深度前馈网络

1. 深度前馈网络（`deep feedfoward network`）也称作前馈神经网络（`feedforward neural network`）或者多层感知机（`multilayer perceptron:MLP`），它是最典型的深度学习模型。
    - 卷积神经网络就是一种特殊的深度前馈网络。
    - 深度前馈网络也是循环神经网络的基础。

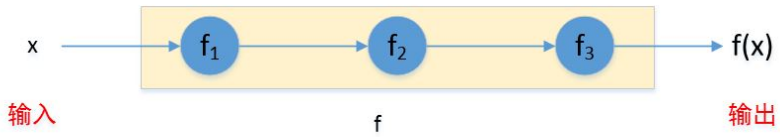
一、基础

1.1 基本概念

1. 深度前馈网络的目标是近似某个函数 f^* 。
    - 分类器 $y=f^*(\vec{\mathbf{x}})$ 将输入 $\vec{\mathbf{x}}$ 映射到它的真实类别 y ，其中 f^* 是真实的映射函数。
    - 深度前馈网络定义另一个映射 $y'=f(\vec{\mathbf{x}};\vec{\theta})$，并且学习参数 $\vec{\theta}$ 从而使得 f 是 f^* 的最佳近似。
2. 深度前馈网络之所以称作前馈的（`feedforward`），是因为信息从输入 $\vec{\mathbf{x}}$ 到输出 y 是单向流动的，并没有从输出到模型本身的反馈连接。

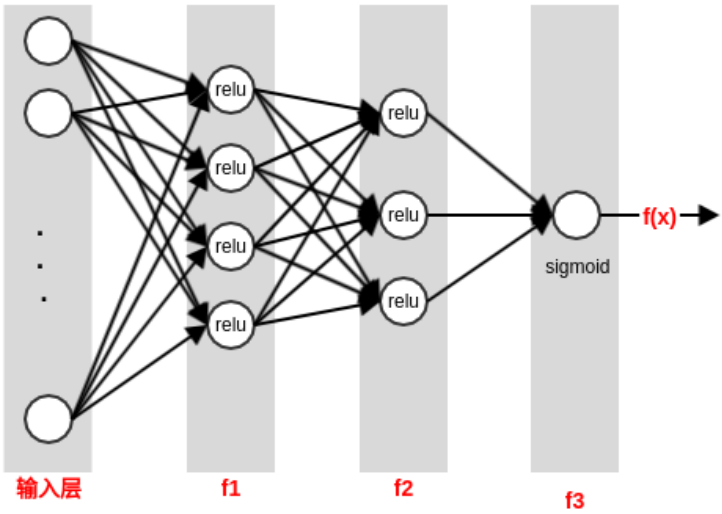
   如果存在反馈连接，则这样的模型称作循环神经网络（`recurrent neural networks`）。
3. 深度前馈网络通常使用许多不同的函数复合而成，这些函数如何复合则由一个有向无环图来描述。最简单的情况：有向无环图是链式结构。

   假设有三个函数 f_1,f_2,f_3 组成链式复合结构，则：$f(\vec{\mathbf{x}})=f_3(f_2(f_1(\vec{\mathbf{x}})))$ 。其中： f_1 被称作网络的第一层， f_2 为网络第二层， f_3 称为网络第三层。链的全长称作模型的深度。



    - 深度前馈网络的最后一层也称作输出层。输出层的输入为 $f_2(f_1(\vec{\mathbf{x}}))$，输出为 $f(\vec{\mathbf{x}})$ 。
    - 给定训练样本 $(\vec{\mathbf{x}},y)$，要求输出层的输出 $f(\vec{\mathbf{x}})\approx y$，但是对于其他层并没有任何要求。
        - 因为无法观测到除了输出层以外的那些层的输出，因此那些层被称作隐层(`hidden layer`)。
        - 学习算法必须学习如何利用隐层来配合输出层来产生想要的结果。
        - 通常每个隐层的输出都是一个向量而不是标量，这些隐层的输出向量的维数决定了深度前馈网络的宽度。
4. 也可以将每一层想象成由许多并行的单元组成，每个单元表示一个向量到标量的函数：每个单元的输入来自于前一层的许多单元，单元根据自己的激活函数来计算单元的输出。

   因此每个单元类似于一个神经元。



1.2 特征学习

1. 线性模型简单高效，且易于求解。但是它有个明显的缺陷：**模型的能力被局限在线性函数中，因此它无法理解任意两个输入变量间的非线性相互作用** 。

   解决线性模型缺陷的方法是：采用核技巧，将线性模型作用在 $\phi(\vec{\mathbf{x}})$ 上，而不是原始输入 $\vec{\mathbf{x}}$ 上。其中 ϕ 是一个非线性变换。

   可以认为：通过 ϕ，提供了 $\vec{\mathbf{x}}$ 的一个新的 `representation` 。

2. 有三种策略来选择这样的非线性变换 ϕ 。

   - 使用一个通用的 ϕ，如无限维的 ϕ（采用基于 `RBF` 核的核技巧）。

     当 ϕ 具有足够高的维数，则总是有足够的能力来适应训练集，但是对于测试集的泛化往往不佳。这是因为：通用的 ϕ 通常只是基于局部平滑的原则，并没有利用足够多的先验知识来解决高级问题。

   - 手动设计 ϕ 。

     这种方法对于专门的任务往往需要数十年的努力（如语音识别任务）。

   - 通过模型自动学习 ϕ。

     这是深度学习采用的策略。以单层隐层的深度前馈网络为例：$y=f(\vec{\mathbf{x}};\vec{\theta},\vec{\mathbf{w}})=\phi(\vec{\mathbf{x}};\vec{\theta})^T\vec{\mathbf{w}}$ 。此时有两个参数：

     - 参数 $\vec{\theta}$ ： 从一族函数中学习 ϕ ，其中 ϕ 定义了一个隐层。
     - 参数 $\vec{\mathbf{w}}$ ： 将 $\phi(\vec{\mathbf{x}})$ 映射到所需输出。

3. 深度学习中，将 `representation` 参数化为 $\phi(\vec{\mathbf{x}};\vec{\theta})$，并使用优化算法来寻找 $\vec{\theta}$ 从而得到一个很好的 `representation` 。

   - 如果使用一个非常宽泛的函数族 $\phi(\vec{\mathbf{x}};\vec{\theta})$，则能获得第一种方案的好处：适应能力强。
   - 如果将先验知识编码到函数族 $\phi(\vec{\mathbf{x}};\vec{\theta})$ 中，则能获得第二种方案的好处：有人工先验知识。

   因此深度学习的方案中，只需要寻找合适的、宽泛的函数族 $\phi(\vec{\mathbf{x}};\vec{\theta})$ ，而不是某一个映射函数 $\phi(\vec{\mathbf{x}})$ 。

4. 通过特征学习来改善模型不仅仅适用于前馈神经网络，也适用于几乎所有的深度学习模型。

1.3 训练

1. 训练一个深度前馈网络和训练一个线性模型的选项相同：选择优化算法、代价函数、输出单元的形式。

除此之外还需要给出下列条件：

- 由于深度前馈网络引入了隐层的概念，因此需要选择适用于隐层的激活函数。激活函数接受隐层的输入值，给出了隐层的输出值。
- 深度前馈网络的网络结构也需要给出，其中包括：有多少层网络、每层网络有多少个单元、层级网络之间如何连接。

2. 深度神经网络训练时需要计算复杂函数的梯度，通常这采用反向传播算法(`back propagation`)和它的现代推广来完成。

1.4 示例

1. `XOR`函数是关于两个二进制值 x_1, x_2 的运算，其中 $x_1, x_2 \in \{0, 1\}$，要求：

$$\begin{aligned}\text{xor}(0,1) &= 1\\ \text{xor}(1,0) &= 1\\ \text{xor}(0,0) &= 0\\ \text{xor}(1,1) &= 0\end{aligned}$$

令想要学习的目标函数为：$y = f^*(\vec{\mathbf{x}}) = \text{xor}(x_1, x_2)$，其中 $\vec{\mathbf{x}} = (x_1, x_2)^T$，即 x_1, x_2 为输入 $\vec{\mathbf{x}}$ 的两个分量。

