 $K = 2, F = 3, S = 2, P = 1$ 。

就是说，有2个滤波器，滤波器的尺寸是 3×3 ，它们的步长是2.因此，输出数据体的空间尺寸是 $(5-3+2)/2+1=3$ 。注意输入数据体使用了零填充 $P = 1$ ，所以输入数据体外边缘一圈都是0。下面的例子在绿色的输出激活数据上循环演示，展示了其中每个元素都是先通过蓝色的输入数据和红色的滤波器逐元素相乘，然后求其总和，最后加上偏差得来。



译者注：请点击图片查看动画演示。如果gif不能正确播放，请读者前往[斯坦福课程官网](#)查看此演示。

用矩阵乘法实现：卷积运算本质上就是在滤波器和输入数据的局部区域间做点积。卷积层的常用实现方式就是利用这一点，将卷积层的前向传播变成一个巨大的矩阵乘法：

1. 输入图像的局部区域被**im2col**操作拉伸为列。比如，如果输入是

[227x227x3]，要与尺寸为11x11x3的滤波器以步长为4进行卷积，就取输入中的[11x11x3]数据块，然后将其拉伸为长度为11x11x3=363的列向量。重复进行这一过程，因为步长为4，所以输出的宽高为(227-11)/4+1=55，所以得到 $m2co$ 操作的输出矩阵X_col的尺寸是[363x3025]，其中每列是拉伸的感受野，共有 $55 \times 55 = 3,025$ 个。注意因为感受野之间有重叠，所以输入数据体中的数字在不同的列中可能有重复。

2. 卷积层的权重也同样被拉伸成行。举例，如果有96个尺寸为[11x11x3]的滤波器，就生成一个矩阵W_row，尺寸为[96x363]。
3. 现在卷积的结果和进行一个大矩阵乘 $\text{np.dot}(W_row, X_col)$ 是等价的了，能得到每个滤波器和每个感受野间的点积。在我们的例子中，这个操作的输出是[96x3025]，给出了每个滤波器在每个位置的点积输出。
4. 结果最后必须被重新变为合理的输出尺寸[55x55x96]。

这个方法的缺点就是占用内存太多，因为在输入数据体中的某些值在X_col中被复制了多次。但是，其优点是矩阵乘法有非常多的高效实现方式，我们都可使用（比如常用的BLAS API）。还有，同样的 $m2co$ 思路可以用在汇聚操作中。

反向传播：卷积操作的反向传播（同时对于数据和权重）还是一个卷积（但是是和空间上翻转的滤波器）。使用一个1维的例子比较容易演示。

1x1卷积：一些论文中使用了1x1的卷积，这个方法最早是在论文[Network in Network](#)中出现。人们刚开始看见这个1x1卷积的时候比较困惑，尤其是那些具有信号处理专业背景的人。因为信号是2维的，所以1x1卷积就没有意义。但是，在卷积神经网络中不是这样，因为这里是对3个维度进行操作，滤波器和输入数据体的深度是一样的。比如，如果输入是[32x32x3]，那么1x1卷积就是在高效地进行3维点积（因为输入深度是3个通道）。

扩张卷积：最近一个研究（[Fisher Yu和Vladlen Koltun的论文](#)）给卷积层引入了一个新的叫扩张（*attribution*）的超参数。到目前为止，我们只讨论了卷积层滤波器是连续的情况。但是，让滤波器中元素之间有间隙也是可以的，这就叫做扩张。举例，在某个维度上滤波器w的尺寸是3，那么计算输入x的方式是： $w[0]*x[0] + w[1]*x[1] + w[2]*x[2]$ ，此时扩张为0。如果扩张为1，那么计算为： $w[0]*x[0] + w[1]*x[2] + w[2]*x[4]$ 。换句话说，操作中存在1的间隙。在某些设置中，扩张卷积与正常卷积结合起来非常有用，因为在很少的层数内更快地汇集输入图片的大尺度特征。比如，如果上下重叠2个3x3的卷积层，那么第二个卷积层的神经元的感受野是输入数据体中5x5的区域（可以成这些神经元的有效感受野是5x5）。如果我们对卷积进行扩张，那么这个有

