

第五章 深度前馈网络

5.1 实例：学习异或 XOR

异或问题：XOR 函数（“异或”逻辑）是两个二进制值 x_1 和 x_2 的运算。只有当两个值不同才会返回 1，其他情况为 0。

目的就是学习目标函数 $y = f^*(\mathbf{x})$ 得到正确的输出。通过建立一个模型 $y = f(\mathbf{x}; \boldsymbol{\theta})$ 并通过学习算法不断优化参数 $\boldsymbol{\theta}$ 使得 f 尽可能逼近 f^* 。

按照之前的方法，可以将该问题看做是回归问题，然后使用均方误差损失函数

$$J = \frac{1}{4} \sum_{\mathbf{x} \in \mathbb{X}} (f^*(\mathbf{x}) - f(\mathbf{x}; \boldsymbol{\theta}))^2$$

如果选择线性模型，则有

$$f(\mathbf{x}, \boldsymbol{\omega}, b) = \mathbf{x}^T \boldsymbol{\omega} + b$$

通过正规方程求解可得 $\boldsymbol{\omega} = 0$ 和 $b = \frac{1}{2}$ 。这个线性模型在任意一点都会输出 0.5，由图 42 可知，线性模型无法表示异或问题。

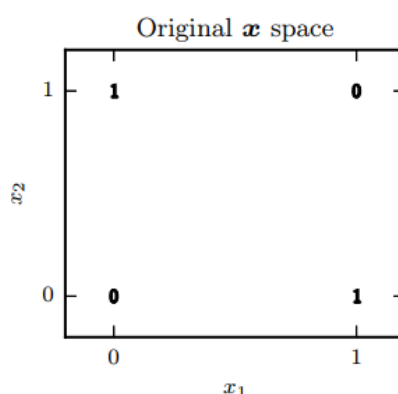


图 42 异或问题

这里可以引入一个简单的前馈神经网络，该网络中有一隐藏层并且隐藏层中包含两个单元，如图 43 所示。这个前馈网络有一个通过函数 $f^{(1)}(\mathbf{x}, \mathbf{W}, \mathbf{c})$ 计算得到的隐藏单元

的向量 \mathbf{h} 。这些隐藏单元的值随后被用作第二层的输入。第二层就是这个网络的输出层。输出层仍然只是一个线性回归模型，只不过现在它作用于 \mathbf{h} 而不是 \mathbf{x} 。网络现在包含链接在一起的两个函数： $\mathbf{h} = f^{(1)}(\mathbf{x}, \mathbf{W}, \mathbf{c})$ 和 $y = f^{(2)}(\mathbf{h}, \mathbf{w}, b)$ ，完整的模型是 $f(\mathbf{x}, \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x}))$ 。

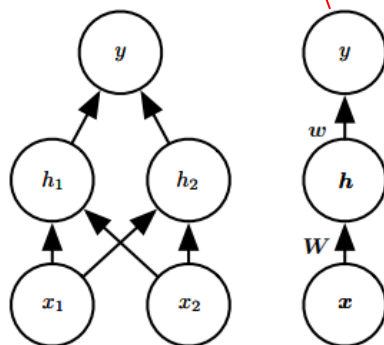


图 43 前馈网络结构（两种表达）

注意，这里的 $f^{(1)}$ 不能是线性函数，否则前馈网络作为一个整体对于输入仍然是线性的。假设 $f^{(1)}(\mathbf{x}) = \mathbf{W}^T \mathbf{x}$ 且 $f^{(2)}(\mathbf{h}) = \mathbf{h}^T \mathbf{w} = \mathbf{w}^T \mathbf{h}$ ，则 $f(\mathbf{x}) = \mathbf{w}^T \mathbf{W}^T \mathbf{x} = \mathbf{x}^T (\mathbf{W} \mathbf{w}) = \mathbf{x}^T \mathbf{w}'$ 。

现代网络中推荐使用的是由激活函数 $g(z) = \max\{0, z\}$ 定义的整流线性单元（rectified linear unit, ReLU），如图 44 所示。