假设模型给出了一个函数 $\hat{y} = f(\vec{\mathbf{x}}; \vec{\theta})$，希望学习参数 $\vec{\theta}$，使得 f 尽可能接近 f^*。

2. 考虑一个简单的数据集 $\mathbb{D} = \{(0,0)^T, (0,1)^T, (1,0)^T, (1,1)^T\}$。希望 f 在这四个点上都尽可能接近 f^*。

采用`MSE`损失函数：$J(\vec{\theta}) = \frac{1}{4}\sum_{\vec{\mathbf{x}}\in\mathbb{D}}[f^*(\vec{\mathbf{x}}) - f(\vec{\mathbf{x}}; \vec{\theta})]^2$。

3. 假设选择一个线性模型：$f(\vec{\mathbf{x}}; \vec{\mathbf{w}}, b) = \vec{\mathbf{x}}^T\vec{\mathbf{w}} + b$。通过最小化 $J(\vec{\theta})$，可以得到它的解为：

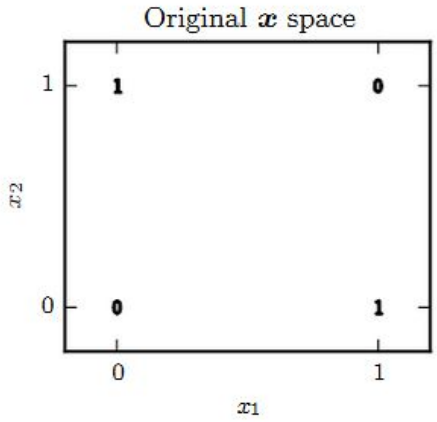
$$\vec{\mathbf{w}} = \vec{\mathbf{0}}, \quad b = \frac{1}{2}$$

即：$f(\vec{\mathbf{x}}; \vec{\mathbf{w}}, b) = \frac{1}{2}$。这意味着：线性模型将在每一点都是输出 0.5，因此它并不是`xor`函数的一个很好的拟合。

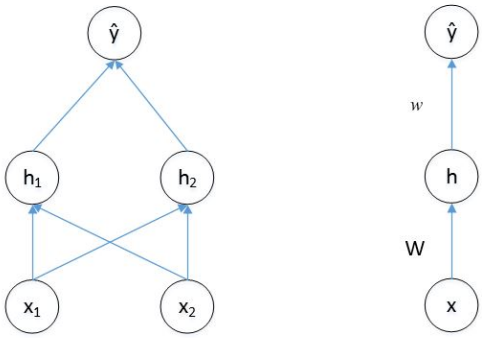
从下图可知：

- 当 $x_1 = 0$ 时，函数的输出随着 x_2 的增加而增加。
- 当 $x_1 = 1$ 时，函数的输出随着 x_2 的增加而减少。

因此导致了 $w_2 = 0$；同理 $w_1 = 0$。



4. 假设采用一个简单的深度前馈网络。该网络结构如下，它有一层隐层，并且隐层中包含两个单元。



- 第一层为隐层，对应于函数： $f_1(\vec{\mathbf{x}};\mathbf{W},\vec{\mathbf{c}})$，其输入为 $\vec{\mathbf{x}}=(x_1,x_2)^T$ ，输出为 $\vec{\mathbf{h}}=(h_1,h_2)^T$ 。
- 第二层为输出层，对应于函数： $f_2(\vec{\mathbf{h}};\vec{\mathbf{w}},b)$，其输入为 $\vec{\mathbf{h}}$ ，输出为 $\hat{y}$ 。

令输出层仍然是一个线性回归模型，即： $f_2(\vec{\mathbf{h}};\vec{\mathbf{w}},b)=\vec{\mathbf{h}}^T\vec{\mathbf{w}}+b$ 。则完整的模型为： $f(\vec{\mathbf{x}};\mathbf{W},\vec{\mathbf{c}},\vec{\mathbf{w}},b)=f_2(f_1(\vec{\mathbf{x}}))$ 。

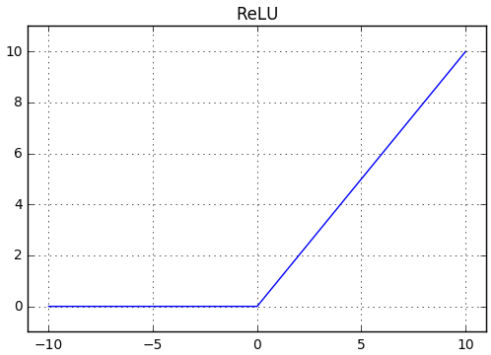
5. 大多数神经网络中， f_1 的构造过程为：先使用仿射变换，然后通过一个激活函数。其中：激活函数不需要参数控制，仿射变换由参数控制。

   令 $\vec{\mathbf{h}}=g(\mathbf{W}^T\vec{\mathbf{x}}+\vec{\mathbf{c}})$，其中 $\mathbf{W}^T\vec{\mathbf{x}}+\vec{\mathbf{c}}$ 就是仿射变换， $g(\cdot)$ 为激活函数。

   假设隐层的激活函数是线性的，则 f_1 也是线性的，暂时忽略截距项，则 $f_1(\vec{\mathbf{x}})=\mathbf{W}^T\vec{\mathbf{x}};f_2(\vec{\mathbf{h}})=\vec{\mathbf{w}}^T\vec{\mathbf{h}}$。即： $f(\vec{\mathbf{x}})=\vec{\mathbf{w}}^T\mathbf{W}^T\vec{\mathbf{x}}$ 。

   令： $\vec{\mathbf{w}}'=\mathbf{W}\vec{\mathbf{w}}$，则有： $f(\vec{\mathbf{x}})=\vec{\mathbf{x}}^T\vec{\mathbf{w}}'$。即：前馈神经网络整体也是线性的。根据前面讨论，线性模型无法拟合 `xor` 函数。因此 f_1 必须是非线性函数。

6. 现代神经网络中，默认推荐的激活函数为修正线性单元(`rectified linear unit:ReLU`): $g(z)=\max\{0,z\}$ 。



整个网络为： $f(\vec{\mathbf{x}};\mathbf{W},\vec{\mathbf{c}},\vec{\mathbf{w}},b)=\vec{\mathbf{w}}^T\max\{0,\mathbf{W}^T\vec{\mathbf{x}}+\vec{\mathbf{c}}\}+b$ 。

其中一个解为：

$$\mathbf{W}=\begin{bmatrix}1&1\\1&1\end{bmatrix},\quad\vec{\mathbf{c}}=\begin{bmatrix}0\\-1\end{bmatrix},\quad\vec{\mathbf{w}}=\begin{bmatrix}1\\-2\end{bmatrix},\quad b=0$$

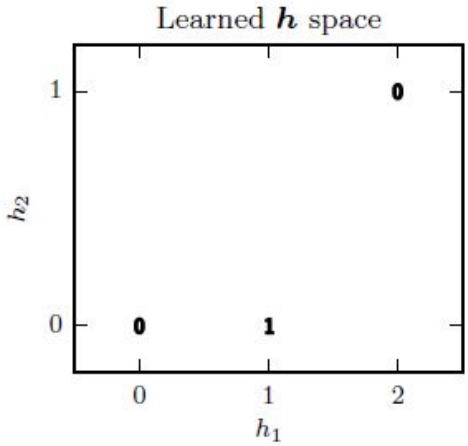
令 $\mathbf{X}$ 表示输入矩阵，每个样本占用一行。则对于输入空间中的全部四个点，输入矩阵为：

$$\mathbf{X}=\begin{bmatrix}0&0\\0&1\\1&0\\1&1\end{bmatrix}$$

根据 $\vec{\mathbf{h}}=g(\mathbf{W}^T\vec{\mathbf{x}}+\vec{\mathbf{c}})$，有：

$$\mathbf{H}=\begin{bmatrix}0&0\\1&0\\1&0\\2&1\end{bmatrix}$$

$\mathbf{H}$ 的每一行表示一个样本 $\vec{\mathbf{x}}_i$ 对应的隐单元 $\vec{\mathbf{h}}_i$ 。可以看到：隐层改变了样本之间的关系。



$f(\mathbf{X})=\mathbf{H}\vec{\mathbf{w}}+\vec{\mathbf{0}}$，得到：

$$\begin{bmatrix}0&0\\0&1\\1&0\\1&1\end{bmatrix}\xrightarrow{f}\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$$

.