效感受野就会迅速增长。

汇聚层

通常，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 2×2 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 2×2 的区域）。深度保持不变。汇聚层的一些公式：

- 输入数据体尺寸 $W_1 \cdot H_1 \cdot D_1$
- 有两个超参数：
 - 空间大小 F
 - 步长 S
- 输出数据体尺寸 $W_2 \cdot H_2 \cdot D_2$ ，其中

$$W_2 = (W_1 - F)/S + 1$$

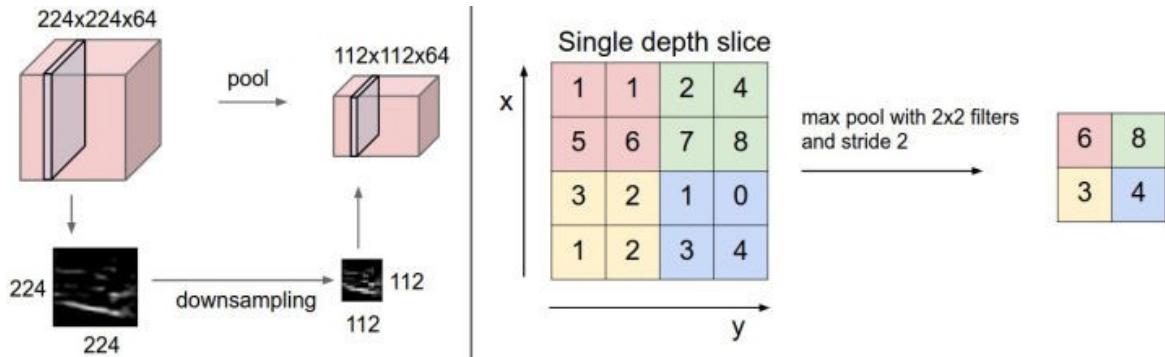
$$H_2 = (H_1 - F)/S + 1$$

$$D_2 = D_1$$

- 因为对输入进行的是固定函数计算，所以没有引入参数
- 在汇聚层中很少使用零填充

在实践中，最大汇聚层通常只有两种形式：一种是 $F = 3, S = 2$ ，也叫重叠汇聚（overlapping pooling），另一个更常用的是 $F = 2, S = 2$ 。对更大感受野进行汇聚需要的汇聚尺寸也更大，而且往往对网络有破坏性。

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（ L_2 -norm pooling）。平均汇聚历史上比较常用，但是现在已经很少使用了。因为实践证明，最大汇聚的效果比平均汇聚要好。



汇聚层在输入数据体的每个深度切片上，独立地对其进行空间上的降采样。左边：本例中，输入数据体尺寸[224x224x64]被降采样到了[112x112x64]，采取的滤波器尺寸是2，步长为2，而深度不变。右边：最常用的降采样操作是取最大值，也就是最大汇聚，这里步长为2，每个取最大值操作是从4个数字中选取（即2x2的方块区域中）。

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。

不使用汇聚层：很多人不喜欢汇聚操作，认为可以不使用它。比如在[Striving for Simplicity: The All Convolutional Net](#)一文中，提出使用一种只有重复的卷积层组成的结构，抛弃汇聚层。通过在卷积层中使用更大的步长来降低数据体的尺寸。有发现认为，在训练一个良好的生成模型时，弃用汇聚层也是很重要的。比如变化自编码器（VAEs：variational autoencoders）和生成性对抗网络（GANs：generative adversarial networks）。现在看起来，未来的卷积网络结构中，可能会很少使用甚至不使用汇聚层。

归一化层

在卷积神经网络的结构中，提出了很多不同类型的归一化层，有时候是为了实现在生物大脑中观测到的抑制机制。但是这些层渐渐都不再流行，因为实践证明它们的效果即使存在，也是极其有限的。对于不同类型的归一化层，可以看看Alex Krizhevsky的关于[cuda-convnet library API](#)的讨论。