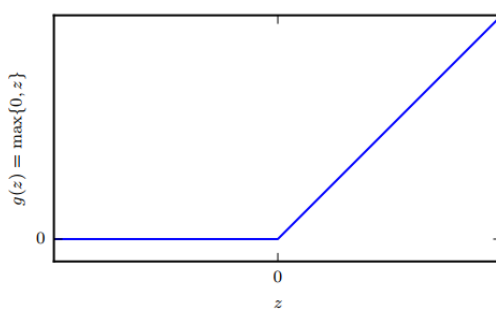


图 44 整流线性激活函数

我们现在可以指明我们的整个网络是

$$f(\mathbf{x}, \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^T \max\{0, \mathbf{W}^T \mathbf{x} + \mathbf{c}\} + b$$

令

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

以及 $\mathbf{b}=0$.

输入矩阵表示为:

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

第一步为输入矩阵乘以第一层的权重矩阵并加上偏置向量 \mathbf{c}

$$\mathbf{XW} + \mathbf{c} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

对上面结果进行整流线性变换得到了在 \mathbf{h} 空间上的坐标

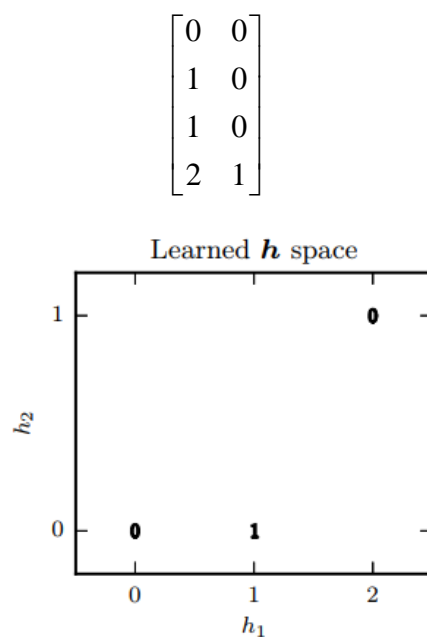


图 45 学习到的 \mathbf{h} 空间

最后乘以权重向量 \mathbf{w} 得

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

5.2 基于梯度的学习

5.2.1 代价函数

大多数情况下，参数模型定义为一个分布 $p(\mathbf{y}|\mathbf{x};\boldsymbol{\theta})$ ，并且简单地使用最大似然原理，这意味着使用训练数据模型预测间的交叉熵作为代价函数。

5.2.1.1 使用最大似然学习条件分布

大多数现代的神经网络使用最大似然来训练。这意味着代价函数就是负的对数似然，它与训练数据和模型分布间的交叉熵等价。这个代价函数表示为

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

贯穿神经网络设计的一个反复出现的主题是代价函数的梯度必须足够的大和具有足够的预测性，来为学习算法提供一个好的指引。饱和（变得非常平）的函数破坏了这一目标，因为它们把梯度变得非常小。这在很多情况下都会发生，因为用于产生隐藏单元或者输出单元的输出的激活函数会饱和。负的对数似然帮助我们在很多模型中避免这个问题。很多输出单元都会包含一个指数函数，这在它的变量取绝对值非常大的负值时会造成饱和。负对数似然代价函数中的对数函数消除了某些输出单元中的指数效果。

5.2.1.2 学习条件统计量

感觉不太重要，略。

5.2.2 输出单元

代价函数的选择与输出单元的选择紧密相关。大多数时候，我们简单地使用数据分布和模型分布间的交叉熵。选择如何表示输出决定了交叉熵函数的形式。任何可用作输出的神经网络单元，也可以被用作隐藏单元。这里，我们着重讨论将这些单元用作模型输出时的情况。

5.2.2.1 用于高斯输出分布的线性单元

一种简单的输出单元是基于仿射变换的输出单元，仿射变换不具有非线性。这些单元往往被直接称为线性单元。

给定特征 \mathbf{h} ，线性输出单元层产生一个向量 $\mathbf{y} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$ 。

线性输出层经常被用来产生条件高斯分布的均值：

$$p(\mathbf{y} | \mathbf{x}) = \mathcal{N}(\mathbf{y}; \mathbf{y}, \mathbf{I})$$

最大化其对数似然此时等价于最小化均方误差。

5.2.2.2 用于 Bernoulli 输出分布的 sigmoid 单元

许多任务需要预测二值型变量 y 的值。具有两个类的分类问题可以归结为这种形式。

此时最大似然的方法是定义 y 在 \mathbf{x} 条件下的 Bernoulli 分布。

Bernoulli 分布仅需单个参数来定义。神经网络只需要预测 $P(y=1|\mathbf{x})$ 即可。为了使这个数是有效的概率，它必须处在区间 $[0,1]$ 中。

