目录

一、反向传播算法

反向传播算法是机器学习领域常用的一种算法，是一种利用链式法则递归计算表达式的梯度的方法。

所谓链式法则即求导法则中的链式法则，如果函数有嵌套，可以逐层求导。

核心问题是：给定函数 $f(x)$，其中 x 是输入数据的向量，如何计算其关于 x 的梯度，也

之所以关注这个问题，是因为在神经网络中需要优化损失函数使得性能更好，这种优化的过程经常需要沿着梯度的反方向来进行，所以求梯度是一种常用的方式。

1. 梯度的理解。

这个其实很好理解，梯度即是各个变量的偏导数按照一定顺序组成的向量。和高等数学中的梯度概念是完全一样的，不再赘述。

要理解：函数关于每个变量的偏导数指明了整个表达式关于该变量的敏感程度。

2. 链式法则求复合表达式的梯度。

考虑 $f(x)$=z(x+y)，我们拆解成 $f=q*z$ 与 $q=x+y$，这样我们可以利用链式法则来计算

这个梯度：

$$\frac{df}{dz}=q$$

$$\frac{df}{dq}=z$$

$$\frac{dq}{dx}=\frac{dq}{dy}=1$$

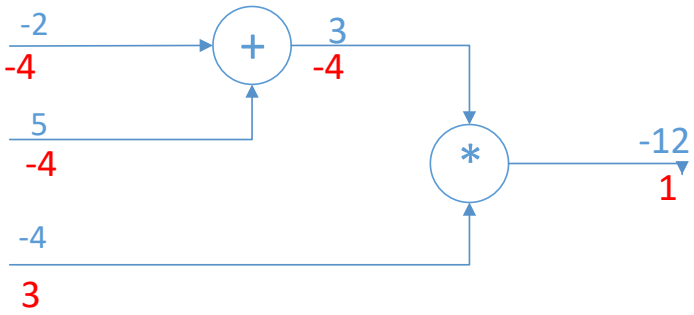
所以根据链式求导法则:

$$\frac{df}{dx}=\frac{df}{dq}\frac{dq}{dx}=z$$

$$\frac{df}{dy}=\frac{df}{dq}\frac{dq}{dy}=z$$

实际操作中，只是简单的将两个梯度数值相乘即可，示例代码：

```
"""
#设置输入
x=-2;y=5;z=-4
#进行前向传播，即计算函数的值
q=x+y      #q  3
f=q*z      #f  -12

dqdx=1;
dqdy=1;      #q对x和y的梯度

#进行逆向传播，首先处理f=q*z
dfdz=q     #所以关于z的梯度是3
dfdq=z     #所以关于q的梯度是-4

dfdx=dfdq*dqdx
dfdy=dfdq*dqdy

print("dfdz,dfdx,dfdy分别是：")
print(dfdz,dfdx,dfdy)

```



上图的计算路线显示了计算的视觉化过程，前向传播从输入计算到输出，反向传播从尾部开始前向地计算梯度（红色表示），可以认为梯度是从计算中回流。

我们把每个运算符想象成一个门单元，在整个计算路线图中，每个门单元都会得到一些输入并立即计算出两样东西：

① 这个门的输出值。

② 其输出值关于输入值的局部梯度。（这个主要是由运算形式决定的？）

门单元完成这两件事完全是独立的，不需知道计算路中的其他地方的细节，然而，一旦前向传播完毕，在反向传播的过程中，门单元将获得整个网络的输出在自己的输出值上的梯度，链式法则指出，们单元应该将回传的梯度乘以它对其输入的局部梯度，从而得到整个网络的输出对于该门单元的每个输入值的梯度。

所以这样来说，反向传播可以看作是门单元之间通过梯度信号互相通信，只要让它们的输入沿着梯度方向变化，无论他们自己的输出值在何种程度上上升或者降低，都是为了让整个网络的输出更高（建立在这个门沿着梯度方向变化）。

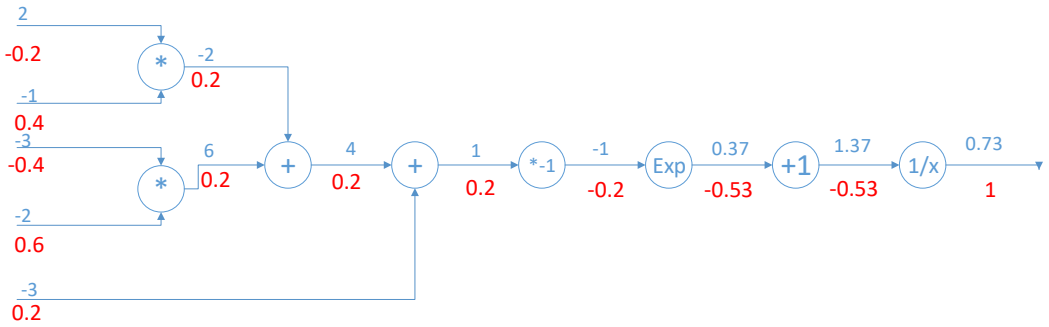
比如对于加法门的 3 来说，其梯度是负的，所以这个数如果变小的话可以使得最后的梯度增加。又比如说-4 的输入，如果让其沿着其梯度方向（正），增加，就会导致整体的输出增加。但必须注意，由于梯度的定义，这种沿着梯度的增加只在比较小的范围内有效，因为梯度本身就是会随着变量的变化变换的。

3. Sigmoid 例子

$$f(w,x)=\frac{1}{1+e^{-(w_0x_0+w_1x_2+w_2)}}$$

这个函数作为一个激活函数是经常用的，除了简单的加法和乘法之外，还有倒数，指数等，我们把这样的一个函数的前向和后向传播算法画出来看一下：

假设输入是$[w_0,x_0,w_1,x_2,w_2]=[2,-1,-3,-2,-3]$



这样写出来的前向传播和后向传播的可视化图是这样的，这种算法应当是要掌握的。算的时候只要牢记链式法则就 ok 了。

在这个函数中，我们是对 w 和 x 的点积进行操作，我们现在把这个看成一个向量，并且把

这个函数记作：

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

求导：

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

我们发现这个是可以化简的，由于指数函数的特殊求导法则，使得这个形式很美观整齐。
我们把中操作封装起来，在实际的操作中这也是非常有用的。
测试代码：

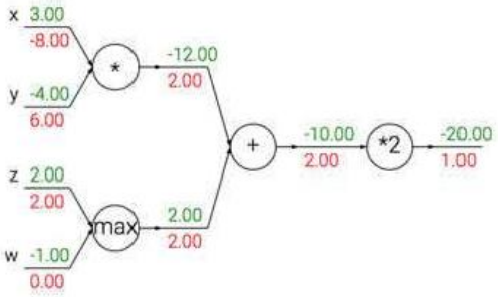
```
import math
w=[2,-3,-3]
x=[-1,-2]
dot=w[0]*x[1]+w[1]*x[1]+w[2]
f=1.0/(1+math.exp(-dot))

ddot=(1-f)*f     #函数求导
#回传
dx=[w[0]*ddot,w[1]*ddot]
dw=[x[0]*ddot,x[1]*ddot,1*ddot]
print(dx)
print(dw)

```

[0.3932238664829637, -0.5898357997244456]

[-0.19661193324148185, -0.3932238664829637, 0.19661193324148185]

输出和我在可视化图中计算的略有不同，主要原因是对一些指数做了近似。
我们也可以看到，许多时候在反向传播的时候会用到前向传播所计算的值，所以在编程的时候最好对这些中间变量进行缓存，这样在反向传播的时候就可以用到了。

4. 回传流中的模式



神经网络中最常用的三种门单元分别是加，乘法，最大值，看上面这个例子。
加法操作将梯度相等得分发给它的输入，取最大值操作将梯度路由给更大的输入，乘法门拿取输入激活数据，对他们进行交换，然后乘以梯度，也就是一个变量的值实际上是影响的另一个变量的梯度。
从上可知：
加法门单元：把输入的梯度相等得分发给它所有的输入，这一行为与输入值在前向传播的值无关，因为加法操作所有的梯度都是 1，所以所有输入的梯度就等于输出的梯度。

取最大值门单元：对梯度做路由，和加法门不同，取最大值门将梯度传给其中一个输入，这个输入应该是前向传播中最大的那个输入，其余的是 0。
乘法门单元：这个相对不容易解释，他的局部梯度就是输入值，但是是交换之后的，对于乘法来说，一个变量的梯度实际上是收到另外一个变量的影响的。
非直观影响：注意一种特殊情况，如果乘法门单元的其中一个输入非常小，另一个非常大，那么乘法门的操作将不会那么只管：它将比较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，$f = w^T x_i$ 说明输入数据的大小对权重梯度的大小有影响，例如我们如果把所有的输入乘以 1000，那么对权重梯度的影响将对增加 1000 倍，这样就必须通过降低学习率来弥补，这就是为什么数据预处理的关系重大，将直接影响到学习率。

5. 用向量化操作来计算梯度

上述考虑的都是单个变量的情况，但是所有的概念都是适用于矩阵操作的，只是在操作的时候要注意关注维度和转置操作。

```
41 import numpy as np
42 W=np.random.randn(5,10)
43 X=np.random.randn(10,3)
44 D=W.dot(X)
45
46 #假设我们得到了D的梯度
47 dD=np.random.randn(*D.shape)
48 dW=dD.dot(X.T)
49 dX=W.T.dot(dD)
50 print(dW)
51 print(dX)
```

不用去记最后的表达，因为可以通过维度匹配推导出来，要记住的是，python 或者说 numpy 里面的 dot 就是矩阵乘法。

二、Softmax 和 SVM 分类器的损失函数对比

1. SVM 线性分类用的损失函数的形式是：

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

这个是不带正则化惩罚的形式，对于一个参数 W 来说，算的的 S 是分值，其中有个是正确分类，有的是不正确的分类，这个形式对分类结果的评分是做一个评价，评分越低说明效果越好。

举一个例子说明是如何计算的，比如我们有三个分类，并且得到了评分 $s = [13, -7, 11]$，其中第一个是正确分类，即 $y_i = 0$，同时把 Δ 设置为 10，上面的公式就是将不正确的分类加起来，所以我们得到正确的部分。

$$L_i = \max(0, -7 - 13 + 10) + \max(0, 11 - 13 + 10) = 8$$

我们可以看到第一部分算出来的是 0，第二部分算出来的是 8，通俗的来讲，这个损失函数干了这么一件事：我们并不是只关心正确的分类的评分，而且要求不正确的评分和正确评分的差距足够大（这样显然是可以提高分类器性能的，这里是 10），如果没有大于这个值，我们就要计算不正确分类的损失，所有加起来当作当前参数评价的损失函数。
这个公式还有一种写法：这是用线性分类器的情况。

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

2. 但是，上面的损失函数有一个问题，假设我们有一个数据集和一个权重集能够正确的分类每个数据，即所有的边界条件都满足，对所有的 i 来说损失函数都是 0.问题是这样的 W 并不是 w 欲以的，可能有很多相似的 W 都能正确的分类所有的数据，举个例子，如果 W 可以分类正确所有数据，那么对其进行倍乘 λW 也可以分类所有数据，我们希望对 W 添加一些偏好，对其他权重则不添加，来消除这种模糊性，这一点是能够实现的，方法就是向损失函数添加一个正则化惩罚（regularization penalty）$R(W)$，最常用的就是 L2 范数，即：

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

这样就能给出完整的多类 SVM 损失函数了，由两部分构成：数据损失（data loss），即所有样本的平均损失，以及正则化损失：

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

如要展开也是可以的：

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

这样处理最好的一个性质就是可以对大数值权重进行惩罚提高其泛化能力，意味着没有那个维度能够独自对整体分值有过大影响，比如：对于数据 $x = [1,1,1,1]$ 来说，有两个权重：

$w_1 = [1,0,0,0]$，$w_2 = [0.25,0.25,0.25,0.25]$，两者的数据损失为：$w_1^T x = w_2^T x = 1$，但是正则化惩罚不一样，$w_1$ 为 1，而 w_2 是 0.25,因此，根据正则项惩罚来看，w_2 的性能要更好，从直观上来看，w_2 的权重更分散，这样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几个维度，这一效果将提高模型的泛化能力从而避免过拟合。

3. 另外：上面对于超参数 Δ 的设置一笔带过了，那么究竟该设置为什么值呢？需要通过交叉验证来求么？现在看来，绝大多数情况下设置为 1 都是安全的，Δ 和 λ 是两个不同的超参数，但是它们一起控制同一个权衡，即损失函数中数据损失和正则化损失之间的权衡，权重 W 的大小对分类器评分有直接影响:缩小 W 分类之间的差距也变小，因此不同分类分值之间的边界具体值从某些角度看起来是没有意义的，因为这些分值主要是通过 W 来控制的，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）。

4. softmax 分类器。

SVM 和 softmax 是最常用的两个线性分类器，softmax 的损失函数与 SVM 的损失函数不同，softmax 分类器可以理解为逻辑回归分类器在面对多分类问题的一般化归纳，softmax 的输出是一个归一化的概率，这一点在后面讨论，在 softmax 分类器中，函数映射 $f(x_i, W) = Wx_i$ 不变，只是将这些评分值是做每一个分类的未归一化的对数概率，并且将折叶损失替换为交叉熵损失（cross-entropy loss）：

