CS231n课程笔记翻译：神经网络笔记

原文如下

内容列表：

快速简介

在不诉诸大脑的类比的情况下，依然是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$s = W_2max(0, W_1x)$。其中W_1的含义是这样的：举个例子来说，它可以是一个[100x3072]的矩阵，其作用是将图像转化为一个100维的过渡向量。函数$max(0, -)$是非线性的，它会作用到每个元素。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$s = W_3max(0, W_2max(0, W_1x))$，其中$W_1, W_2, W_3$是需要进行学习的参数。中间隐层的尺寸是网络的超参数，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

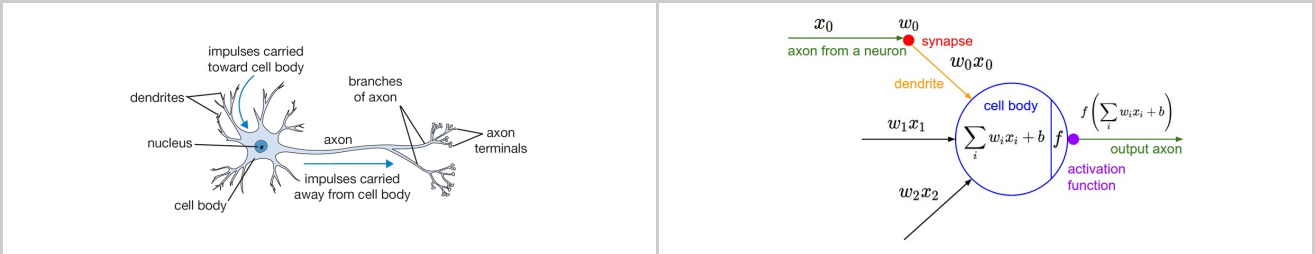
神经网络算法领域最初是被对生物神经系统建模这一目标启发，但随后与其分道扬镳，成为一个工程问题，并在机器学习领域取得良好效果。然而，讨论将还是从对生物系统的一个高层次的简略描述开始，因为神经网络毕竟是从这里得到了启发。

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_0x_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(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$\sigma(\Sigma_i w_i x_i + b)$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$\boldsymbol{w}$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]。

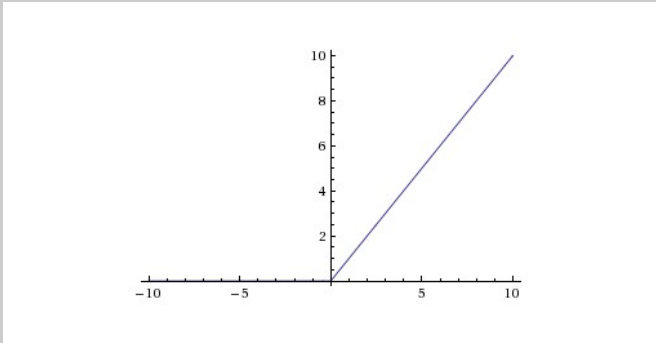
——

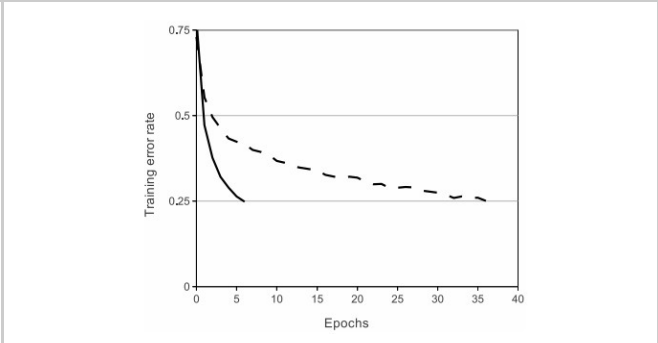
Sigmoid。sigmoid非线性函数的数学公式是$\sigma(x) = 1/(1 + e^{-x})$，函数图像如上图的左边所示。在前一节中已经提到过，它输入实数值并将其“挤压”到0到1范围内。更具体地说，很大的负数变成0，很大的正数变成1。在历史上，sigmoid函数非常常用，这是因为它对于神经元的激活频率有良好的解释：从完全不激活（0）到在求和后的最大频率处的完全饱和（**saturated**）的激活（1）。然而现在sigmoid函数已经不太受欢迎，实际很少使用了，这是因为它有两个主要缺点：