7. 在使用深度前馈网络逼近 `xor` 函数中，参数的求解可以通过简单的猜测来求解。但是对于复杂的函数逼近问题中，通常使用基于梯度的优化算法。
   - 这里给出的 `xor` 问题的解是损失函数的全局最小点，也可以通过梯度下降法找到该点。
   - 在实践中，梯度下降法通常难以找出像这样的容易理解的、整数值的解。

二、损失函数

2.1 损失函数的非凸性

1. 在线性模型中，对于线性回归模型，可以直接求解出解析解。对于 `logistic` 回归或者 `SVM`，其损失函数为凸的。

   凸优化算法可以保证全局收敛，而且理论上保证从任何一种参数出发都可以收敛。实际计算中，可能遇到数值稳定性问题。

2. 神经网络和线性模型的最大区别是：神经网络的非线性导致大多数的损失函数都是非凸的。损失函数非凸导致两个问题：
   - 基于梯度的优化算法仅仅能够使得损失函数到达一个较小的值，而不是全局最小值。
   - 基于梯度的优化算法无法保证从任何一个初始参数都能收敛，它对于参数的初值非常敏感。

     对于神经网络：
     - 通常将所有的权重值初始化为小的随机数。

- 通常将偏置初始化为零或者小的正值。

  不用负数是因为如果偏置的初始值为负，很可能导致某些神经元一直处于未激活状态。

2.2 代价函数的选取

1. 深度学习的一个重要方面是代价函数的选取。
   - 代价函数给出的是单个样本的损失，损失函数是代价函数在所有样本上的和。
   - 通常神经网络的代价函数与传统模型（如线性模型）的代价函数相同。
2. 大多数现代的神经网络采用最大似然准则，令代价函数为负的对数似然函数。因此损失函数为：

$$J(\vec{\theta}) = -\mathbb{E}_{\vec{\mathbf{x}},y\sim\hat{p}_{data}}\log p_{model}(y\mid\vec{\mathbf{x}};\vec{\theta})$$

   其中：
   - $\hat{p}_{data}$ 为样本的经验分布：

$$\hat{p}_{data}(\vec{\mathbf{x}}_i,y_i) = \begin{cases}\frac{1}{N}\delta(\vec{\mathbf{x}}-\vec{\mathbf{x}}_i,y-y_i), & (\vec{\mathbf{x}}_i,y_i)\in\mathbb{D}\\ 0, & \text{else}\end{cases}$$

     $\delta(\cdot)$ 为狄拉克函数，它仅在原点处非0，在其它所有位置都为 0 ，其在整个定义域上的积分为 1 。 N 为数据集 $\mathbb{D}$ 的大小。

   - p_{model} 为对数据建立的模型， $\vec{\theta}$ 为模型参数。 代价函数的具体形式取决于 p_{model}的形式，随不同的模型而改变。

     如： $p_{model}(y\mid\vec{\mathbf{x}};\vec{\theta}) = \mathcal{N}(y;f(\vec{\mathbf{x}};\vec{\theta}),\mathbf{I})$ ，则 $J(\vec{\theta}) = \frac{1}{2}\mathbb{E}_{\vec{\mathbf{x}},y\sim\hat{p}_{data}}||y-f(\vec{\mathbf{x}};\vec{\theta})||^2+\text{const}$ ，常数项包含了高斯分布的方差，与 $\vec{\theta}$ 无关。

     因此可以看到：此时的最大似然估计等价于最小均方误差。

   - $\mathbb{E}_{\vec{\mathbf{x}},y\sim\hat{p}_{data}}\log p_{model}(y\mid\vec{\mathbf{x}};\vec{\theta})$ 其实就是样本的经验分布 $\hat{p}_{data}$ 与模型 p_{model} 的交叉熵 $H(\hat{p}_{data},p_{model})$ 。
3. 使用最大似然准则来导出代价函数的优势是：减轻了为每个模型设计代价函数的负担。一旦明确了一个模型 $p_{model}(y\mid\vec{\mathbf{x}})$ ，则自动地确定了一个代价函数 $-\log p_{model}(y\mid\vec{\mathbf{x}})$ 。
4. 代价函数的梯度必须足够大且能够计算。
   - 如果代价函数非常平缓，则代价函数的梯度非常小。若梯度很小甚至消失，会导致求解模型参数的迭代过程无法推进。
   - 如果代价函数太大导致发生上溢出时，数值计算会出现问题。用负的对数似然函数作为代价函数可以避免这个问题。
5. 均方误差和平均绝对误差这两种代价函数，在使用基于梯度的优化方法时，经常会产生非常小的梯度。

   这也是使用负的对数似然函数作为代价函数的一个重要原因。

三、输出单元

1. 代价函数的选取和输出单元的类型紧紧相关。
2. 任何类型的输出单元，也可以用作隐单元。

3.1 线性输出单元

1. 最简单的输出单元为线性单元：它基于仿射变换，不包含非线性。
   - 给定特征 $\vec{\mathbf{h}}$，单个线性输出单元的输出为： $\hat{y} = \vec{\mathbf{w}}^T\vec{\mathbf{h}}+b$ 。

- 若输出层包含多个线性输出单元，则线性输出层的输出为： $\hat{\vec{y}}=\mathbf{W}^T\vec{\mathbf{h}}+\vec{\mathbf{b}}$ 。

2. 线性输出层经常用于学习条件高斯分布的均值： $p(y\mid\vec{\mathbf{x}})=\mathcal{N}(y;\hat{y},\mathbf{I})$。

   给定 $\vec{\mathbf{x}}$ 的条件下， y 的分布为均值为 $\hat{y}$、方差为 `1` 的高斯分布。此时：最大化对数似然函数等价于最小化均方误差。

3. 最大似然准则也可以用于学习高斯分布的协方差矩阵。但是由于协方差矩阵的特点（对称的、正定的），因此用线性输出层来描述这种限制是困难的。所以通常采用其他类型的输出单元来学习协方差矩阵。

4. 线性模型不会饱和，因此可以方便的使用基于梯度的优化算法。

3.2 sigmoid 输出单元

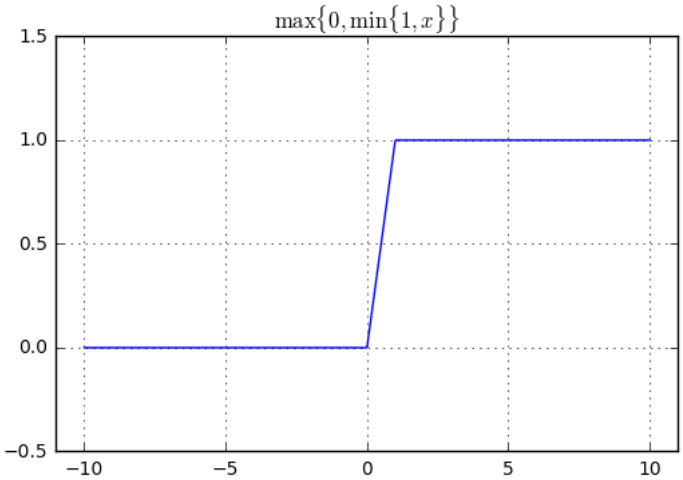
1. `sigmoid` 单元：用于 `Bernoulli` 分布的输出。

2. 二类分类问题可以用伯努利分布来描述。由于伯努利分布只需要一个参数来定义，因此神经网络只需要预测 $P(y=1\mid\vec{\mathbf{x}})$，它必须位于区间 `[0,1]` 之间。

   - 一种方案是采用线性单元，但是通过阈值来使它位于 `[0,1]` 之间：

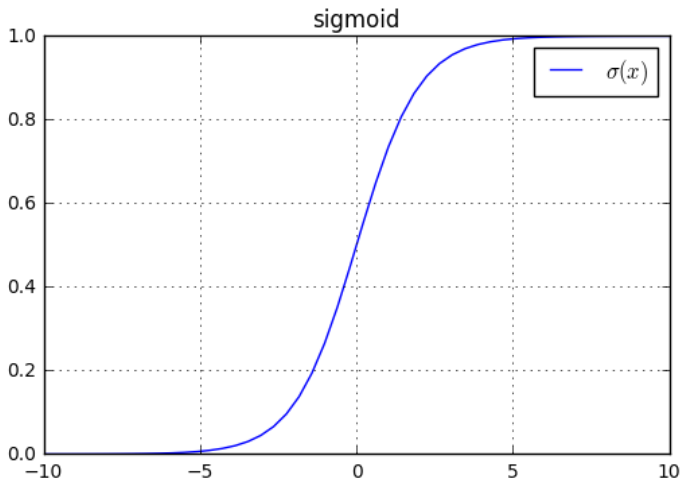
     $$P(y=1\mid\vec{\mathbf{x}})=\max\{0,\min\{1,\vec{\mathbf{w}}^T\vec{\mathbf{h}}+b\}\}$$