基于使用 sigmoid 输出单元结合最大似然的方法可以保证无论何时模型给出了错误的答案时，总能有一个较大的梯度。

可以认为 sigmoid 输出单元具有两个部分。首先，它使用一个线性层来计算 $\mathbf{z} = \mathbf{w}^T \mathbf{h} + \mathbf{b}$ 。接着，它使用 sigmoid 激活函数将 \mathbf{z} 转化成概率。

5.2.2.3 用于 Multinoulli 输出分布的 softmax 单元

softmax 函数的形式为

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

使用最大化对数似然训练 softmax 来输出目标值 y

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

<https://peterroelants.github.io/posts/cross-entropy-softmax/>

以下是使用 softmax 函数和交叉熵实现多分类的具体过程。

(1) Softmax 函数

softmax 函数以 c 维向量 \mathbf{z} 为输入，输出 0 到 1 的 c 维实数向量 \mathbf{y} ，该函数是标准化的指数函数，定义如下：

$$y_c = \varsigma(\mathbf{z})_c = \frac{e^{z_c}}{\sum_{d=1}^C e^{z_d}} \quad \text{for } c = 1 \dots C$$

分母 $\sum_{d=1}^C e^{z_d}$ 的作用保证 $\sum_{c=1}^C y_c = 1$ 。作为神经网络的输出层，softmax 函数可以用图形表示为一个有 c 个神经元的层。

给定输入 \mathbf{z} ，可以写出 $t=c$ 类 ($c=1 \dots C$) 的概率

$$\begin{bmatrix} P(t=1 | \mathbf{z}) \\ \vdots \\ P(t=C | \mathbf{z}) \end{bmatrix} = \begin{bmatrix} \varsigma(\mathbf{z})_1 \\ \vdots \\ \varsigma(\mathbf{z})_C \end{bmatrix} = \frac{1}{\sum_{d=1}^C e^{z_d}} \begin{bmatrix} e^{z_1} \\ \vdots \\ e^{z_C} \end{bmatrix}$$

(2) Softmax 函数求导

为了在神经网络中使用 softmax 函数，需要计算其导数。对于 $\Sigma_C = \sum_{d=1}^C e^{z_d}$ 如果定义 $c=1 \dots C$ ，有 $y_c = e^{z_c} / \Sigma_C$ ，则 softmax 的输出 \mathbf{y} 关于输入 \mathbf{z} 的导数 $\partial y_i / \partial z_j$ 可以按下式计算：

$$\begin{aligned} \text{if } i = j: \quad \frac{\partial y_i}{\partial z_i} &= \frac{\frac{\partial e^{z_i}}{\Sigma_C}}{\frac{\partial z_i}{\partial z_i}} = \frac{e^{z_i} \Sigma_C - e^{z_i} e^{z_i}}{\Sigma_C^2} = \frac{e^{z_i}}{\Sigma_C} \frac{\Sigma_C - e^{z_i}}{\Sigma_C} = \frac{e^{z_i}}{\Sigma_C} \left(1 - \frac{e^{z_i}}{\Sigma_C}\right) = y_i(1 - y_i) \\ \text{if } i \neq j: \quad \frac{\partial y_i}{\partial z_j} &= \frac{\frac{\partial e^{z_i}}{\Sigma_C}}{\frac{\partial z_j}{\partial z_j}} = \frac{0 - e^{z_i} e^{z_j}}{\Sigma_C^2} = -\frac{e^{z_i}}{\Sigma_C} \frac{e^{z_j}}{\Sigma_C} = -y_i y_j \end{aligned}$$