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。更多细节请查看[神经网络](#)章节。

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。因此，将此两者相互转化是可能的：

- 对于任一个卷积层，都存在一个能实现和它一样的前向传播函数的全连接层。权重矩阵是一个巨大的矩阵，除了某些特定块（这是因为有局部连接），其余部分都是零。而在其中大部分块中，元素都是相等的（因为参数共享）。
- 相反，任何全连接层都可以被转化为卷积层。比如，一个 $K = 4096$ 的全连接层，输入数据体的尺寸是 $7 \times 7 \times 512$ ，这个全连接层可以被等效地看做一个 $F = 7, P = 0, S = 1, K = 4096$ 的卷积层。换句话说，就是将滤波器的尺寸设置为和输入数据体的尺寸一致了。因为只有一个单独的深度列覆盖并滑过输入数据体，所以输出将变成 $1 \times 1 \times 4096$ ，这个结果就和使用初始的那个全连接层一样了。

全连接层转化为卷积层：在两种变换中，将全连接层转化为卷积层在实际运用中更加有用。假设一个卷积神经网络的输入是 $224 \times 224 \times 3$ 的图像，一系列的卷积层和汇聚层将图像数据变为尺寸为 $7 \times 7 \times 512$ 的激活数据体（在 AlexNet 中就是这样，通过使用 5 个汇聚层来对输入数据进行空间上的降采样，每次尺寸下降一半，所以最终空间尺寸为 $224/2/2/2/2=7$ ）。从这里可以看到，AlexNet 使用了两个尺寸为 4096 的全连接层，最后一个有 1000 个神经元的全连接层用于计算分类评分。我们可以将这 3 个全连接层中的任意一个转化为卷积层：

- 针对第一个连接区域是 $[7 \times 7 \times 512]$ 的全连接层，令其滤波器尺寸为 $F = 7$ ，这样输出数据体就为 $[1 \times 1 \times 4096]$ 了。
- 针对第二个全连接层，令其滤波器尺寸为 $F = 1$ ，这样输出数据体为 $[1 \times 1 \times 4096]$ 。
- 对最后一个全连接层也做类似的，令其 $F = 1$ ，最终输出为 $[1 \times 1 \times 1000]$

实际操作中，每次这样的变换都需要把全连接层的权重 W 重塑成卷积层的滤波器。那么这样的转化有什么作用呢？它在下面的情况下可以更高效：让卷积网络在一张更大的输入图片上滑动（**译者注**：即把一张更大的图片的不同区域都分别带入到卷积网络，得到每个区域的得分），得到多个输出，这样的转化可以让我们在单个向前传播的过程中完成上述的操作。

举个例子，如果我们想让 224×224 尺寸的浮窗，以步长为32在 384×384 的图片上滑动，把每个经停的位置都带入卷积网络，最后得到 6×6 个位置的类别得分。上述的把全连接层转换成卷积层的做法会更简便。如果 224×224 的输入图片经过卷积层和汇聚层之后得到了 $[7 \times 7 \times 512]$ 的数组，那么， 384×384 的大图片直接经过同样的卷积层和汇聚层之后会得到 $[12 \times 12 \times 512]$ 的数组（因为途径5个汇聚层，尺寸变为 $384/2/2/2/2 = 12$ ）。然后再经过上面由3个全连接层转化得到的3个卷积层，最终得到 $[6 \times 6 \times 1000]$ 的输出（因为 $(12 - 7)/1 + 1 = 6$ ）。这个结果正是浮窗在原图经停的 6×6 个位置的得分！（**译者注**：这一段的翻译与原文不同，经过了译者较多的修改，使更容易理解）

面对 384×384 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 224×224 块进行多次评价，其效果和使用把全连接层转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。

自然，相较于使用被转化前的原始卷积神经网络对所有36个位置进行迭代计算，使用转化后的卷积神经网络进行一次前向传播计算要高效得多，因为36次计算都在共享计算资源。这一技巧在实践中经常使用，一次来获得更好的结果。比如，通常将一张图像尺寸变得更大，然后使用变换后的卷积神经网络来对空间上很多不同位置进行评价得到分类评分，然后在求这些分值的平均值。

