FDU 神经网络 2. 前馈神经网络

本文参考以下教材:

- Deep Learning (I. Goodfellow, Y. Bengio, A. Courville) Chapter 6
- 深度学习 (I. Goodfellow, Y. Bengio, A. Courville, 赵申剑等译) 第 6 章
- 神经网络与深度学习 (邱锡鹏) 第 4 章

欢迎批评指正!

2.1 An Introduction

前馈神经网络 (Feedforward Neural Network) 又称**多层感知机** (Multi-Layer Perceptron, MLP)
这里的 "前馈" 是指模型的输出和模型之间没有**反馈** (feedback) 连接.

前馈神经网络可以有很多层.
第一层称为**输入层** (input layer)，最后一层称为**输出层** (output layer)
训练样本 (x, y) 直接指明了输出层必须产生一个接近 $y \approx f_\star(x)$ 的值 $f(x)$ 来最好地实现对 $f_\star(\cdot)$ 的近似.
然而训练样本没有直接指明中间层应该怎么做，因此我们称中间层为**隐藏层** (hidden layer)

2.1.1 一个简单的例子

我们先使用前馈网络解决一个简单的任务: 学习 XOR (异或) 函数.

XOR	0	1
0	0	1
1	1	0

显然这是一个非线性函数.
将数据点 $(0,0), (0,1), (1,0), (1,1)$ 绘制在二维平面后可以发现它们不是线性可分的.
不过我们可以通过一个非线性变换使得数据点在新的表示下是线性可分的:

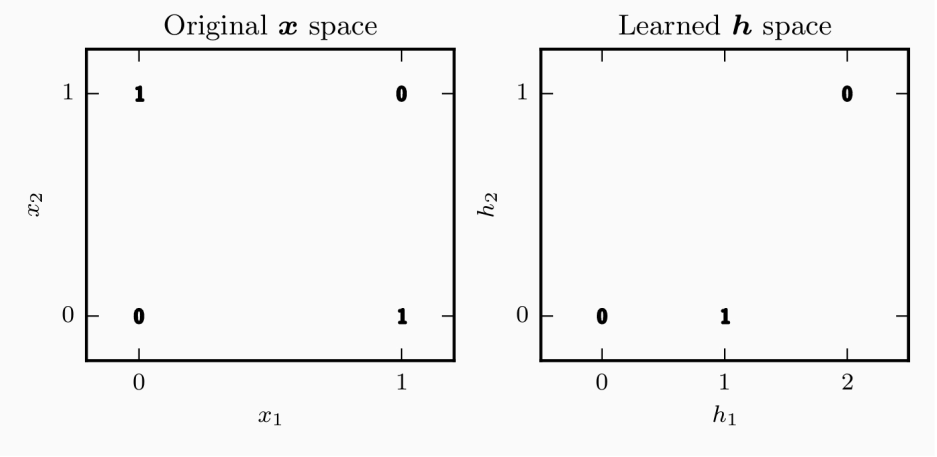


图 6.1: 通过学习一个表示来解决 XOR 问题。图上的粗体数字标明了学得的函数必须在每个点输出的值。*(左)* 直接应用于原始输入的线性模型不能实现 XOR 函数。当 $x_1 = 0$ 时，模型的输出必须随着 x_2 的增大而增大。当 $x_1 = 1$ 时，模型的输出必须随着 x_2 的增大而减小。线性模型必须对 x_2 使用固定的系数 w_2。因此，线性模型不能使用 x_1 的值来改变 x_2 的系数，从而不能解决这个问题。*(右)* 在由神经网络提取的特征表示的变换空间中，线性模型现在可以解决这个问题了。在我们的示例解决方案中，输出必须为 1 的两个点折叠到了特征空间中的单个点。换句话说，非线性特征将 $\boldsymbol{x} = [1, 0]^\top$ 和 $\boldsymbol{x} = [0, 1]^\top$ 都映射到了特征空间中的单个点 $\boldsymbol{h} = [1, 0]^\top$。线性模型现在可以将函数描述为 h_1 增大和 h_2 减小。在该示例中，学习特征空间的动机仅仅是使得模型的能力更大，使得它可以拟合训练集。在更现实的应用中，学习的表示也可以帮助模型泛化。

一个简单的前馈网络如下:

$$
\begin{aligned}
h &= f^{(1)}(x; W, c) = g(W^{\mathrm{T}} x + c) \\
y &= f^{(2)}(h; w, b) = w^{\mathrm{T}} h + b
\end{aligned}
$$

其中第一层通过作用一个非线性的**激活函数** (activation function) $g(\cdot)$ 来得到输入 x 的非线性表示 h
第二层接收第一层的输出 h，作用一个线性函数后得到最终的输出 y.
因此整个网络的数学表示是:

$$
\begin{aligned}
y &= f(x; W, c, w, b) \\
&= f^{(2)}(f^{(1)}(x; W, c); w, b) \\
&= w^{\mathrm{T}} g(W^{\mathrm{T}} x + c) + b
\end{aligned}
$$

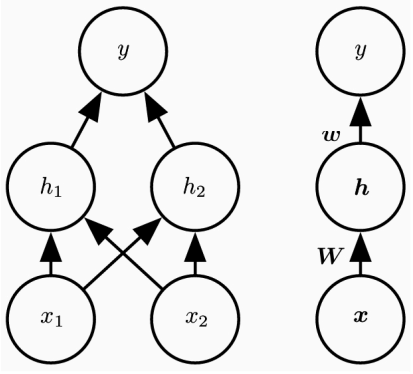


图 6.2: 使用两种不同样式绘制的前馈网络的示例。具体来说，这是我们用来解决 XOR 问题的前馈网络。它有单个隐藏层，包含两个单元。*(左)* 在这种样式中，我们将每个单元绘制为图中的一个节点。这种风格是清楚而明确的，但对于比这个例子更大的网络，它可能会消耗太多的空间。*(右)* 在这种样式中，我们将表示每一层激活的整个向量绘制为图中的一个节点。这种样式更加紧凑。有时，我们对图中的边使用参数名进行注释，这些参数是用来描述两层之间的关系的。这里，我们用矩阵 $\boldsymbol{W}$ 描述从 $\boldsymbol{x}$ 到 $\boldsymbol{h}$ 的映射，用向量 $\boldsymbol{w}$ 描述从 $\boldsymbol{h}$ 到 y 的映射。当标记这种图时，我们通常省略与每个层相关联的截距参数。

我们现在可以给出 XOR 问题的一个解:

- 取激活函数 $g(\cdot)$ 为**整流线性单元** (Rectified Linear Unit, ReLU) $g(z) := \max\{0, z\}$

