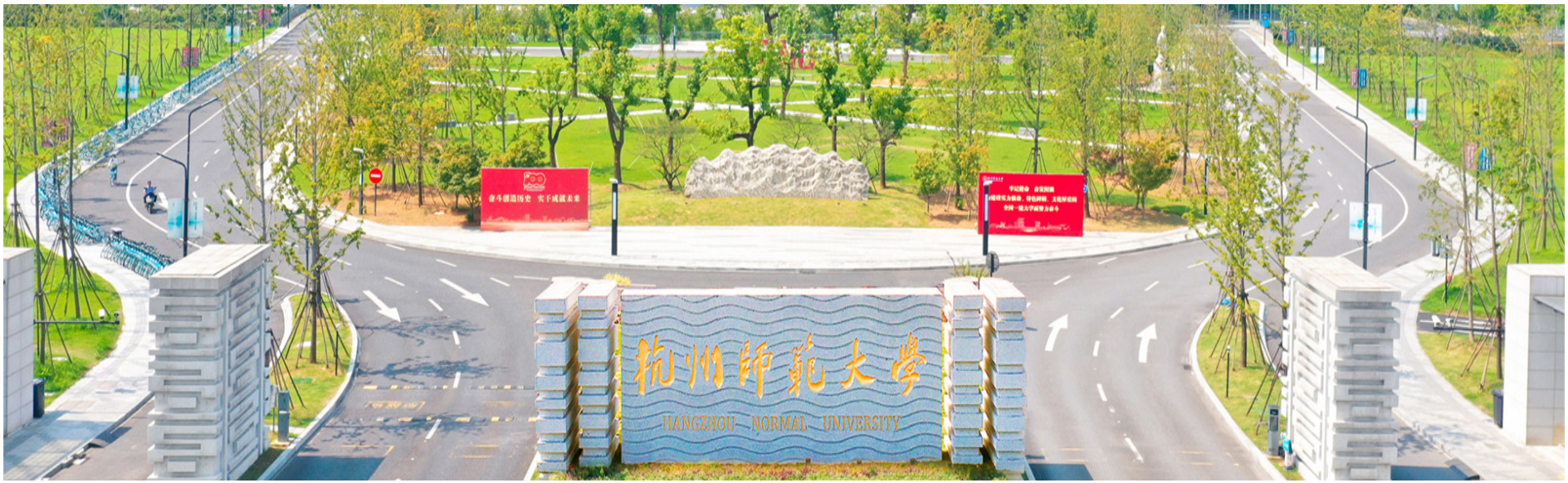
第七讲 - 神经网络基础

张建章

阿里巴巴商学院
杭州师范大学

2025-02-01



目录

1. 神经网络的起源与定义

最初源自 McCulloch 与 Pitts（1943）提出的“神经元”模型，旨在用命题逻辑描述生物神经元的计算行为。当代神经网络已脱离早期生物学启发，更侧重数学与工程实现。

现代神经网络的含义：由大量**计算单元（unit）**组成，每个单元将一组实数向量作为输入，经过加权求和和非线性映射后输出单一标量。

2. “深度学习”与网络架构

深度（deep）网络：当网络层数众多时，称之为“深度学习”模型（deep learning），能够自动从原始数据中学习到丰富的特征表示。

最小网络的表达能力：仅含**单隐层（one hidden layer）**的前馈网络，理论上即可逼近任意函数。

3. 与逻辑回归的比较

与逻辑回归共享加权求和和 sigmoid 等数学工具，但神经网络更具表达能力。逻辑回归依赖**手工设计的特征模板**；神经网络则普遍**避免手工设计特征**，直接以词向量为原始输入学习特征，这一过程也被称为**表示学习（representation learning）**。

目录

神经网络的基本构成单元是计算单元。每个计算单元**接收一组输入值**，并对其进行计算，最终**产生一个输出**。其计算过程如下：

① 首先，单元对输入值进行加权求和，并加入一个偏置项：

$$z = b + \sum_{i=1}^{n} w_i x_i$$

其中，w 为权重向量，x 为输入向量，b 为偏置项。为了简便，常使用向量表示该加权求和过程：

$$z = w \cdot x + b$$

② 接下来，神经单元会对加权和 z 应用一个非线性激活函数 $f(z)$，将其转换为输出 y：

$$y = f(z)$$

常见的激活函数包括sigmoid函数，它将输出映射到区间 $(0,1)$：

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

在此公式中，y 即为单元的**激活值**，它是神经网络中**单一计算单元的最终输出**。除了sigmoid函数，常用的激活函数还包括**tanh（双曲正切函数）**和**ReLU（修正线性单元）**。

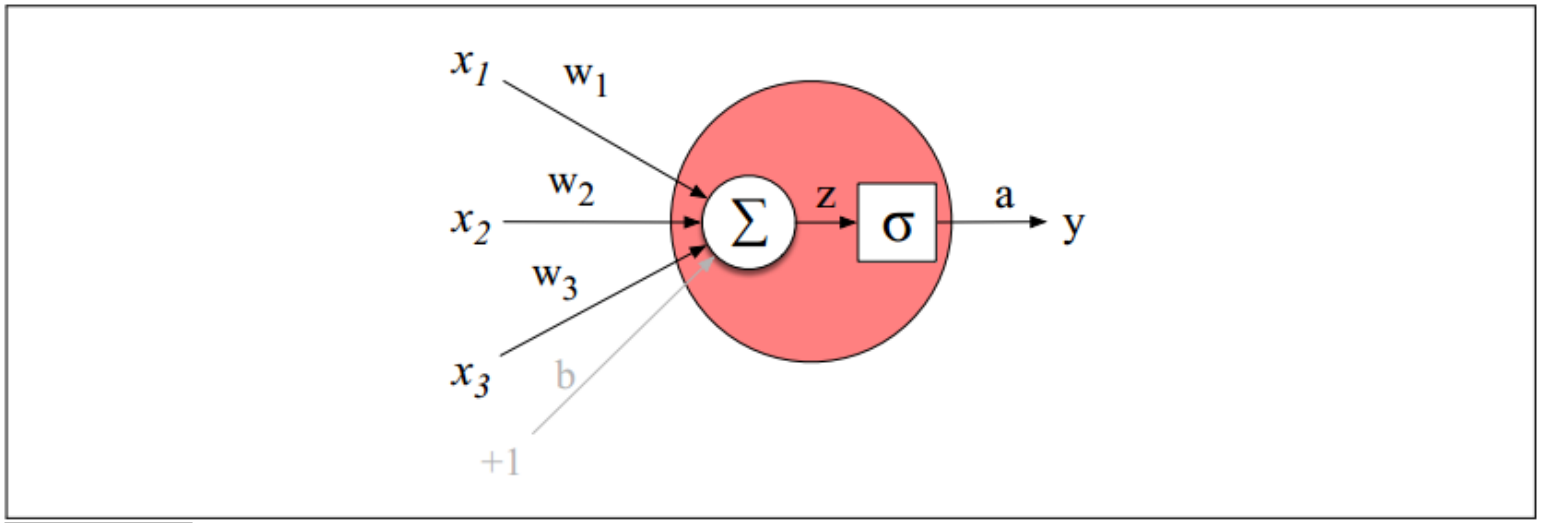


Figure 7.2 A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, z, and the output of the sigmoid, a. In this case the output of the unit y is the same as a, but in deeper networks we'll reserve y to mean the final output of the entire network, leaving a as the activation of an individual node.

