7 神经网络

> “机器在单位数量较多时可以表现出非常复杂的行为。”
>
> ——艾伦·图灵（1948年）《智能机器》，第6页

神经网络是语言处理的基本计算工具，而且是非常古老的工具。它们之所以被称为“神经”网络，是因为其起源于**McCulloch-Pitts神经元模型**（McCulloch和Pitts，1943年），这是一种将生物神经元简化为可以用命题逻辑描述的计算元素的模型。然而，现代语言处理中的神经网络已不再依赖这些早期的生物启发。

现代神经网络实际上是一个由小型计算单元组成的网络，每个单元接收一个输入值向量并输出一个单一的值。本章介绍应用于分类的神经网络。我们介绍的架构称为**前馈网络(feedforward network)**，因为计算是从一层单元到下一层单元逐步进行的。现代神经网络的使用通常被称为**深度学习**，因为现代网络通常非常“**深**”（即拥有许多层）。

神经网络在数学上与逻辑回归有许多相似之处。但神经网络比逻辑回归更强大，事实上，最简单的神经网络（技术上只有一层“隐藏层”）就可以学习任何函数。

神经网络分类器与逻辑回归的另一点不同是：在逻辑回归中，我们通过开发基于领域知识的各种丰富特征模板，将回归分类器应用于许多不同任务。而在使用神经网络时，通常避免使用手工设计的特征，转而构建神经网络，直接将原始词汇作为输入，并在学习分类的过程中自动学习特征。我们在第6章中已经看到了这种嵌入表示学习的例子。特别是非常深的网络在表示学习方面表现得尤为出色。因此，对于那些提供足够数据以自动学习特征的任务，深度神经网络是正确的工具。

在本章中，我们将介绍作为分类器的前馈网络，并将其应用于语言建模的简单任务：为词序列分配概率并预测后续词汇。在接下来的章节中，我们将介绍神经网络模型的许多其他方面，例如**循环神经网络**（第8章）、**Transformer**（第9章）和掩码语言建模（第11章）。

7.1 神经单元

神经网络的构建模块是单个计算单元。一个单元接收一组实数作为输入，对其进行某种计算，并生成输出。

核心上，神经单元通过对其输入进行加权求和，其中有一个额外的项称为偏置项。给定一组输入值 $x_1...x_n$ ，单元有一组相应的权重 $w_1...w_n$ 和一个偏置 b ，因此加权和 z 可以表示如下:

$$z = b + \sum_i w_i x_i$$

我们用权重向量 w 、标量偏置 b 和输入向量 x 来表示 z ，并将求和替换为更简便的点积（内积）：

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

如方程7.2所定义， z 只是一个实数。

最后，神经单元不直接将线性函数 z 作为输出，而是对 z 应用一个非线性函数 f 。我们将此函数的输出称为单元的激活值 a 。由于我们只是在建模单个单元，该节点的激活值实际上就是整个网络的最终输出，通常我们称之为 y 。因此， y 的值定义如下:

$$y = a = f(z)$$

接下来我们将讨论三种常用的非线性函数 f （包括 Sigmoid、tanh 和整流线性单元 ReLU），但为了教学方便，我们从 Sigmoid 函数开始，因为我们在第5章中已经见过它:

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

Sigmoid 函数（如图7.1所示）有许多优点；它将输出映射到 (0, 1) 范围内，这在压缩异常值时非常有用。并且它是可微的，正如我们在第??节中看到的，这对学习过程非常有帮助。