注意：如果 $i = j$ ，该导数类似于逻辑函数的导数。

(3) Softmax 函数的交叉熵损失函数

为了获得 softmax 函数的交叉熵损失函数，我们从似然函数出发，给你模型的参数集合 θ 可以预测每个输入样本的正确分类，同逻辑损失函数的推导。最大似然可以写为：

$$\operatorname{argmax}_{\theta} \mathcal{L}(\theta | \mathbf{t}, \mathbf{z})$$

似然 $\mathcal{L}(\theta | \mathbf{t}, \mathbf{z})$ 可以重写为给定参数 θ 下生成 \mathbf{t} 和 \mathbf{z} 的联合概率： $P(\mathbf{t}, \mathbf{z} | \theta)$ ，写成条件概率形式为：

$$P(\mathbf{t}, \mathbf{z} | \theta) = P(\mathbf{t} | \mathbf{z}, \theta) P(\mathbf{z} | \theta)$$

由于我们不关心 \mathbf{z} 的概率，因此可将似化简为： $\mathcal{L}(\theta | \mathbf{t}, \mathbf{z}) = P(\mathbf{t} | \mathbf{z}, \theta)$ 。当 θ 固定时又等于 $P(\mathbf{t} | \mathbf{z})$ 。由于每一个 t_i 与整个 \mathbf{z} 相互独立，且 \mathbf{t} 中只有一个类被激活，于是：

$$P(\mathbf{t} | \mathbf{z}) = \prod_{i=c}^C P(t_c | \mathbf{z})^{t_c} = \prod_{i=c}^C \zeta(\mathbf{z})_c^{t_c} = \prod_{i=c}^C y_c^{t_c}$$

通过最小化负对数似然来实现最大化似然：

$$-\log \mathcal{L}(\theta | \mathbf{t}, \mathbf{z}) = \xi(\mathbf{t}, \mathbf{z}) = -\log \prod_{i=c}^C y_c^{t_c} = -\sum_{i=c}^C t_c \cdot \log(y_c)$$

从而得到交叉熵损失函数 ξ 。注意对于 2 分类问题，输出 $t_2 = 1 - t_1$ ，所以此时有同逻辑回归一样的损失函数： $\xi(\mathbf{t}, \mathbf{y}) = -t_c \log(y_c) - (1 - t_c) \log(1 - y_c)$ 。

n 个样本的交叉熵损失函数为：

$$\xi(T, Y) = \sum_{i=1}^n \xi(\mathbf{t}_i, \mathbf{y}_i) = -\sum_{i=1}^n \sum_{i=c}^C t_{ic} \cdot \log(y_{ic})$$

其中， t_{ic} 当且仅当样本 i 属于 c 类时为 1， y_{ic} 为样本 i 属于 c 类的输出概率。

(4) Softmax 函数的交叉熵损失函数求导

损失函数关于 z_i 的导数 $\partial \xi / \partial z_i$ 为：

$$\begin{aligned}
\frac{\partial \xi}{\partial z_i} &= -\sum_{j=1}^c \frac{\partial t_j \log(y_j)}{\partial z_i} = -\sum_{j=1}^c t_j \frac{\partial \log(y_j)}{\partial z_i} = -\sum_{j=1}^c t_j \frac{1}{y_j} \frac{\partial y_j}{\partial z_i} \\
&= -\frac{t_i}{y_i} \frac{\partial y_i}{\partial z_i} - \sum_{j \neq i}^c \frac{t_j}{y_j} \frac{\partial y_j}{\partial z_i} = -\frac{t_i}{y_i} y_i (1 - y_i) - \sum_{j \neq i}^c \frac{t_j}{y_j} (-y_j y_i) \\
&= -t_i + t_i y_i + \sum_{j \neq i}^c t_j y_i = -t_i + \sum_{j=1}^c t_j y_i = -t_i + y_i \sum_{j=1}^c t_j \\
&= y_i - t_i
\end{aligned}$$

注意已经对上面的 $i = j$ 和 $i \neq j$ 进行了求导 $\partial y_j / \partial z_i$ 。

5.3 隐藏单元

整流线性单元是隐藏单元极好的默认选择。许多其他类型的隐藏单元也是可用的。决定何时使用哪种类型的隐藏单元是困难的事（尽管整流线性单元通常是一个可接受的选择）。我们这里描述对于每种隐藏单元的一些基本直觉。这些直觉可以用来建议我们何时来尝试一些单元。通常不可能预先预测出哪种隐藏单元工作得最好。设计过程充满了试验和错误，先直觉认为某种隐藏单元可能表现良好，然后用它组成神经网络进行训练，最后用验证集来评估它的性能。