其他常用激活函数

1. tanh（双曲正切函数）

$$y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

tanh函数是平滑的，且在整个定义域内可微，保证了优化过程中的梯度计算。

2. ReLU（修正线性单元）

$$y = \text{ReLU}(z) = \max(0, z)$$

与sigmoid和tanh不同，ReLU的梯度在正区间始终为1，因此能有效避免**梯度消失**问题，尤其是在网络层数较多的情况下，能加速训练过程。

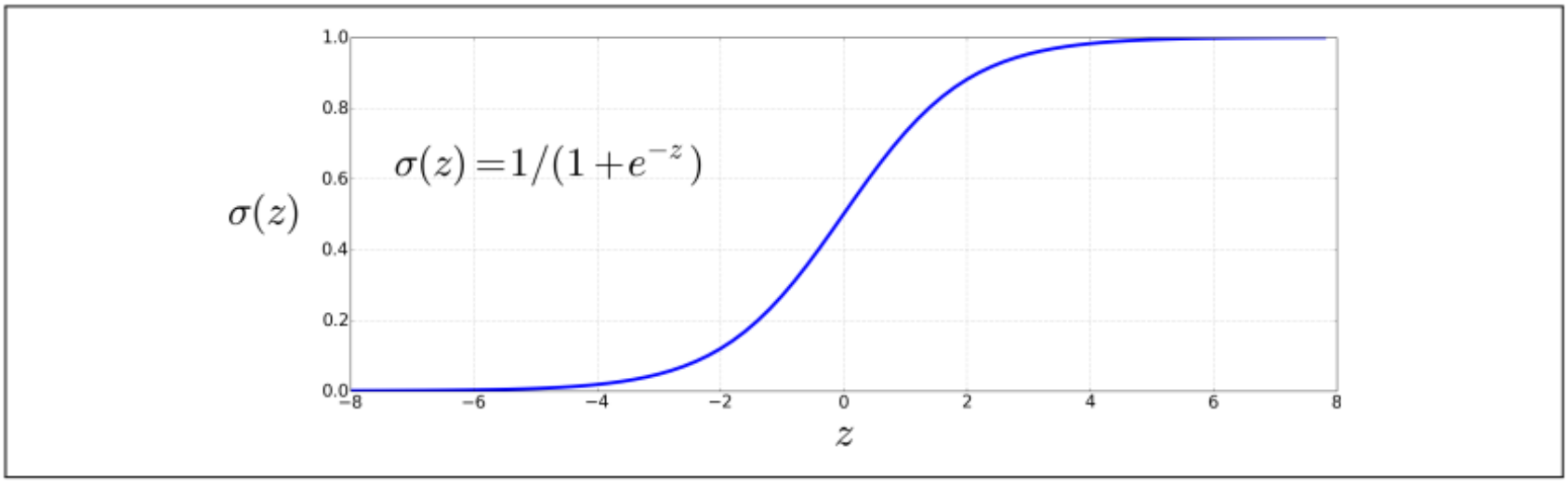


Figure 7.1 The sigmoid function takes a real value and maps it to the range $(0, 1)$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

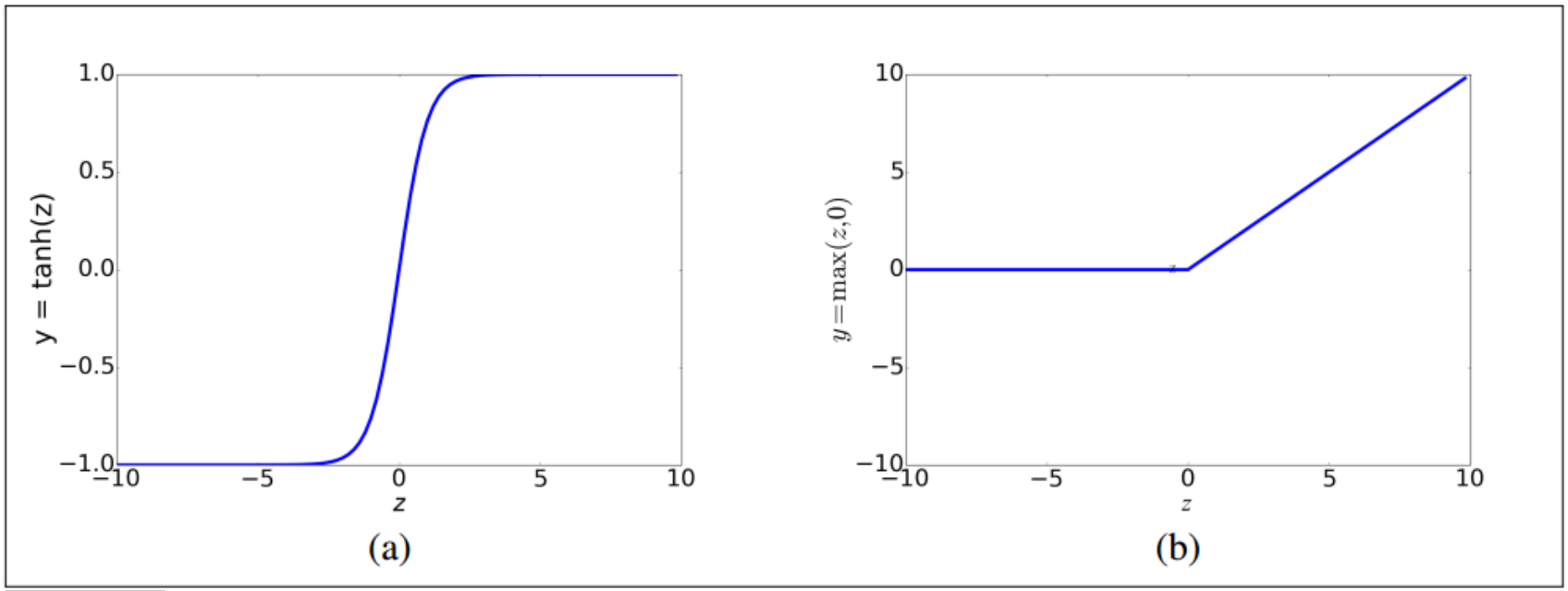


Figure 7.3 The tanh and ReLU activation functions.

目录

Minsky 和 Papert（1969）证明了**单个神经单元无法计算其输入的某些简单函数**。例如，感知机 (一种典型的单个神经元)虽然可以计算逻辑函数AND和OR，但无法计算XOR函数。三种逻辑函数的真值表如下：

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

1. 感知机 (Perceptron)

Perceptron 是一种简单的**人工神经元模型**，是神经网络的最早实现之一。它由 Rosenblatt 在 1958 年提出，主要用于二分类任务。感知机模型的核心思想是模拟生物神经元的工作方式，其中**输入信号通过加权求和后与一个阈值（偏置）进行比较，根据比较结果决定输出**。

$$y = \begin{cases} 0, & \text{如果 } w \cdot x + b \leq 0 \\ 1, & \text{如果 } w \cdot x + b > 0 \end{cases}$$

感知机的关键特点是它**使用线性分类器来划分输入空间，且它只能解决线性可分问题**。例如，AND 和 OR 等简单的逻辑运算可以通过感知机实现。下图显示了使用单一感知机建模AND和OR逻辑函数的**一组参数** (可行的参数有无穷多组)。

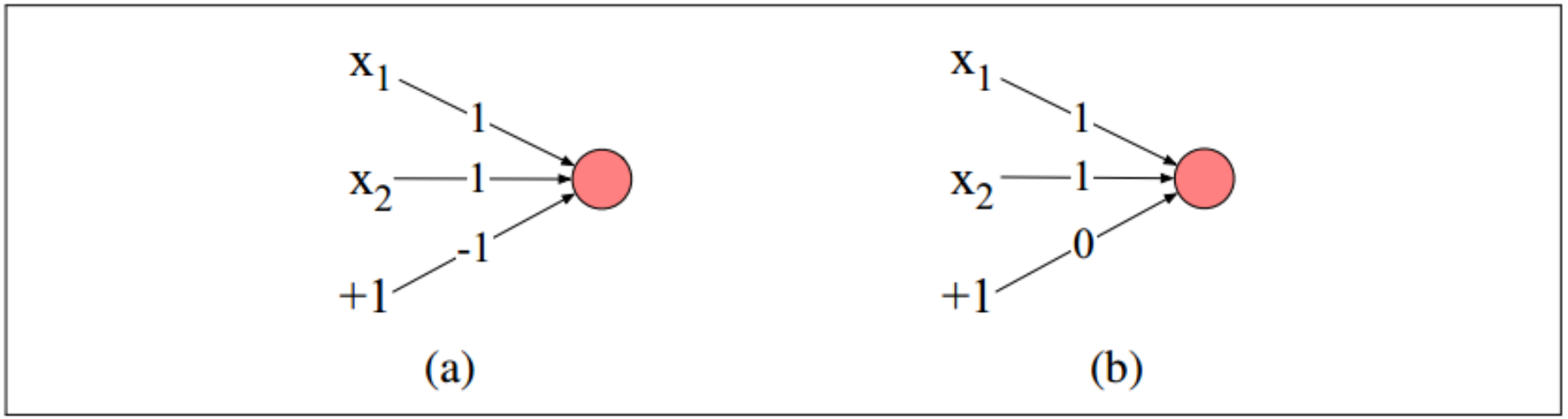


Figure 7.4 The weights w and bias b for perceptrons for computing logical functions. The inputs are shown as x_1 and x_2 and the bias as a special node with value $+1$ which is multiplied with the bias weight b. (a) logical AND, with weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) logical OR, with weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = 0$. These weights/biases are just one from an infinite number of possible sets of weights and biases that would implement the functions.