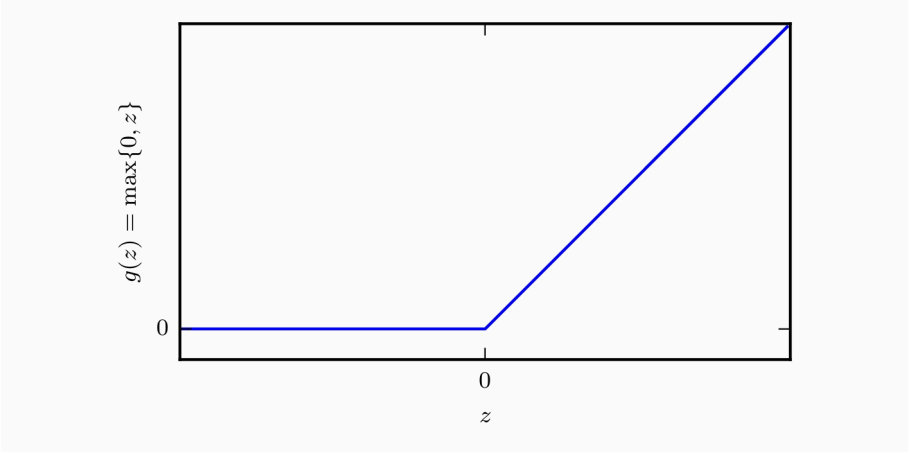


图 6.3: 整流线性激活函数。该激活函数是被推荐用于大多数前馈神经网络的默认激活函数。将此函数用于线性变换的输出将产生非线性变换。然而，函数仍然非常接近线性，在这种意义上它是具有两个线性部分的分段线性函数。由于整流线性单元几乎是线性的，因此它们保留了许多使得线性模型易于使用基于梯度的方法进行优化的属性。它们还保留了许多使得线性模型能够泛化良好的属性。计算机科学的一个通用原则是，我们可以从最小的组件构建复杂的系统。就像图灵机的内存只需要能够存储 0 或 1 的状态，我们可以从整流线性函数构建一个万能函数近似器。

- 取参数 W, c, w, b 为:

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

我们可以验证它能够完美地拟合 XOR 函数在数据点 $(0,0), (0,1), (1,0), (1,1)$ 上的取值:
(简单起见，我们将数据点堆叠成设计矩阵 X，统一计算模型输出 y)

$$
\begin{aligned}
y &= w^{\mathrm{T}} g(XW + 1_4 c^{\mathrm{T}}) + b \cdot 1_4 \\
&= \begin{bmatrix} 1 \\ -2 \end{bmatrix}^{\mathrm{T}} \max \left\{ 0, \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \end{bmatrix}^{\mathrm{T}} \right\} + 0 \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 \\ -2 \end{bmatrix}^{\mathrm{T}} \max \left\{ 0, \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 0 & -1 \\ 0 & -1 \\ 0 & -1 \\ 0 & -1 \end{bmatrix} \right\} \\
&= \begin{bmatrix} 1 \\ -2 \end{bmatrix}^{\mathrm{T}} \max \left\{ 0, \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \right\} \\
&= \begin{bmatrix} 1 \\ -2 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}
\end{aligned}
$$

$$
\left(\text{Design matrix } X := \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \right)
$$

在这个例子中，我们简单地指定了网络结构，然后说明它在特定的参数配置下得到的误差为零.
在实际情况中，可能会有数十亿的模型参数以及数十亿的训练样本，所以不能像上述例子那样简单地猜解.
基于梯度的优化算法可以帮助我们找到参数的近似解.

2.1.2 万能近似定理

具有隐藏层的前馈网络提供了一种万能近似框架.
具体来说，**万能近似定理** (universal approximation theorem) 表明:
一个前馈神经网络如果具有线性输出层和至少一层具有某种 "压缩" 性质的激活函数的隐藏层，
只要给予网络足够数量的隐藏单元，
它就能以任意精度来近似任何从一个有限维空间到另一个有限维空间的 Borel 可测函数.
此外，前馈网络的导数也可以任意好地来近似函数的导数.

Borel 可测的概念超出了本课程的范畴，
我们只需知道定义在 $\mathbb{R}^n$ 的有界闭子集上的连续函数都是 Borel 可测的.
这意味着对于我们试图学习的绝大多数函数，都存在一个足够大的前馈神经网络能够表示这个函数.
然而我们不能保证训练算法能够学得这个函数.
Occam 剃刀原理告诉我们没有普遍优越的机器学习算法.
这说明不存在万能的过程既能拟合训练样本，又能够选择一个函数来泛化到训练集中没有的点.

Montufar 定理指出:
具有 d 个输入、深度为 l、每个隐藏层具有 n 个单元的深度整流网络可以描述的线性区域的数量是:

$$
O\left(\binom{n}{d}^{d(l-1)} n^d \right)
$$

即是深度 l 的指数级.
无论是理论上还是经验上，更深的模型似乎确实在广泛的任务中泛化得更好.

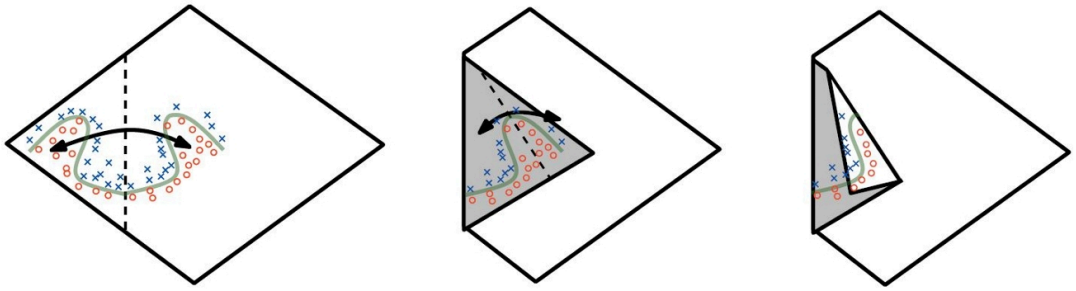


图 6.5: 关于更深的整流网络具有指数优势的一个直观的几何解释，来自 Montufar *et al.* (2014)。*(左)*绝对值整流单元对其输入中的每对镜像点有相同的输出。镜像的对称轴由单元的权重和偏置定义的超平面给出。在该单元顶部计算的函数（绿色决策面）将是横跨该对称轴的更简单模式的一个镜像。*(中)* 该函数可以通过折叠对称轴周围的空间来得到。*(右)* 另一个重复模式可以在第一个的顶部折叠（由另一个下游单元）以获得另外的对称性（现在重复四次，使用了两个隐藏层）。经 Montufar *et al.* (2014) 许可改编此图。

2.2 激活函数

为了增强神经网络的表示能力和学习能力，激活函数需要具备以下性质:

- ① 连续可导 (允许在少数点上不可导) 的非线性函数，方便数值优化.
- ② 激活函数及其导函数尽可能简单，方便计算.
- ③ 激活函数的导函数的值域要在一个合适的区间内，保证训练的稳定性.