     令 $z=\vec{\mathbf{w}}^T\vec{\mathbf{h}}+b$，则上式右侧就是函数 $\max\{0,\min\{1,z\}\}$，函数图象如下。

     该函数有个问题：当 z 位于 `[0,1]` 之外时，模型的输出 $P(y=1\mid\vec{\mathbf{x}})$ 对于 z 的梯度都为 0。根据反向传播算法，此时 $P(y=1\mid\vec{x})$ 对于参数 $\vec{\mathbf{w}}$ 和参数 b 的梯度都为零。从而使得梯度下降算法难以推进。

   - 另一种方案就是采用 `sigmoid` 单元： $P(y=1\mid\vec{\mathbf{x}})=\sigma(\vec{\mathbf{w}}^T\vec{\mathbf{h}}+b)$ ，其中 $\sigma(\cdot)$ 就是 `sigmoid` 函数。

虽然 `sigmoid` 函数也存在饱和的问题，但是它比 $\max\{0,\min\{1,z\}\}$ 要稍微缓解。

3. `sigmoid` 输出单元有两个部分：首先它用一个线性层来计算 $z=\vec{\mathbf{w}}^T\vec{\mathbf{h}}+b$；然后它使用 `sigmoid` 激活函数将 z 转化成概率。

   根据：

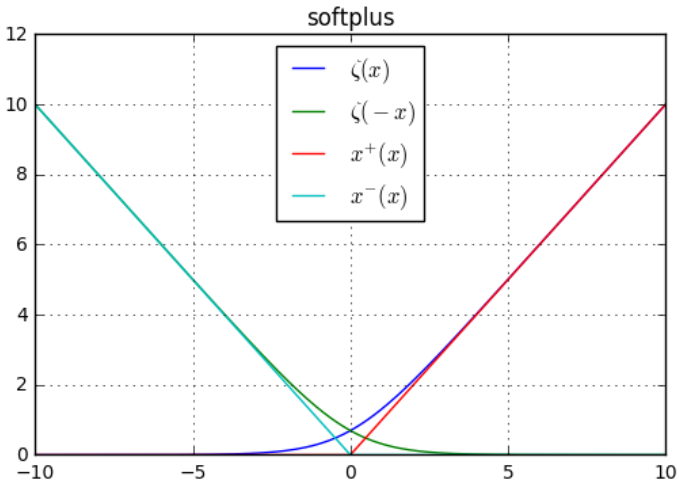
$$P(y=1\mid\vec{\mathbf{x}})=\sigma(z)=\frac{\exp(z)}{\exp(z)+\exp(0)}$$
$$P(y=0\mid\vec{\mathbf{x}})=1-\sigma(z)=\frac{\exp(0)}{\exp(z)+\exp(0)}$$

   则有：$P(y\mid\vec{\mathbf{x}})=\frac{\exp(yz)}{\sum_{y'}\exp(y'z)},y\in\{0,1\}$。即：$P(y\mid\vec{\mathbf{x}})=\sigma((2y-1)z),y\in\{0,1\}$

4. `sigmoid` 单元的代价函数通常采用负的对数似然函数：

$$J(\vec{\theta})=-\log P(y\mid\vec{\mathbf{x}})=-\log\sigma((2y-1)z)=\zeta((1-2y)z),\ \text{for } y\in\{0,1\}$$

   其中 $\zeta(x)=\log(1+\exp(x))$，它是函数 $x^+=\max(0,x)$ 的一个近似。



   可以看到，只有当 $(1-2y)z$ 取一个非常大的负值时，代价函数才非常接近于0。因此代价为0发生在：

   - $y=1$ 且 z 为一个较大的正值，此时表示正类分类正确。
   - $y=0$ 且 z 为一个较大的负值，此时表示负类分类正确。

   当 z 符号错误时（即 z 为负数，而 $y=1$；或者 z 为正数，但是 $y=0$），$(1-2y)z=|z|$，则 `softplus` 函数会渐进地趋向于 $|z|$，且其梯度不会收缩。这意味着基于梯度的学习可以很快地改正错误的 z。

5. 当使用其他代价函数时（如均方误差），代价函数会在任何 $\sigma(z)$ 饱和时饱和，此时梯度会变得非常小从而无法学习。

   因此最大似然函数总是训练 `sigmoid` 输出单元的首选代价函数。

3.3 softmax 输出单元

1. `softmax` 单元：用于 `multinoulli` 分布的输出。
2. 当表示一个具有 n 个可能取值的离散型随机变量分布时，可以采用 `softmax` 函数。它可以视作 `sigmoid` 函数的扩展：

$$\vec{\mathbf{z}}=\mathbf{W}^T\vec{\mathbf{h}}+\vec{\mathbf{b}}$$

$$\hat{y}_i=P(y=i\mid\vec{\mathbf{x}})=\text{softmax}(\vec{\mathbf{z}})_i=\frac{\exp(z_i)}{\sum_j\exp(z_j)},\quad i=1,2,\cdots,K$$

$$\hat{\vec{\mathbf{y}}}=(\hat{y}_1,\hat{y}_2,\cdots,\hat{y}_K)^T$$

$\hat{y}_i$ 表示类别为 i 的概率。

3. 当所有输入都加上一个相同常数时，`softmax`的输出不变。即：$\text{softmax}(\vec{\mathbf{z}})=\text{softmax}(\vec{\mathbf{z}}+c)$ 。

   根据该性质，可以导出一个数值稳定的`softmax`函数的变体：

   $$\text{softmax}(\vec{\mathbf{z}})=\text{softmax}(\vec{\mathbf{z}}-\max_i z_i)$$

4. `softmax`函数是`argmax`函数的软化版本，而不是`max`函数的软化版本。
   - `argmax`函数的结果为一个独热向量（只有一个元素为1，其余元素都是0），且不可微。
   - `softmax`函数是连续可微的。

     当某个输入最大($z_i=\max_i z_i$)，且 z_i 远大于其他的输入时，对应的位置输出非常接近 1 ，其余的位置的输出非常接近 0 。
   - `max`函数的软化版本为 $\text{softmax}(\vec{\mathbf{z}})^T\vec{\mathbf{z}}$ 。
5. 假设真实类别为 k ，则`softmax`输出的对数似然函数为： $\log\text{softmax}(\vec{\mathbf{z}})_k=z_k-\log\sum_j\exp(z_j)$ 。

   其中：第一项 z_k 不会饱和（它的梯度不会为零），第二项近似为 $\max_j z_j$ 。

   为了最大化对数似然函数：第一项鼓励 z_k 较大，第二项鼓励所有的 z_j 较小。此时意味着：若真实类别为 k ，则 z_k 较大，其它的 z_j 较小。
6. 基于对数似然函数的代价函数为： $-\log\text{softmax}(\vec{\mathbf{z}})_k=\log\sum_j\exp(z_j)-z_k$ 。

   因此代价函数惩罚那个最活跃的预测（最大的 z_j ）。如果 $z_k=\max_j z_j$，则代价函数近似为零。
7. 当输入是绝对值较小的负数时， $\text{softmax}(\vec{\mathbf{z}})_i$ 的计算结果可能为 0 。此时 $\log\text{softmax}(\vec{\mathbf{z}})$ 趋向于负无穷，非数值稳定的。

   因此需要设计专门的函数来计算 $\log\text{softmax}$ ，而不是将 softmax 的结果传递给 $\log$ 函数。
8. 除了负对数似然，其他的许多代价函数对`softmax`函数不适用（如均方误差代价函数）。