这里列出的隐藏单元并不是在所有的输入点上都是可微的，但在实践中，梯度下降对这些机器学习模型仍然表现得很好，部分原因是神经网络训练算法通常不会达到代价函数的局部最小值，而是仅仅显著地减小它的值，我们也并不期望训练能够实际到达梯度为 0 的点，所以代价函数的最小值对应于梯度未定义的点是可以接受的。

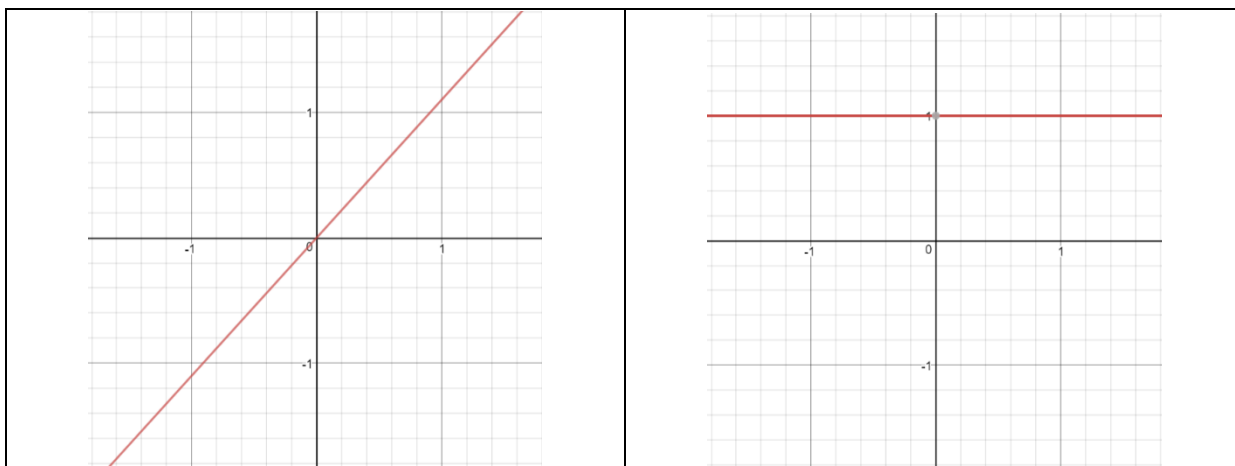
以下内容参照

https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#linear。

5.3.1 线性函数（Liner）

直线函数其激活与输入成正比（即神经元的加权和）。

激活函数	导数
$R(z, m) = \{z * m\}$	$R'(z, m) = \{m\}$



优点:

- 给出了一个激活范围，所以它不是二进制激活;
- 当然可以把几个神经元连接在一起，如果超过 1 个触发，我们就可以取最大值 (或 softmax)，并以此为基础做出决定。

缺点:

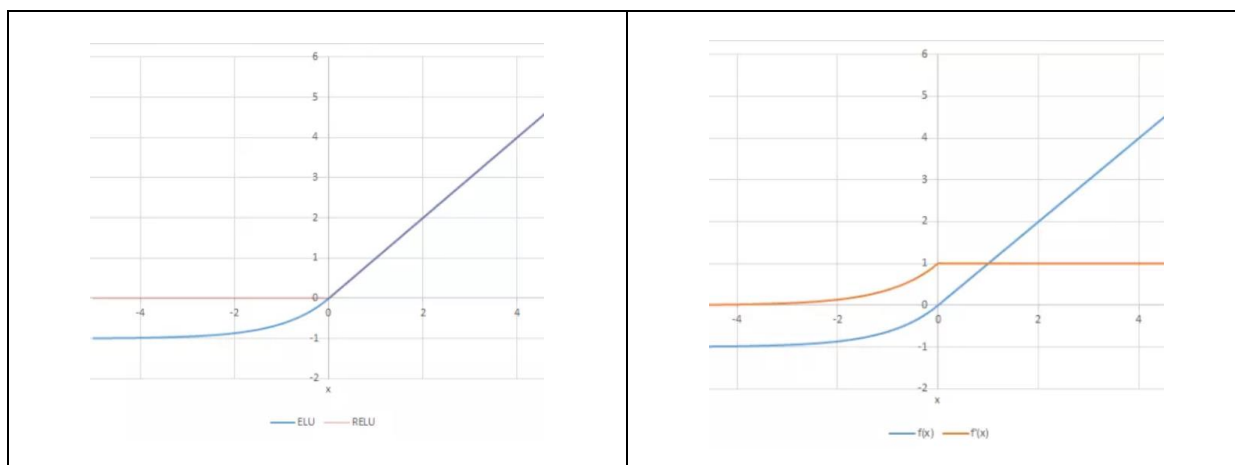
- 该函数的导数是常数，也就是说，梯度和 x 没有关系;
- 它是一个恒定的梯度，下降也是恒定的梯度;
- 如果预测有误差，则反向传播产生的变化是常数，不依赖于输入(x)的变化。