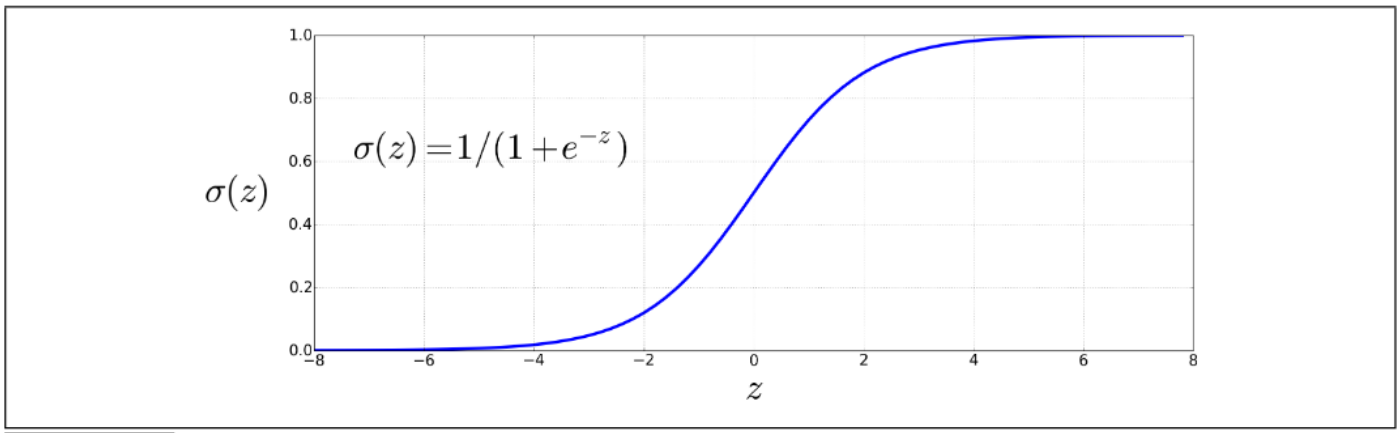

图 7.1 sigmoid 函数采用实数值并将其映射到范围 (0, 1)。它在 0 附近几乎呈线性，但离群值会被压缩到 0 或 1。

代入方程式7.2 代入方程式7.3 给出了神经单元的输出：

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1+\exp(-(\mathbf{w} \cdot \mathbf{x} + b))}$$

图7.2显示了一个基本神经单元的最终示意图。在这个示例中，单元接收三个输入值 x_1 、 x_2 和 x_3 ，计算加权和，将每个值分别乘以权重 w_1 、 w_2 和 w_3 ，再加上一个偏置项 b ，然后将求和结果传递给一个 Sigmoid 函数，最终输出一个介于0和1之间的值。

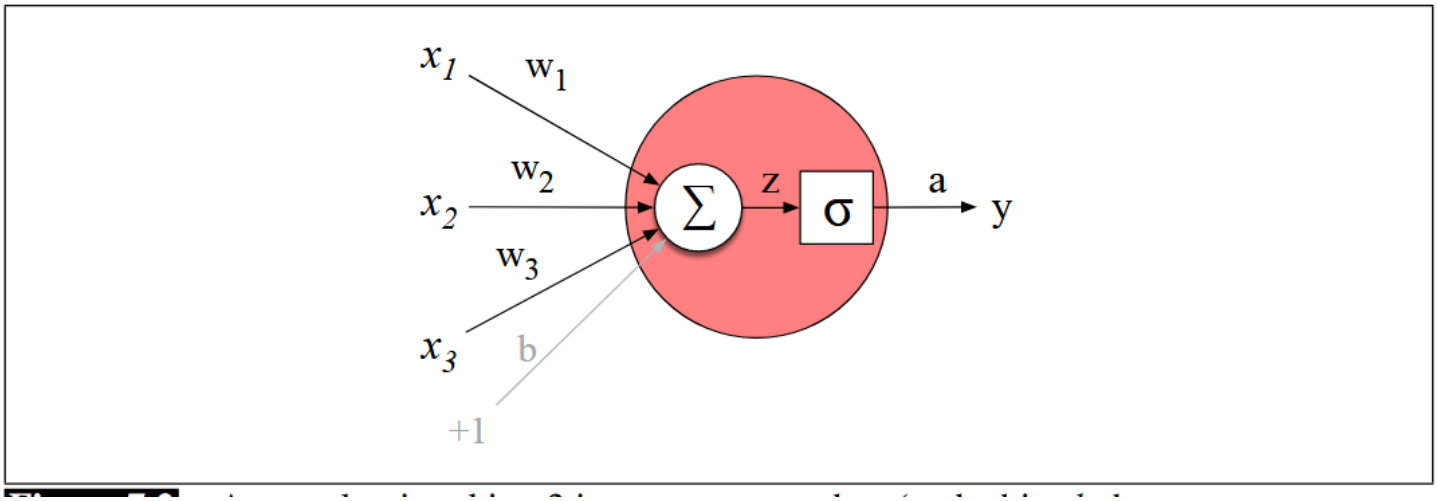

图 7.2 一个神经单元，采用 3 个输入 x1、x2 和 x3（以及偏置 b，我们将其表示为钳制在 +1 的输入的权重）并产生输出 y。我们包括一些方便的中间变量：求和的输出 z 和 sigmoid 的输出 a。在这种情况下，单元 y 的输出与 a 相同，但在更深的网络中，我们将保留 y 表示整个网络的最终输出，而将 a 作为单个节点的激活。