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

$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$ 这个函数被称作 softmax 函数，输入是一个向量，输出也是一个向量，且是压缩过的，且和为 1。以归一化概率的角度来理解这个公式是比较简单的。至于加正则化惩罚，则和 SVM 是一样的。

上面的式子还有一个小毛病，就是求指数之后可能会变得很大，因为指数的增长率是很快的，所以一般处理的时候会加上一个偏置，给分子和分母都乘以常数 C，并变换到求和之中，这样就能在数学上得到一个等价的公式：

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

C 的值按理说可以随意选择，我们通常把 C 设置为：$\log C = -\max f_j$。这是一个技巧，简单来说就是把数据平移到 0 以下（最大值为 0），示例代码：

```
7
8 import numpy as np
9 f=np.array([456,456,789])
10 p=np.exp(f)/np.sum(np.exp(f))
11 print(p)
12
13
14
15 ff=np.array([123,456,789])
16 ff-=np.max(f)
17 pp=np.exp(ff)/np.sum(np.exp(ff))
18 print(pp)
```

另外：softmax 还有一个优势在于其求导出来的结果非常简单：这里我还没有完全看懂，总之在反向传播的时候，导数很好求。

5. 总结:

在实际使用中，SVM 和 softmax 经常是相似的，两种分类器的表现性能也差不多，SVM 可以说更加“局部目标化”，既可以看作一个特性，也是一个劣势，比如对评分为[10,-2,3]（第一个是正确分类），那么一个 SVM（Δ=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,就会认为损失值是 0.而并不关注具体分数细节（[10,9,9]与[10,-100,-100]并没有什么区别），对于 softmax 来说，情况就会不同，前者计算出来的损失值是远高于后者的，正确分类总能得到更高的可能性，损失值也总是能够更小一些，也就是说，SVM 不会关注过多的细节，举例来说，一个汽车的分类器应该把主要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来进行区分，因为区分青蛙得到的评分已经够低了。

三、神经网络。

1. 快速简介：

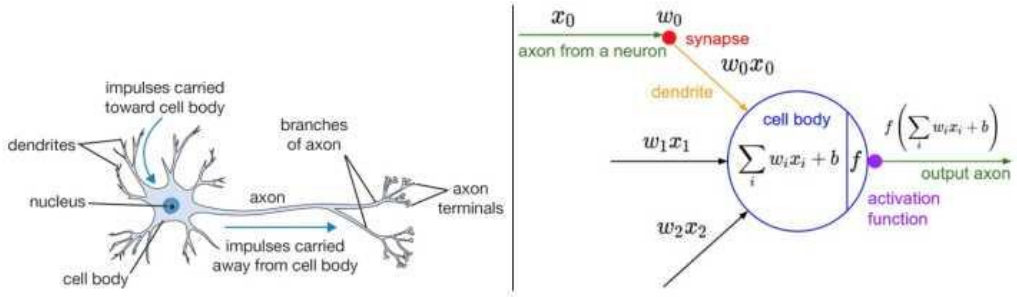
在线性分类中，我们是使用 $S = Wx$ 来计算不同视觉类别的评分，比如我们使用的 CIFAR-10 数据中，x 是一个[3072*1]的向量，W 是[10*3072]的矩阵，所以输出是一个[10*1]的评分向量。

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[100*3072]的矩阵，作用是将图像转换为一个 100 维的过度向量，

函数 max(0,--)是非线性函数，会作用到每个元素，W_2 是一个[10*100]的矩阵，最后会得到一个[10*1]的评分，非线性是非常重要的，这是改变的关键点，参数W_1, W_2 可以通过梯度下降来学习到，梯度可以由反向传播中通过链式法则来求导得出。

一个三层的神经网络可以类比看作： $S = W_3 \max(0, W_2 \max(0, W_1 x))$

2. 单个神经元建模



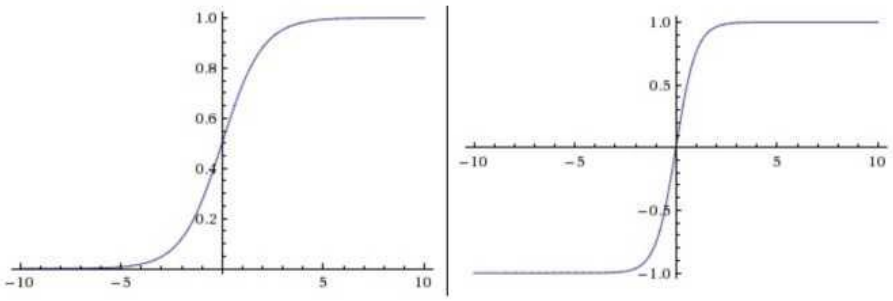
这个图很能说明问题，每个神经元都对它的输入和权重进行点积，然后加上偏差，最后用激活函数（就是前面说的那个非线性函数），本例中使用的是 sigmoid 函数 $\sigma(x) = 1/(1+e^{-x})$

```
class Neuron(object):
  # ...
  def forward(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
```

这是一个典型的神经元前向传播的代码。

这样来看，就像在线性分类器中那样，神经元有能力喜欢或者不喜欢输入空间中的某些区域（通过激活函数），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。

常用激活函数：



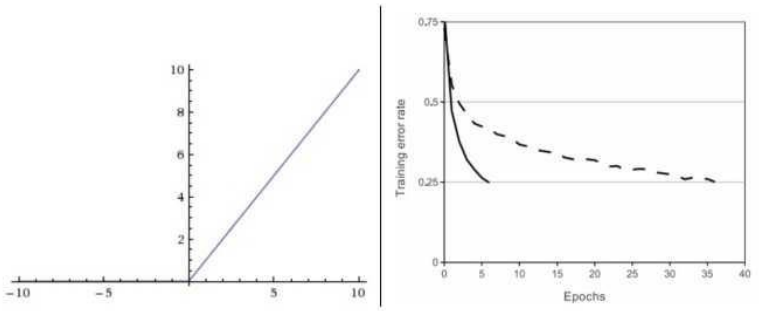
左边是 Sigmoid 非线性函数，将实数压缩到[0,1]之间。右边是 tanh 函数，将实数压缩到[-1,1]。历史上来说 Sigmoid 是用的比较多的函数，是因为对神经元的激活频率有良好的解释：从完全不激活（0）到求和后的最大频率处的完全饱和的激活（1）。但是现在很少用了，主要是由于下面两个缺点：

① 函数饱和使得梯度小时，在接近 0 或者 1 的时候梯度几乎降为 0，在反向传播的时候，这个局部梯度会与整个损失函数关于该门单元的输出的梯度相乘，因此如果局部梯度非常小，相乘的结果也会接近 0，这就会有效的“杀死”梯度，几乎没有信号可以通过神经元传到权重再到数据了，另外，为了防止饱和，必须对权重矩阵初始化特别留意，一旦初始化权重过大，大多数神经元都会饱和，就会导致网络几乎不在学习了。

② 另外，Sigmoid 的输出不是零中心的，这个结果不是我们想要的，因为这就会把原本零中心的数据转换成不是零中心的，这一情况将会影响梯度下降的操作，如果输入神经元的数据总是整数，那么关于 w 的梯度在反相传播的过程中将全部是正或者负（与 f 有关），这会导致梯度下降权重更新的时候出现 z 字型下降。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。（没有完全理解这块说的）。

Tanh 函数是一个零中心的，也存在饱和问题，在实际使用中比 sigmoid 要受欢迎一些。具体形式为：

$\tanh(x) = 2/sigma(2x) - 1$ 就是一个放大的 sigmoid 神经元。



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，当大于 1 时斜率为 1，研究表明，使用 ReLU 要比 tanh 收敛快 6 倍。

ReLU 的函数形式为： $f(x) = \max(0, x)$，形式很简单，就是取了一个阈值。近些年很流行用这个。

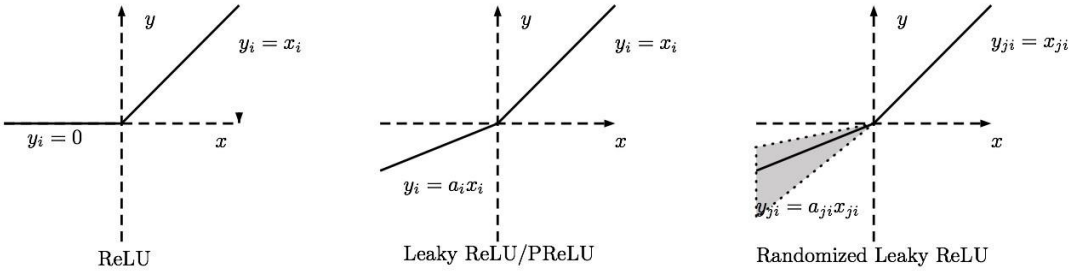
优点：随机梯度下降收敛速度快。非饱和和非线性导致的。

优点：计算快，比起上面说的两种（有指数运算），这个知识简单的阈值计算。

优点：训练的是每次都值激活一部分神经元,一些神经元输出为 0，这样就造成了网络的稀疏性，减少了参数的相互依存关系，缓解了过拟合问题（这个问题还有生物学解释：神经学家发现人的大脑在同一时刻也之激活 1-4%的神经元）。

缺点：训练的时候，ReLU 可能会比较脆弱的死掉，当一个很大的梯度流过 Relu 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。（这个也没有很理解），解决这个问题的方法之一是降低学习率。

关于 ReLU 还有一些改进版本：



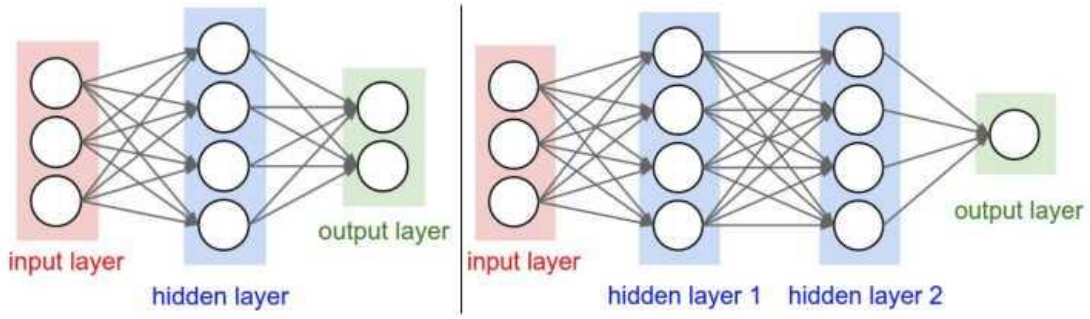
Maxout，这是最新提出的一种神经元，对局权重和数据的内积采用这样一种形式：

$\max(W_1 x+b_1, W_2 x+b_2)$，这实际上是对 ReLU 和 leaky-ReLU 的一般化归纳。这样虽然客服了 ReLu 的缺点，但是这使得每个神经元的参数倍增，导致整体参数的数量倍增。

值得注意的是，一般不在同一网络中混用不同类型的神经元参数，虽然这样做没有什么根本性的理由。

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。

3. 神经元层结构



先看这样一个图，我们一般不把输入层也算在内，比如左边是 2 层，右边是三层。层与层之间的神经元是全连接的，同一层的神经元不连接。

另外：输出层是没有激活函数的。

网络尺寸：衡量网络尺寸主要有两种：神经元的个数和参数的个数。（假设上面的输入都是 3*1）

- 第一个网络：4+2 个神经元。共有[3*4]+[4*2]=20 个参数，并且还有 4+2 个偏置，共 26 个可学习的参数，
- 第二个网络：4+4+1=0 个神经元，共有[3*4]+[4*4]+[4*1]=32 个权重，还有 9 个偏置，共 41 个可学习的参数。

现代神经网络可能包含有一亿个参数，可由 10-20 层构成（这就是深度学习了）

前向传播举例：用第二个网络举例：一个层的所有强度可以存放在一个单独的矩阵当中，每个神经元的权重在 W 的一行中，用矩阵乘法就可以计算该层中所有神经元的激活数据。

示例程序如下：

```
# 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)
```

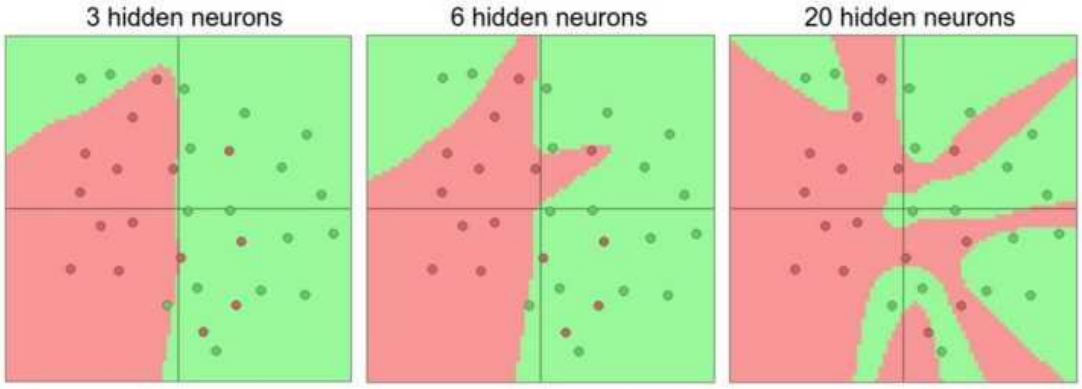
```
out = np.dot(W3, h2) + b3 # 神经元输出(1x1)
```