2.2.1 ReLU 型函数

(1) ReLU 函数

ReLU (Rectified Linear Unit) 函数定义为 $\mathrm{ReLU}(z) = \max\{0_d, z\}$ $(\forall\, z \in \mathbb{R}^d)$
整流线性单元通常建立在仿射变换之上，形如 $h = \mathrm{ReLU}(W^{\mathrm{T}}x + b)$
它只要处于激活状态，其梯度就不仅大而且一致.
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.

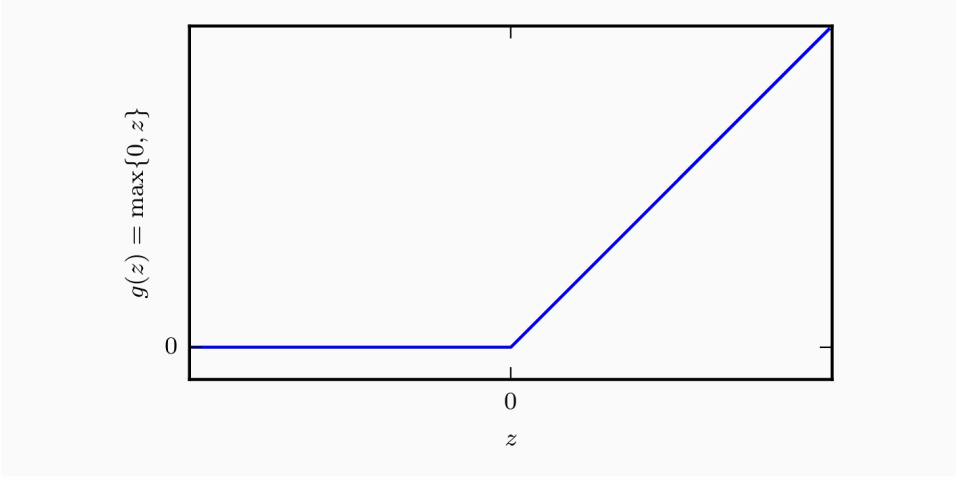


图 6.3: 整流线性激活函数。该激活函数是被推荐用于大多数前馈神经网络的默认激活函数。将此函数用于线性变换的输出将产生非线性变换。然而，函数仍然非常接近线性，在这种意义上它是具有两个线性部分的分段线性函数。由于整流线性单元几乎是线性的，因此它们保留了许多使得线性模型易于使用基于梯度的方法进行优化的属性。它们还保留了许多使得线性模型能够泛化良好的属性。计算机科学的一个通用原则是，我们可以从最小的组件构建复杂的系统。就像图灵机的内存只需要能够存储 0 或 1 的状态，我们可以从整流线性函数构建一个万能函数近似器。

ReLU 函数的缺陷在于它在未激活状态下的导数永远是 0，这会导致很多问题.
例如当第一个隐藏层中的 ReLU 神经元在所有的训练数据上都不能被激活时，
这个神经元参数的梯度将永远是 0_d，在后续的训练过程中都不能被激活.
类似地，这种现象也可能发生在其他隐藏层，称为**死亡 ReLU 问题** (dying ReLU problem)
幸运的是，整流线性单元的很多变种都能保证它们对于所有输入都有非零的梯度.

(2) Leaky ReLU 函数

带泄露的 ReLU (Leaky ReLU) 在输入小于 0 时保持一个很小的梯度 γ，以避免永远不能被激活.
其定义如下:

$$\begin{gathered}\mathrm{LeakyReLU}(z) := \max\{0_d, z\} + \beta \min\{0_d, z\} \quad (\forall\, z \in \mathbb{R}^d) \\ \text{where } \beta \text{ is a constant} \\ \Updownarrow \\ \mathrm{LeakyReLU}_i(z) := \begin{cases} z_i, & \text{if } z_i > 0 \\ \beta z_i, & \text{if } z_i \le 0 \end{cases} \quad (i = 1, \ldots, d)\end{gathered}$$

其中 β 是一个绝对值很小的常数 (或者说超参)，例如 $\beta = 0.01$
当 $\beta < 1$ 时，它可写成 $\mathrm{LeakyReLU}(z) = \max\{z, \beta z\}$，即一个简单的 maxout 单元.

(3) PReLU 函数

带参数的 ReLU (Parametric ReLU, PReLU) 引入的不是常数而是参数.
其定义为:

$$\begin{gathered}\mathrm{PReLU}(z) := \max\{0_d, z\} + \lambda \odot \min\{0_d, z\} \quad (\forall\, z \in \mathbb{R}^d) \\ \text{where } \lambda = [\lambda_1, \ldots, \lambda_d]^{\mathrm{T}} \text{ is learnable} \\ \Updownarrow \\ \mathrm{PReLU}_i(z) := \begin{cases} z_i, & \text{if } z_i > 0 \\ \lambda_i z_i, & \text{if } z_i \le 0 \end{cases} \quad (i = 1, \ldots, d)\end{gathered}$$

其中 $\lambda = [\lambda_1, \ldots, \lambda_d]^{\mathrm{T}}$ 是可学习的参数.

(4) ELU 函数

指数线性单元 (Exponential Linear Unit, ELU) 函数是一个近似零中心化的非线性函数.
其定义为:

$$\begin{gathered}\mathrm{ELU}(z) := \max\{0_d, z\} + \beta(\exp(z) - 1_d) \quad (\forall\, z \in \mathbb{R}^d) \\ \text{where } \beta \text{ is a constant} \\ \Updownarrow \\ \mathrm{ELU}_i(z) := \begin{cases} z_i, & \text{if } z_i > 0 \\ \beta(\exp(z_i) - 1), & \text{if } z_i \leq 0 \end{cases} \quad (i = 1, \ldots, d)\end{gathered}$$

其中 β 是一个常数 (或者说超参)，用于调整输出均值在 0_d 附近.

(5) Softplus 函数

Softplus 函数是 ReLU 函数的平滑化，其定义为:

$$\mathrm{softplus}(z) := \log(1_d + \exp(z)) \quad (\forall\, z \in \mathbb{R}^d)$$

它保留了 ReLU 函数单侧抑制、宽兴奋边界的特性，却没有保留稀疏激活性.
巧合的是，其一维形式的导数恰为 Sigmoid 函数:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}z}\mathrm{softplus}(z) &= \frac{\mathrm{d}}{\mathrm{d}z}\log(1 + \exp(z)) \\ &= \frac{1}{1 + \exp(z)} \cdot \exp(z) \\ &= \sigma(z)\end{aligned} \quad \left(\text{note that } \sigma(z) := \frac{1}{1 + \exp(-z)} = \frac{\exp(z)}{\exp(z) + 1}\right)$$

并且我们有:

$$\begin{aligned}\log \sigma(-z) &= \log(\frac{1}{1 + \exp(z)}) \\ &= -\log(1 + \exp(z)) \\ &= -\mathrm{softplus}(z)\end{aligned} \quad (\forall\, z \in \mathbb{R})$$