- *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，几乎就有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$\boldsymbol{f = w^T x + b}$中每个元素都$\boldsymbol{x > 0}$），那么关于$\boldsymbol{w}$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$\boldsymbol{f}$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，那么从此所以流过这个神经元的梯度将都变成0。也就是说，这个ReLU单元在训练中将不可逆转的死亡，因为这导致了数据多样化的丢失。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

$$f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$$

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。

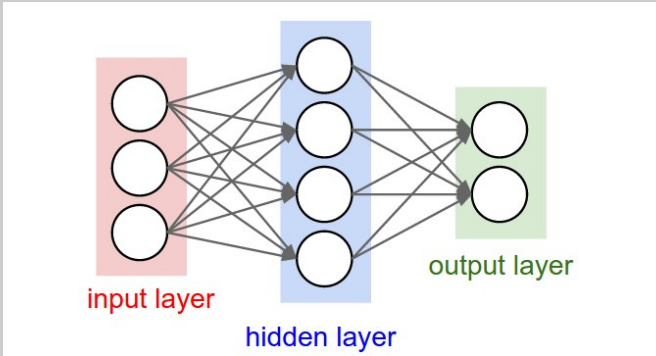
一句话：“*那么该用那种呢？*”用ReLU非线性函数。注意设置好学习率，或许可以监控你的网络中死亡的神经元占的比例。如果单元死亡问题困扰你，就试试Leaky ReLU或者Maxout，不要再用sigmoid了。也可以试试tanh，但是其效果应该不如ReLU或者Maxout。

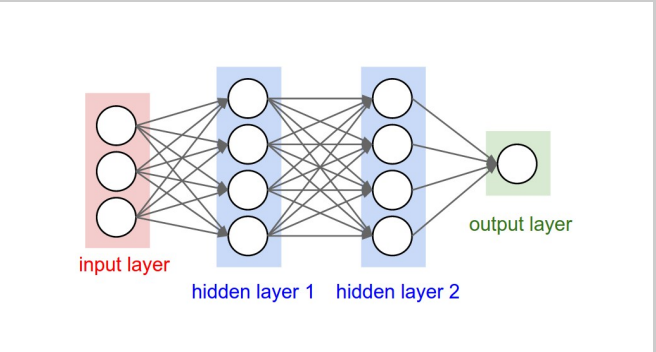
神经网络结构

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不会有激活函数的（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），[3x4]+[4x2]=20个权重，还有4+2=6个偏置，共26个可学习的参数。
- 第二个网络有4+4+1=9个神经元，[3x4]+[4x4]+[4x1]=32个权重，4+4+1=9个偏置，共41个可学习的参数。

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[3x1]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[4x3]，所有单元的偏置储存在**b1**中，尺寸[4x1]。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[4x4]矩阵，存储着第二个隐层的连接，**W3**是[1x4]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**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a, b, 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：

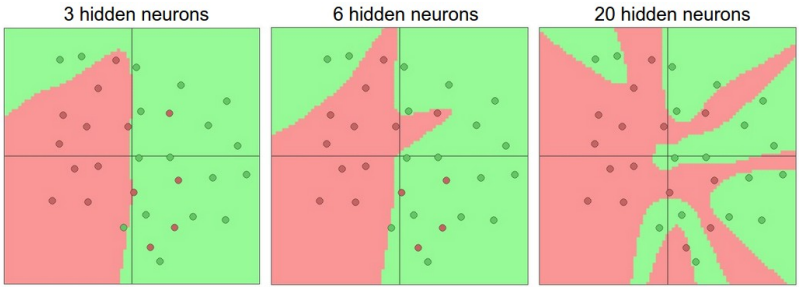
- [Deep Learning]()的[Chapter6.4]()，作者是Bengio等。
- [Do Deep Nets Really Need to be Deep?]()
- [FitNets: Hints for Thin Deep Nets]()

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。

——

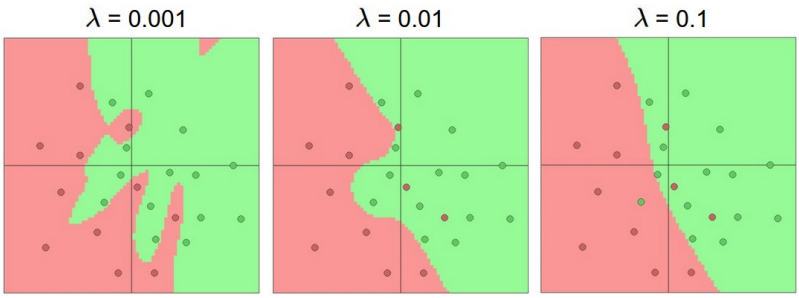
在上图中，可以看见有更多神经元的神经网络可以表达更复杂的函数。然而这既是优势也是不足，优势是可以分类更复杂的数据，不足是可能造成对训练数据的过拟合。过拟合（**Overfitting**）是网络对数据中的噪声有很强的拟合能力，而没有重视数据间（假设）的潜在基本关系。举例来说，有20个神经元隐层的网络拟合了所有的训练数据，但是其代价是把决策边界变成了许多不相连的红绿区域。而有3个神经元的模型的表达能力只能用比较宽泛的方式去分类数据。它将数据看做是两个大块，并把个别在绿色区域内的红色点看做噪声。在实际中，这样可以在测试数据中获得更好的泛化（**generalization**）能力。

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，应该进尽可能使用大网络，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