5.3.2 指数线性单元 (Exponential Linear Unit, ELU)

指数线性单元往往可以更快的收敛损失到零，产生更准确的结果。不同于其他的激活函数，ELU 有一个额外的正的常数 α 。

除负输入外，ELU 与 RELU 非常相似。它们都是非负输入的恒等函数形式。另一方面，ELU 慢慢变得平滑直至输出收敛到 $-\alpha$ ，而 RELU 在负半轴骤减为水平直线。

激活函数	导数
$R(z) = \begin{cases} z & z > 0 \\ \alpha \cdot (e^z - 1) & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ \alpha \cdot e^z & z < 0 \end{cases}$



优点:

- ELU 慢慢变得平滑,直到它的输出等于 $-\alpha$ 而 ReLU 却是骤减 (不平滑)。
- ELU 是 ReLU 的一个强大替代品。
- 与 ReLU 不同, ELU 可以产生负的输出。

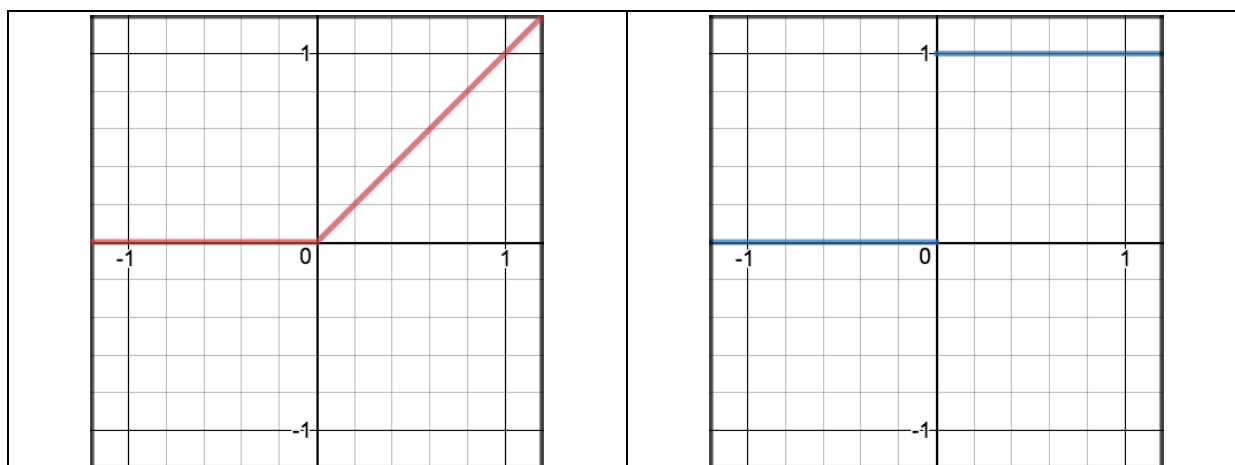
缺点:

对于 $x > 0$, $[0, \infty]$ 的输出范围会导致激活函数爆炸。

5.3.3 整流线性单元 (Rectified Linear Units, ReLU)

整流线性单元易于优化,因为它们和线性单元非常类似。线性单元和整流线性单元的唯一区别在于整流线性单元在其一半的定义域上输出为零。这使得只要整流线性单元处于激活状态,它的导数都能保持较大。它的梯度不仅大而且一致。整流操作的二阶导数几乎处处为 0, 并且在整流线性单元处于激活状态时,它的一阶导数处处为 1。这意味着相比于引入二阶效应的激活函数来说,它的梯度方向对于学习来说更加有用。