表达能力：

已经证明：至少含有一个隐含层的神经网络可以近似任何连续函数。

对于神经网络来说，虽然理论上一层就可以近似任意函数，但是实际实践中的效果来看层数的增加可以带来性能的提升，实践中三层的神经网络比二层的要好，然而继续加深则很少有太大的帮助，卷积神经网络确完全不同，卷积神经网络的深度是一个极端重要的因素，对于该现象的一个直观解释：图像拥有层次化结构（比如脸是由眼睛等组成，而眼睛又由边缘组成）所以多层处理对于这种数据有直观意义。

设置层的数量和尺寸：要不要用隐含层，每一层用多少神经元？这都是在实际问题中需要关注的问题。

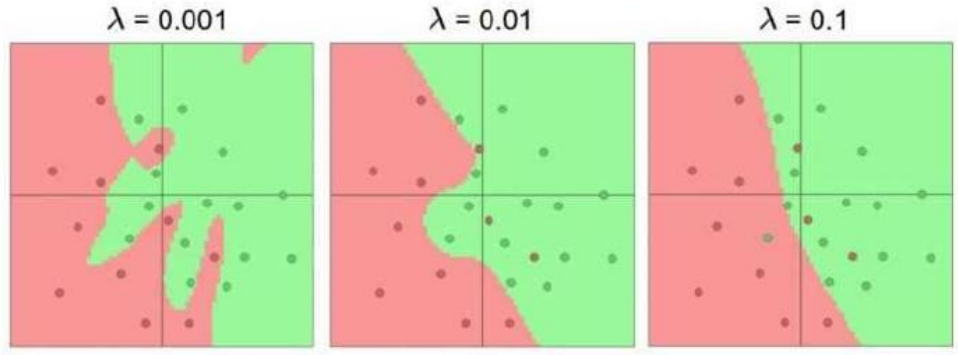


对一个平面的二分类问题，上图显示了一个隐含层不同数量的神经元的分类效果。可以看出随着神经元数量的增加模型会更加复杂可以处理更复杂的数据，但是缺点是造成了过拟合，所谓**过拟合**是网络对数据中的噪声有很强的你和能力，而没有注重数据潜在的基本关系，比如最后一个含有 20 个隐含层的网络，看上去分类了所有测试数据，但是这些数据有的可能是噪声引起的，这样做的代价是把决策边界变成了许多不相连的红色区域，而三个神经元的表达能力知识用比较宽泛的方式去分类数据，并把绿色区域中的红色点看作是噪声，在实际中，这样的模型在测试数据中反而可以获得更好的泛化能力。

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。

不要减少神经元数目的主要原因在于小网络更难用梯度下降法等局部方法来训练，虽然说小型网络的损失函数局部极小值更少，更容易收敛到这些局部极小值，但是这些最小值一般性能都很差，损失值很高，而相反的是，大网络拥有更多的局部极小值，但就实际损失来看，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。

正则化强度是控制神经网络过拟合的好方法。



这个图显示的是有 20 个隐藏神经元时，随着正则化强度增加，决策面变得更加平滑，所以需要记住的是：

不应该因为害怕过拟合而使用小网络，相反，应该尽可能的使用大网络，然后使用正则化技巧来控制过拟合。

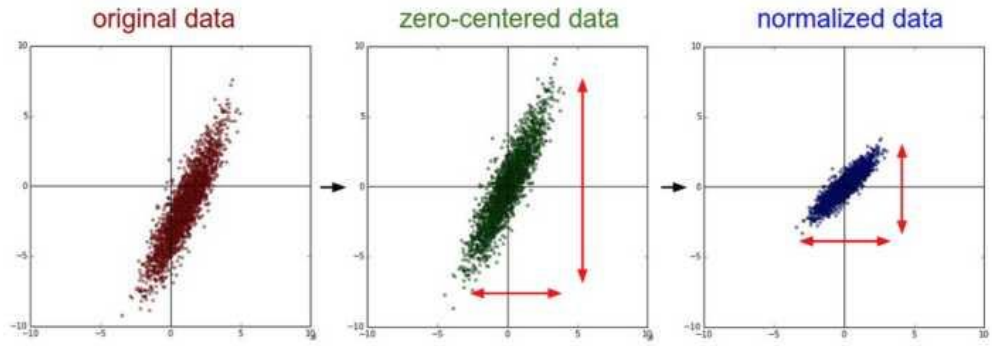
4. 设置数据和模型：

① 数据预处理

数据预处理我们有三个常用符号，数据矩阵 X，X 的大小是[N*D],其中 N 是数据数量，D 是数据维数。

均值减法：即在一个维度上的每个数都减去这个维度的均值，也称中心化

归一化：有两种方法可以实现所有维度的归一化，使其数值范围内都近似相等，第一种是现对数据做零中心化，然后每个维度都除以其标准差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值和最小是是 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据范围时才有意义，图像处理中像素的范围几乎都是一致的（0-255）这个预处理步骤不是很有必要，但是需要注意的是预处理操作的重要性几乎等同于算法本身。



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：这是另一种预处理方式，现对数据进行零中心化，然后计算协方差矩阵，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
```

协方差矩阵的第（i，j）个数据表示的是第 i 和和第 j 和维度的协方差，对角线是方差，而且协方差矩阵是对称的和半正定的。因为降维还需要求出特征值和特征向量来，所以一般对其进行 SVD 分解。

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

U 的列是特征向量，S 是装有奇异值的一维数组，S 中的元素是特征值的平方，为了去除数据相关性，把已经零中心化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
```

U 是标准正交向量的集合（列均模为 1 且列之间相互正交），所以这个操作可以看作是对数据 X 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]
```

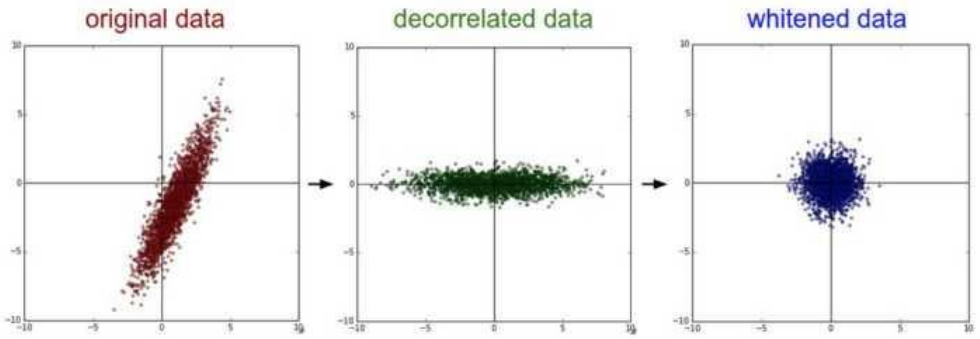
降维的优点是显而易见的，保留了方差较大方向的 100 维（程序为例）数据，基本可以包含了原始数据的大部分有用信息，这样可以节省时间和存储空间而并不会损失过多性能。

白化：还有一个经常看到的是白化，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。几何解释是：如果数据服从多变量的高斯分布，那么经过白化之后，会是一个均值为 0，且协方差相等的矩阵。代码：

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

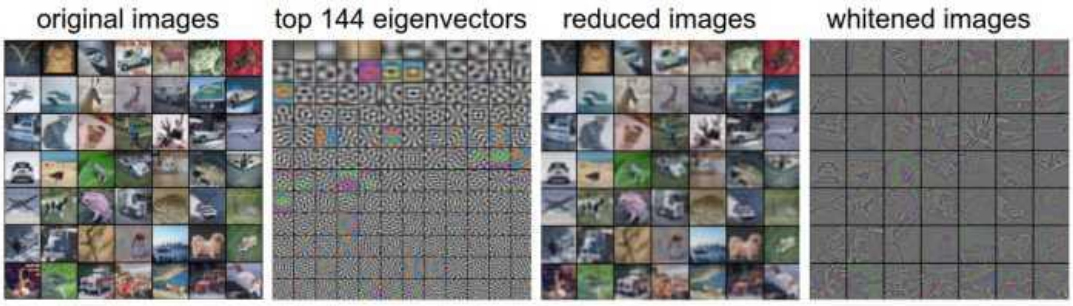
注意这里的输入是特征基准上的数据，1e-5 的加入是为了防止分母为 0，这个变换的一个缺陷是在变换的过程中可以夸大了数据中的噪声，这是一位内将所有的维度都拉到相同的数值范围，这些维度中也包含了哪些拥有绩效方差而大都是噪声的维度，实际的操作中可以用更强的平滑来解决（例如采用比 1e-5 更大的值）。

（这段的理解可以这么来说，方差较小的维度包含了较少的信息，理应被忽略，但是白化操作对这些维度和方差较大的维度一视同仁都进行了归一化处理，这就相当于这些方差较小的维度（可能是噪声）就被放大了）。



通过这个图可以直观一些。

中间是 PCA 操作的数据，变换到了协方差矩阵的基准轴上，进行了解相关操作，最右边是进行白化操作，对各个方向上进行了压缩。



这个图就更好了，在 CIFAR-10 的数据集上进行了一些可是话操作，我们可以看到，经过 PCA 降维之后的数据（这里用了 144 维）还是保留了原图大部分的信息，而白化操作把 144 维在各个维度上进行压缩，较低的频率（颜色，平滑区域）可以忽略了，较高的频率（边缘）就被夸大了。

实际操作的时候，卷积神经网络就不会采用 pca 降维这些类似的操作了（因为卷积本身就会降维，我是这样理解的），然而对像素进行中心化和归一化还是很常见的。

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。

② 权重初始化

开始训练网络之前，需要初始化网络参数。

首先，全零初始化是不行的，一开始就全是 0，那么每个神经元就算出同样的输出，反向传播时就会得到童谣的梯度，从而进行参数更新时就会进行同样的参数更新，神经元之间就是去了不对称性。

小随机数初始化：是一个不错的选择，可以生成高斯分布的也可以是均匀分布的，从实践结果上来看，对算法的结果影响都极小。但是并不是小数值就会取得好的结果，例如，一个神经网络的层中的权重值很小，那么在反向传播时就会计算出非常小的梯度（梯度和权重是成比例的），这就会很大程度上衰减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：上面的做发存在一个问题，随着数据量的增长，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，这样神经元的输出的方差就归一化到 1 了，也就是说，建议神经元的权重向量初始化到 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的神经元起始时有近似同样的输出分布，实践证明这样也可以提高收敛的速度。

推导过程如下：假设权重 w, x 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。检查 S 的方差：

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

这个推导第二步我没有完全看懂，第三部是我们假设输入和权重的均值都是 0，但这不是一般化的情况，ReLU 单元中均值就是正的，最后一步我们假设所有的 $w_i x_i$ 都具有相同的分布，从推到中可以看出，如果想要 s 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 来说，$Var(aX) = a^2 Var(X)$，这说明可以基于一个标准高斯分布，然后乘以 $\sqrt{1/n}$ 使其方差为：1/n，于是得出：w = np.random.randn(n) / sqrt(n)。

最新的论文指出（何凯明大神），针对 ReLU 神经元，神经元的方差应该是 2/n。代码是：np.random.randn(n) * sqrt(2.0/n)，这个形式是神经元网络算法中使用 Relu 神经元时当前的最佳推荐。

稀疏初始化：另一个处理非标定方差的方法是将所有的权重该矩阵置为 0 之后，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（权重数值由一个小的高斯分布生成），一个比较典型的数目是 10 个。

偏置初始化：一般而言初始化为 0 就可以了，有些人也喜欢用 0.01 这样的小数值，但是也并不是总能提高算法行性能，所以通常情况下还是用 0 来做。

③ 批量归一化

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！

这段我是直接复制过来的。没有很懂。

④ 正则化

有很多的方法可以控制神经网络的容量来防止其过拟合。

L2 正则化：最常用的方法可以通过乘法目标中的所有参数的平方来实现，向目标函数

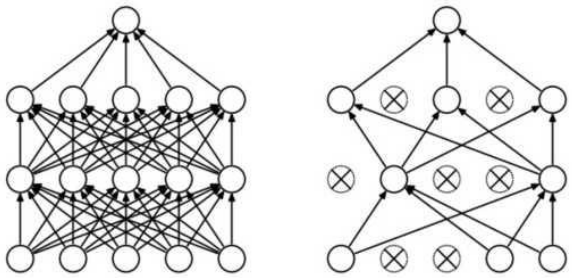
中加入一个$\frac{1}{2}\lambda w^2$来实现，1/2 的作用是关于 w 求导之后为：λw而不是多一个 2，和前面说到的正则化是一样的意思，倾向于对大权重进行乘法而使得权重更加分散平均，这样旧时的网络更倾向于关注使用所有特征而不是严重依赖某小部分的特征。另外，使用 L2 正则化意味着所有的权重都以 w += -lambda * W 向着 0 线性下降的。

L1 正则化：这是另一个常用的正则化方法，队医每一个 w 我们都向目标函数加入一个$\lambda|w|$，也可对 L1 和 L2 进行组合：$\lambda_1|w|+\lambda_2 w^2$，这也被称作 Elastic net regularizaton。

L1 有一个有趣的性质：会让权重向量在最优化过程中变得稀疏（即非常接近 0）。也就是说，使用 L1 正则化的神经元到最后使用的是他们最重要的输入数据的稀疏子集，对于噪声几乎是不变的了，相较于 L1,L2 正则化的权重向量大多是分散的小数字，在实践中，除非某种特别关注的特征，一般 L2 要比 L1 的效果要好。

最大范式约束（max norm constraints）：这种形式是给神经元中的权重向量设置上限，并使用投影梯度下降来确保这一约束，这种约束的有点在于即使学习率设置过高的时候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。也有研究者发文称这种正则化的效果更好。

随机失活（dropout）：这也是一个简单而又有效的方法，与前面的几种方法互为补充，在训练的时候随机失活的方式是让神经元以超参数 P 的概率被激活或者被设置为 0。



上图来源与论文 Dropout: A Simple Way to Prevent Neural Networks from Overfitting,展示了其核心思路，可以认为这种方法是对完整网络随机抽取一些子集，每次基于输入数据只更新子网络的参数（然而，数量巨大的子网络并不是相互独立的，因为共享参数）测试过程中如果不使用随机失活，可以理解为对数量巨大的子网络做了模型集成（model ensemble），以此来计算出一个平均预测。

一个三层随机网络的普通版随机失活可以用下面代码实现：

可以看出，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（利用随机数做掩膜加上去让神经元的输出为 0），反向传播保持不变，但是肯定要把两个遮罩加上去了。

Predict 函数则不需要进行随机失活了，但是这里对两个隐层的输出都要乘以 p 以调整其数值范围，这一点非常重要，因为我们希望神经网络的输出与训练的时候的输出是一致的，以 P=0.5 为例，测试神经元必须把他们的输出减半，因为训练的时候输出只有一半（因为有一般（概率）的神经元在训练传输的时候失活了），那么在测试的时候我们对每一层都要乘上这个概率以保证训练和测试的结果数值范围是一致的。