但对于 XOR 这类非线性可分问题，**单一的感知机**无法解决，使用多层神经元构建的神经网络可以解决。因此，构建多个单一神经元互相连接和堆叠的神经网络模型是有必要的。

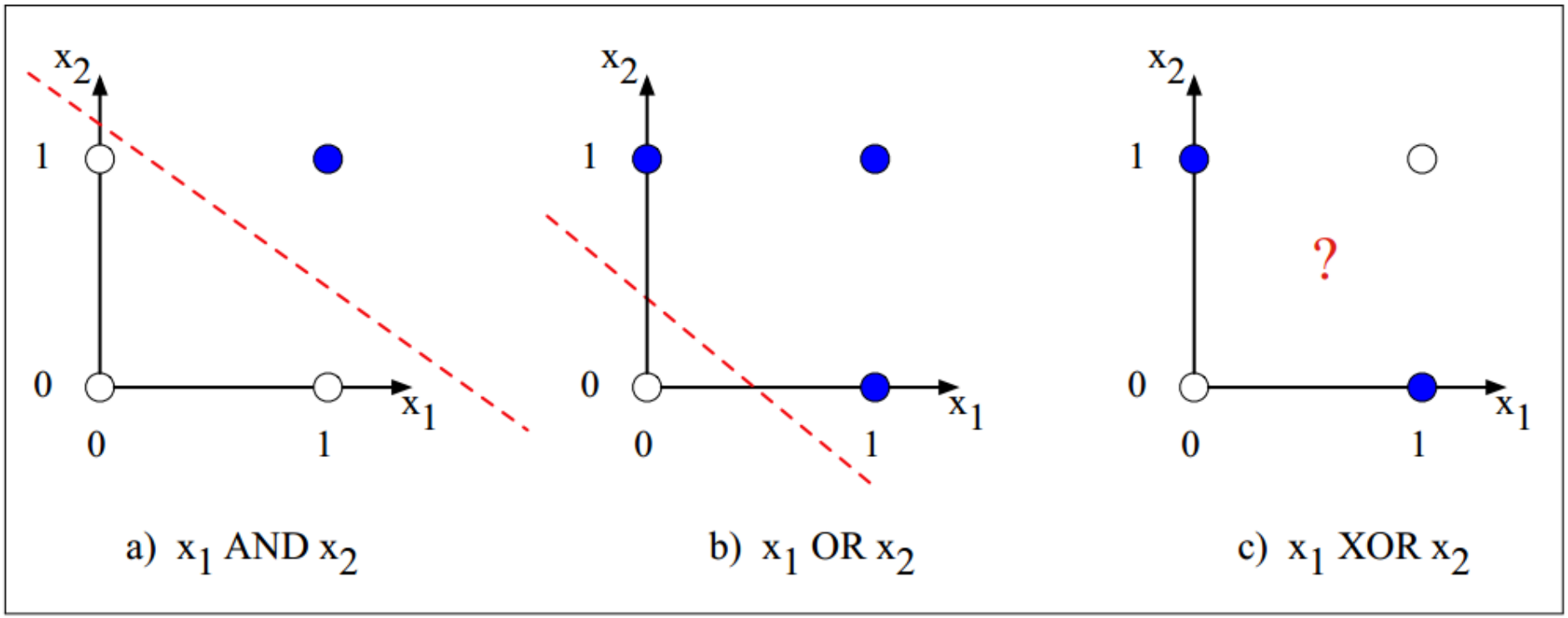


Figure 7.5 The functions AND, OR, and XOR, represented with input x_1 on the x-axis and input x_2 on the y-axis. Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

使用两层ReLU神经元解决XOR问题

虽然单个感知机无法计算 XOR，但可以通过多层网络来解决这一问题。下面举例说明，使用两层基于 ReLU 激活函数的神经单元解决 XOR 问题，下图的神经网络包括三个ReLU神经元，连边上标注了一组可行的权重值和偏置项：

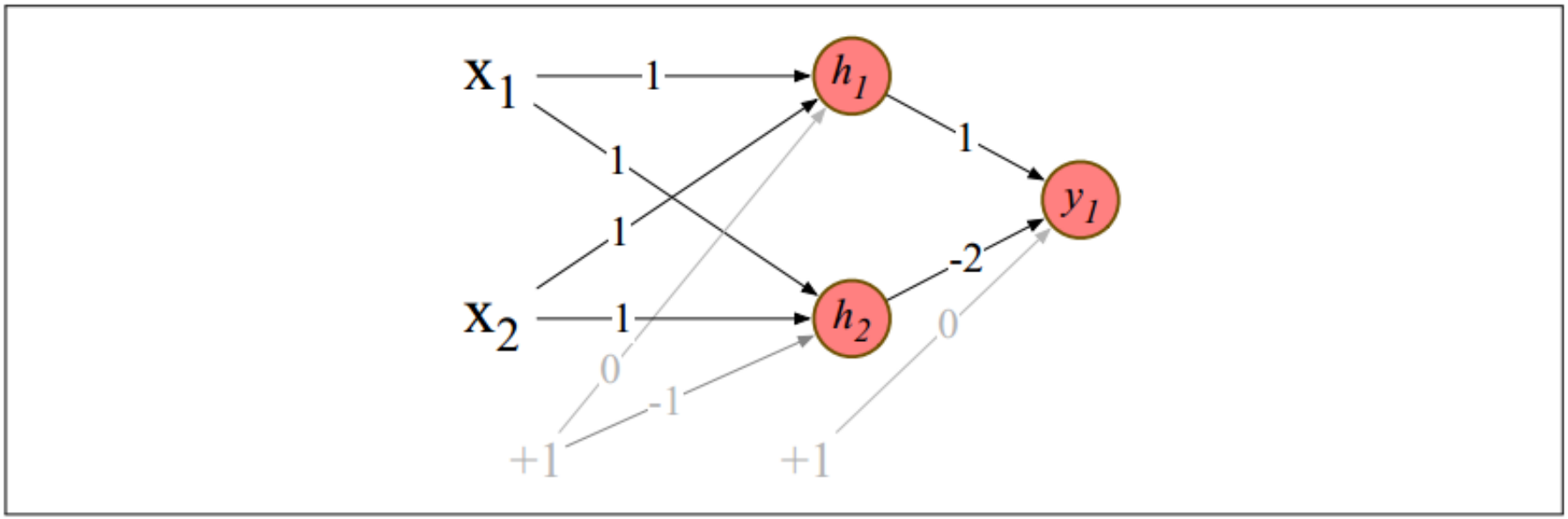


Figure 7.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them h_1, h_2 (h for "hidden layer") and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with the bias weights/units in gray.