激活函数	导数
$R(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$



优点：

- 避免和纠正了消失梯度问题；
- ReLu 的计算开销比 tanh 和 sigmoid 要小，因为它涉及到更简单的数学运算。

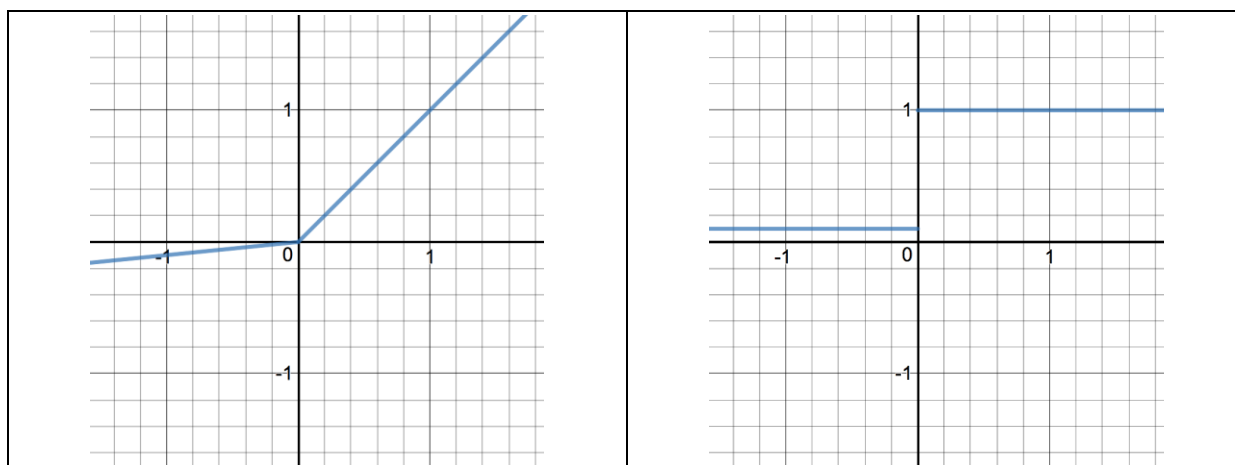
缺点：

- 它的一个限制是，它应该只用在神经网络模型的隐藏层中；
- 有些梯度在训练过程中很脆弱，甚至会死亡。它会导致权重更新再也不会任何数据点上激活。简单地说，ReLu 会导致神经元死亡；
- 对于 ReLu 区域($x < 0$)的激活，因为梯度为 0，在下降过程中不会调整权重。这意味着，那些进入这种状态的神经元将停止对误差/输入的变化做出反应(仅仅因为梯度为 0，没有任何变化)。这就是所谓的死亡 ReLu 问题。
- ReLu 的范围是 $[0, \text{inf}]$ 。这意味着它会破坏激活。

5.3.4 渗漏整流线性单元 (Leaky ReLU)

当 $z < 0$ ，leaky ReLU 允许一个小的、非零、恒定梯度 α （通常 $\alpha = 0.01$ ）。

激活函数	导数
$R(z) = \begin{cases} z & z > 0 \\ \alpha z & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ \alpha & z < 0 \end{cases}$



优点:

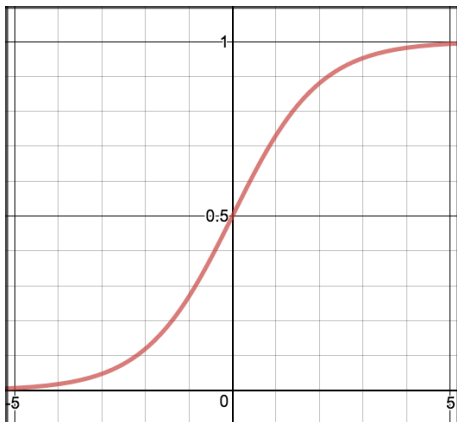
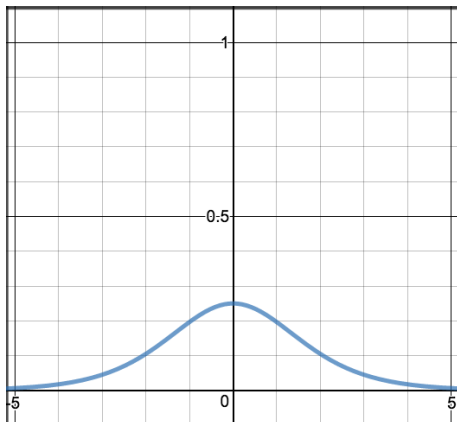
- Leaky ReLU 试图通过一个小的负斜率(0.01 左右)来修复 ReLU 死亡问题。