让我们通过一个例子来直观理解一下。假设我们有一个单元，其权重向量和偏置如下：

$$\begin{aligned}\mathbf{w} = &\quad [0.2, 0.3, 0.9] \\ b = &\quad 0.5\end{aligned}$$

如果该单元接收到以下输入向量，它将会输出什么呢?

$$\mathbf{x} = [0.5, 0.6, 0.1]$$

最终的输出 y 将如下所示:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}} = \frac{1}{1 + e^{-(.5*.2 + .6*.3 + .1*.9 + .5)}} = \frac{1}{1 + e^{-0.87}} = .70$$

在实际操作中，Sigmoid 通常不再作为激活函数使用。一个非常类似但表现几乎总是更好的函数是 tanh 函数，如图7.3a所示。tanh 是 Sigmoid 的一种变体，输出范围为 -1 到 +1：

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

最简单的激活函数，也可能是使用最广泛的，是整流线性单元 ReLU（如图7.3b所示）。当 z 为正时，ReLU 的输出就是 z ，否则输出为0:

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

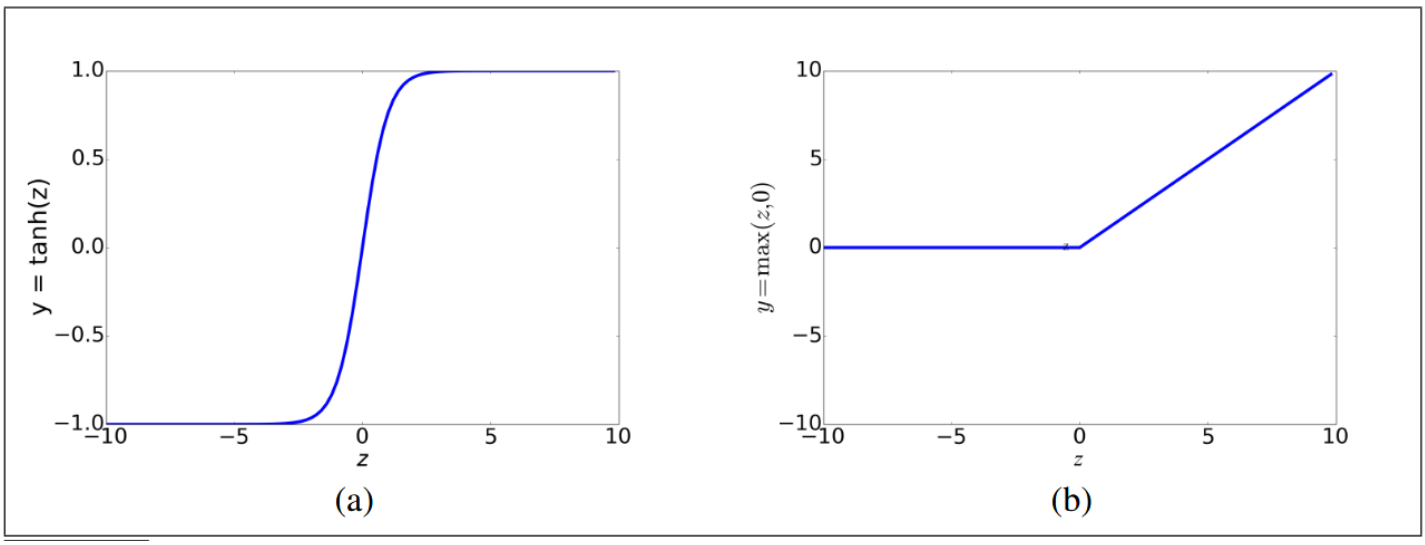

图 7.3 tanh 和 ReLU 激活函数。

这些激活函数具有不同的特性，使它们适用于不同的语言应用或网络架构。例如，tanh 函数具有平滑可微性，并且可以将异常值映射回均值附近。另一方面，ReLU 函数由于非常接近线性，具有一些良好的特性。在 Sigmoid 或 tanh 函数中，z 的值非常大时，y 的值趋于饱和，也就是非常接近1，并且其导数非常接近0。