最后，如果我们想用步长小于32的浮窗怎么办？用多次的向前传播就可以解决。比如我们想用步长为16的浮窗。那么先使用原图在转化后的卷积网络执行向前传播，然后分别沿宽度，沿高度，最后同时沿宽度和高度，把原始图片分别平移16个像素，然后把这些平移之后的图分别带入卷积网络。（**译者注**：这一段的翻译与原文不同，经过了译者较多的修改，使更容易理解）

- [Net Surgery](#)上一个使用Caffe演示如何在进行变换的IPython Note教程。

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。在本节中将讨论在卷积神经网络中这些层通常是如何组合在一起的。

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。换句话说，最常见的卷积神经网络结构如下：

INPUT -> [[CONV -> RELU]^N -> POOL?^M] -> [FC -> RELU]^K -> FC

其中*指的是重复次数，POOL?指的是一个可选的汇聚层。其中 $N \geq 0$ ，通常 $N \leq 3, M \geq 0, K \geq 0$ ，通常 $K < 3$ 。例如，下面是一些常见的网络结构规律：

- **INPUT -> FC**，实现一个线性分类器，此处 $N = M = K = 0$ 。
- **INPUT -> CONV -> RELU -> FC**
- **INPUT -> [CONV -> RELU -> POOL]² -> FC -> RELU -> FC**。此处在每个汇聚层之间有一个卷积层。
- **INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL]³ -> [FC -> RELU]² -> FC**。此处每个汇聚层前有两个卷积层，这个思路适用于更大更深的网络，因为在执行具有破坏性的汇聚操作前，多重的卷积层可以从输入数据中学习到更多的复杂特征。

几个小滤波器卷积层的组合比一个大滤波器卷积层好：假设你一层一层地重叠了3个 3×3 的卷积层（层与层之间有非线性激活函数）。在这个排列下，第一个卷积层中的每个神经元都对输入数据体有一个 3×3 的视野。第二个卷积层上的神经元对第一个卷积层有一个 3×3 的视野，也就是对输入数据体有 5×5 的视野。同样，在第三个卷积层上的神经元对第二个卷积层有 3×3 的视野，也就是对输入数据体有 7×7 的视野。假设不采用这3个 3×3 的卷积层，而是使用一个单独的 7×7 的感受野的卷积层，那么所有神经元的感受野也是 7×7 ，但是就有一些缺点。首先，多个卷积层与非线性的激活层交替的结构，比单一卷积层的结构更能提取出深层的更好的特征。其次，假设所有的数据有 C 个通道，那么单独的 7×7 卷积层将会包含 $C \times (7 \times 7 \times C) = 49C^2$ 个参数，而3个 3×3 的卷积层的组合仅有 $3 \times (C \times (3 \times 3 \times C)) = 27C^2$ 个参数。直观说来，最好选择带有小滤波器的卷积层组合，而不是用一个带有大的滤波器的卷积层。前者可以表达出输入数据中更多个强力特征，使用的参数也更少。唯一的不足是，在进行反向传播时，中间的卷积层可能会导致占用更多的内存。

最新进展：传统的将层按照线性进行排列的方法已经受到了挑战，挑战来自谷歌的Inception结构和微软亚洲研究院的残差网络（Residual Net）结构。这两个网络（下文案例学习小节中有细节）的特征更加复杂，连接结构也不同。

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。现在先介绍设置结构尺寸的一般性规则，然后根据这些规则进行讨论：

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 3×3 或最多 5×5 ），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。比如，当 $F = 3$ ，那就使用 $P = 1$ 来保持输入尺寸。当 $F = 5, P = 2$ ，一般对于任意 F ，当 $P = (F - 1)/2$ 的时候能保持输入尺寸。如果必须使用更大的滤波器尺寸（比如 7×7 之类），通常只用在第一个面对原始图像的卷积层上。