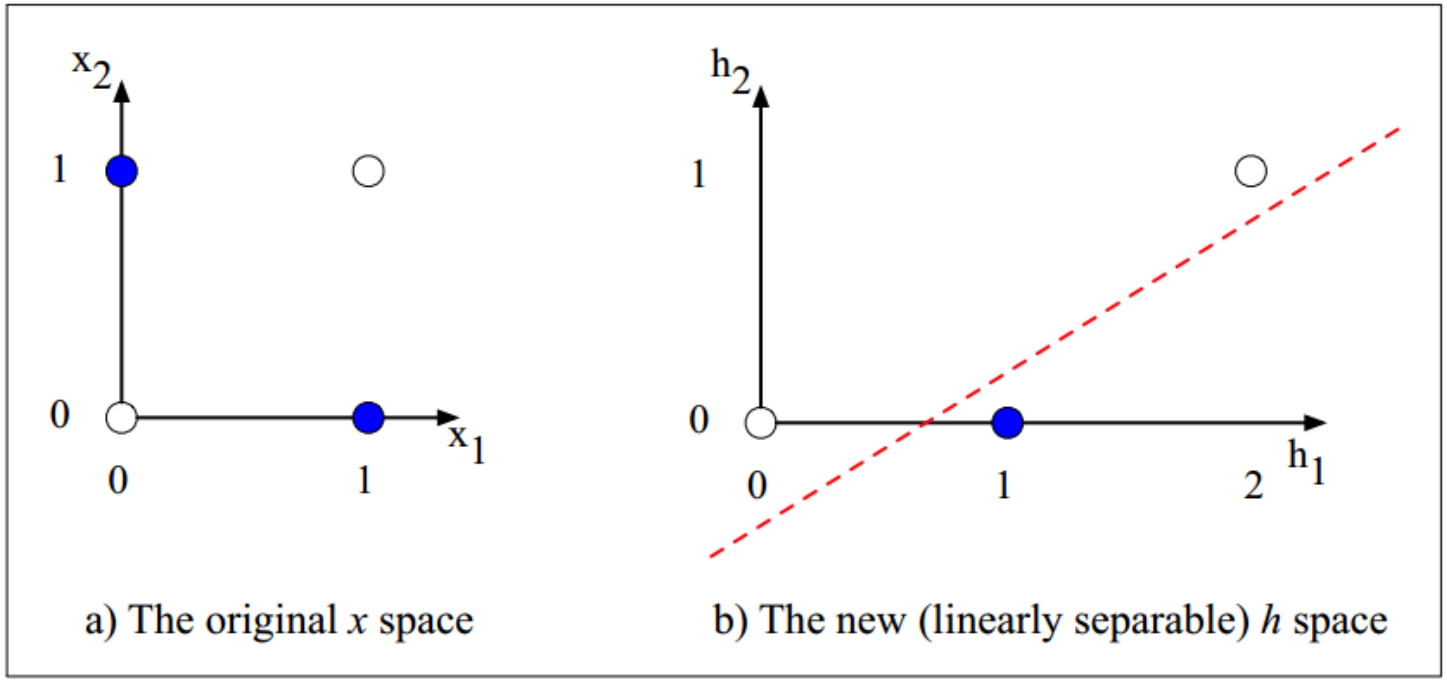
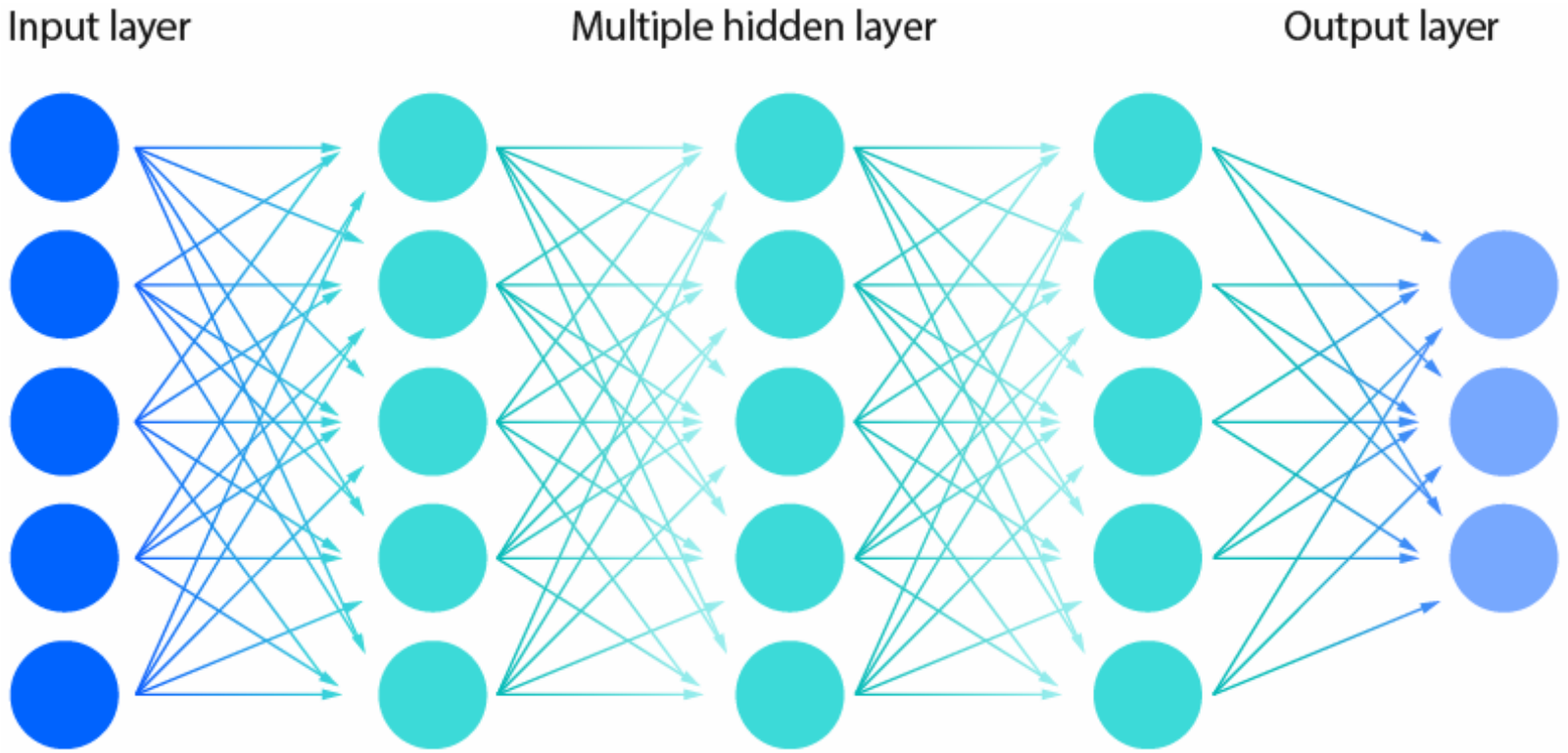


Figure 7.7 The hidden layer forming a new representation of the input. (b) shows the representation of the hidden layer, **h**, compared to the original input representation **x** in (a). Notice that the input point [0, 1] has been collapsed with the input point [1, 0], making it possible to linearly separate the positive and negative cases of XOR. After Goodfellow et al. (2016).

上图表明，经过隐藏层计算，神经网络将原始输入的线性不可分的表示空间转换为新的线性可分的表示空间。因此，神经网络的优势在于**通过层次化结构自动学习输入数据的特征表示**，而非依赖于手工设计的特征，从而克服了单一感知机无法计算 XOR 的限制。这一思想是深度学习的核心之一 (即，representation learning, 表示学习)。

目录

前馈神经网络（Feedforward Neural Network）：该网络由多个层组成，其中**每个层之间的单元相互连接**，没有反馈环路。每一层的输出会传递给上一层，计算过程**逐层向前推进**。



前馈网络在历史上也常被称为多层感知机（MLP，Multilayer Perceptrons），但实际上，现代神经网络的单元不仅仅是早期感知机的简单阶跃函数，而是使用了多种激活函数（如ReLU、sigmoid等）进行更复杂的非线性变换。

1. 前馈网络（Feedforward Network）定义

前馈网络是一个多层网络，通常由输入层、隐藏层和输出层构成。每一层的单元都会接收来自上一层的所有输出，进行加权求和，并通过非线性激活函数处理，进而传递到下一层。每一层的单元都被称为神经单元（neural unit）。标准的前馈网络是**全连接的 (Fully-Connected)**，即，隐藏层/输出层中的每个单元都与上一层所有单元相连。