下图给出了 ReLU 型函数的图示:

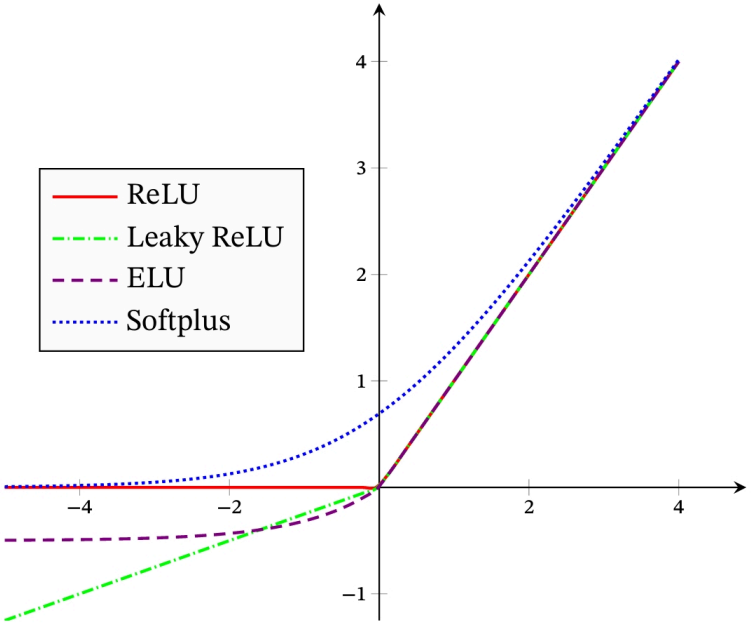


图 4.4 ReLU、Leaky ReLU、ELU 以及 Softplus 函数

2.2.2 Sigmoid 型函数

Sigmoid 型函数 $f(\cdot)$ 是指一类 S 型两端饱和函数.
一维情况下，它满足:

$$\begin{cases} \lim_{x\to-\infty} f'(x) = 0 \\ \lim_{x\to\infty} f'(x) = 0 \end{cases}$$

常用的一维 Sigmoid 型函数有 Logistic Sigmoid 函数 $\sigma(\cdot)$ 和双曲正切函数 $\tanh(\cdot)$.

(1) Sigmoid 函数

Sigmoid 函数 $\sigma(\cdot)$ 的定义如下:

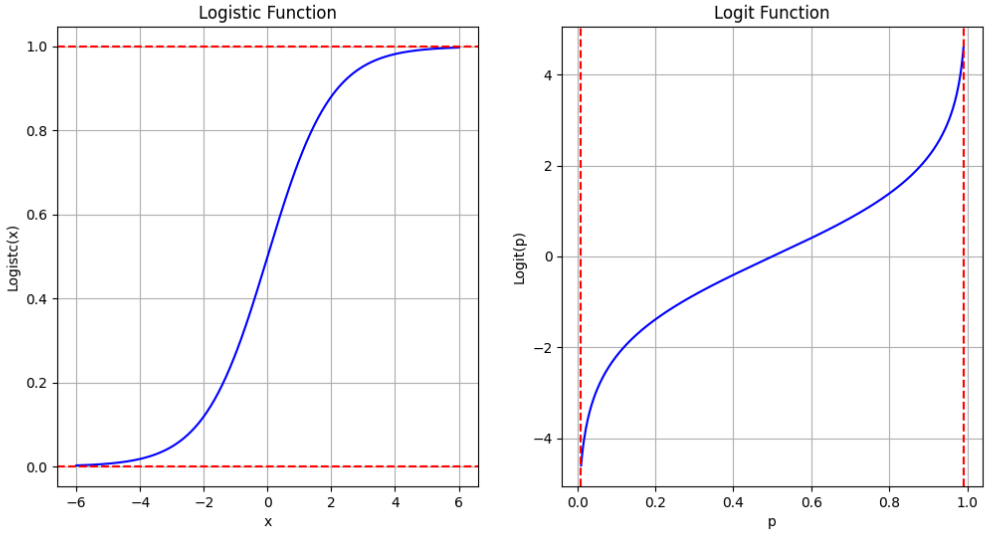
$$\begin{aligned} \sigma(z) &:= \frac{1}{1+\exp(-z)} \\ &= \frac{\exp(z)}{\exp(z)+1} \end{aligned} \qquad (\forall\, z \in \mathbb{R})$$

它把实数域的输入挤压到 $(0,1)$，因此其输出可以直接看作概率分布，
也可看作一个软性门 (soft gate)，用来控制其他神经元输出信息的数量.
当输入值在 0 附近时，Sigmoid 函数近似为线性函数 $\sigma(z) \approx \frac{1}{4}z + \frac{1}{2}$ (一阶 Taylor 近似)
当输入值靠近两端 $(\pm\infty)$ 时，Sigmoid 函数对输入进行挤压.

$$
\begin{aligned}
\sigma(-z) &= \frac{1}{1+\exp(z)} \\
&= \frac{\exp(-z)}{\exp(-z)+1} \\
&= 1 - \frac{1}{1+\exp(-z)} \\
&= 1 - \sigma(z)
\end{aligned}
$$

$$
\begin{aligned}
\sigma'(z) &= \frac{\exp(-z)}{(1+\exp(-z))^2} \\
&= \frac{\exp(-z)}{1+\exp(-z)} \cdot \frac{1}{1+\exp(-z)} \\
&= \frac{1}{\exp(z)+1} \cdot \frac{1}{1+\exp(-z)} \\
&= \sigma(-z) \cdot \sigma(z) \\
&= (1-\sigma(z))\sigma(z)
\end{aligned}
$$

Sigmoid 函数的反函数称为分对数函数 $\mathrm{Logit}(p) = \log(\frac{p}{1-p}) \quad (p \in [0,1])$



使用 Sigmoid 函数可以表征一个条件 Bernoulli 分布.
考虑 Sigmoid 神经元:

$$\sigma(z) = \sigma(w^{\mathrm{T}}x + b)$$

其中分对数 (logit) $z = w^{\mathrm{T}}x + b$
假定非归一化的条件概率 $\widetilde{\mathrm{P}}\{Y=1|x\}$ 取对数后关于 y, z 满足:
(简单起见，省略 w, b 的记号)

$$\log \widetilde{\mathrm{P}}\{Y=1|x\} = yz$$

则我们可以逐步推导出归一化的条件概率:

$$\widetilde{\mathrm{P}}\{Y=1|x\}=\exp(yz)$$
$$\Updownarrow$$
$$\begin{aligned}\mathrm{P}\{Y=1|x\}&=\frac{\exp(yz)}{\exp(0\cdot z)+\exp(1\cdot z)}\\&=\frac{\exp(z)}{1+\exp(z)}\\&=\sigma(z)\\\hline\mathrm{P}\{Y=0|x\}&=1-\sigma(z)\\&=\sigma(-z)\end{aligned}$$