   `softmax` 函数将在很多情况下饱和，饱和意味着梯度消失，而梯度消失会造成学习困难。

   `softmax`函数饱和时，此时基于`softmax`函数的代价函数也饱和；除非它们能将`softmax`转化为成其它形式，如对数形式。
9. `softmax` 函数饱和的一般化形式：对于`softmax`函数，它有多个输出值；当输入值之间的差异较大时，某些输出值可能饱和。

   当某个输入最大($z_i=\max_i z_i$)，且 z_i 远大于其他的输入时：$\text{softmax}(\vec{\mathbf{z}})_i$ 将饱和到 1 ，$\text{softmax}(\vec{\mathbf{z}})_j, j\neq i$ 将饱和到 0 。

3.4 其他输出单元

1. 任何其他类型的输出单元都可以应用到神经网络，这些输出单元通常使用负的对数似然作为代价函数。

   如果定义了一个条件分布 $p_{model}(y\mid\vec{\mathbf{x}};\vec{\theta})$，则最大似然准则建议使用 $-\log p_{model}(y\mid\vec{\mathbf{x}};\vec{\theta})$ 作为代价函数。

四、隐单元

1. 隐单元的设计是一个非常活跃的研究领域，并且目前还没有明确的指导性理论，难以决定何时采用何种类型的隐单元。
   - 通常不能预先判断哪种类型的隐单元工作的最好，所以设计过程中需要反复试错，通过测试集评估其性能来选择合适的隐单元。
   - 一般默认采用的隐单元是修正线性单元，但是仍然有许多其他类型的隐单元。
2. 某些隐单元可能并不是在所有的输入上都是可微的。如：修正线性单元 $g(z)=\max\{0,z\}$ 在 $z=0$ 处不可微，这使得在该点处梯度失效。

   事实上梯度下降法对这些隐单元的表现仍然足够好，原因是：
   - 神经网络的训练算法通常并不会达到代价函数的局部最小值，而仅仅是显著地降低代价函数的值即可。

     因此实际上训练过程中一般无法到达代价函数梯度为零的点，所以代价函数取最小值时梯度未定义是可以接受的。
   - 不可微的点通常只是在有限的、少数的点上不可微，在这些不可微的点通常左导数、右导数都存在。

     神经网络训练的软件实现通常返回左导数或者右导数其中的一个，而不是报告导数未定义或者产生一个错误。

     这是因为计算机计算 0 点的导数时，由于数值误差的影响实际上不可能计算到理论上 0 点的导数，而是一个微小的偏离：向左侧偏离就是左导数，向右侧偏离就是右导数。
3. 大多数隐单元的工作都可以描述为下面三步：
   - 接受输入向量 $\vec{\mathbf{x}}$ 。
   - 计算仿射变换 $z=\vec{\mathbf{w}}^T\vec{\mathbf{x}}+b$ 。
   - 用非线性函数 $g(z)$ 计算隐单元输出。

     函数 $g(z)$ 也称作激活函数，大多数隐单元的区别仅仅在于激活函数 $g(z)$ 的形式不同。
4. 神经网络的隐层由多个隐单元组成，隐层的输出为： $g(\mathbf{W}^T\vec{\mathbf{x}}+\vec{\mathbf{b}})$ 。

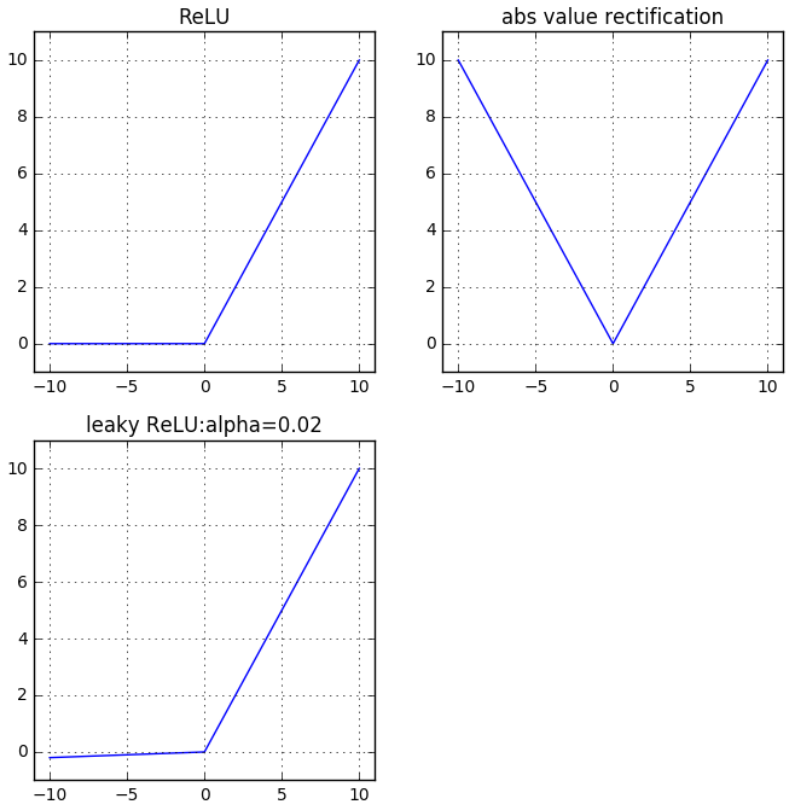
4.1 修正线性单元

1. 修正线性单元采用激活函数 $g(z)=\max\{0,z\}$ ，它和线性单元非常类似，区别在于：修正线性单元在左侧的定义域上输出为零。
   - 优点：采用基于梯度的优化算法时，非常易于优化。

     当修正线性单元处于激活状态时，导数为常数`1`；当修正线性单元处于非激活状态时，导数为常数`0`。修正线性单元的二阶导数几乎处处为零。
   - 缺点：无法通过基于梯度的方法学习那些使得修正线性单元处于非激活状态的参数，因为此时梯度为零。
2. 对于修正线性单元 $\vec{\mathbf{h}}=g(\mathbf{W}^T\vec{\mathbf{x}}+\vec{\mathbf{b}})$，初始化时可以将 $\vec{\mathbf{b}}$ 的所有元素设置成一个小的正值（如0.1），从而使得修正线性单元在初始时尽可能的对训练集中大多数输入呈现激活状态。
3. 有许多修正线性单元的扩展存在，这些扩展保证了它们能在各个位置都保持非零的梯度。

   大多数扩展的表现与修正线性单元相差无几，偶尔表现的更好。
4. 修正线性单元的三个扩展：当 $z<0$ 时，使用一个非零的斜率 α ：
   $h=g(z,\alpha)=\max(0,z)+\alpha\min(0,z)$ 。
   - 绝对值修正`absolute value rectification`： 使用 $a=-1$，此时 $g(z)=|z|$ 。
   - 渗透修正线性单元`leaky ReLU`：将 α 固定成一个类似 0.01 的小值。
   - 参数化修正线性单元`parametric ReLU`：将 α 作为学习的参数。

此时 α 是与训练集相关的。不同的训练集，学得的 α 会不同。



4.2 maxout 单元

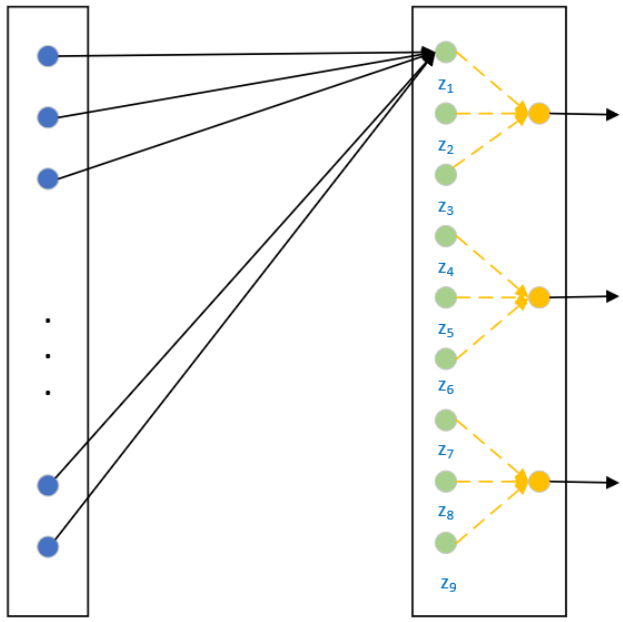
1. `ReLU` 单元及其扩展都基于一个原则：越接近线性，则模型越容易优化。采用更容易优化的线性，这种一般化的原则也适用于除了深度前馈网络之外的网络。
2. `maxout` 单元是修正线性单元的进一步扩展。`maxout` 单元并不是作用于 $\vec{\mathbf{z}}$ 的每个元素 z_i，而是将 $\vec{\mathbf{z}}$ 分成若干个小组，每个组有 k 个元素：

$$\begin{aligned}\mathbb{G}_1 &= \{z_1, z_2, \cdots, z_k\}\\ \mathbb{G}_2 &= \{z_{k+1}, z_{k+2}, \cdots, z_{2k}\}\\ &\vdots\\ \mathbb{G}_i &= \{z_{(i-1)k+1}, z_{(i-1)k+2}, \cdots, z_{ik}\}\\ &\vdots\end{aligned}$$