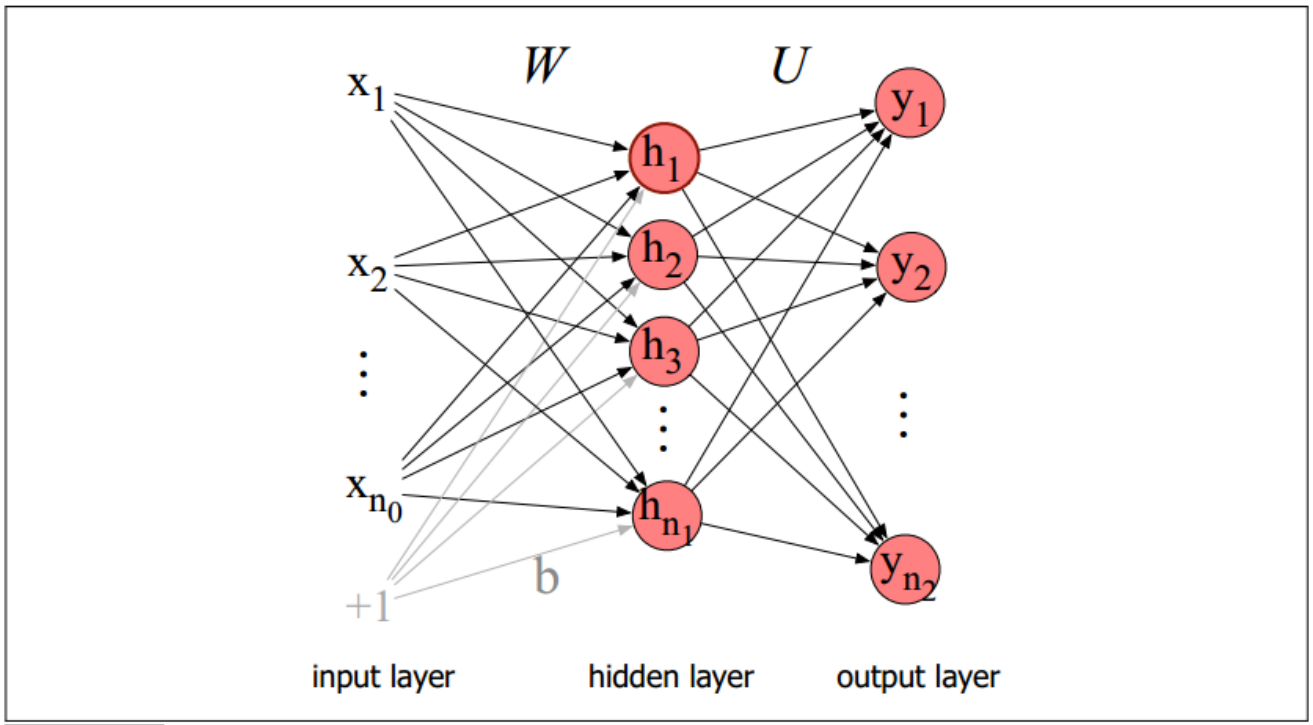


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

2. 网络的层次结构

一个典型的前馈网络由三类节点构成：

- **输入层（Input Layer）**：接收原始特征向量 $x \in \mathbb{R}^{n_0}$，例如文本分类中可能来自词嵌入、手工特征或其他表示。

- **隐藏层（Hidden Layer）**：是神经网络的核心部分，由多个神经元组成，每个单元对输入进行加权求和并应用激活函数。隐藏层通过权重矩阵 W 和偏置 b 将输入数据转化为新的表示。每个单元执行两步操作：

① 线性变换：计算加权和

$$z_j = \sum_{i=1}^{n_0} W_{ji}\, x_i \; + \; b_j,$$

或等价地矩阵形式 $z = Wx + b$。

② 非线性激活：应用激活函数 σ（可为 sigmoid、tanh、ReLU 等），得到隐藏层激活向量 $h = \sigma(z) = \sigma(Wx + b)$。

- **输出层（Output Layer）**：根据隐藏层的输出进行最终的预测。输出层的大小取决于任务的性质，例如二分类任务通常有一个输出节点，而多分类任务则有多个输出节点，输出结果是一个概率分布。以多分类为例，将隐藏层表示 $h \in \mathbb{R}^{n_1}$ 再次做线性变换 $z' = Uh$，并通过 softmax 函数归一化为概率分布：

$$y_i = \frac{\exp(z'_i)}{\sum_{k=1}^{n_2} \exp(z'_k)}, \quad y \in \mathbb{R}^{n_2}, \sum_i y_i = 1.$$

简洁表示：可用以下矩阵计算公式表示上述带单隐层的前馈网络的前向计算过程。

$$\begin{aligned} h &= \sigma(Wx + b) \\ z &= Uh \\ y &= \mathrm{softmax}(z) \end{aligned}$$

其中 $W \in \mathbb{R}^{n_1 \times n_0}$, $b \in \mathbb{R}^{n_1}$, $U \in \mathbb{R}^{n_2 \times n_1}$，输出 $y \in \mathbb{R}^{n_2}$ 满足 $\sum_i y_i = 1$。

3. 与逻辑回归的关系

一层网络：等价于逻辑回归（无隐藏层，只做线性变换+softmax）。

多层网络：则在输入与输出之间引入一层或多层非线性投影，相当于“自动生成”的中间特征，再由逻辑回归层做分类。

等价视角：单层网络（逻辑回归）即无隐藏层的前馈网；增加隐藏层即相当于在特征与分类器之间加入了**非线性投影**，使网络可学习复杂函数。隐藏层可自动“抽取”输入中的高阶、非线性组合特征；**多个隐藏层（深度网络）更擅长分层表示，适合大规模、数据充足的任务**。

多层网络的计算

对于深度神经网络，每一层的计算过程可以递归表示。通过增加更多层，网络能够学习到更复杂的特征表示，从而提高分类任务的性能。每一层的计算步骤包括：

- 计算每一层的加权和 $z[i] = W[i] \cdot a[i-1] + b[i]$
- 应用激活函数 $a[i] = g[i](z[i])$
- 最终输出 $y = a[n]$，其中 n 为网络的总层数。

1. 统一的层次符号表示

① 层号与参数

- 以方括号“[ ]”标注层号：第 0 层为输入层，第 1、2…n 层分别为隐藏层和输出层。

- 每层的权重矩阵记作 $W[i]$，偏置向量记作 $b[i]$；第j层包含 n_i 个神经元。

② 激活与中间变量

- $a[i]$ 表示第 i 层的输出激活（activation），其中 $a[0]$即输入向量 x；

- $z[i]$ 表示第 i 层在施加非线性函数之前的线性变换结果，$z[i] = W[i]\ a[i–1] + b[i]$。

③ 激活函数

- 每层可选不同的非线性函数$g[i](\cdot)$，隐藏层常用ReLU或tanh，输出层若做分类则选softmax。

如此，深度为n的前馈网络的前向计算可统一写作：

for $i = 1...n$:

$$z[i] = W[i]\ a[i–1] + b[i]$$

$$a[i] = g[i](z[i])$$

$$\hat{y} = a[n]$$

其中$\hat{y}$即网络的最终输出（分类概率分布）。

2. Logits 与 softmax

- 最后一层的线性变换结果$z[n]$常称为“logits”，即 softmax 之前的**打分向量**。

- 通过$softmax(z[n])$将logits归一化为概率分布，用以多分类任务。

3. 非线性激活的必要性

- 若网络全部采用线性激活（或无激活函数），则任意多层的线性变换可以合并为一次线性变换：

$$z^{(2)} = W^{(2)}(W^{(1)}x+b^{(1)})+b^{(2)} = (W^{(2)}W^{(1)})x+(W^{(2)}b^{(1)}+b^{(2)}) = W'x+b'$$