于是我们有 $\mathrm{P}\{Y=y|x\}=\sigma((2y-1)z)$ (其中 $y\in\{0,1\}$)

Sigmoid 函数 $\sigma(z)$ 在 z 取非常小的负值时会饱和到 0，在 z 取非常大的正值时会饱和到 1.
我们可以通过取对数来使其 "膨胀"，得到合适的损失函数 (具体来说是 softplus 损失函数):

$$\begin{aligned}\mathcal{L}(x,w,b)&=-\log\mathrm{P}\{Y=y|x\}\\&=-\log\sigma((2y-1)z)\quad(\text{note that }\operatorname{softplus}(x)=\log(1+\exp(x))=-\log\sigma(-x))\\&=\operatorname{softplus}((1-2y)z)\end{aligned}$$

根据 Softplus 函数图像可知，它仅仅在 $(1-2y)z$ 取绝对值非常大的负值时才会饱和.
因此饱和只会出现在模型已经得到正确答案时——当 $y=1$ 且 z 取非常大的正值时，或者 $y=0$ 且 z 取非常小的负值时.
当 z 的符号错误时，我们有 $(1-2y)z=|z|$.
随着 $|z|$ 增大，Softplus 函数渐近趋向 $|z|$，其导数渐近趋向 1.
这意味着基于梯度的学习可以很快地改正错误的 z.

如果我们使用均方误差之类的其他损失函数，则它们会在 $\sigma(\cdot)$ 饱和时饱和.
这种情况一旦发生，梯度会变得非常小以至于不能用来学习，无论此时模型给出的是正确还是错误的答案.

总之，Sigmoid 神经元在其大部分定义域内都饱和:
当 z 取绝对值很大的正值时，它饱和到一个高值;
当 z 取绝对值很大的负值时，它饱和到一个低值;
只有当 z 接近 0 时它们才对输入强烈敏感.
Sigmoid 神经元的广泛饱和性会使得基于梯度的学习变得非常困难.
因此我们不鼓励将 Sigmoid 神经元用作前馈网络中的隐藏单元 (它更多应用在循环网络等情景中)
只有使用一个合适的损失函数来抵消 Sigmoid 神经元的广泛饱和性时，
它才可以作为输出单元与基于梯度的学习相兼容.

(2) Softmax 函数

Softmax 函数可以视为 $\operatorname{argmax}(\cdot)$ 函数的平滑版本 (如果我们认为 $\operatorname{argmax}(\cdot)$ 的输出是 one-hot 向量的话)
同时它也可以视为 Sigmoid 函数在高维情况的推广:

$$\begin{aligned}\operatorname{softmax}(z) &:= \frac{\exp(z)}{1_d^{\mathrm{T}} \exp(z)} \\ &= \frac{1}{\sum_{i=1}^{d} \exp(z_i)} \begin{bmatrix} \exp(z_1) \\ \vdots \\ \exp(z_d) \end{bmatrix} \quad (\forall\, x \in \mathbb{R}^d)\end{aligned}$$

它可以用来表示一个具有 n 个可能取值的离散型随机变量的分布.
和 Sigmoid 类似地，我们可以用 Softmax 函数得到一个 Multinoulli 分布.
考虑 Sigmoid 神经元:

$$\operatorname{softmax}(z) = \operatorname{softmax}(W^{\mathrm{T}} h + b)$$

假定归一化的条件概率 $\mathrm{P}\{Y = i|x\}$ 满足:

$$\mathrm{P}\{Y = i|x\} = \frac{\exp(z_i)}{\sum_{j=1}^{d} \exp(z_j)} = \operatorname{softmax}_i(z)$$

取对数以抵消 Softmax 中的 $\exp$，便得到:

$$\log \mathrm{P}\{Y = i|x\} = \log(\operatorname{softmax}_i(z)) = z_i - \log(\sum_{j=1}^{d} \exp(z_j))$$

- 第一项表示输入 z_i 总是对代价函数有直接的贡献.
  因为这一项不会饱和，所以即使 z_i 对第二项的贡献很小，学习依然可以进行.
  当最大化对数似然时，第一项鼓励 z_i 被抬高，而第二项则鼓励 z 的所有元素被压低.
- 注意到第二项 $-\log(\sum_{j=1}^{d} \exp(z_j))$ 可以近似为 $\max_j z_j$.
  我们从中得到的直觉是，最大化对数似然等价于强烈地惩罚最活跃的不正确预测.
  如果某个样本的正确答案已经具有了 Softmax 的最大输入，那么第一项和第二项将大致抵消.
  于是这个样本对于整体训练贡献很小，训练贡献主要由未被正确分类的样本产生.

对数似然以外的很多目标函数对 Softmax 神经元都不起作用.
由于没有用对数来抵消指数，当指数函数的变量取非常小的负值时会造成梯度消失，从而无法学习.
其中均方误差对于 Softmax 神经元来说是一个很差的损失函数，
即使模型做出高度可信的不正确预测，模型也很难通过学习来改变其输出.

(3) 双曲正切函数

双曲正切函数的定义如下:

$$\begin{aligned}\tanh(z) &:= \frac{\sinh(z)}{\cosh(z)} \\ &= \frac{e^z - e^{-z}}{e^z + e^{-z}} \\ &= \frac{e^{2z} - 1}{e^{2z} + 1} \\ &= 2\frac{e^{2z}}{e^{2z} + 1} - 1 \\ &= 2\sigma(2z) - 1\end{aligned}$$

因此双曲正切函数可以视为将 Sigmoid 函数纵向放大 2 倍并将对称中心下移至坐标原点得到的.