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 2×2 感受野（即 $F = 2$ ）的最大值汇聚，步长为2（ $S = 2$ ）。注意这一操作将会把输入数据中75%的激活数据丢弃（因为对宽度和高度都进行了2的降采样）。另一个不那么常用的设置是使用 3×3 的感受野，步长为2。最大值汇聚的感受野尺寸很少有超过3的，因为汇聚操作过于激烈，易造成数据信息丢失，这通常会导致算法性能变差。

减少尺寸设置的问题：上文中展示的两种设置是很好的，因为所有的卷积层都能保持其输入数据的空间尺寸，汇聚层只负责对数据体从空间维度进行降采样。如果使用的步长大于1并且不对卷积层的输入数据使用零填充，那么就必须非常仔细地监督输入数据体通过整个卷积神经网络结构的过程，确认所有的步长和滤波器尺寸互相吻合，卷积神经网络的结构美妙对称地联系在一起。

为什么在卷积层使用1的步长？在实际应用中，更小的步长效果更好。上文也已经提过，步长为1可以让空间维度的降采样全部由汇聚层负责，卷积层只负责对输入数据体的深度进行变换。

为何使用零填充？使用零填充除了前面提到的可以让卷积层的输出数据保持和输入数据在空间维度的不变，还可以提高算法性能。如果卷积层值进行卷积而不进行零填充，那么数据体的尺寸就会略微减小，那么图像边缘的信息就会过快地损失掉。

因为内存限制所做的妥协：在某些案例（尤其是早期的卷积神经网络结构）中，基于前面的各种规则，内存的使用量迅速飙升。例如，使用64个尺寸为 3×3 的滤波器对 $224 \times 224 \times 3$ 的图像进行卷积，零填充为1，得到的激活数据体尺寸是 $[224 \times 224 \times 64]$ 。这个数量就是一千万的激活数据，或者就是72MB的内存（每张图就是这么多，激活函数和梯度都是）。因为GPU通常因为内存导致性

能瓶颈，所以做出一些妥协是必须的。在实践中，人们倾向于在网络的第一个卷积层做出妥协。例如，可以妥协可能是在第一个卷积层使用步长为2，尺寸为7x7的滤波器（比如在ZFnet中）。在AlexNet中，滤波器的尺寸的11x11，步长为4。

案例学习

下面是卷积神经网络领域中比较有名的几种结构：

- **LeNet**：第一个成功的卷积神经网络应用，是Yann LeCun在上世纪90年代实现的。当然，最著名还是被应用在识别数字和邮政编码等的LeNet结构。
- **AlexNet**：AlexNet卷积神经网络在计算机视觉领域中受到欢迎，它由Alex Krizhevsky，Ilya Sutskever和Geoff Hinton实现。AlexNet在2012年的ImageNet ILSVRC 竞赛中夺冠，性能远远超出第二名（16%的top5错误率，第二名是26%的top5错误率）。这个网络的结构和LeNet非常类似，但是更深更大，并且使用了层叠的卷积层来获取特征（之前通常是只用一个卷积层并且在其后马上跟着一个汇聚层）。
- **ZF Net**：Matthew Zeiler和Rob Fergus发明的网络在ILSVRC 2013比赛中夺冠，它被称为ZFNet（Zeiler & Fergus Net的简称）。它通过修改结构中的超参数来实现对AlexNet的改良，具体说来就是增加了中间卷积层的尺寸，让第一层的步长和滤波器尺寸更小。
- **GoogLeNet**：ILSVRC 2014的胜利者是谷歌的Szeged等实现的卷积神经网络。它主要的贡献就是实现了一个奠基模块，它能够显著地减少网络中参数的数量（AlexNet中有60M，该网络中只有4M）。还有，这个论文中没有使用卷积神经网络顶部使用全连接层，而是使用了一个平均汇聚，把大量不是很重要的参数都去除掉了。GoogLeNet还有几种改进的版本，最新的一个是Inception-v4。
- **VGGNet**：ILSVRC 2014的第二名是Karen Simonyan和Andrew Zisserman实现的卷积神经网络，现在称其为VGGNet。它主要的贡献是展示出网络的深度是算法优良性能的关键部分。他们最好的网络包含了16个卷积/全连接层。网络的结构非常一致，从头到尾全部使用的是3x3的卷积和2x2的汇聚。他们的预训练模型是可以在网络上获得并在Caffe中使用的。VGGNet不好的一点是它耗费更多计算资源，并且使用了更多的参数，导致更多的内存占用（140M）。其中绝大多数的参数都是来自于第一个全连接层。后来发现这些全连接层即使被去除，对于性能也没有什么影响，这样就显著降低了参数数量。
- **ResNet**：残差网络（Residual Network）是ILSVRC2015的胜利者，由