损失了多层网络的表达能力，退化为单层线性模型。

- 因此必须在各层之间引入非线性函数（如 ReLU、tanh 等），才能真正提升模型的表示能力。

4. 偏置项的“虚拟输入”替换

为了简化符号，可将偏置b视作一个恒为1的“虚拟输入”a_0的权重：

- 在每层输入向量最前端添加一维$a_0 = 1$；
- 原有的偏置b即对应这条虚拟连接的权重$W[:, 0]$。

这样可将$h = \sigma(Wx + b)$简化为 $h = \sigma(Wx)$，其中x已隐含$a_0 = 1$，并将b合并进W矩阵的第一列。

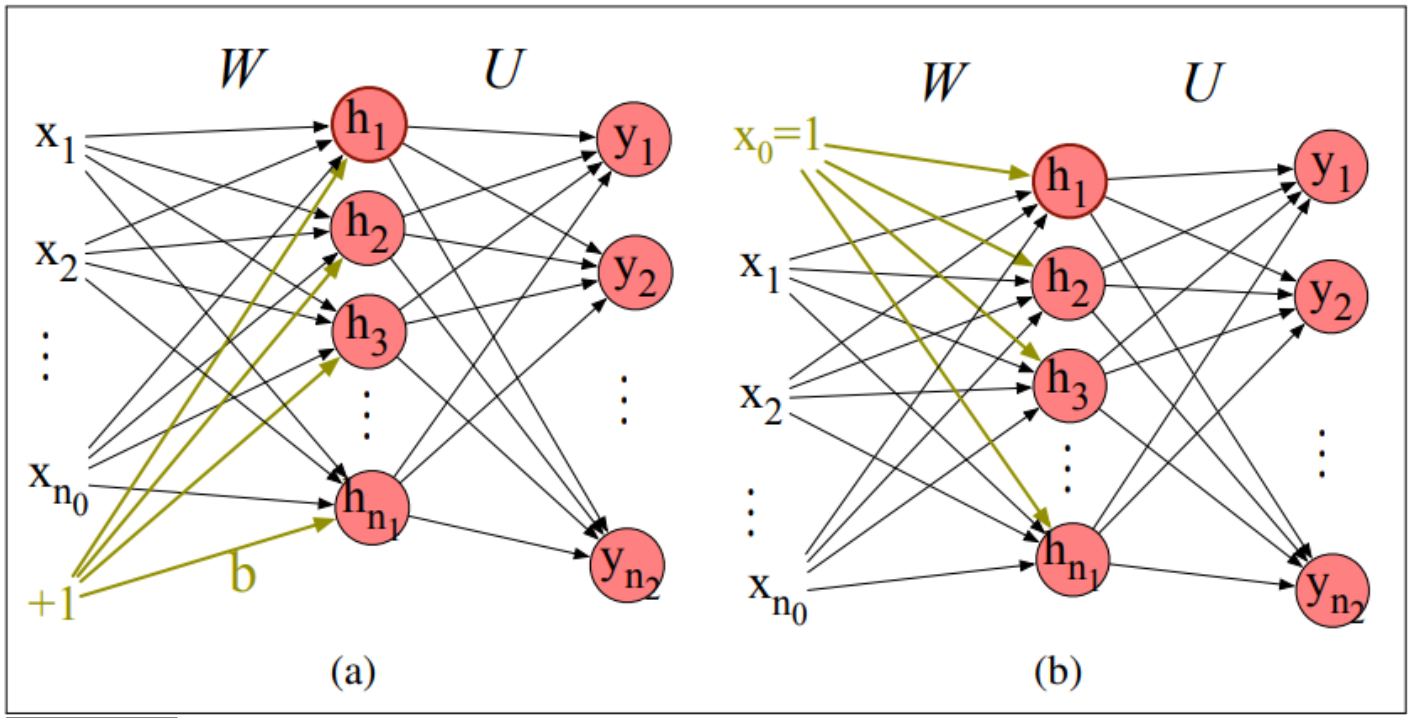


Figure 7.9 Replacing the bias node (shown in a) with x_0 (b).

目录

1. 情感分析的2层前馈网络

从一个简单的2层情感分类器开始，假设将逻辑回归分类器扩展为一个包含隐藏层的前馈网络。输入特征可以是手工设计的标量特征，例如：

- x_1：文档中单词的计数，
- x_2：正面词汇表中单词的计数，
- x_3：如果文档中包含“no”，则为1，否则为0，等等。

输出层 $\hat{y}$ 可以有两个节点（分别代表正面和负面情感），或者三个节点（分别代表正面、负面和中立情感）。输出的每个节点值表示对应情感的概率。

该网络结构的计算公式如下：

$$x = [x_1, x_2, \ldots, x_N] \quad (\text{每个} x_i \text{ 为手工设计的特征})$$

$$h = \sigma(Wx + b)$$

$$z = Uh$$

$$\hat{y} = \text{softmax}(z)$$

其中：

- x 是输入特征向量，
- W 是权重矩阵，
- b 是偏置向量，
- σ 是激活函数（例如 sigmoid 或 ReLU），
- U 是输出层的权重矩阵，
- z 是输出层的得分向量，经过 softmax 转化为概率分布 $\hat{y}$。

2. 非线性特征的表示学习

通过在逻辑回归模型中加入隐藏层，前馈神经网络能够表示特征之间的非线性交互关系。这使得网络能够更好地学习和分类情感，而不仅仅依赖于线性特征。这样，网络不仅仅是对输入特征的线性组合，而是通过多层变换来捕捉复杂的模式。

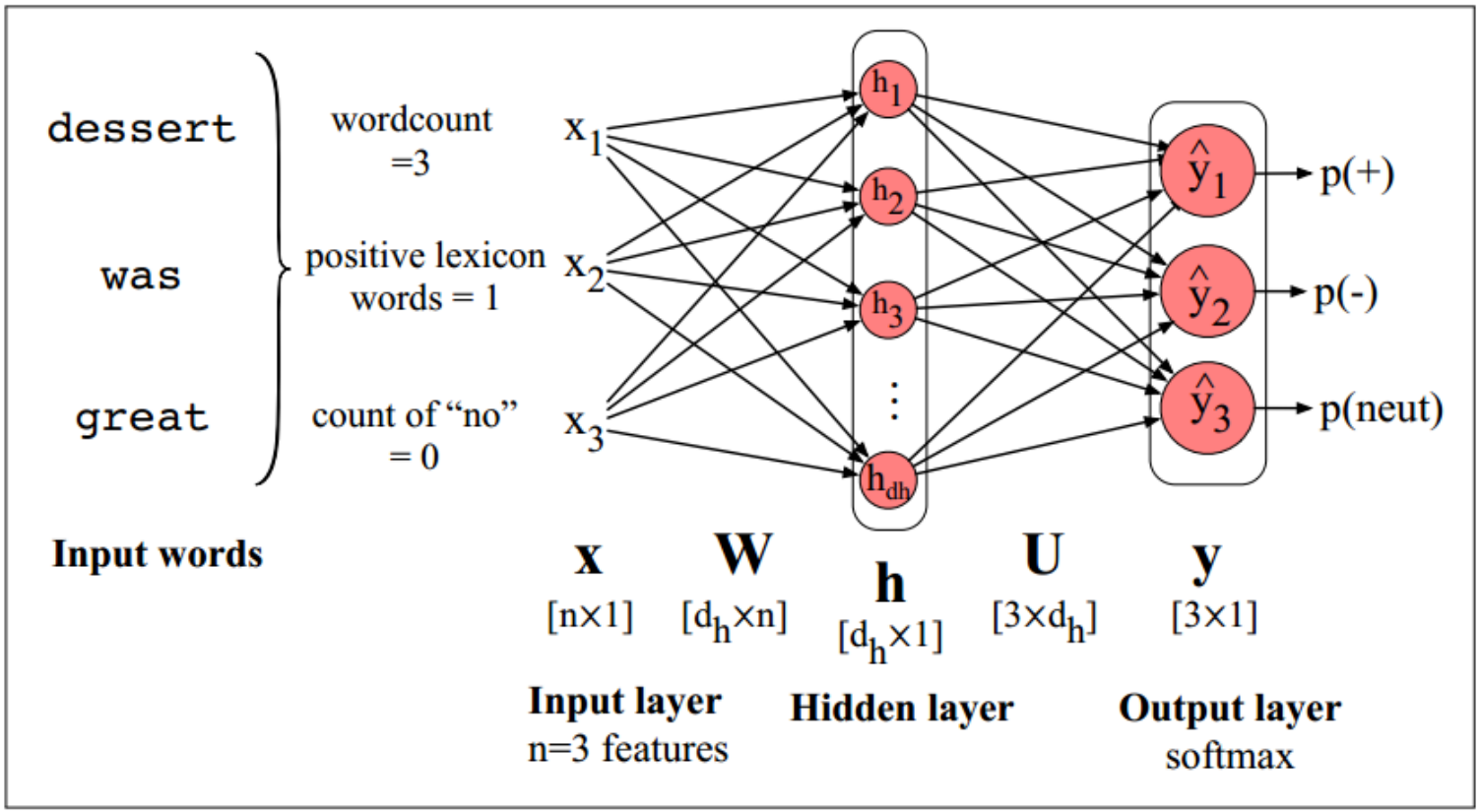


Figure 7.10 Feedforward network sentiment analysis using traditional hand-built features of the input text.