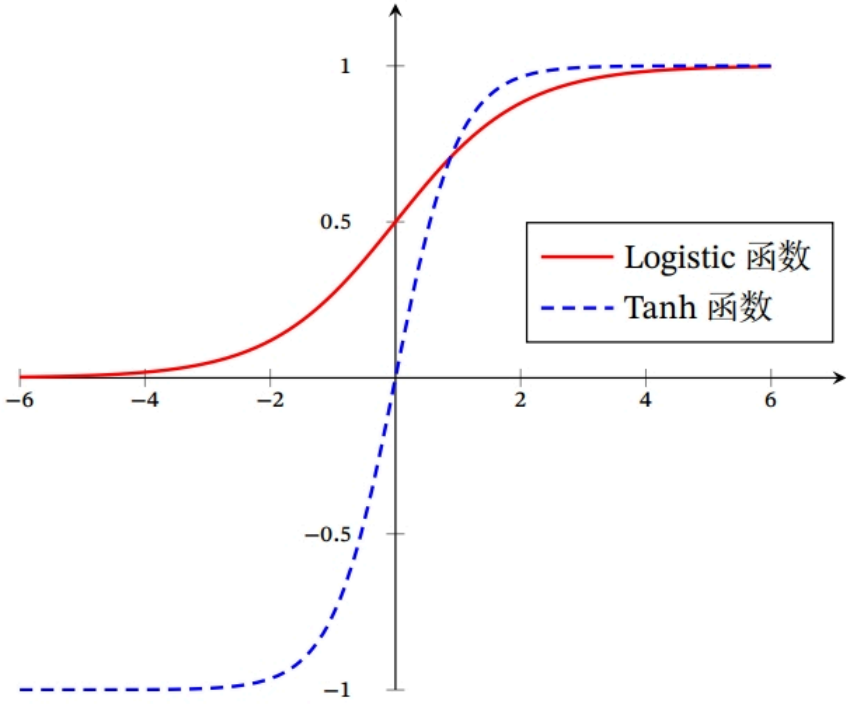


图 4.2　Logistic 函数和 Tanh 函数

由此可知，双曲正切函数是**零中心化的** (zero-centered)，而 Logistic 函数不是零中心化的.
非零中心化的输出会使得下一层的神经元的输入发生**偏置偏移** (bias shift)，使得梯度下降的收敛速度变慢.
此外，当输入值在 0 附近时，双曲正切函数近似为单位函数 $\tanh(z) \approx z$ (一阶 Taylor 近似)
因此双曲正切函数通常要比 Sigmoid 函数的表现更好.

2.2.3 其他

(1) Maxout 函数

Maxout 函数是多个输入到单个输出的非线性映射关系:

$$\text{maxout}(x) := \max_{i=1,\dots,d} z_i \quad (\text{where } z = W^{\mathrm{T}}x + b \in \mathbb{R}^d)$$

它可以看作任意凸函数的分段线性近似，并且在有限的点上是不可微的.

(2) Swish 函数

Swish 函数是一种自门控 (self-gated) 激活函数，其定义如下:

$$\text{swish}(z) = z \cdot \sigma(\beta z)$$

其中 $\sigma(\cdot)$ 是 Sigmoid 函数，β 是一个门控常数 (或者说超参)
它提供了一种软性的门控机制:
当 $\sigma(\beta z)$ 接近于 1 时，门处于 "开" 的状态，激活函数的输出近似于输入 z 本身.
当 $\sigma(\beta z)$ 接近于 0 时，门处于 "关" 的状态，激活函数的输出近似于 0

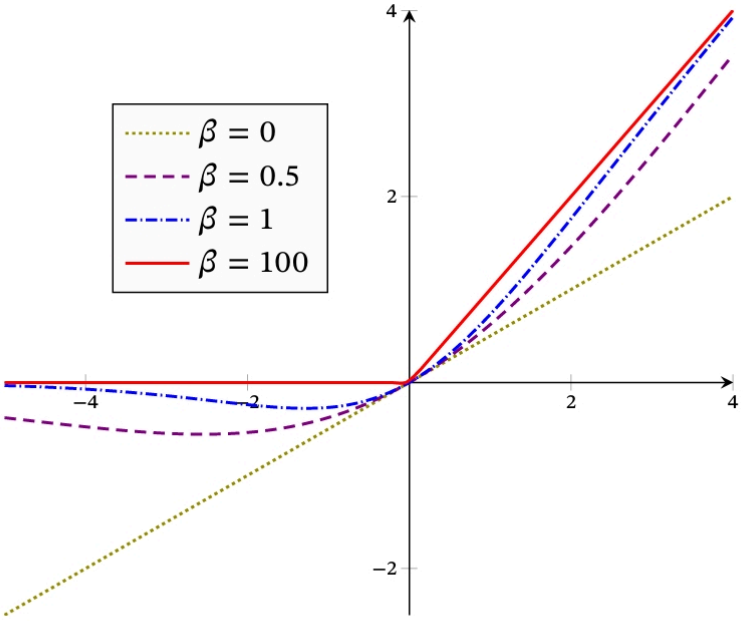


图 4.5 Swish 函数

当 $\beta = 0$ 时，Swish 函数变为线性函数 $x/2$
当 $\beta \to \infty$ 时，Swish 函数近似于 ReLU 函数.
因此它可看作线性函数 $x/2$ 和 ReLU 函数之间的非线性插值函数.

(3) GELU 函数

GELU (Gaussian Error Linear Unit, Gauss 误差线性单元) 的定义如下:

$$\mathrm{GELU}(x) := x \cdot \mathrm{P}\{N(\mu, \sigma^2) \leq x\}$$

其中 $\mathrm{P}\{N(\mu, \sigma^2) \leq x\}$ 是 Gauss 分布 $N(\mu, \sigma^2)$ 的累积分布函数.
超参数 μ, σ^2 一般设为 $\mu = 0, \sigma = 1$

由于零均值 Gauss 分布的累积分布函数是 S 型函数 (尽管无法用初等函数表达),
故 GELU 函数可用 $\tanh$ 函数或 Sigmoid 函数 $\sigma(\cdot)$ 来近似:

$$\begin{aligned}\mathrm{GELU}(x) &\approx 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right)\right) \\ &\approx x \cdot \sigma(1.702x)\end{aligned}$$

2.3 前馈神经网络

2.3.1 记号

在前馈神经网络中，每一层的神经元可以接收前一层神经元的信号，并产生信号输出到下一层.
第 0 层称为输入层，最后一层称为输出层，其他中间层称为隐藏层.

- 对于二分类问题，输出层可以只放一个 Sigmoid 神经元，
  直接输出类别 $y = 1$ 的条件概率 $\mathrm{P}\{y = 1|x\}$

- 对于多分类问题 (设类别总数为 K)，输出层可以放 K 个神经元，激活函数为 softmax 函数. 直接输出每个类别的条件概率 $\mathrm{P}\{y=k|x\}\ (k=1,\ldots,K)$

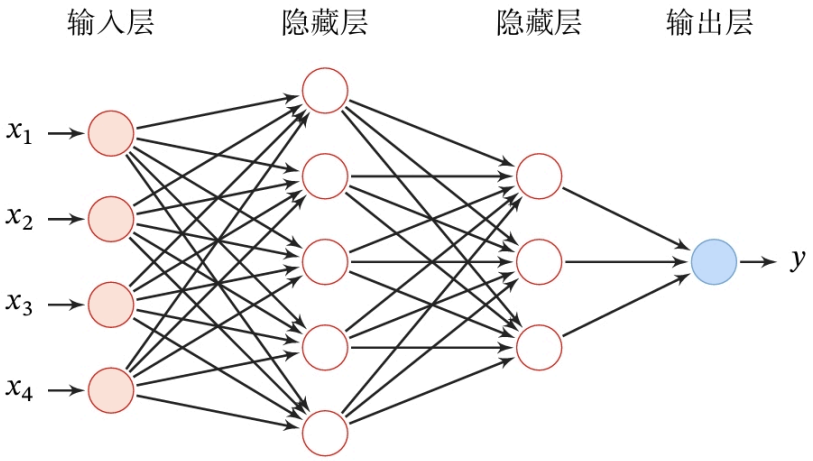


图 4.7 多层前馈神经网络

记号约定如下:

- 我们记输入层为第 0 层，输入值 $x \in \mathbb{R}^{M_0}$，记 $a^{(0)} = x \in \mathbb{R}^{M_0}$
- 神经网络的层数 (忽略输入层) 记为 L
- 第 l 层神经元个数记为 M_l
- 第 l 层神经元的激活函数记为 $f_l(\cdot)$
- 第 $l-1$ 层到第 l 层的权重矩阵记为 $W^{(l)} \in \mathbb{R}^{M_l \times M_{l-1}}$，偏置记为 $b^{(l)} \in \mathbb{R}^{M_l}$
- 第 l 层神经元的输入 (净活性值) $z^{(l)} \in \mathbb{R}^{M_l}$
- 第 l 层神经元的输出 (活性值) $a^{(l)} \in \mathbb{R}^{M_l}$

于是前馈神经网络的迭代格式如下:

$$\begin{gathered}\text{Input } a^{(0)} := x \\ \hline z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)} \\ a^{(l)} = f_l(z^{(l)})\end{gathered}$$

即 $a^{(l)} = f_l(z^{(l)}) = f_l(W^{(l)} a^{(l-1)} + b^{(l)})$
换言之，第 l 层神经元相当于对上一层的活性值 $a^{(l-1)}$ 进行一次仿射变换和一次非线性变换.
或者说 $z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)} = W^{(l)} f_{l-1}(z^{(l-1)}) + b^{(l)}$
换言之，第 l 层神经元的输入值是由上一层的输入值 $z^{(l-1)}$ 经过一次非线性变换和一次仿射变换得到的.
因此前馈神经网络就是逐层的信息传递:

$$x = a^{(0)} \mapsto z^{(1)} \mapsto a^{(1)} \mapsto z^{(2)} \mapsto \cdots \mapsto a^{(L-1)} \mapsto z^{(L)} \mapsto a^{(L)} = \phi(x; W, b)$$
$$\text{where } W = \{W^{(1)}, \ldots, W^{(L)}\} \text{ and } b = \{b^{(1)}, \ldots, b^{(L)}\}$$

其中 W, b 代表网络中所有层的权重和偏置.

2.3.2 梯度下降

对于 K 分类问题，我们通常使用交叉熵损失函数.
考虑样本 (x, y)，其中 $x \in \mathbb{R}^d$，而 $y \in \mathbb{R}^K$ 为标签对应的 one-hot 向量表示.
设 $\hat{y}$ 为分类模型输出的条件概率向量，则交叉熵损失函数为:

$$\begin{aligned}\mathcal{L}(y, \hat{y}) &:= -y^{\mathrm{T}} \log(\hat{y}) \\ &= -\sum_{k=1}^{K} y_k \log(\hat{y}_k) \\ &\quad (\text{where } y \in \{0,1\}^K \text{ and } \hat{y} \in [0,1]^K \text{ such that } 1_K^{\mathrm{T}} \hat{y} = 1)\end{aligned}$$

给定训练集 $\mathcal{D}_{\text{train}} := \{(x^{(n)}, y^{(n)})\}_{n=1}^{N}$，其结构化风险函数为:

$$\begin{aligned}\text{Risk}(W, b) &:= \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}(y^{(n)}, \hat{y}^{(n)}) + \frac{1}{2} \lambda \|W\|_{\text{F}}^2 \\ &= \frac{1}{N} \sum_{n=1}^{N} (y^{(n)})^{\mathrm{T}} \log(\hat{y}^{(n)}) + \frac{1}{2} \lambda \sum_{l=1}^{L} \|W^{(l)}\|_{\text{F}}^2\end{aligned}$$

其中 $\|\cdot\|_{\mathbb{F}}$ 是 Frobenius 范数.

在梯度下降中，第 l 层的参数 $W^{(l)}$ 和 $b^{(l)}$ 的迭代格式为:

$$\begin{aligned}W^{(l)} &\leftarrow W^{(l)} - \alpha \frac{\partial \text{Risk}(W, b)}{\partial W^{(l)}} \\ &= W^{(l)} - \alpha \left(\frac{1}{N} \sum_{n=1}^{N} \frac{\partial \mathcal{L}(y^{(n)}, \hat{y}^{(n)})}{\partial W^{(l)}} + \lambda W^{(l)} \right) \\ \hline b^{(l)} &\leftarrow b^{(l)} - \alpha \frac{\partial \text{Risk}(W, b)}{\partial b^{(l)}} \\ &= b^{(l)} - \alpha \left(\frac{1}{N} \sum_{n=1}^{N} \frac{\partial \mathcal{L}(y^{(n)}, \hat{y}^{(n)})}{\partial b^{(l)}} \right)\end{aligned}$$

其中 α 为学习率 (或者说步长)
我们通常使用反向传播算法来高效地计算上述迭代格式中的偏导.

2.3.3 反向传播算法

> **写在前面:**
> 按最优化的记号，关于 z 的梯度应该用 ∇_z 符号表示 (标量对列向量求梯度得到的是同形的列向量),
> 但为了方便我们使用 $\frac{\partial}{\partial z}$ 来代表梯度，尽管 $\frac{\partial}{\partial z}$ 符号在数学上代表导数 (标量对列向量求导数得到的是转置后同形的行向量)
> 机器学习的记号真是太混乱啦! 而且与数学的记号不兼容.
> 无论如何，现在我们认为 $\frac{\partial}{\partial z}$ 代表梯度.
> (邱锡鹏老师书上的记号也是这样默认的，不过着实让我困惑了一阵)

首先我们计算梯度 $\frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(L)}} \in \mathbb{R}^{M_L}$:

$$\begin{aligned}\frac{\partial \mathcal{L}(y, \hat{y})}{\partial z^{(L)}} &= \frac{\partial \hat{y}}{\partial z^{(L)}} \cdot \frac{\partial \mathcal{L}(y, \hat{y})}{\partial \hat{y}} \quad \left(\text{note that } \begin{cases}\hat{y} = a^{(L)} = f_l(z^{(L)}) \\ \mathcal{L}(y, \hat{y}) = -y^{\mathrm{T}} \log(\hat{y})\end{cases}\right) \\ &= \frac{\partial f_l(z^{(L)})}{\partial z^{(L)}} \cdot (y \oslash \hat{y}) \quad (\text{where } \oslash \text{ denotes pointwise division})\end{aligned}$$