缺点:

- 它具有线性,不能用于复杂的分类。在某些用例中,它落后于 Sigmoid 和 Tanh。

5.3.5 Sigmoid

Sigmoid 接受一个实值作为输入,并输出 0 到 1 之间的另一个值。它易于实现,具有激活函数的所有优良特性:非线性、连续可微、单调、输出范围固定。

激活函数	导数
$S(z) = \frac{1}{1 + e^{-z}}$	$S'(z) = S(z) \cdot (1 - S(z))$
	

优点:

- 它本质上是非线性的。这个函数的组合也是非线性的;

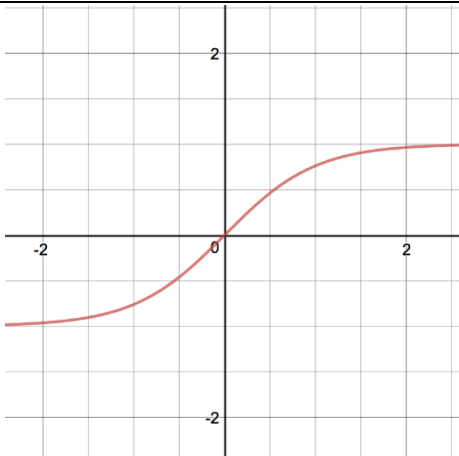
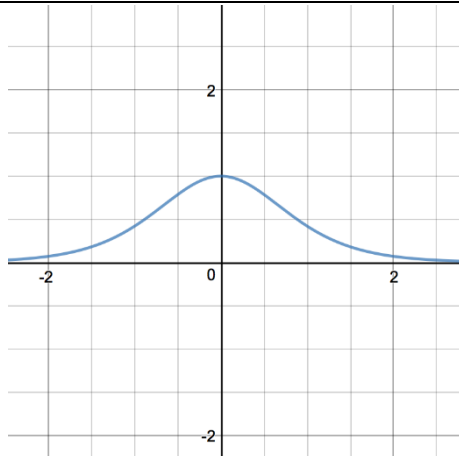
- 不同于阶跃函数它将给出一个模拟激活；
- 它也有一个平滑的梯度；
- 适应于分类器；
- 与线性函数的输出范围 $(-\infty, \infty)$ 相比，激活函数的输出总是在范围 $(0,1)$ 内。我们将激活绑定在一个范围内，样就不会破坏激活了。

缺点：

- 在 sigmoid 函数的两端，Y 值对 x 的变化响应很少；
- 这就导致了梯度消失的问题；
- 它的输出不是以 0 为中心的。它使梯度更新在不同的方向走得太远。输出在 0 和 1 之间，这使得优化更加困难；
- sigmoid 易饱和并杀死梯度；
- 网络拒绝进一步学习或者非常慢(取决于用例，直到梯度/计算受到浮点值限制)。

5.3.6 Tanh

Tanh 将实数压缩到 $[-1,1]$ 范围，是非线性的。但与 Sigmoid 不同的是，它的输出是以 0 为中心的。因此，在实际应用中，tanh 非线性总是优于 sigmoid 非线性。

激活函数	导数
$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$\tanh'(z) = 1 - \tanh(z)^2$
	

优点：

- tanh 的梯度比 sigmoid 更强(导数更陡)。

缺点：

- Tanh 也有消失梯度问题。

5.3.7 Softmax

Softmax 函数计算事件在 n 个不同事件上的概率分布。一般来说，这个函数将计算每个目标类在所有可能目标类上的概率。随后计算的概率将有助于确定给定输入的目标类。