导数为零会导致学习过程中的问题，因为正如我们将在第7.5节中看到的那样，我们通过反向传播误差信号来训练网络，网络中的每一层都会乘以梯度（偏导数）；梯度接近0时，误差信号会越来越小，直到太小而无法用于训练，这个问题被称为梯度消失问题。而 ReLU 不会有这个问题，因为对于较大的 z 值，ReLU 的导数为1，而不是接近0。

7.2 XOR 问题

在神经网络发展的早期，人们意识到，神经网络的强大之处，如同激发它们的真实神经元一样，源于将这些单元组合成更大的网络。

Minsky 和 Papert（1969年）证明了多层网络的必要性，这是最巧妙的论证之一。他们证明了单个神经单元无法计算某些非常简单的输入函数。考虑计算两个输入的基本逻辑函数的任务，比如 AND、OR 和 XOR。下面是这些函数的真值表：

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

这个例子首先是在感知器（感知机）上展示的，感知器是一种非常简单的神经单元，具有二进制输出且没有非线性激活函数。感知器的输出 y 是 0 或 1，计算方式如下（使用与方程7.2中相同的权重 w、输入 x 和偏置 b ）：

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

构建一个可以计算二进制输入的逻辑 AND 和 OR 函数的感知器非常容易；图7.4显示了所需的权重。

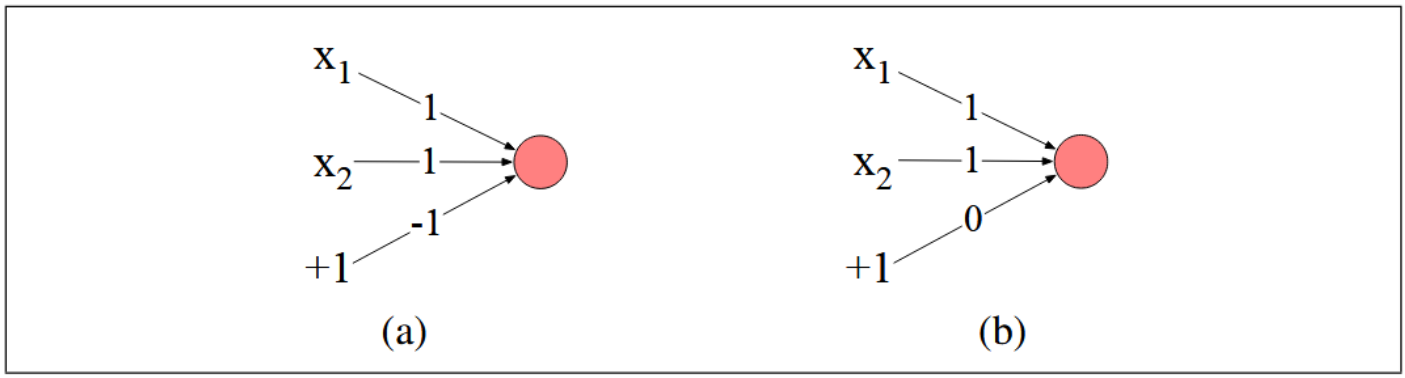

图 7.4 用于计算逻辑函数的感知器的权重 w 和偏差 b。输入显示为 x1 和 x2，偏差作为特殊节点，其值 +1 乘以偏差权重 b。 (a) 逻辑与，权重 w1 = 1 和 w2 = 1，偏置权重 b = －1。 (b) 逻辑或，权重 w1 = 1 和 w2 = 1，偏差权重 b = 0。这些权重/偏差只是实现这些功能的无数可能的权重和偏差集合中的一个。

然而，事实证明，无法构建一个感知器来计算逻辑 XOR！(值得花点时间自己尝试一下！)