然后 `maxout` 单元对每个组输出其中最大值的元素：

$$g(\vec{\mathbf{z}})_i = \max_{z_j \in \mathbb{G}_i} z_j$$

- 假设 $\vec{\mathbf{z}}$ 是 100维的， $k = 20$，则 $g(\vec{\mathbf{z}})$ 为 5维的向量。
- 至于如何分组，则没有确定性的指导准则。



3. `maxout` 单元的 $\vec{\mathbf{z}}$ 通常是通过对输入 $\vec{\mathbf{x}}$ 执行多个仿射变换而来。

   设 $\vec{\mathbf{x}}=(x_1,\cdots,x_n)^T\in\mathbb{R}^n$ ，`maxout` 单元有 k 个分组，输出为 $y_1,\cdots,y_n$ ：

$$\begin{aligned}y_1&=\max(\vec{\mathbf{w}}_{1,1}^T\cdot\vec{\mathbf{x}}+b_{1,1},\cdots,\vec{\mathbf{w}}_{1,k}^T\cdot\vec{\mathbf{x}}+b_{1,k})\\&\vdots\\y_n&=\max(\vec{\mathbf{w}}_{n,1}^T\cdot\vec{\mathbf{x}}+b_{n,1},\cdots,\vec{\mathbf{w}}_{n,k}^T\cdot\vec{\mathbf{x}}+b_{n,k})\end{aligned}$$

   令

$$\mathbf{W}_1=\begin{bmatrix}\vec{\mathbf{w}}_{1,1}^T\\\vec{\mathbf{w}}_{2,1}^T\\\cdots\\\vec{\mathbf{w}}_{n,1}^T\end{bmatrix}\quad\cdots\quad\mathbf{W}_k=\begin{bmatrix}\vec{\mathbf{w}}_{1,k}^T\\\vec{\mathbf{w}}_{2,k}^T\\\cdots\\\vec{\mathbf{w}}_{n,k}^T\end{bmatrix}\quad\vec{\mathbf{b}}_1=\begin{bmatrix}b_{1,1}\\b_{2,1}\\\cdots\\b_{n,1}\end{bmatrix}\quad\cdots\quad\vec{\mathbf{b}}_k=\begin{bmatrix}b_{1,k}\\b_{2,k}\\\cdots\\b_{n,k}\end{bmatrix}$$

$$\vec{\mathbf{y}}=[y_1,y_2,\cdots,y_n]^T$$

   则有：$\vec{\mathbf{y}}=\max(\mathbf{W}_1\vec{\mathbf{x}}+\vec{\mathbf{b}}_1,\cdots,\mathbf{W}_k\vec{\mathbf{x}}+\vec{\mathbf{b}}_k)$ 。

4. `maxout` 单元的优点：
   - 接近线性，模型易于优化。
   - 经过 `maxout` 层之后，输出维数降低到输入的 $\frac{1}{k}$ 。这意味着下一层的权重参数的数量降低到 `maxout` 层的 $\frac{1}{k}$ 。
   - 由多个分组驱动，因此具有一些冗余度来抵抗遗忘灾难 `catastrophic forgetting` 。

     遗忘灾难指的是：网络忘记了如何执行它们过去已经训练了的任务。

5. 在卷积神经网络中， `max pooling` 层就是由 `maxout` 单元组成。

6. `maxout` 单元提供了一种方法来学习输入空间中的多维度线性分段函数。

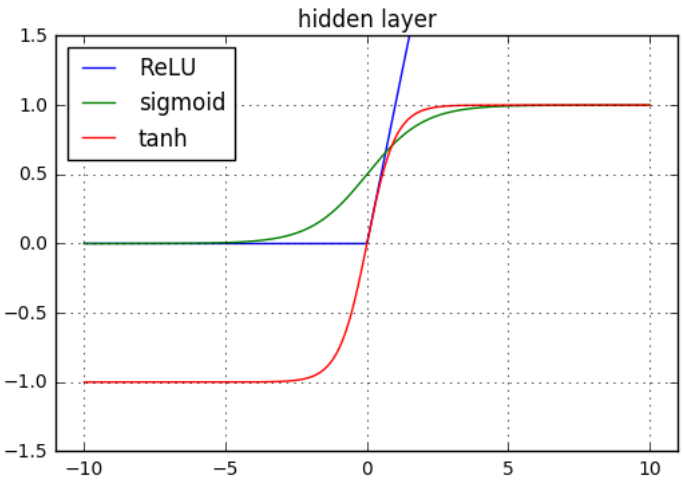
   它可以学习具有 k 段的线性分段的凸函数，因此 `maxout` 单元也可以视作学习激活函数本身。

7. 使用足够大的 k， `maxout` 单元能够以任意程度逼近任何凸函数。
    - 特别的， $k=2$ 时的 `maxout` 单元组成的隐层可以学习实现传统激活函数的隐层相同的功能，包括：`ReLU` 函数、绝对值修正线性激活函数、 `leakly ReLU` 函数、参数化 `ReLU` 函数，也可以学习不同于这些函数的其他函数。
    - 注意： `maxout` 层的参数化与 `ReLU` 层等等这些层不同（最典型的， `maxout` 层的输出向量的维数发生变化），因此即使 `maxout` 层实现了 `ReLU` 函数，其学习机制也不同。

4.3 sigmoid / tanh 单元

1. `sigmoid` 单元和 `tanh` 单元：其激活函数分别为 `sigmoid` 函数和 `tanh` 函数。
2. 在引入修正线性单元之前，大多数神经网络使用 `sigmoid` 函数 $g(z)=\sigma(z)$，或者双曲正切函数 $g(z)=\tanh(z)$ 作为激活 函数。这两个激活函数密切相关，因为 $\tanh(z)=2\sigma(2z)-1$ 。

   与修正线性单元不同， `sigmoid` 单元和 `tanh` 单元在其大部分定义域内都饱和，仅仅当 z 在 0 附近才有一个较高的梯度，这会使得基于梯度的学习变得非常困难。因此，现在不鼓励将这两种单元用作深度前馈网络中的隐单元。
    - 如果选择了一个合适的代价函数（如对数似然函数）来抵消了 `sigmoid` 的饱和性，则这两种单元可以用作输出单元（而不是隐单元）。
    - 如果必须选用 `sigmoid` 激活函数时， `tanh` 激活函数通常表现更佳。因为 `tanh` 函数在 0点附近近似于单位函数 $g(z)=z$ 。



3. `sigmoid` 激活函数在前馈网络之外的神经网络中更为常见。

   有一些网络不能使用修正线性单元，因此 `sigmoid` 激活函数是个更好的选择，尽管它存在饱和问题。
    - 循环神经网络：修正线性单元会产生信息爆炸的问题。
    - 一些概率模型：要求输出在 `0~1` 之间。

4.4 其他隐单元

1. 存在许多其他种类的隐单元，但它们并不常用。经常会有研究提出一些新的激活函数，它们是前述的标准激活函数的变体，并表示它们表现得非常好。

   除非这些新的激活函数被明确地证明能够显著地改进时，才会发布这些激活函数。如果仅仅与现有的激活函数性能大致相当，则不会引起大家的兴趣。
2. 隐单元的设计仍然是个活跃的研究领域，许多有效的隐单元类型仍有待挖掘。
3. 线性隐单元：它完全没有激活函数 $g(z)$，也可以认为是使用单位函数 $g(z)=z$ 作为激活函数 。