```
import numpy as np

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):
    #三层神经网络的前向传播
    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(H2,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
    #参数更新   略

def predict(X):
    H1=np.maximum(0,np.dot(W1,X)+b1)*p
    H2=np.maximum(0,np.dot(W2,H1)+b2)*p
    out=np.dot(W3,H2)+b3
```

这样操作不好的性质是我们必须在测试的时候对激活数据进行数值范围调整，既然测试性能如此关键，实际上更倾向于使用反向随机失活（inverted dropout）,在训练的时候对数值范围进行调整，这样做还有的一个好处就是无论是否使用随机失活，预测方法的代码都是一样的，不需要改变，改正后的代码如下：

```
import numpy as np

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):
    #三层神经网络的前向传播
    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
    H1*=U1    #drop
    H2=np.maximum(0,np.dot(H2,H1)+b2)
    U2=(np.random.rand(*H2.shape)<p)/p  #第二个随机失活遮罩，注意除以p进行数值范围调整
    H2*=U2   #drop
    out=np.dot(W3,H2)+b3

    #反向传播计算梯度  略
    #参数更新   略

def predict(X):
    H1=np.maximum(0,np.dot(W1,X)+b1)
    H2=np.maximum(0,np.dot(W2,H1)+b2)
    out=np.dot(W3,H2)+b3
```

随机失活发布之后，很快有大量的研究为什么它的时间效果和好，以及其和正则化方法之间的关系，有兴趣可以参考其他一些相关论文。

偏置正则化：对偏置参数（b）的正则化并不常见，因为其在矩阵乘法中并不和数据产生互动，所以并不需要控制其在数据维度上的效果，实际使用中，对偏置进行正则化也很少会使模型的性能下降，所以还是有人这么做的

每层正则化：对于不同的层进行不同强度的正则化（λ不同），也算一个方法，不过这么做的人也很少。

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曾后面使用随机失活也比较常见，P 值一般默认设置为 0.5，也可在验

证集上调参。

5. 损失函数。

在上面重点讨论了损失函数的正则化部分，可以看作是对模型复杂程度的某种乘法，损失函数的第二个部分是数据损失，这是一个有监督学习的问题，用于衡量分类算法的预测结果和真是标签的一致性，数据损失对所有样本的数据损失求一个平均值，也就是说：

$L=\frac{1}{N}\sum_i L_i$，N 是训练集中样本的数量。一个常见的损失函数的形式是：

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

这是 SVM 的损失函数的折页损失，有学者提除平方折页损失效果更好，另外一种常用的分类器是 softmax 分类器，它使用交叉熵损失：

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

。

还存着着几个问题：

类别数目巨大：标签量非常庞大时，直接用输出对应标签就不是一个好办法了，这时候用分层 soft max（hierarchical softmax）了，分层 softmax 是将标签分解成一个树，每个标签都表示成树上的一个节点，那么这样能表示的类别就可以指数增长了，树的每个节点都训练成一个 soft max 分类器在左枝和右枝之间做决策，树的结构对于算法的最终结果影响巨大，而且一般需要具体问题具体分析。

属性分类（attribute）：上面两个损失公式的前提都是假设每个样本都只有一个正确的标签，但是如果标签是一个二值向量，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？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器，列入，针对每个分类的二分类器都会采用下面的公式：

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

上式中，求和是对所有分类 j, y_{ij} 的值是 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，这样，当该类被正确分类并展示的时候，分值向量 f_j 为正，否则为负，当一个正样本的分小于 1，或者一个负样本的得分大于-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类（0，1），其中对于分类 1 的概率是：

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

对于分类 0 的概率是：

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

$\sigma(w^T x+b) > 0.5$ 或者 $w^T x+b > 0$ 时，那么样本就被正确分类（y=1），然后损失函数最大化这个对数似然函数，问题简化为：

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

上式中，假如标签 y_{ij} 非 0 即 1，上面的函数看起来很复杂，f 梯度计算却比较简单：

$$\frac{\partial L_i}{\partial f_j} = y_{ij} - \sigma(f_j)$$

回归问题：回归问题一般是预测实数的值的问题，比如预测股市或者房价，预测图片中某个东西的长度等，对于这个问题，通常是计算预测值和真实值之间的损失，然后用 L2 或者 L1 范式度量差异，对于某个样本来说，L2 范式计算如下：

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

之所以要进行平方，是因为梯度计算起来更加简单，平方也是一个单调函数，可以不用改变最优参数，L1 范式则是要把每个维度的绝对值加起来：

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

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} 表示预测值和真实值之间的差异，关于该维度的梯度就很容易求得：

L2: δ_{ij} L1: $sign(\delta_{ij})$,也就是说，评分值的梯度要么与误差的差值成比例，要不就是从插值中继承符号（sign）。

需要注意的是：L2 损失比起较为稳定的 softmax 损失来，优化上要困难很多，直观而言，他需要网络具有一个特别的性质，即对于每个输入都要输出一个确切的正确值，而在 softmax 就不是这样，因为每个评分的准确值并不是很重要(归一化指数概率了)。还有，L2 损失的鲁棒性不好，因为异常值会导致很大的梯度，所以在面对一个回归问题时，首先考虑将输出二值化是否真的不够用，比如对一个产品进行星级预测，分为 1-5 星，用 5 个独立的分类起来打分的效果比使用一个回归损失要好很多，如果确信分类不适用，那么就使用 L2 损失吧，但是要注意：L2 损失非常脆弱，在网络中使用 dropout（随机失活）并不是好主意（尤其在 L2 损失的上一层）。

当面对一个回归任务，首先考虑是不是必须使用回归，一般而言，尽量把输出转换成二分类，然后进行分类，从而变成一个分类问题。

结构化预测（structure prediction）：结构话损失指标签可以是任意的结构，图标，树都可以，这种问题就比较复杂了，这里不提了。

6. 梯度检查。

理论上进行梯度检查很简单，就是把解析梯度和数值计算得到的梯度进行比较，然而从实际操作层面的角度来说，这个过程复杂且容易出错下面提供一些技巧：

使用中心化公式：

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

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

在进行梯度检查每个维度的时候，会要求计算两次损失函数，所以计算资源的消耗也是两倍，但是近似的梯度值会准确很多，要理解这一点，需要对 f(x+h)和 f(x-h)使用泰勒级数展开，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)。

使用相对误差来比较：比较数值梯度和解析梯度的时候，用绝对差值么？这样得到的标准通常是不统一的，在比较相似度上，经常采用的一种方式是归一化或者相对化，我们这里使用相对值：

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

这样的结果就是很好的。

在实践中：相对误差

>1 $1e^{-2}$:通常意味着梯度可能出错。

$1e^{-4} - 1e^{-2}$：这个值勉强凑活。

$1e^{-4}$：这个值对于有不可导点的函数还是差不多的，如果目标函数中没有使用 kink(使用 tanh 和 softmax)，那么相对误差还是有点高。

$< 1e^{-7}$：这个数量级效果就非常好了。

随着网络的加深，相对误差就会累积越来越高，所以如果是十层的话，那么 $1e^{-2}$ 可能也不是什么坏结果，因为误差一致在积累（回忆反向传播那个图）。所以这些是相对的，如果对于一个浅层的而且目标函数可导的函数，这个结果就不太现实了。

使用双精度：和使用单精度浮点数来进行梯度检查相比，会有效降低相对误差。

保持在浮点数的有效范围：建议通读《What Every Computer Scientist Should Konw About Floating-Point Artthmetic》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，在这个范围中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：越过不可导点的梯度检查会导致梯度检查不准确，要尽量避免这种情况，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。(这里没有完全懂)。

使用少量数据点：解决上面不可导点的方法是使用更少的数据点，因为含有不可导点的损失函数的数据越少，当然不可导点就越少，如果梯度检查对 2-3 个数据点都有效的话，那么基本上对整个批量的数据梯度检查也是没有问题的

谨慎设置步长：这里的步长是 h，理论上来说 h 是越小越好，但是实际操作的时候由于数值精度的因素，一般调整到$1e^{-4}$或者$1e^{-6}$这个数量级是可以的。

在操作的特性模式中梯度检查：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，即使在该点上检查成功了，也不能保证全局上梯度的实现都是正确的，为了安全起见，最好让网络学习一段时间，等到损失函数开始下降之后然后再进行梯度检查，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，从而掩盖了梯度不正确的事实。

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，需要注意的正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，这样就会掩盖数据损失梯度的不正确实现。因此，推荐关闭正则化对数据损失单独做梯度检查，然后对正则化做单独梯度检查，对正则化做单独检查有两种方式，一种可以是去掉数据损失部分，一种是增加正则化强度，保证其效果在梯度检查中是无法忽视的，这样不正确实现就会容易被观察到。

关闭随机失活和数据扩张：这些不确定效果的操作在数值梯度计算时会产生很大的误差，关闭这些操作不好的一点就是无法对它们进行梯度检查（比如随机失活的反向传播就会失效），因此一个更好的解决方案是在计算 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：实际中，梯度可能有上百万的参数，这种情况下，只检查其中一些维度然后假设其他维度是正确的。要注意：确认在所有不同的参数中都抽取一部分来检查梯度，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。

7. 合理性检查

在进行优化之前，最好进行一些合理性检查。

寻找特定情况的正确损失值：使用小参数进行初始化时，确保得到的损失值与期望是一致的，最好先单独检查数据损失，比如对于一个 CIDAR-10 的 soft max 分类器，一般期望它的初始损失值是 2.302，因为初始时预测每个分类的概率都是 0.1，然后 soft max 损失值正确分类的负对数概率为-ln(0.1)=2.302,类似的情况还有一些，所以如果没有看到这些损失值，那么初始化中就可能出现问题。

提高正则化强度导致损失值变大：显而易见不用解释，如果这个满足不了，肯定是有问题。

对小数据子集过拟合：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），确保能得到 0 的损失（这时候不加正则化），如果不能通过这么一个正常性检查，不然进行整个数据集的训练是没有意义的，但是要注意，对小数据集的过拟合并不代表万事大吉，依然可能有不正确的实现。

8. 检查学习过程

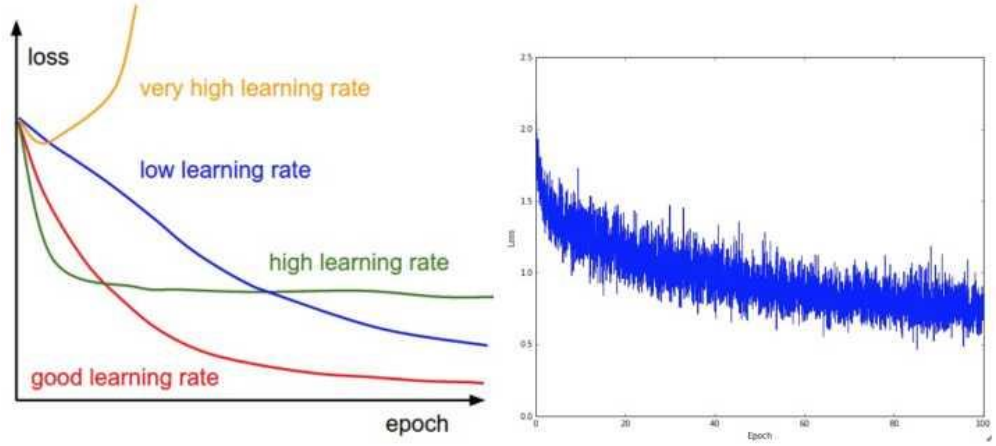
在训练神经网络的时候，应该跟踪多个重要数值，这些数值输出的图标是观察训练进程的一扇窗口，是直观理解不同的超参数设置效果的工具，从而得知如何修改超参数以获得更高效的学习过程，在下面的图表中，x 轴通常表示周期单位（epochs），一个周期意味着每个数据都被观察过了一遍。

损失函数：训练期间第一个要跟踪的数值就是损失函数，它在前向传播时对每个独立的批数据进行计算，下面的图显示了损失值随时间的变化：

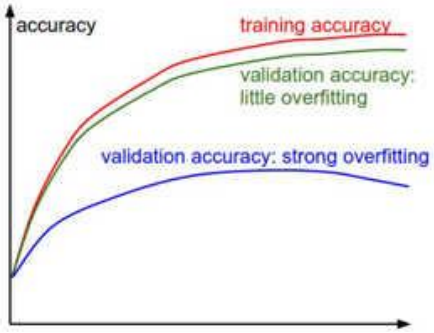
作图显示的是不同学习率的结果，过低的学习率倒是算法的改善是线性的，高一些的学习率会看起来几何指数下降，更高的学习率会让损失值更快的下降，但可能停留在一个不好的损

失值上（绿）。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网络，这个损失值曲线看起来比较合理，而且指出了批数据的数量可能有点小（因为损失值的噪声很大）。

损失值的震荡程度和批尺寸（batch size）有关，当批尺寸为 1 的时候，震荡会相对较大，当批尺寸就是整个数据集是震荡会最小，因为每个梯度更新都是单调得优化损失函数（除非学习率设置过高）。



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，这个能够展示模型过拟合的程度



在训练集和验证集准确率中间的空隙指明了模型过拟合的程度，图中蓝色的验证集曲线显示相较于训练集，验证集的准确里低了很多，这说明模型有很强的过拟合，遇到这种请款报告就应该增大正则化强度（更强的 L2 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基本一致，这说明模型容量还不够大：应该通过增加参数数量来让模型更大一些。

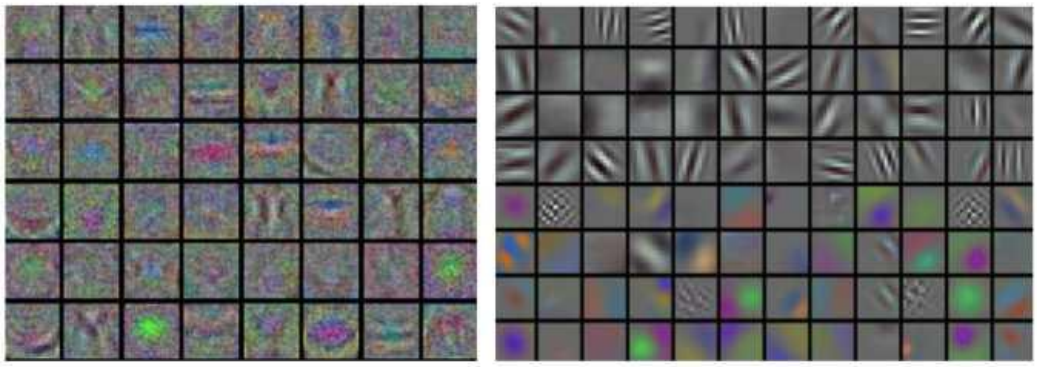
权重更新比例：最后一个应该跟踪的量是权重中跟新值的数量和全部值数量之间的比例，需要对每个参数集的更新比例进行单独计算和跟踪。一个经验性的结论这个比例应该在$1e^{-3}$左右，如果过小，可能是学习率有点小，过高则说明学习率太高了。下面看下具体例子。