何恺明等实现。它使用了特殊的跳跃链接，大量使用了批量归一化 (batch normalization)。这个结构同样在最后没有使用全连接层。读者可以查看何恺明的演讲 ([视频](#), [PPT](#))，以及一些使用Torch重现网络的[实验](#)。ResNet当前最好的卷积神经网络模型 (2016年五月)。何开明等最近的工作是对原始结构做一些优化，可以看论文[Identity Map pings in Deep Residual Networks](#)，2016年3月发表。

VGGNet的细节：我们进一步对VGGNet的细节进行分析学习。整个VGGNet中的卷积层都是以步长为1进行3x3的卷积，使用了1的零填充，汇聚层都是以步长为2进行了2x2的最大值汇聚。可以写出处理过程中每一步数据体尺寸的变化，然后对数据尺寸和整体权重的数量进行查看：

```
INPUT: [224x224x3] memory: 224*224*3=150K weights: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M weights:
(3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M weights:
(3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K weights: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M weights:
(3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M weights:
(3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K weights: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K weights:
(3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K weights:
(3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K weights:
(3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K weights: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K weights:
(3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K weights:
(3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K weights:
(3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K weights: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K weights:
(3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K weights:
(3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K weights:
(3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K weights: 0
FC: [1x1x4096] memory: 4096 weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 weights: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 weights: 4096*1000 = 4,096,000
```

```
TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2  
for bwd)  
TOTAL params: 138M parameters
```

注意，大部分的内存和计算时间都被前面的卷积层占用，大部分的参数都用在后面的全连接层，这在卷积神经网络中是比较常见的。在这个例子中，全部参数有140M，但第一个全连接层就包含了100M的参数。

计算上的考量

在构建卷积神经网络结构时，最大的瓶颈是内存瓶颈。大部分现代GPU的内存是3/4/6GB，最好的GPU大约有12GB的内存。要注意三种内存占用来源：

- 来自中间数据体尺寸：卷积神经网络中的每一层中都有激活数据体的原始数值，以及损失函数对它们的梯度（和激活数据体尺寸一致）。通常，大部分激活数据都是在网络中靠前的层中（比如第一个卷积层）。在训练时，这些数据需要放在内存中，因为反向传播的时候还会用到。但是在测试时可以聪明点：让网络在测试运行时候每层都只存储当前的激活数据，然后丢弃前面层的激活数据，这样就能减少巨大的激活数据量。
- 来自参数尺寸：即整个网络的参数的数量，在反向传播时它们的梯度值，以及使用momentum、Adagrad或RMSProp等方法进行最优化时的每一步计算缓存。因此，存储参数向量的内存通常需要在参数向量的容量基础上乘以3或者更多。
- 卷积神经网络实现还有各种零散的内存占用，比如成批的训练数据，扩充的数据等等。

一旦对于所有这些数值的数量有了一个大略估计（包含激活数据，梯度和各种杂项），数量应该转化为以GB为计量单位。把这个值乘以4，得到原始的字节数（因为每个浮点数占用4个字节，如果是双精度浮点数那就是占用8个字节），然后多次除以1024分别得到占用内存的KB，MB，最后是GB计量。如果你的网络工作得不好，一个常用的方法是降低批尺寸（batch size），因为绝大多数的内存都是被激活数据消耗掉了。