3. 特征表示的学习

目前，大多数NLP任务中，不再依赖手工设计的特征，而是依赖深度学习自动从数据中学习特征。例如，单词的表示可以通过词嵌入（如 word2vec 或 GloVe）来学习，这些嵌入能够捕捉单词之间的语义相似性。对于情感分析，神经网络会学习到如何通过这些嵌入表示上下文信息，并根据上下文推断出情感类别。

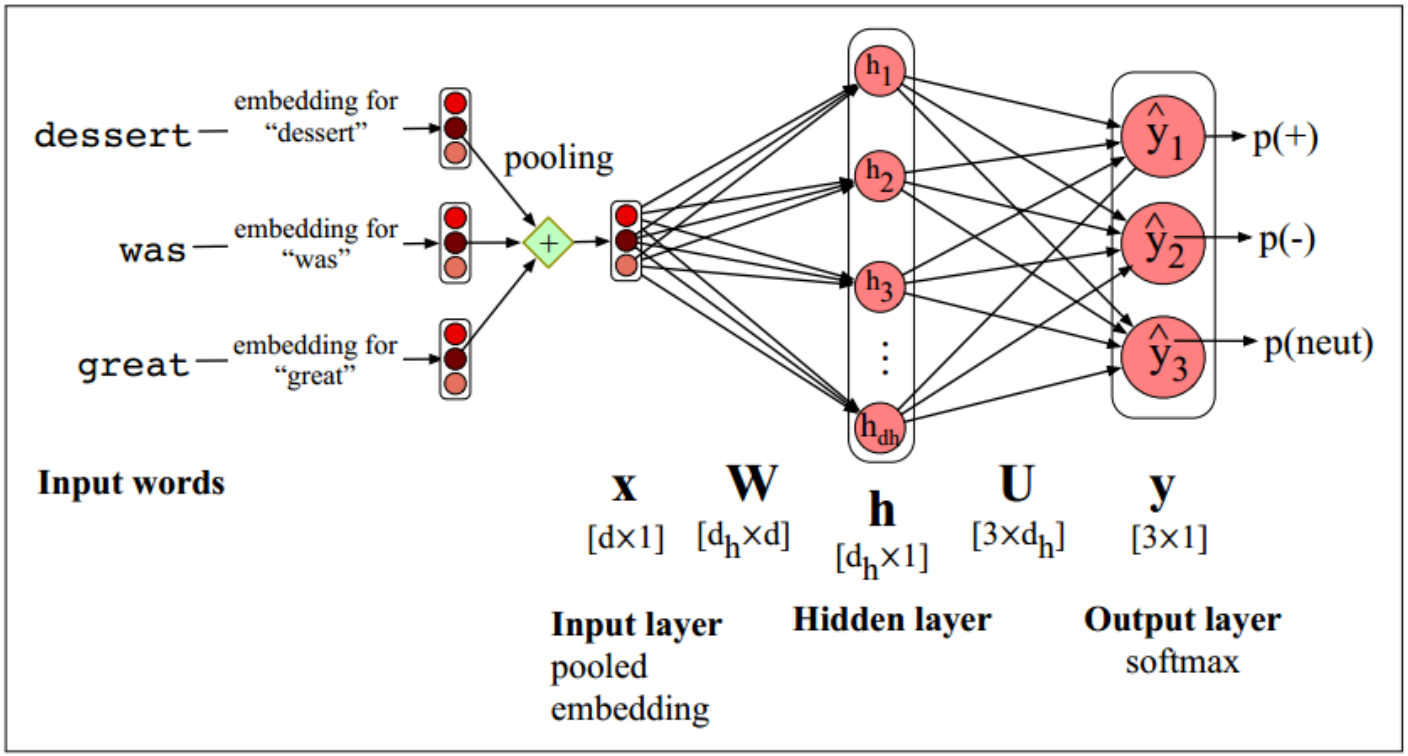


Figure 7.11 Feedforward network sentiment analysis using a pooled embedding of the input words.

4. 池化操作

在处理文本时，通常会将多个单词的词嵌入汇总为一个整体表示。最简单的方式是通过池化（pooling）函数，例如平均池化。假设有一个由多个词组成的文本，每个词的嵌入为 $e(w_1), e(w_2), \ldots, e(w_n)$，可以通过对所有词的嵌入求平均来得到该文本的整体表示：

$$x_{\text{mean}} = \frac{1}{n} \sum_{i=1}^{n} e(w_i)$$

这种方法能够有效地**将文本中的信息压缩成一个固定维度的表示**，便于后续的分类任务。

其他池化方式：除了平均池化，还可以使用其他池化策略，如最大池化（max pooling）。最大池化会选择每个维度的最大值，以捕捉文本中最重要的特征。这些池化方式有助于提升模型的泛化能力和性能。

通过将前馈网络应用于情感分析等NLP分类任务，模型能够通过自动学习特征表示，**避免手工设计特征的繁琐过程**，并且能够更好地捕捉文本中的复杂模式。这种方法为NLP任务提供了强大的灵活性和效果。

在二分类任务中，从概率论角度看，单节点+Sigmoid与双节点+Softmax在功能上等价：

$$\text{sigmoid}(z) = \frac{1}{1+e^{-z}} \quad \Longleftrightarrow \quad \text{softmax}([0, z]) = \left[e^0/(1+e^z),\ e^z/(1+e^z)\right],$$

但双节点+Softmax更符合「统一多分类框架」的设计思路，也在实践中带来更好的数值与工程便利性。

目录

1. 问题设定与总体流程

输入：样本特征向量 x（可以是原始特征或嵌入表示）。

网络输出：$\hat{y}$，通常为各类别的预测概率。

目标：找到所有层的权重 $W^{[i]}$ 和偏置 $b^{[i]}$（统称参数 θ），使得对训练集中所有样本计算的总损失最小。

核心步骤：

① 定义损失函数，衡量 $\hat{y}$ 与 y 的差距；

② 计算损失相对于各参数的梯度；

③ 利用梯度下降（或其变种）迭代更新参数。

2. 损失函数（Loss Function）

多分类使用交叉熵损失（Cross-Entropy Loss），与逻辑回归完全一致，输出概率向量 $\hat{y} \in \mathbb{R}^K$，真标签为 one-hot 向量 y）：

$$L_{\mathrm{CE}}(\hat{y}, y) = -\sum_{k=1}^{K} y_k \log \hat{y}_k = -\log \hat{y}_c \quad (c \text{ 为正确类别})$$

该损失兼具概率解释与数值稳定性，易于与 softmax 输出层搭配使用。

3. 梯度计算（Gradient Computation）

- 对于浅层（单层）网络，可直接将损失对权重 w_j 求导，得到类似逻辑回归的更新公式：$\frac{\partial L_{\mathrm{CE}}(\hat{y},y)}{\partial w_{k,i}} = -(y_k - \hat{y}_k)x_i$

- 对于深层网络，需要**链式法则（chain rule）**将误差反向传播至各层参数。

关键算法：误差反向传播（Backpropagation），实质为对“计算图”中每个节点依次应用链式法则，先完成一次前向传播，计算各层中间变量，再做一次后向传播，累计梯度。

计算图 (Computation Graph)

1. 构建计算图：将网络计算拆解为节点（加、乘、激活等基本操作）与有向边（数据流）。

2. 前向传播：按顺序计算各节点的值，直至输出并计算损失 L。

3. 后向传播：从损失节点向前，依次计算每条边上的梯度（上游梯度 $\times$ 本地梯度），最终得到对每个参数的偏导数。

4. 参数更新：以梯度下降为例，

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta},$$

其中 η 为学习率。

下面以一个简单的两层神经网络做二分类为例，演示计算图与反向传播计算$\partial L/\partial z$，简便起见，输出层包含一个使用sigmoid函数作为激活函数的神经元。