```
# 假设参数向量为W，其梯度向量为dW
param_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
```

这个程序跟踪的是二范数的改变，这些矩阵通常是相关的，所以这样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。

第一层的可视化：最后，对于图像像素数据来说的话，把第一层特征可视化会有帮助：



比如上图，左图中充满了噪声，这暗示了网络中可能出现了问题：网络没有收敛，学习率设置不恰当，正则化权重偏低，都有可能出现。有图的特征就不错，平滑干净且种类繁多，说明训练过程中进行良好。

9. 参数更新

介绍一些在实践中常用的参数更新的方式。

普通更新：最简单的形式就是沿着负梯度方向改变参数，因为梯度指向的是上升方向

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

其中 learning_rate 是一个超参数，是一个固定量，只要学习率合适，就总能优化。

动量更新：这是另一种方法，这种方法在深度网络上几乎总能取得比较好的收敛速度。可以看成是从物理角度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（U=mgh），用随机数字初始化参数等于在某个位置给质点的初速度设为 0，最优化的过程就可以看作是模拟参数向量（即质点）在地形上滚动的过程。

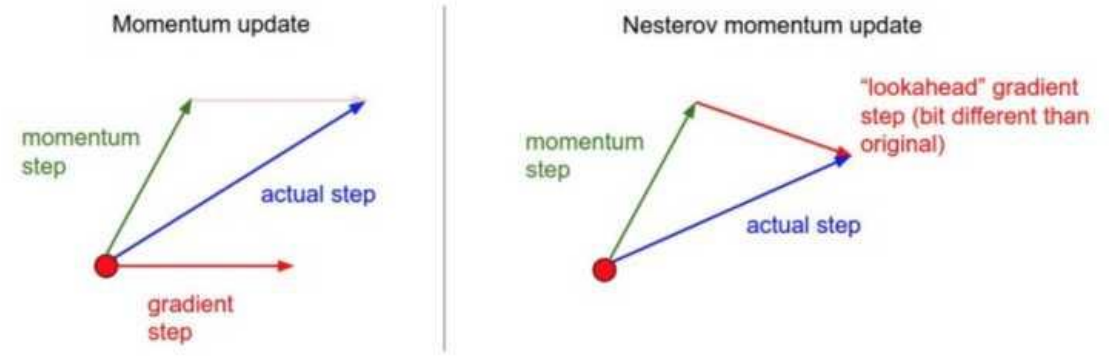
作用于质点的力与梯度的潜在能量（$F=-\nabla U$）有关，质点所受的力就是损失函数的负梯度，还有因为 $F=ma$，所以在这个观点下负梯度与质点的加速度是成比例的，这个和随机梯度下降（SDG）是不同的，在普通版本中，梯度直接影响位置，而这里物理观点建议知识影响速度，然后速度再影响位置。

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

这里引入了一个初始化为 0 的变量 v 和一个超参数 mu，这个变量 mu 可以理解为动量（一般设置为 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，不然质点就会在山底永远不停下来，通过交叉验证，这个参数通常设置为[0.5,0.9,0.95,0.99]中的一个比较好，和学习率随着时间退火（下面讨论）类似，动量随着时间变换的设置有时能略微改善最优化的结果，其中动量在学习过程的后阶段会上升。一个典型的设置就是一开始设置为 0.5，在后面的多个周期中慢慢升到 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。

Nestorov 动量更新：与普通动量更新有些许不同，最近也很流行，理论上在凸函数上能得到

更好的收敛，在实践中也确实比普通版的动量表现要好一些。
其核心思路就是，当参数来到 x 位置时，观察上面的动量更新的公式可以发现，动量部分（去掉第二部分即梯度部分），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 看作是“向前看”，这个点就在我们一会要停止的点的附近(就差了一个负梯度*学习率)，因此计算 x+mu*v 的梯度而不是旧位置的梯度就有意义了。
下面这个图可以说明这个问题：



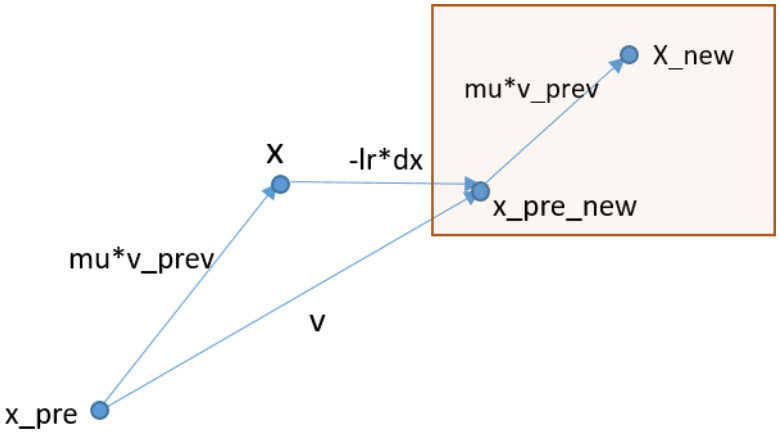
Nesterov 动量。既然我们知道动量将会把我们带到绿色箭头指向的点，我们就不要在原点（红色点）那里计算梯度了。使用 Nesterov 动量，我们就在这个“向前看”的地方计算梯度。代码如下，除了计算梯度，其他的都和原来传统动量更新方法一致。

```
    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
```

画个图可能更好理解这一点改写，注意这里的 x 实际上对应的是上图的 x，而且每次都存储的是这个位置（参数），我们把原来的 x 记作 x_pre 吧。



这样看的话上面的程序就很简单了，牢记这里的 x 实际上是 x_head 就行了。
v_prev = v　这个就是存一份，因为在第三个式子里我们需要回到 x_pre 进行更新。

v = mu * v - learning_rate * dx　这个就是更新向量，和上面的都是一样的。
x += -mu * v_prev + (1 + mu) * v 这个之所以看着复杂但是实际上并不，首先-mu*v 是回到 x_pre 的位置，然后加上 v 到新的 x_pre 位置，然后再加上 mu*v 更新到新的 x_head，这里用 x 表示了。
学习率退火：这个通俗的理解可以认为学习率应该不是一成不变的，因为刚开始优化的时候学习率应该是大一点好，这样能够快速收敛，到了后面接近最优点应该小一些好，这样能够更精确得寻找到最优点不会带来无规则的跳动，但是怎么减小也是一个问题。一般用下面三种方法进行学习率退火：

- 随步数衰减：每几个周期根据一些因素降低学习率，典型的方法是每 5 个周期将学习率减少一般，或者每 20 个周期减少为原来的 0.1。常用的一种具体操作是：使用一个固定的学习率进行训练的同时来观察验证集正确率，每当验证集正确率停止下降，就乘以一个常数（比如 0.5）来降低学习率。
- 指数衰减：$\alpha=\alpha_0 e^{-kt}$，先快后慢，是符合客观需求的，t 是迭代次数，也可以以周期作为单位。
- 1/t 衰减：$\alpha=\alpha_0/(1+kt)$，和上面的指数差不多。

实践中，随着步数衰减的随机失活更受欢迎，因为它使用的超参数（衰减系数和以周期为单位的步数）比 k 更有解释性，如果有足够的计算资源，可以让衰减更加缓慢一些，让训练的时间更长一些。
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，迭代如下：

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

这里的$[Hf(x)]$是 Hessian 矩阵，存放的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，直观解释上，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校的参数更新，在曲率大的时候小步前进，曲率小的时候大步前进，这里没有学习率这个超参数，这也是一个巨大优势。
但是这个更新方法很难用到实际深度学习中去，主要是因为 Hessian 矩阵的计算以及求逆是非常耗资源(包括内存和计算资源)，这样，就有各种拟牛顿方法被发明出来来近似转置 Hessian 矩阵，这些方法中最流行的是 L-BFGS，这个方法是使用随时间变换的梯度中的信息来隐式近似（实际上整个矩阵从来没有被计算过）。
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。
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。
逐参数适应学习率方法：
上面所说的方法都是对学习率进行全局操作，且对所有参数都是一样的，学习率调参是一个很耗费计算资源的过程，所以很多工作投入到如何进行自适应学习率调参的方法，甚至是逐个参数适应学习率调参，这些方法依然需要其他超参数设定，但是可以使用更广范围的超参数而且能比原始的学习率方法有更好的表现。
Adagrad：