- 如果网络的每一层都是由线性变换组成，则网络作为整体也是线性的。这会降低网络的表达能力，因此线性隐单元较少使用。
- 线性隐单元提供了一种减少网络中参数的数量的有效方法。

  假设有一个隐层，它有 n 个输入， p 个输出，隐向量为 $\vec{\mathbf{h}}=g(\mathbf{W}^T\vec{\mathbf{x}}+\vec{\mathbf{b}})$ ，则 $\mathbf{W}$ 为 $n\times p$ 维的矩阵，需要 np 个参数。

  可以用两层来代替该层：第一层使用权重矩阵 $\mathbf{U}$，且没有激活函数；第二层使用权重矩阵 $\mathbf{V}$，且为常规激活函数。这对应了 $\mathbf{W}$ 的因式分解： $\vec{\mathbf{h}}=g(\mathbf{V}^T\mathbf{U}^T\vec{\mathbf{x}}+\vec{\mathbf{b}})$ 。

  假设第一层产生了 q 个输出，则有：第一层的输入为 n 个，输出为 q 个；第二层的输入为 q 个，输出为 p 个。

  整体参数为 $(n+p)q$ 个。当 q 很小时，可以满足 $(n+p)q<np$，从而减少了参数。

  事实上第一层的输出 $\vec{\mathbf{h}}_1=\mathbf{U}^T\vec{\mathbf{x}}$ 实际上对应着对输入 $\vec{\mathbf{x}}$ 的一个编码（或者称作降维）：将 n 维的向量压缩到 q 维。如果 q 非常小，则输入 $\vec{\mathbf{x}}$ 信息压缩的过程中会丢失大量的信息。

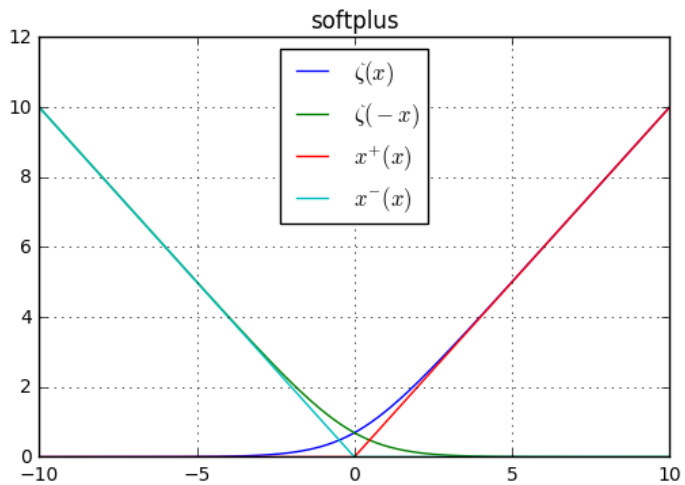
4. `softmax` 隐单元：激活函数为 `softmax` 函数。
   - `softmax` 单元既可以用作输出单元，也可以用作隐单元。
   - `softmax` 单元可以很自然地表示具有 k 个可能取值的离散型随机变量的概率分布，因此它也可以视作一种开关。
5. 径向基函数隐单元：激活函数为径向基函数 `RBF` ：

$$h_i=\exp\left(-\frac{1}{\sigma_i^2}||\mathbf{W}_{:,i}-\vec{\mathbf{x}}||^2\right)$$

   其中 $\mathbf{W}_{:,i}$ 表示权重矩阵的第 i 列。

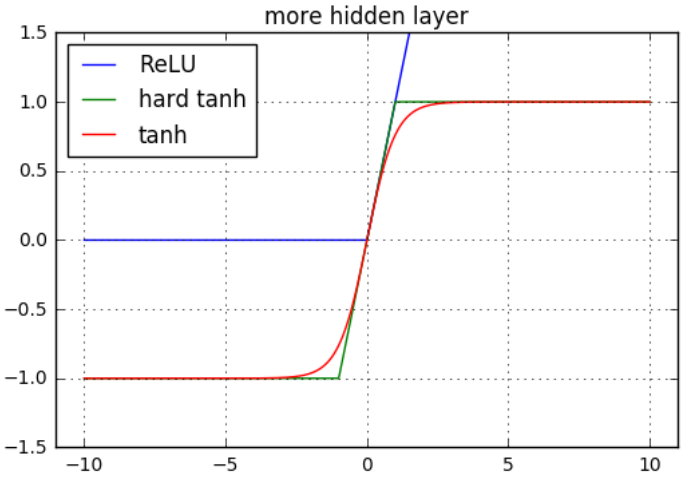
   径向基函数在 $\vec{\mathbf{x}}$ 接近 $\mathbf{W}_{:,i}$ 时非常活跃，但是对于大部分 $\vec{\mathbf{x}}$ 该函数值都饱和到 0，因此难以优化。
6. `softplus` 隐单元：激活函数为 `softplus` 函数： $g(z)=\zeta(z)=\log(1+e^z)$ 。这种单元是修正线性单元的平滑版本。

   通常不鼓励使用 `softplus` 激活函数，而推荐使用修正线性单元。理论上 `softplus` 函数处处可导，且它不完全饱和，所以人们希望它具有比修正线性单元更好的性能。但是根据经验来看，修正线性单元效果更好。



7. 硬双曲正切隐单元：激活函数为硬双曲正切函数： $g(z)=\max(-1,\min(1,z))$ 。

   它的形状和 `tanh` 以及修正线性单元类似，但是它是有界的且在 $z=1,-1$ 点不可导。



4.5 激活函数对比

1. `sigmoid` 主要缺点：
   - 容易饱和从而使得梯度消失。当激活函数取值在接近0或者1时会饱和，此时梯度为近似为0。
   - 函数输出不是零中心的。这会导致后续神经元的输出数值总是正数。
2. `tanh` ：
   - 优点：函数输出是零中心的。
   - 缺点：容易饱和从而使得梯度消失。

   `tanh` 激活函数几乎在所有场合都是优于 `sigmoid` 激活函数的。但是有一种情况例外：如果要求函数输出是 `0~1` 之间（比如表征某个概率），则二者之间必须用 `sigmoid` 。
3. `relu` ：
   - 优点：对随机梯度下降的收敛有巨大的加速作用，而且非常容易计算。
   - 缺点：可能导致神经元死掉。

     当一个很大的梯度流过 `relu` 神经元时，可能导致梯度更新到一种特别的状态：在这种状态下神经元无法被其他任何数据点再次激活。此后流过这个神经元的梯度将变成 0，该单元在训练过程中不可逆的死亡。

     如果学习率设置的过高，可能会发现网络中大量神经元都会死掉。整个训练过程中，这些神经元都不会被激活。
4. `leaky relu` ：为了解决 `relu` 死亡神经元的问题的尝试，但是效果并不明显。
5. `maxout` ： 它是 `relu` 和 `leaky relu` 的一般性归纳，以单输出的两段为例：

$$\max(\vec{\mathbf{w}}_1^T\cdot\vec{\mathbf{x}}+b_1,\vec{\mathbf{w}}_2^T\cdot\vec{\mathbf{x}}+b_2)$$

   - 当分成两段且 $b_1=0,\vec{\mathbf{w}}_1=\vec{\mathbf{0}}$ 时，就退化为单输出的 `relu` 激活函数。
   - `maxout` 单元拥有 `relu` 单元的所有优点，而没有它的缺点。
   - `maxout` 单元的缺点：

     对于 k 段的 `maxout` ， 假设其输出为 n 维的，输出为 $y_1,\cdots,y_n$ ，则有：
     $\vec{\mathbf{y}}=\max(\mathbf{W}_1\vec{\mathbf{x}}+\vec{\mathbf{b}}_1,\cdots,\mathbf{W}_k\vec{\mathbf{x}}+\vec{\mathbf{b}}_k)$ 。

     相对于 `relu` 单元 $\max(\vec{\mathbf{0}},\mathbf{W}\vec{\mathbf{x}}+\vec{\mathbf{b}})$， 可以看到： `maxout` 单元的参数数量增加到 k 倍。

> 注意，前面提到：`maxout` 下一层的权重参数的数量降低到`maxout`层的 $\frac{1}{k}$ 。 而这里说明的是：`maxout` 单元本层的参数数量是`relu` 的 k 倍。二者不矛盾。