这个重要结果背后的直觉在于，感知器是一个线性分类器。对于二维输入 x_1 和 x_2，感知器方程 $w_1x_1 + w_2x_2 + b = 0$ 是一条直线的方程。（我们可以通过将其转换为标准线性格式来看出：$x_2 = (-w_1/w_2)x_1 + (-b/w_2)$。 这条线在二维空间中充当决策边界，线的一侧的所有输入点被分配为输出0，另一侧的所有输入点被分配为输出1。如果我们有超过两个输入，决策边界将变为超平面，而不是直线，但概念是相同的，都是将空间分为两类。

图7.5显示了逻辑输入（00、01、10 和 11）的可能组合，以及为 AND 和 OR 分类器绘制的一条线。请注意，根本无法画出一条直线将 XOR 的正例（01 和 10）与负例（00 和 11）分开。我们称 XOR 不是一个线性可分的函数。当然，我们可以用曲线或其他函数来划分边界，但无法用一条直线来完成。

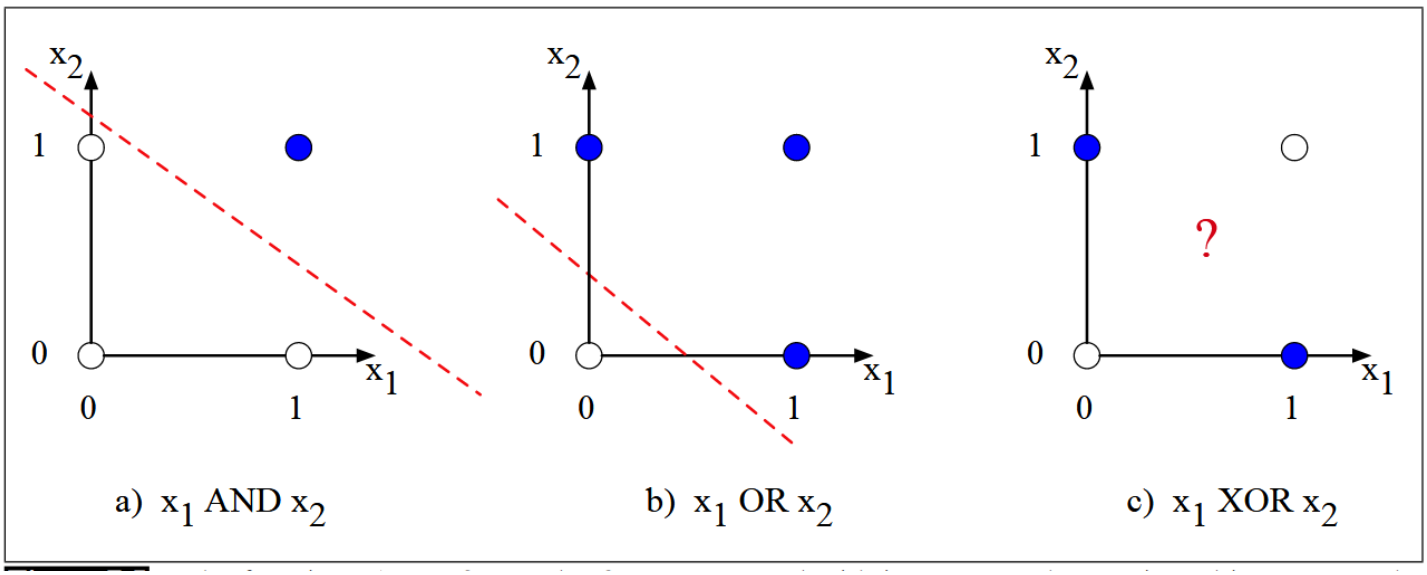

图 7.5 函数 AND、OR 和 XOR，用 x 轴上的输入 x1 和 y 轴上的输入 x2 表示。实心圆圈表示感知器输出 1，白色圆圈表示感知器输出 0。无法绘制一条线来正确区分 XOR 的两个类别。人物造型仿照 Russell 和 Norvig (2002)。

7.2.1 解决方案：神经网络

虽然单个感知器无法计算 XOR 函数，但通过多层感知器单元组成的网络可以计算 XOR 函数。不过，与其用简单的感知器网络来展示，不如按照 Goodfellow 等人（2016年）的方法，用两层基于 ReLU 的单元来计算 XOR 函数。图7.6展示了输入通过两层神经单元处理的示意图。中间层（称为 h ）有两个