```
# 假设有梯度和参数向量 x
```

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

这是一种适应性学习率算法，这里把每一个元素的更新的梯度用 dx 的梯度的平方根来了一个加权，如果梯度较大的话就用较小的学习率，注意这里是逐元素进行的，eps 是防止分母为零设置的，一般$1e^{-8} \to 1e^{-4}$之间，这里平方根的操作很重要，如果去掉的话算法表现会差很多。一个缺点是深度学习中学习率被证明通常是过于激进的，会导致模型过早停止学习。

RMSprop：是一个非常高效但是没有公开发表过的方法，这个方法简单地修改了一下上述那个方法，单调地降低了学习率：

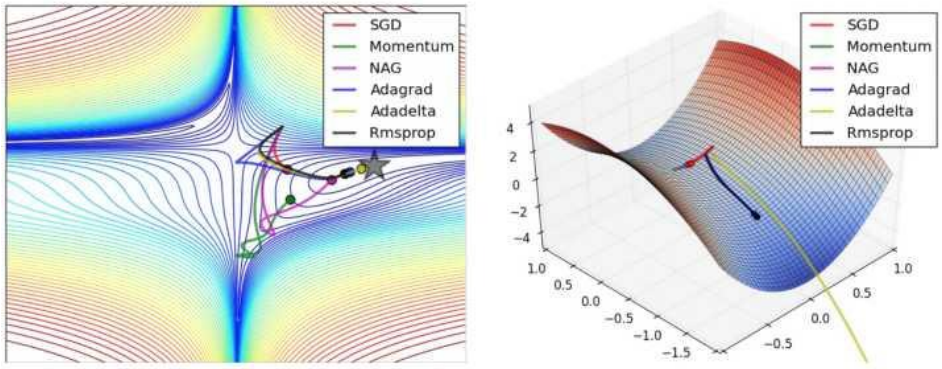
```
    cache =  decay_rate * cache + (1 - decay_rate) * dx**2
    x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

可以认为这是一种插值，让 cache 更加平滑，decay_rate 是一个超参数，常用的值是[0.9,0.99,0.999]。其他的和 Adagrad 是一样的，但是这个更新不会让学习率单调变小（是这样的，因为更新之后的 cache 并不一定会比原来大）。

Adam：这是最近才提出来的一种方法，简化版的代码如下：

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

这样一看马上就明白了，相对与 RMSprop 来说，对梯度也进行了插值，使用其平滑版本，而不是用原始梯度向量 dx，推荐的参数为：**eps=1e-8, beta1=0.9, beta2=0.999**。实际操作中推荐这种方法作为默认算法，一般回避 RMSprop 效果好一些。完整的 Adam 更新包含了一个偏置矫正机制，因为一开始 m，v 两个矩阵初始为 0，在没有完全热身之前存在偏差（因为对他们的插值权重挺大的），应该也可以先使用 **Adagrad** 方法跑一段时间，然后换成 Adam，不知道可以不？



这里这个图比较了集中方法的收敛速度，动图见:http://cs231n.github.io/neural-networks-3/ 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。

10. 超参数调优：

神经网络无法避免一些超参数，最常用的有：

- 初始化学习率。

- 学习率衰减方式
- 正则化强度（L2 惩罚，随机失活强度等）

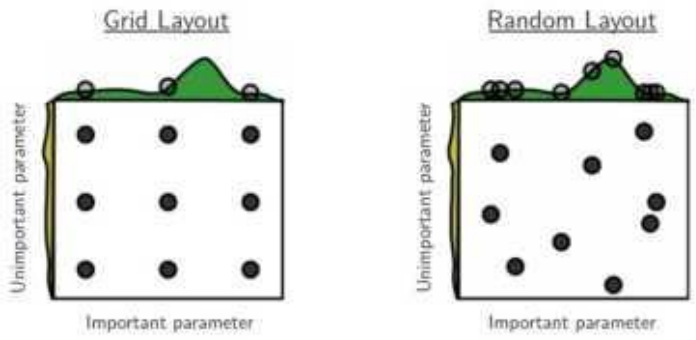
但是也是可以看到的，有些参数是相对不敏感的，比如在逐参数适应学习方法中，对于动量及其时间表的设置等，下面给出一些其他额外的要点和技巧。

实现：较大的神经网络往往需要几天甚至几周的时间来进行调参，一个设计代码的思路是：用仆程序持续的随机设置参数然后进行优化。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，然后想文件系统写一个模型的记录点（记录各种统计数据，比如损失值的变化等），这个文件最好是可以共享的，然后还有一个主程序，可以启动或者结束计算集群中的仆程序，有时候也可以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：虽然说的是交叉验证，但是很多时候使用的是一个验证集。

超参数范围：在对数尺度进行超参数搜索，一般一个典型的学习率初始化应该是：learning_rate = 10 ** np.random.uniform(-6, 1)。最起码看起来是这样，也就是说，从标准分布中生成一个随机数，然后再成为 10 的阶数，对于正则化强度，可以采用同样的策略。这是因为学习率和正则化强度对于训练的动态进程都有乘的效果。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。dropout=np.random.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 中指明：随机选择比网格化的选择更加有效，在实践中也更容易实现。



这是论文中的核心说明图，通常，有些超参数比其余的更重要，通过随机搜索，可以更精确的发现那些比较重要的参数的好数值。

对于边界上的优化值要小心：这种情况下一般发生在用一个不好的范围内搜索超参数的时候，比如我们使用 learning_rate = 10 ** np.random.uniform(-6, 1)来搜索，一旦我们找到一个比较好的值，要确定这个值是否出现在这个范围的边界上，如果是的话，就要注意了，可能就错过更好的搜索范围了。

从粗到细分阶段搜索：实践中，先进行粗略范围搜索（比如 10**（-6，1）），然后根据好的结果出现。缩小范围继续搜索，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：这是一个专门的研究领域了，有兴趣可以参考相关文献。

11. 评价：

在实践的时候，有一个总能提升神经网络几个百分点准确率的方法，就是在训练的时候单独

训练几个独立的模型，然后在测试的时候平均他们的预测结果。集成模型的数量增加，算法的结果也单调上升（但是提升效果越来越少）。还有模型之间的差异越大，可能提升效果越好，进行集成主要有下面集中方法：

- 同一个模型，不同初始化：使用交叉验证来取得最好的超参数，然后用最好的参数来训练不同初始化条件的模型。
- 在交叉验证中发现最好的模型：可以通过交叉验证取得最好的超参数，然后选择其中最好的几个来进行集成。这样做比较简单，因为在集成之后不用再训练了。
- 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，这样做多样性是不足的，但是实践中效果还不错，代价也很小。
- 训练的时候跑参数的平均值：训练的过程中，如果如果损失值相比于前一次的权重出现指数级下降时，就在内存中对权重做一个备份，这样就能对前几次循环中的网络中的参数进行平均，会发现这样一个“平滑”过的版本的权重总能得到更少的误差，直观的理解就是目标函数是一个碗状的，这样的平均更可能接触到碗底。

模型集成的一个劣势就是会在测试集上花费更多的时间。

12. 总结：

训练一个神经网络需要：

- 利用小批量数据进行梯度检查，还要注意各种错误。
- 进行合理性检查，确认初始值损失是合理的，在小数据集上可以取得 100%的正确率。
- 训练时，跟踪损失函数的值，训练集和验证集的准确率，还可以跟踪更新的参数数量相较于总参数量的比例（1/1000 左右较好），对于输入是图像的，或者卷积神经网络，可以可视化第一层权重。
- 推荐的两个更新方法是：SGD+Nesterov 动量方法，或者 Adam 方法。
- 学习率退火，随着训练学习率要下降。
- 使用随机搜索来搜索最优超参数，分阶段从粗到细。
- 进行模型集成可以获得额外的性能提高。

四、卷积神经网络。

1. 结构概述。

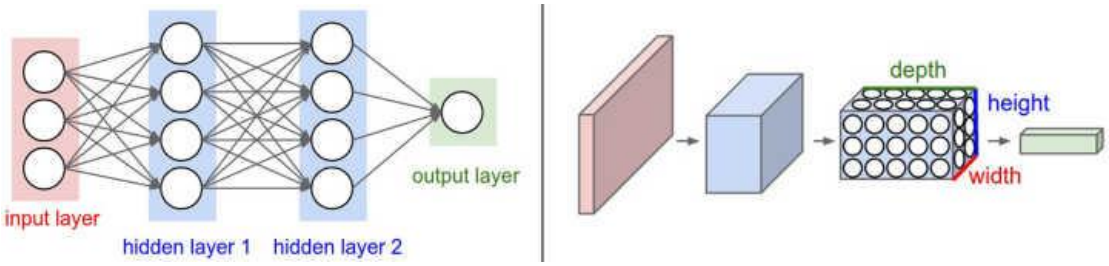
和前面说的常规神经网络是非常相似的：都由神经元构成，神经元中有具有学习能力的权重和偏置，每个神经元都能得到一些输入数据，进行内积运算后再进行激活函数运算。整个网络依旧是一个可导的评分函数：输入是原始的图像数据，输出是不同类别的评分，在最后一个层（往往是全连接层），网络依旧有一个损失函数（比如 SVM 或 softmax），并且在神经网络中我们实现的各种技巧和要点也同样适用于卷积神经网络。

变化是我们假定驶入是图像，基于该假设，我们向结构中添加了一些特有的性质，这些特有的属性使得前向传播函数看起来更加高效，并且可以大幅降低网络中的参数数量。

对于常规神经网络来说，神经网络的输入是一个向量，然后在一系列隐层中对其做变换，每个隐层都有若干神经元构成，每个神经元都与前一层所有的神经元相连，但是在一个隐层中的神经元之间是不互相连接的，最后的层称作输出层，在分类问题中，输出被看作是不同类别的评分值。

常规网络尺寸对于大尺寸的图像效果不是很好，对于 CIFAR-10 数据集中，每个图片的尺寸是 32*32*3，所以对于常规神经网络的第一个隐层来说，每个单独的神经元都要有 32*32*3=3072 个权重，这个看起来还可以接受，但如果像素扩大呢，比如是 200*2003=12000 呢，这样一个神经元就要 120000 个权重，网络中肯定不止一个神经元，那么参数的量就会快速增加，这种连接方式效率很低，大量的参数也很快会导致网络过拟合。

神经元的三维排列：输入如果全部是图像的话，可以将结构调整得更加合理，获得不小的优势，与常规的神经网络不同，卷积神经网络的各层中的神经元是 3 维排列的：宽，高，深，这里的深度对应的是激活数据体的第三个维度，而不是指整个网络的深度，比如 CIFAR-10 数据集中，数据体的维度是 32*32*3（宽，高，深）。我们会看到，层中的神经元将只与前一层中的一小块区域连接，而不是采用全连接的方式，对于用来分类 CIFAR-10 的卷积网络，其最后的输出是 1*1*10，因为在卷积神经网络结构在最后部分会把全尺寸的图像压缩为包含分类评分的一个向量，向量在深度方向排列，下面是例子：



右边是一个卷积神经网络，图中网络将它的神经元都排列成 3 个维度（宽，高，深）。卷积神经网络的每一层都将 3D 他的输入变换为神经元 3D 的激活数据并输出，这个例子中，红色是输入层装的图像，所以它的维度是图像的维度是一样的（比如这个数据集中是 32*32*3），深度代表了三个通道。

卷积神经网络是由层构成的，每一层都有一个简单的 API：用一些含或者不含参数的可导函数，将输入的 3D 数据变换为输出的 3D 数据。

2. 用来构建卷积神经网络的各种层。

卷积神经网络主要由三种类型的层构成：卷积层，汇聚层（pooling），和全连接层。全连接层和常规神经网络中的一样，通过这些层叠加起来就可以构成一个完整的神经网络，pooling 层翻译成池化我觉得挺好的。

举个例子然后再详述：一个用于 CIFAR-10 图像数据分类的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：

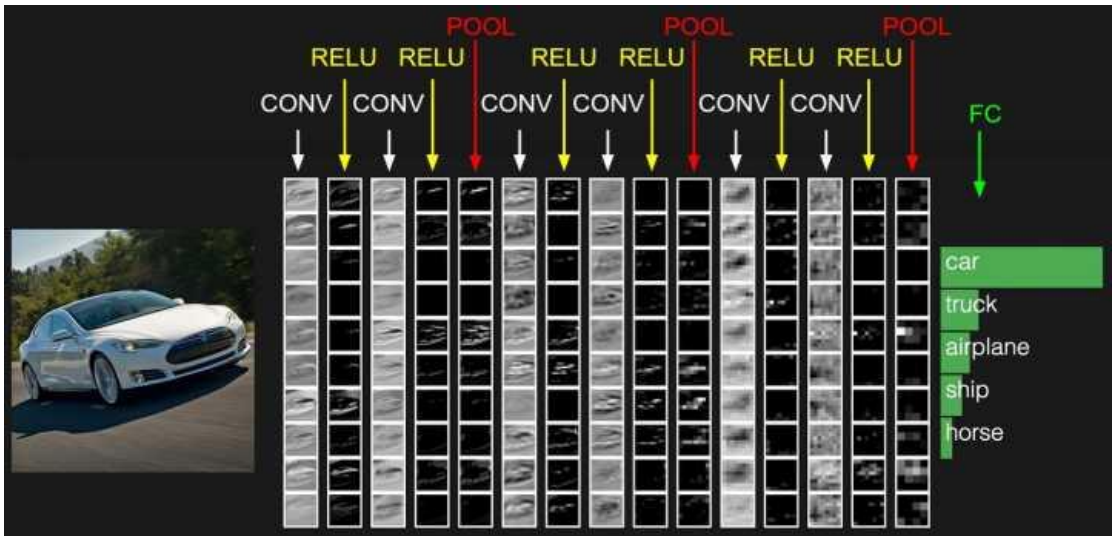
- 输入[32*32*3]存有原图像的原始像素值。
- 卷积层中，神经元与输入层中的一个局部区域相连，每个神经元都计算自己与输入层相连的小区域与自己权重的内积，卷积层会计算所有神经元的输出，如果我们使用 12 个滤波器（也叫做核），得到的数据体的维度就是（32*32*12）。
- 汇聚层进行下采样，（只在宽度和高度上进行，深度上进行采样没有意义），尺寸可以变成[16*16*12]
- 全连接层会计算分类的评分，将数据尺寸变为[1*1*10],其中 10 个数字对应十个类别的评分值，全连接层与常规神经网络一样，其中每个神经元都与前一层的所有神经元相连。