其次我们计算梯度 $\frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}} \in \mathbb{R}^{M_l}$ 和 $\frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l+1)}} \in \mathbb{R}^{M_{l+1}}$ 之间的递推关系:

$$
\begin{aligned}
\frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}} &= \frac{\partial a^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l+1)}}{\partial a^{(l)}} \cdot \frac{\partial \mathcal{L}(y,\hat{y})}{z^{(l+1)}} \quad \left(\text{note that } \begin{cases} a^{(l)} = f_l(z^{(l)}) \\ z^{(l+1)} = W^{(l+1)}a^{(l)} + b^{(l+1)} \end{cases}\right) \\
&= \frac{\partial f_l(z^{(l)})}{\partial z^{(l)}} \cdot (W^{(l+1)})^{\mathrm{T}} \cdot \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l+1)}} \in \mathbb{R}^{M_l}
\end{aligned}
$$

其中 $\odot$ 代表 Hadamard 乘积.
于是我们有:

$$
\begin{aligned}
\frac{\partial \mathcal{L}(y,\hat{y})}{\partial W^{(l)}} &= \frac{\partial z^{(l)}}{\partial W^{(l)}} \cdot \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}} \quad (\text{note that } z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}) \\
&= (a^{(l-1)})^{\mathrm{T}} \otimes \left(I_{M_l} \cdot \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}} \right) \\
&\quad (\text{use } \otimes \text{ to represent the production of 3D tensor and column vector}) \\
&= (a^{(l-1)})^{\mathrm{T}} \otimes \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}} \\
&= \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}} (a^{(l-1)})^{\mathrm{T}} \\
\hline
\frac{\partial \mathcal{L}(y,\hat{y})}{\partial b^{(l)}} &= \frac{\partial z^{(l)}}{\partial b^{(l)}} \cdot \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}} \quad (\text{note that } z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}) \\
&= I_{M_l} \cdot \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}} \\
&= \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}}
\end{aligned}
$$

其中 $\otimes$ 代表 Kronecker 乘积.

使用反向传播算法的前馈神经网络训练过程可以分成以下 ③ 步:

- ① 前馈计算得到每一层的输入 $z^{(l)}$ 和激活值 $a^{(l)}$ ($l = 1, \ldots, L$)

$$
\begin{aligned}
& x = a^{(0)} \mapsto z^{(1)} \mapsto a^{(1)} \mapsto z^{(2)} \mapsto \cdots \mapsto a^{(L-1)} \mapsto z^{(L)} \mapsto a^{(L)} = \hat{y} \\
\hline
& \text{for } l = 1 : L \text{ do:} \\
& \quad z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)} \\
& \quad a^{(l)} = f_l(z^{(l)}) \\
& \text{end}
\end{aligned}
$$

- ② 反向传播计算 $\mathcal{L}(y,\hat{y})$ 关于每一层输入 $z^{(l)}$ 的梯度: (注意是梯度，不是导数)

$$
\begin{aligned}
& \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(L)}} \mapsto \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(L-1)}} \mapsto \cdots \mapsto \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(1)}} \\
\hline
& \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(L)}} = \frac{\partial f_l(z^{(L)})}{\partial z^{(L)}} \cdot (y \oslash \hat{y}) \\
& \text{for } l = L-1 : -1 : 1 \text{ do:} \\
& \quad \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}} = \frac{\partial f_l(z^{(l)})}{\partial z^{(l)}} \cdot (W^{(l+1)})^{\mathrm{T}} \cdot \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l+1)}} \\
& \text{end}
\end{aligned}
$$

- ③ 计算每一层参数的梯度，并更新参数:

$$
\begin{aligned}
&\frac{\partial \mathcal{L}(y,\hat{y})}{\partial W^{(l)}} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}}(a^{(l-1)})^{\mathrm{T}} \\
&\frac{\partial \mathcal{L}(y,\hat{y})}{\partial b^{(l)}} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}} \\
&\hline
&W^{(l)} \leftarrow W^{(l)} - \alpha\left(\frac{\partial \mathcal{L}(y,\hat{y})}{\partial W^{(l)}} + \lambda W^{(l)}\right) \quad (l = 1,\dots,L) \\
&b^{(l)} \leftarrow b^{(l)} - \alpha\frac{\partial \mathcal{L}(y,\hat{y})}{\partial b^{(l)}}
\end{aligned}
$$

其中 $\lambda > 0$ 为正则化系数.

(神经网络与深度学习, 算法 4.1)

简单起见，我们只总结使用反向传播算法的随机梯度下降:

(要使用全样本的话，只需在最后一步更新参数时使用梯度的均值即可)

Input: train set $\mathcal{D}_{\text{train}} = \{(x^{(n)}), y^{(n)}\}_{n=1}^{N},$ validation set $\mathcal{D}_{\text{valid}}$
learning rate $\alpha > 0,$ regularization coefficient $\lambda > 0,$
number of layers $L \geq 2,$ number of neurons $M_l > 0\ (1 \leq l \leq L)$

Randomly initialize $W = \{W^{(1)},\dots,W^{(L)}\}$ and $b = \{b^{(1)},\dots,b^{(L)}\}$
Repeat:
  Randomly shuffle the train set D_{train}
  for $n = 1 : N$ do:
    choose $(x, y) = (x^{(n)}, y^{(n)})$
    $a^{(0)} = x$
    for $l = 1 : L$ do: (Forward computation of each layer's input and activation values)
      $z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$
      $a^{(l)} = f_l(z^{(l)})$
    end
    $\hat{y} = a^{(L)}$
    $\dfrac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(L)}} = \dfrac{\partial f_l(z^{(L)})}{\partial z^{(L)}} \cdot (y \oslash \hat{y})$
    for $l = L-1 : -1 : 1$ do: (Backward propagation)
      $\dfrac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}} = \dfrac{\partial f_l(z^{(l)})}{\partial z^{(l)}} \cdot (W^{(l+1)})^{\mathrm{T}} \cdot \dfrac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l+1)}}$
      $\dfrac{\partial \mathcal{L}(y,\hat{y})}{\partial W^{(l)}} = \dfrac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}}(a^{(l-1)})^{\mathrm{T}}$
      $\dfrac{\partial \mathcal{L}(y,\hat{y})}{\partial b^{(l)}} = \dfrac{\partial \mathcal{L}(y,\hat{y})}{\partial z^{(l)}}$
    end
    for $l = 1 : L$ do: (Update the parameters)
      $W^{(l)} \leftarrow W^{(l)} - \alpha\left(\dfrac{\partial \mathcal{L}(y,\hat{y})}{\partial W^{(l)}} + \lambda W^{(l)}\right)$
      $b^{(l)} \leftarrow b^{(l)} - \alpha\dfrac{\partial \mathcal{L}(y,\hat{y})}{\partial b^{(l)}}$
    end
  end
Until the error rate of the neural network on the validation set $\mathcal{D}_{\text{valid}}$ no longer decreases
Output parameters $W = \{W^{(1)},\dots,W^{(L)}\}$ and $b = \{b^{(1)},\dots,b^{(L)}\}$

The End