1. 构建计算图

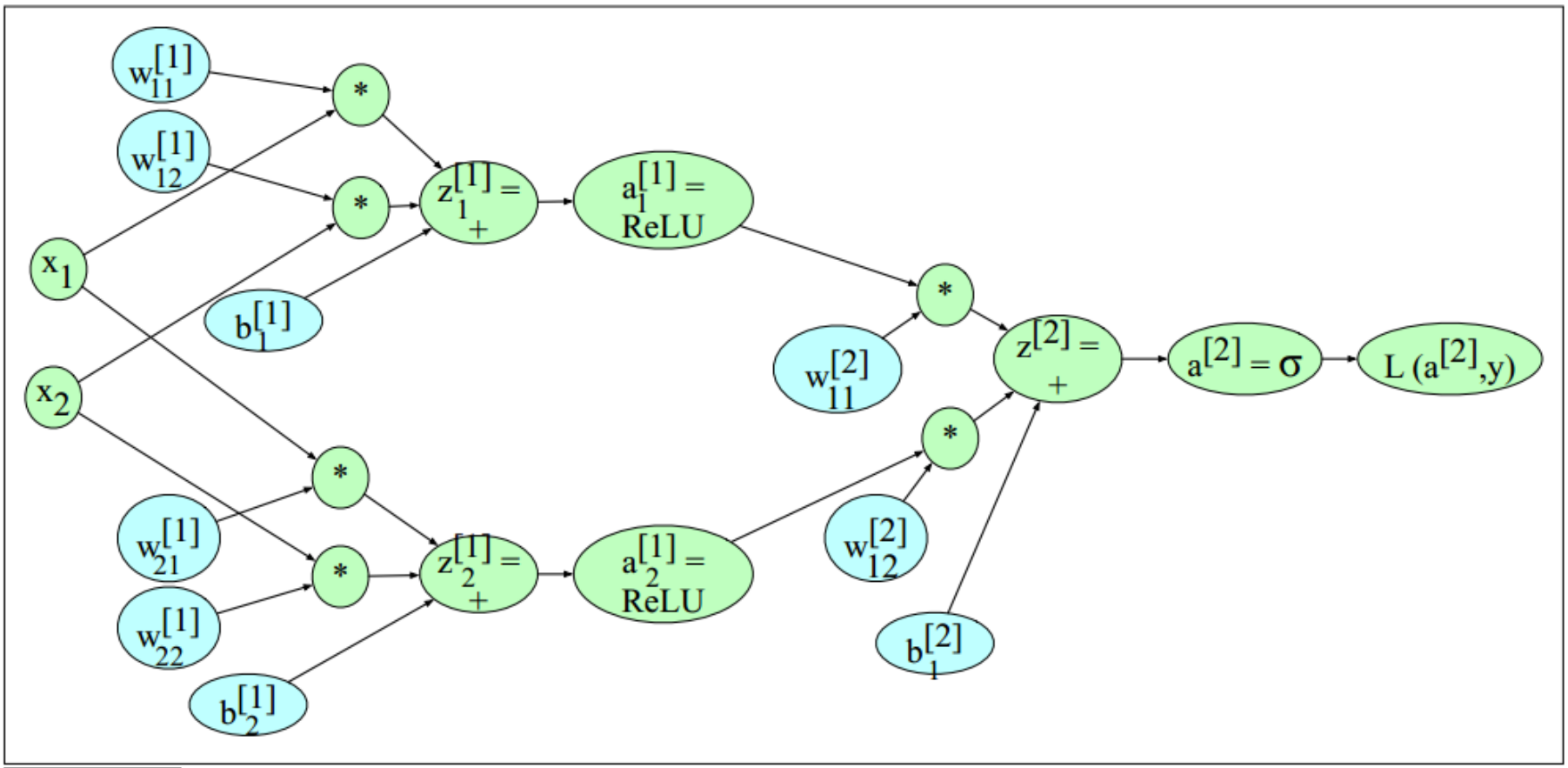


Figure 7.15 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input units and 2 hidden units. We've adjusted the notation a bit to avoid long equations in the nodes by just mentioning the function that is being computed, and the resulting variable name. Thus the * to the right of node $w_{11}^{[1]}$ means that $w_{11}^{[1]}$ is to be multiplied by x_1, and the node $z^{[1]} = +$ means that the value of $z^{[1]}$ is computed by summing the three nodes that feed into it (the two products, and the bias term $b_i^{[1]}$).

2.前向计算（forward pass）

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\, a^{[0]} + b^{[1]} \\
a^{[1]} &= \mathrm{ReLU}\left(z^{[1]}\right) \\
z^{[2]} &= W^{[2]}\, a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma\left(z^{[2]}\right) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

- $a^{[0]} = x$ 是输入向量；

- $W^{[1]}, b^{[1]}$ 是第一层的权重和偏置，计算出预激活向量 $z^{[1]}$；

- 然后用 ReLU 非线性得到隐藏层激活 $a^{[1]}$；

- 第二层（输出层）同理：线性映射得到 $z^{[2]}$，再用 sigmoid（σ）映射成输出概率 $\hat{y}$；

- 最终 $\hat{y} = a^{[2]}$ 即模型对正类（或对应类别）的预测概率。

3. 后向传播

① 交叉熵损失（cross-entropy loss）：

$$\begin{aligned} L_{\mathrm{CE}}(\hat{y}, y) &= -\big[y \log \hat{y} + (1-y) \log(1-\hat{y})\big] \\ &= -\left[y \log a^{[2]} + (1-y) \log\big(1-a^{[2]}\big)\right] \end{aligned}$$

这是二分类（sigmoid 输出）的负对数似然，用作网络的训练目标。

② 各激活函数的导数：

$$\frac{d}{dz}\sigma(z) = \sigma(z)\big(1-\sigma(z)\big),$$

$$\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z),$$

$$\frac{d}{dz}\mathrm{ReLU}(z) = \begin{cases} 0, & z < 0, \\ 1, & z \geq 0. \end{cases}$$

在反向传播时，需要这些局部梯度来应用链式法则。

③ **链式法则（chain rule）**：将损失对输出$a^{[2]}$的梯度，与输出对预激活z的梯度相乘。

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z} = a^{[2]} - y$$

$$\frac{\partial L}{\partial a^{[2]}} = -\Big(y \cdot \frac{\partial \log a^{[2]}}{\partial a^{[2]}} + (1-y) \cdot \frac{\partial \log(1-a^{[2]})}{\partial a^{[2]}} \Big)$$

再利用 $\frac{\partial \log a^{[2]}}{\partial a^{[2]}} = \frac{1}{a^{[2]}}$, $\frac{\partial \log(1-a^{[2]})}{\partial a^{[2]}} = -\frac{1}{1-a^{[2]}}$ 可得

$$\frac{\partial L}{\partial a^{[2]}} = -\Big(\frac{y}{a^{[2]}} + (1-y) \cdot \big(-\frac{1}{1-a^{[2]}}\big) \Big) = -\Big(\frac{y}{a^{[2]}} - \frac{1-y}{1-a^{[2]}} \Big)$$

利用sigmoid函数的导数，可得$\frac{\partial a^{[2]}}{\partial z} = a^{[2]}(1-a^{[2]})$

训练细节

1. 非凸优化与参数初始化

- **非凸性**：神经网络的损失函数通常是**高度非凸的**，多参数、多层结构使得优化问题比传统的逻辑回归复杂得多。

- **随机初始化**：与逻辑回归可将所有权重、偏置初始化为 0 不同，神经网络**须将所有权重初始化为小的随机值**，以打破对称性，保证各隐藏单元在训练初期能学习到不同特征。

2. 输入标准化（Input Normalization）

- **均值归零、方差归一**：将输入特征（或嵌入）**按维度**减去训练集均值、再除以标准差，有助于加快训练收敛、减少梯度消失/爆炸风险。

3. 正则化（Regularization）

- **Dropout**：最常用的正则化技术之一。在每次参数更新时，以概率 p 随机“屏蔽”某些隐藏单元（即将其输出置零），并对其余保留单元的输出进行适当缩放；该操作等价于对不同子网络的随机组合，能有效降低过拟合风险 (Hinton et al. 2012；Srivastava et al. 2014)。

4. 超参数调优（Hyperparameter Tuning）

- **超参数定义：**与网络参数（权重 W、偏置 b）不同，超参数包括学习率 η、mini-batch 大小、网络架构（层数、每层单元数、激活函数类型）、正则化策略（dropout 比例、权重衰减系数）等。

- **调优策略：**超参数不通过梯度下降从训练集直接学习，而需在开发集（dev set）上系统搜索（如网格搜索、随机搜索或贝叶斯优化），以获得最佳泛化性能。

5. 优化算法变体

- 除传统的批量梯度下降（BGD）和随机梯度下降（SGD）外，现代训练过程中常用 Adam、RMSprop、AdaGrad 等**自适应学习率方法**，以加快收敛、减弱手工学习率调节的工作量。

6. 计算框架与硬件加速

- **计算图自动微分：**现代深度学习框架（如 PyTorch、TensorFlow）可对用户定义的计算图自动执行反向传播，免去手工推导和实现梯度的繁琐。利用GPU并行计算矩阵运算，可大幅提升大规模神经网络的训练效率，上述框架已提供对多GPU和分布式训练的支持。

未完待续