这样看来，卷积神经网络一层一层地将图像从原始像素值变换成最终的分类评分值，其中有的层有参数，有的没有（比如下采样，可以认为没有），具体地看，卷积层和全连接层对输入执行变换的时候不仅会用到激活函数，还会用到很多参数（神经元的权值和偏差）。而 RELU 层和汇聚层则是一个固定不变的函数操作，卷积层和全链接层的参数会随着梯度下降被训练。

简单的来说：

- 简单卷积神经网络的结构就是一系列层将输入数据变换为输出数据。
- 卷积神经网络结构中有集中不同类型的层。
- 每个层的输入是 3D 数据，然后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，有的层没有（卷积和全连接有，ReLU 和 pooling 没有）。

- 有的层有超参数，有的没有（卷积，全连接和 pooling 有，ReLU 没有）。



上面是一个直观的例子，左边的输入层存有的是原始的数据，右边的输出层输出的是存有类别评分的向量，数据铺成一列显示了，高维切开，最后一层装有的是不同类别对应的评分，这里只显示了得分最高的 5 个评分值和对应的列别。这是一个小的 VGG 网络。

下面讲解不同的层，层的超参数和连接的情况。

卷积层：

卷积层是卷积神经网络的核心层，产生网络中大部分的计算量。

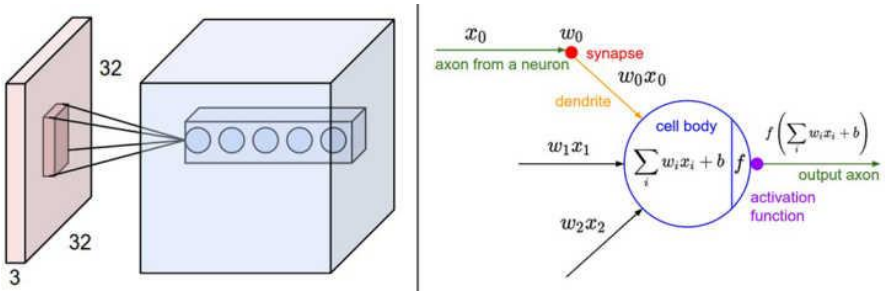
直观来说，卷积层是由一些可学习的滤波器构成的，每个滤波器在空间上（宽，高）都比较小，在深度上和输入是一致的，在宽度和高度上滑动滤波器在输入数据上会得到一个激活图。这就是卷积层的输出了，每个卷积层上我们可能有一个集合的滤波器，比如是 12 个，每个都会生成一个不同的二维激活图，这些激活图映射在深度方向上就生成了输出数据。

局部连接：

处理图像这种高维输入时，让每个神经元都与前一层的所有神经元相连是不现实的，相反的，我们让每个神经元都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），其尺寸是一个超参数，就是空间滤波器的尺寸，在深度方向上，总与输入数据的深度相等，需要注意的是，连接在空间上是局部的，但是在深度上总与输入的深度保持一致。

例 1：假设输入数据体尺寸为[32x32x3]（比如 CIFAR-10 的 RGB 图像），如果感受野（或滤波器尺寸）是 5x5，那么卷积层中的每个神经元会有输入数据体中[5x5x3]区域的权重，共 5x5x3=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。

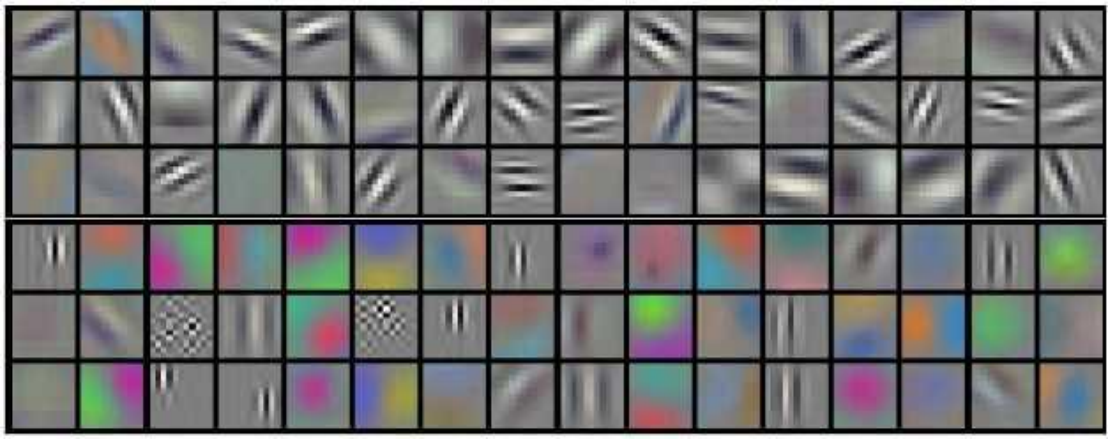
例 2：假设输入数据体的尺寸是[16x16x20]，感受野尺寸是 3x3，那么卷积层中每个神经元和输入数据体就有 3x3x20=180 个连接。再次提示：在空间上连接是局部的（3x3），但是在深度上是和输入数据体一致的（20）。



左边红色是输入数据，蓝色部分是第一个卷积层中的神经元，卷积层中每个神经元都只与数据的一个局部空间相连，但是与输入数据的深度全不相连（所有通道），深度上有多个神经元，它们接受的输入数据是一块区域（就是感受野的大小）
右边是神经元是保持不变的，也是计算内积和激活函数运算，但是被限制到一个局部空间，这个也是有解释的（其实相当于一个非线性滤波器，比如最大值滤波等）。
空间排列：
3 个超参数控制这输出数据的尺寸，分别是深度，步长，以及零填充。下面分别讨论。

- 首先，深度是一个超参数，他的使用和滤波器的数量是一致的，就看有多少个滤波器，每个滤波器对应深度切片上的一层。可以把这些沿着深度方向排列，感受野相同的神经元集合称作**深度列（depth column）**，也有人使用纤维来命名。
- 步长，即滤波器华东的步长，步长为 1 的时候，就是逐像素移动，步长 2 的时候，就是每次移动两个像素，这样的操作也会到左输出宽和高变小
- 滤波具有边界效应，用 0 在边缘条虫是很方便的，也有一个良好的性质，即可以控制输出数据的空间尺寸（即可以保持输入数据尺寸的不变（宽和高））。

参数共享：
使用参数共享主要是来控制参数数量，在上面的例子中，第一个卷积层有 55*55*96=290400 个神经元，每个有 11*11*3=364 个权重和一个偏差，这些结合起来就有 290400*364=10570560 个参数，一层就有这么多参数显然是非常大的了。
合理的一个假设是我们用同一个卷积核去滑动，得到一层二维激活图，这样就可以显著减少参数数量，比如上面的，图片尺寸[227*227*3],感受野 11*11，步长 4，不用 0 填充，因为这样刚好（227-11）/4+1=55 可以整除，卷积层的深度是 96（也就是有不同的 96 个卷积核）。这样算下来的参数就只有：11*11*3*96+96（bias）=34944 个参数，因为每个切片都使用的是同一个卷积核（这样在图像处理上也是可以解释的，应该使用同一个卷积核来对图像进行卷积处理）。



看一张图，这个是 Krizhevsky 这个构架学习到的滤波器的例子，共有 96 个切片。
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 55x55 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一些明确的中心结构时候。这时候我们就应该期望在图片的不同位置学习到完全不同的特征。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）。

用 Numpy 来举个例子，用代码展示上述思路，假设输入数据是 numpy 的数组 x，那么：

- 一个位与(x,y)的深度列（纤维）将会是 X[x,y,:] . （这和 matlab 是一样的几乎）。
- 在深度 d 切片得到的应该是 X[:,:,d].

卷积层的例子：假设输入数据的尺寸 X 的尺寸是：X.shape:(11,11,4),不用零填充，滤波器尺寸是 F=5,扫描步长是 S=2，那么输出数据的尺寸应该是（11-5）/2+1=4；即输出应该是 4*4 的宽高。那么应该是下面这些计算（只展示一部分，按照第一行滑动的一部分）

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

Numpy 中*的操作是树组件的逐元素相乘，权重 W0 是改神经元的权重，也就是滤波器核，b0 是偏差，W0 的尺寸是[5,5,4]，如果要计算第二张激活图的话（切片 2），就应该是下面这样的：

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

因为是切片 2 了，所以权重和偏置都不是第一个那个了，这都是需要网络来学习的。注意一般卷积操作之后通常连接的是 Relu 层，对激活图中的每个元素做激活函数运算，这里并没有显示出来。

到此卷积操作的过程就都展示出来了，其实很简单，和图像处理滤波操作基本是一致的，只是通过另外一种角度来看而已。

这些超参数的设置一般滤波尺寸为 3，5，步长为 1，2（都不是很好），p=1 或者 2（滤波器核为 5 的时候），对应保证输出尺寸和输入一致（宽高）。

http://cs231n.github.io/convolutional-networks/这里有一张动图来描述整个过程，简单易懂，我对这里算是比较熟了，研一的时候手写过各种滤波操作，word 也贴不了动图，就这样了。

用矩阵乘法实现：卷积运算实际上是在滤波器和输入数据的局部区域做点积。卷积层的常用实现方式就是利用了这一点，将卷积层的前向传播变成一个巨大的矩阵乘法。

- 输入数据的局部区域被 im2col 操作拉伸成列，比如，如果输入是[227*223*3]，要与尺寸为 11*11*3 的滤波器以步长为 4 进行卷积，那么输入中的[11*11*3]矩阵快被拉伸成为 11*11*3=363 的一个列向量，重复这一过程，因为步长是 4，所以输出的宽度为：(227-11)/4+1=55，所以 im2col 得到的矩阵尺寸应该是[363*3025]，这个矩阵我们暂且记作 **X_col**，一共有 55*55=3025 这么的多的矩阵块，因为感受野之间有重叠，所以输入数据体中的数字在不同列中可能有重复。（这几乎是肯定的）
- 卷积层的权重同样会被拉伸成列，比如如果我们有 96 的切片，那么就会生成一个[96*363]的矩阵，称作 **W_row**。
- 现在我们只需要进行一个大的矩阵乘法 np.dot(W_row,X_col)就可以计算这许多卷积了，这样进行操作之后,我们得到的尺寸是 96*3025,只需要对每一行都重构成一个 55*55 的矩阵就可以和每个滤波器的矩阵对应了。
- 这个方法的缺点就是占用的内存较多，因为可以看到，输入数据中的某些值被拷贝了多次，优点就是可以利用矩阵乘法进行高效的计算。

卷积操作的反向传播（同时对于数据和权重）还是一个卷积（但是是和空间上反转的滤波器），

使用一个 1 维的例子比较容易演示：（课程上并没有展开）。

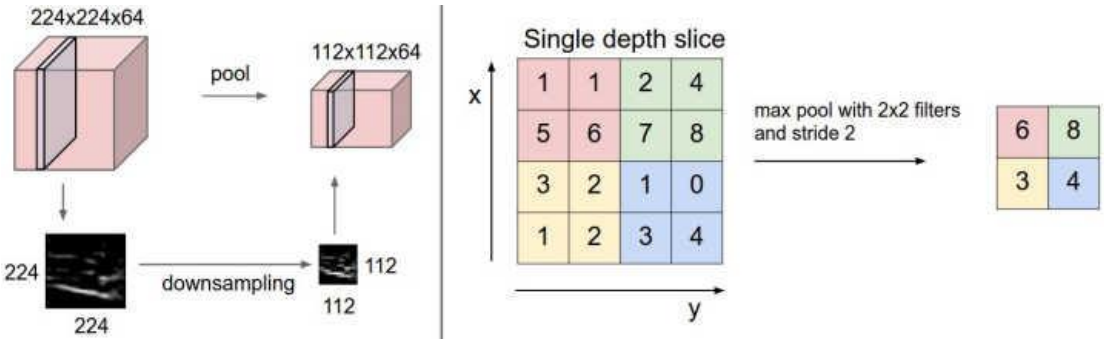
1*1 卷积：某些论文中使用了 1*1 的卷积，一般看到这个还是很疑惑的，特别是有信号处理背景的人，1*1 卷积不久相当于什么都没做么，但是在卷积神经网络中并不是这样，因为这里是对 3 的维度就行操作，滤波器和输入数据的深度是一样的，所以如果输入是[32*32*3]的话，1*1 卷积就是高效地进行三维点积（输入深度是 3 个通道），这样其实可以理解为是在三个通道做不同的加权了，也是有点意义的。