6. 在同一个网络中混合使用不同类型的激活函数非常少见，虽然没有什么根本性的问题来禁止这么做。
7. 激活函数选取准则：
   - 通常建议使用 `relu` 激活函数。
   - 注意设置好学习率，同时监控网络中死亡的神经元占比。

     如果神经元死亡比例过高，则使用 `leaky relu` 或者 `maxout` 激活函数。

五、结构设计

1. 神经网络的结构指的是：神经网络有多少个单元、这些单元如何连接。

   理想的网络结构必须根据具体任务反复试验，并评估验证集的误差来得到。
2. 大多数神经网络被组织成以层为单位，然后层级之间为链式结构。每一层都是前一层的函数，如：第一层为 $\vec{\mathbf{h}}_1 = g_1(\mathbf{W}_1^T\vec{\mathbf{x}} + \vec{\mathbf{b}}_1)$、第二层为 $\vec{\mathbf{h}}_2 = g_2(\mathbf{W}_2^T\vec{\mathbf{h}}_1 + \vec{\mathbf{b}}_2)$、...
3. 链式结构中，主要的结构考虑是：网络的深度（一共多少层）、每一层的宽度（每一层的输出向量的维数）。
   - 即使只有一层隐层的网络也能够适应训练集。
   - 对于更深层的网络，每一层可以使用少得多的单元和参数，并且对于测试集的泛化性能更好。

5.1 通用近似定理

1. 通用近似定理`universal approximation theorem` 表明：

   对于一个具有线性输出层，和至少一层具有任何一种“挤压”性质的激活函数（如`sigmoid`激活函数）的隐层的深度前馈神经网络，只要给予网络足够数量的隐单元，它可以以任意精度来近似任何一个从有限维空间到有限维空间的`Borel`可测函数。前馈神经网络的导数也可以以任意精度来逼近被近似函数的导数。
   - 定义在 $\mathbb{R}^n$ 的有界闭集上的任意连续函数是 `Borel` 可测的，因此可以用神经网络来近似。
   - 神经网络也可以近似这样的任意一个函数：从有限维离散空间映射到另一个有限维离散空间。
   - 虽然原始定理要求特殊的激活函数（该激活函数要求：当自变量的绝对值非常大时，函数饱和），但是后来证明通用近似定理也适用于更广泛的激活函数，其中包括常用的修正线性单元。
2. 通用近似定理表明：无论试图学习什么样的函数，一个很大的深度前馈网络一定能够表示这个函数。

   实际上不能保证训练算法能够学得这个函数，学习可能因为两个不同的原因而失败：
   - 用于训练的优化算法可能找不到合适的参数值来逼近该函数，即：优化算法缩小了模型的有效容量，使得该函数不在有效的解空间内。
   - 训练算法可能由于过拟合而选择了错误的函数。
3. 通用近似定理说明了存在一个足够大的网络能够以任意精度逼近任意函数，但是定理并没有说这个网络有多大。最坏的情况下可能需要指数级的隐单元。
   - 具有单隐层的深度前馈网络足以表示任何函数，但是隐层可能过于庞大以至于无法正确地学习和泛化。

     使用更深的深度前馈网络可以减少所需的单元的数量，并且减少泛化误差。
   - 很多情况下，浅层模型所需的隐单元的数量是输入向量的维数 n 的指数级。
4. 修正线性网络（激活函数都是`relu` 函数）描述的线性区域的数量为输入向量维数 n 的指数级，也是网络深度 l 的指数级。

5.2 网络结构

1. 任何时候当选择一个特定的机器学习算法时，隐含地给定了一个先验知识：算法应该学得什么样的函数。

   选择深度模型则给定了一个先验知识：待学的函数应该是几个更加简单的函数的组合。这意味着：待学习的问题包含一组潜在的因子，这些因子可以根据更简单的潜在因子描述。

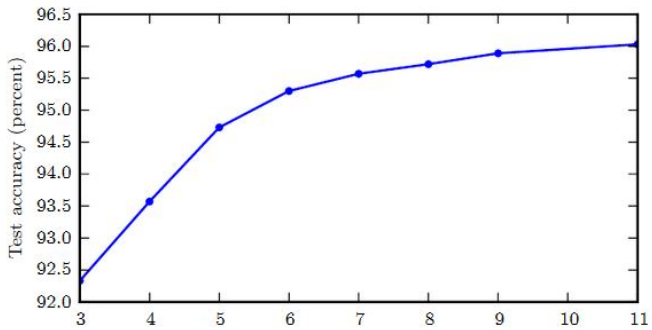
2. 试验表明：更深的网络泛化性能更好，而仅仅增加参数的数量对于泛化性能几乎没有不起作用。

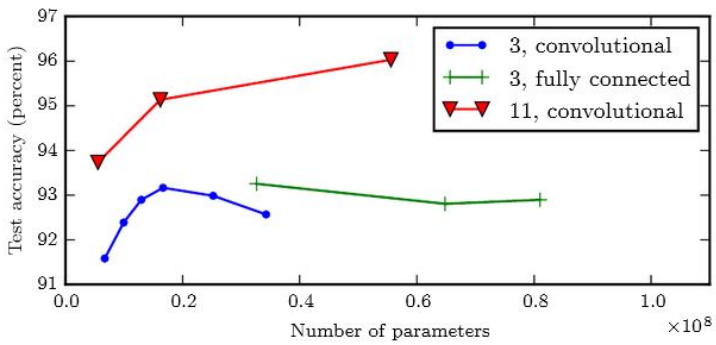
   - 虽然一个2层网络在数学理论上能完美近似所有连续函数，但实际操作中效果相对较差。

     就实践经验而言，深度网络效果比单层效果好。

   - 不要因为担心出现过拟合而使用小网络，而要尽可能使用大网络，然后使用正则化技术来控制过拟合。

   下图依次表示：不同网络层数的网络的泛化性能（测试集的准确率）、不同参数数量和网络层数的网络的泛化性能。





3. 前面介绍的前馈神经网络都是简单的以层为单位的链式结构，主要考虑网络的深度和每层的宽度。实践中的神经网络具有相当的多样性。

   - 卷积神经网络是另一种特殊的结构。
   - 有时候，层不需要位于链中。

     有的神经网络构（如`ResNet`）建了一条主链，然后又添加了额外的结构。如从层 i 连接到层 $i+2$，这使得梯度更容易地从输出层流向输入端。

4. 除了深度与每一层的宽度之外，结构设计考虑的另一个因素是：如何将层与层之间连接起来。

   - 默认的神经网络使用矩阵 $\mathbf{W}$ 给出的线性变换来描述层之间的连接。此时对于本层的每一个单元，其输入为上一层的所有输出。
   - 某些特殊的网络使用更少的连接，上一层的输出只是连接到本层的部分单元。这种策略减少了参数的数量，但是依赖于具体问题。
   - 很难对于通用的神经网络结构给出有效的建议。通常会给出一些特殊的结构，可以在不同的领域工作良好。如：`CNN`在图像识别领域工作很好。

六、历史小记

1. 现代前馈网络的核心思想自20世纪80年代以来没有发生重大变化。

   近年来神经网络性能的大部分改变可归因于两个因素：更大的数据集、更大的网络（由于硬件的强大和软件设施的发展）。

   算法上的部分改变也显著改善了神经网络的性能：

   - 用交叉熵代替均方误差作为损失函数。

     均方误差在20世纪80年代和90年代流行，后来逐渐被交叉熵代替。交叉熵大大提高了`sigmoid`输出单元和`softmax`输出单元的模型的性能。

   - 使用分段线性隐单元（如修正线性单元）来代替`sigmoid`隐单元。

2. 修正线性单元描述了生物神经元的这些特性：
   - 对于某些输入，生物神经元是完全不活跃的。
   - 对于某些输入，生物神经元的输出和输入成比例。
   - 大多数时间，生物神经元位于不活跃的状态。
3. 2006-2012年，人们普遍认为：前馈神经网络如果没有其他模型的辅助，则表现不佳。现在已经知道：当具备合适的资源和工程实践，前馈网络表现的非常好。
4. 前馈网络中基于梯度的学习被用作研究概率模型的工具，它也可以应用于许多其他机器学习任务。
5. 在 2006年，业内使用无监督学习来支持监督学习；目前更常见的是使用监督学习来支持无监督学习。