扩张卷积：目前为止我们讨论的都是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，比如这个就叫做扩张，举例来说，在某个维度上滤波器 w 的尺寸是 3，那么计算输入 x 的方式有：**w[0]*x[0] + w[1]*x[1] + w[2]*x[2],**这就是一个正常的计算，没有扩张，或者说扩张为 0。如果扩张为 1 呢：**w[0]*x[0] + w[1]*x[2] + w[2]*x[4]**，直观上来看是隔一个数据采集一个，本质上是对 x 做了一个下采样然后再做卷积。这样就能在很少的层数快速汇聚输入图片的大尺度特征（这是肯定的，因为下采样了，相当于把图像缩小了，那么大尺度特征也就可以被小滤波器捕捉到）。比如，如果上下重叠两个 3*3 的滤波器，扩张分别是 0 和 1，那么第二个卷积层的神经元的感受野就是 5*5，如果再进行扩张，那么感受野就会快速增长。

汇聚层：

通常，在连续的卷积层之间会周期地插入一个汇聚层，作用是逐渐降低数据体的空间尺寸（这个汇聚层一般是一个线性或者非线性滤波），这样就能减少网络中参数的数量，使得计算资源消耗变少，也能有效控制过拟合。汇聚层使用 max 操作，对于每一个深度切片独立操作（这是肯定的，没有理由对不同的深度层做汇聚），最常用的就是 2*2 的 max 滤波器，且以步长为 2 处理，也就是不重叠，进行最大值下采样，将其中 75%的信息都丢掉，深度保持不变，汇聚层并不额外引入参数，所以设置也比较简单。另外卷积层也很少使用 0 填充。一般最大汇聚只使用 2，或 3 的大小，步长均为 2，不宜用过大的汇聚尺寸，会对网络有比较大的汇聚性。

普通汇聚：除了最大汇聚，汇聚单元还可以使用其他的函数，比如平均汇聚和 L2 范式汇聚，平均汇聚历史上比较常用，但是现在已经很少用了，时间证明，最大汇聚效果比平均汇聚效果要好。看下图，不解释了，其实就是一个下采样。



反向传播：回顾一下反向传播的内容，max(x,y)函数反向传播的时候可以简单地理解为将梯度沿着最大的数回传，因此，在前向传播经过汇聚层的时候，要把池中最大元素的索引记录下来（也叫道岔（switchs）），这样反向传播的时候梯度的路由 就可以很高效。

不使用汇聚层：有人认为可以不适用汇聚层，只使用重复卷积层，卷积中使用更大的步长来快速降低数据尺寸，特别是对于生成模型，抛弃汇聚层可以提高 performance，其实直观看来，这样的下采样确实是可有可无的，抛弃掉也无非是尺寸下降慢点。在未来的卷积网络结构中，可能会很少地使用甚至不使用汇聚层。

归一化层：

卷积神经网络中，提出了许多不同类型的归一化层，有时候是为了实现生物大脑中观测到的抑制机制，但这些曾都渐渐地不流行了，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。 归一化的主要作用就是对数据进行归一化，具体实现不同的归一化方式有不同的实现方式，具体遇到了再看吧。

全连接层：

全连接层中神经元对于前一层中所有的激活数据是全部连接的，这个和常规神经网络中一样，它们的激活可以先用矩阵乘法，然后再加上一个偏差。

把全连接层转换为卷积层：全连接层和卷积层唯一的不同就是卷积层中的神经元只于输入数据中的一个局部区域连接，并且在卷积列的神经元共享参数，两类层中，神经元都是点积计算，所以它们的函数形式是一样的，因此将两者相互转换是可能的：

- 对于任何一个卷积层，都存在一个能和其实现一样前向传播功能的全链接层，权重矩阵是一个巨大的矩阵，除了某些特定块（因为有局部连接），其余部分都是 0，而在其中大部分块中，元素都是相等的（因为参数是共享的）。
- 相反的，任何全链接层都可以被转换成卷积层，比如，一个 K=4096 的全链接层，输入数据的尺寸是 7*7*512,这个全连接层可以被看作是一个滤波器 F=7,S=0,K=4096 的一个卷积层，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数据体，所以输出就会变为：1*1*4096,这个结果就和使用初始的那个全连接层是一样的了。

全连接层转换为卷积层是更有用的，假如一个卷积神经网络的输入是 224*224*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*7*512 的激活数据体（AlexNet 中就是这样，使用了 5 个汇聚层来对输入数据进行下采样，每次采样尺寸下降一半，所有最终的尺寸降为 224/2^5=7），从这里可以看到，AlexNet 使用了两个尺寸为 4096 的全连接层，最后有一个有 1000 个神经元的全连接层用于计算分类评分，我们可以把这三个全连接层的任意一个转换为卷积层：

- 针对第一个连接接区域是[7*7*512]的全连接层，令其滤波器尺寸为 F=7，这样输出数据体就变成[1*1*4096]了。滤波器的尺寸实际上是 7*7*512,因为滤波器深度和输入数据的深度是一样的。
- 针对于第二个全连接层，令其滤波器尺寸为 F=1,这样输出数据就会变为[1*1*4096],这里滤波器尺寸为[1*1*4096]，共有 4096 个这样的滤波器，这样就会得到输出数据为[1*1*4096]。
- 最后一个全连接层也做类似的，滤波器尺寸为[1*1*4096]，共有 1000 个这样的滤波器，最终得到的输出是[1*1*1000].

实际操作中，每次这样的变换都需要把全连接层的权重重塑成卷积层的滤波器，这样做的作用是什么呢？在下面这种情况下可以更高效：让卷积网络在一张更大的输入图片上滑动（把卷积网络整体看作一个滤波器）即把一张更大的图的不同区域都带入卷积网络，从而得到每个区域的评分，得到更多的输出，这样的转换可以让我们在单个向前传播的过程中完成上述操作。

举个例子：

如果想让 224*224（我们这里只考虑宽高，因为深度总是匹配的）尺寸的滑动窗，以步长为 32 在 384*384 的大图上进行滑动，每一部分都带入神经网络，最后得到 6*6 个位置上的评分，如果 224*224 的输入经过卷积和汇聚层之后得到了[7*7*512]的数据，那么更大的 384*384 尺寸的图片经过了同样的卷积神经网络就会的一个[12*12*512]（384/2^5=12）的输出数据，然后经过 3 个全连接层（转为卷积层）的处理，就会得到

[6*6*1000]的输出，（（12-7）/1+1），这个结果正好是 224*224 在 384*384 上以步长 32 滑动所得的 6*6 个区域对应的评分。注意这个 32 不是随便取的，刚好是前面的网络的缩小率（共下采样了 5 次，1/2^5=1/32，这样才能保证刚好前面的每一个滑动对应最后的全连接层之前的 7*7 区域），如果是以更小的步长就不行了。

这样得到的结果和把六个区域分别拿出来进行卷积网络传播得到 6 个评分的结果是一样的，但是显然前向传播 6 次的话有很多区域是被重复传播的，这样做的结果就是可以节省很多计算资源。

总而言之：把全连接层转换为卷积层的一个好处就是可以看作是把整个神经网络当作一个卷积核在更大的图上移动，但是实际上操作的时候只用了一次前向传播，增加了数据的利用率减少了计算消耗。

上面说到，如果不是以 32 为步长进行处理呢？用多次前向传播可以解决这个问题，比如我们想用步长为 16 的舷窗，那么先使用原图像在转换后的卷积网络上执行前向传播，（这样得到是 6*6*1000 的输出），然后分别沿宽度和高度，最后同时沿宽度和高度把平移得到的图片分别带入神经网络。

我大概算了一下，比如上面说的那个网络，如果卷积层下采样 4 次，也就是尺寸降位原来的 1/16，然后第一个全连接层 F=14（224/2^4=14）,其他的参数都不变，这样最后得到的评分还是[1*1*1000]的。

那么在这样一个网络下面我们以 16 步长在 384*384 的图像上移动窗口，这样就会共有（384-224）/16+1=11 的尺寸，全连接层之前得到的尺寸是 24*24 的，因为（384/16=24），这样用全连接层转换的三个卷积层操作的话就会得到（24-14）/1+1=11 的尺寸数据，最后的输出是[11*11*1000]的，和上面需要的是一样的，所以说确实是如果要用一次前向传播就想算出所有子窗口的评分的话那么就是要求降采样率和滑动步长是匹配的。

后来回宿舍和舍友讨论了一下，又有新新的发现，上面说的用 32 为步长进行滑动，这个 32 是由网络决定的，因为把 384*384 的图像输入的话肯定会得到 11*11 的评分（这里把深度略去），这个评分就相当于在原大图上进行 32 为步长滑动送入神经网络所得的评分结果。用较小的步长不行的原因就是原网络是每 32*32 变成了一个像素，现在要每 16*16 变成一个像素，这样肯定是丢失了 3/4 的信息了（丢失的原因就是因为 5 次 pooling）。所以肯定是不行的。

以上面一直再说的这个网络为例，如果用 16*16 的步长的话（224 在 384 上扫描的步长），最后应该得到的是 11*11 的评分（略去深度），但是用原网络（5 次 pooling）只能得到 6*6 的评分，可以肯定这 6*6 的评分肯定是包含在 11*11 的里面的（无非是 16 的步长滑动两次而已）。所以解决方法说是先把原图过一遍神经网络得到 6*6 的评分，然后再进行一些扫描，得到其他的位置的评分，这样做是把两种方法结合起来了，一次前向传播得到 36 个位置的评分，近 1/4 的计算量被节省，剩下的因为用原网络会有信息丢失就只能去逐个扫描了。

3. 卷积神经网络的结构。

卷积神经网络通常由三种层构成（最基础的），卷积层，汇聚层（不特殊说明就是 max）和全连接层（FC）ReLU 激活函数也应该算作一层，逐元素地进行激活函数操作。下面主要讨论这些层通常情况下是如何组合在一起的。

层的排布规律：

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被缩小到一个足够小的尺寸，在某个地方过度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，也就是说，最常见的卷积神经网络结构如下：

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

其中相乘是代表重复次数，POOL？指的是一个可选的汇聚层，其中 N>0，通常 N<=3,K<3。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。

一般来说，几个小滤波器卷积层的组合比一个大的滤波器卷积层好：假设一层一层地叠加了 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，第一个卷积层的每个神经元都对输入数据有一个 3*3 的视野，第二个卷积层对上一个卷积层有一个 3*3 的视野，也就是对输入数据有一个 5*5 的视野（这里步长都是 1 的话），第三层相对输出就是 7*7 的视野，假设不采用这个 3*3 的卷积层而是直接用一个 7*7 的卷积层，所有的神经元的感受野就是 7*7，但是有一些缺点：首先，多个卷积层与非线性激活层的交替结构，比单一卷积层更能提取深层的更好的特征。其次，假设所有的数据都有 C 个通道，那么 7*7 的卷积层将会包含 C*(7*7*C)=49C^2 个参数，如果采用 3 个 3*3 的话，则仅包含 3*（C*（3*3*C））=27C^2 个参数，参数也会少很多，最好选择带有小滤波器的卷积层组合，而不是一个大的滤波器的卷积层，前者可以表达出更多个强力特征，使用的参数也较少，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。

层的尺寸的设置规律：

先介绍一些一般性规则，然后根据这些规则再进行讨论。

- 输入层（包含图像的）：最好能被 2 整除很多次，常用的数字比如 32，64，224，384，和 512。
- 卷积层：应使用小尺寸的滤波器（3，5），推荐步长用，还有一点比较重要就是对数据进行 0 填充以保证卷积层不会改变在空间维度上的尺寸（宽，高）。如果要使用大尺寸滤波器的话，通常只使用在第一个卷积层上。
- 汇聚层：负责对数据空间维度下采样，最长用的是 2*2 的感受野，步长为 2 不重叠，不怎么常用的一个是 3*3 的感受野，步长依然是 2，最大值汇聚的感受野尺寸很少有超过 3 的，因为汇聚操作毕竟是一个暴力采样的过程，很容易造成数据信息丢失。

  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都尺寸互相吻合，卷积神经网络的结构美妙对称地联系在一起。
- 用 1 的步长就是因为实际中小步长的效果更好，步长为 1 可以让空间的下采样全部由 pooling 负责，卷积层只做深度变换。

- 用 0 填充的原因是这样可以保证数据的维度不发生变化（宽高），如果不用 0 填充，那么数据的尺寸就会略微减小，那么边缘信息就会过快损失掉。
- 因为内存限制做的妥协：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 3x3 的滤波器对 224x224x3 的图像进行卷积，零填充为 1，得到的激活数据体尺寸是[224x224x64]。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。

4. 案例：

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。GooLeNet 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 Mappings 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。

5. 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。

五、总结：
看了两周多终于把这个看完了，错别字我都懒得再检查一遍了（以后再看的话再改吧），大部分都是亲手敲上去的，极少部分是复制的，学到了很多，但也还算是系统得过了一遍，有些算法没有涉及到，以后再看吧，比如决策树，SVM 细节等，这个笔记也不全，NN 可 KNN 都没写，因为一开始也没想着要做笔记，因为还是很费时间的，但是笔记做下来还是感觉不亏，学的还算挺满足的，这个基本上是按照知乎上 CS231n 笔记翻译这个专栏的结构来的，图除了一部分自己画的帮助理解的以外，也都来自于此，另外许多文字我都没有作很多修改，基本的逻辑也都保持不变，就是加上了一些自己的理解。如果有人还能看到的话(我会传到自己的 git 上做备份)，希望有帮助。主要还是自己看吧。

Zhxing　2017/11/28　初稿