单元，输出层（称为 y ）有一个单元。图中显示了一组权重和偏置，可以使该网络正确计算 XOR 函数。

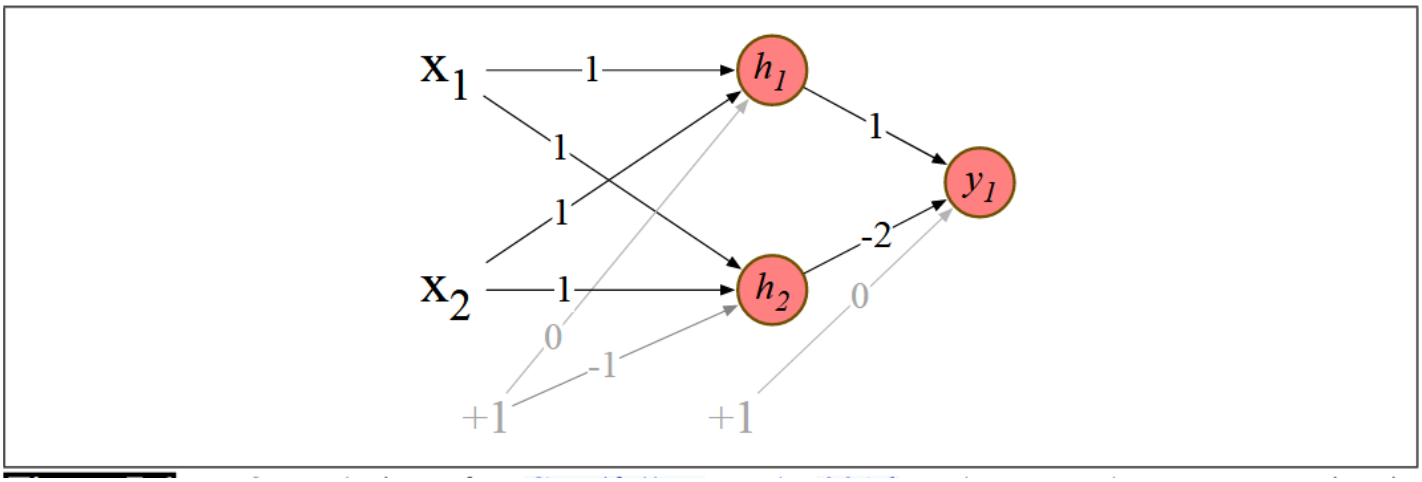

图 7.6 Goodfellow 等人提出的 XOR 解决方案。（2016）。共有三个ReLU单元，分为两层；我们将它们称为 h1、h2（h 表示“隐藏层”）和 y1。和之前一样，箭头上的数字代表每个单元的权重 w，我们将偏差 b 表示为固定为 +1 的单元的权重，偏差权重/单元为灰色。

让我们通过输入 $x = [0, 0]$ 来看看发生了什么。如果我们将每个输入值与相应的权重相乘、求和，然后加上偏置 b ，得到向量 $[0, -1]$ ，然后我们应用整流线性变换，得到 h 层的输出为 $[0, 0]$ 。接着，我们再次与权重相乘、求和并加上偏置（此处为0），结果为0。读者可以继续计算其余3个可能的输入组合，验证对于输入 $[0, 1]$ 和 $[1, 0]$ ，结果 y 值为1，而对于 $[0, 0]$ 和 $[1, 1]$ ，结果为0。

查看中间结果，即两个隐藏节点 h_1 和 h_2 的输出，也很有帮助。前面已经展示了，对于输入 $x = [0, 0]$ ， h 向量是 $[0, 0]$ 。图7.7b显示了所有四个输入对应的 h 层值。注意到对于输入点 $x = [0, 1]$ 和 $x = [1, 0]$ （XOR 输出为1的两个情况），它们的隐藏表示合并为一个点 $h = [1, 0]$ 。这种合并使得 XOR 的正例和负例很容易在线性上分离。换句话说，我们可以将网络的隐藏层视为对输入形成了一种表示。

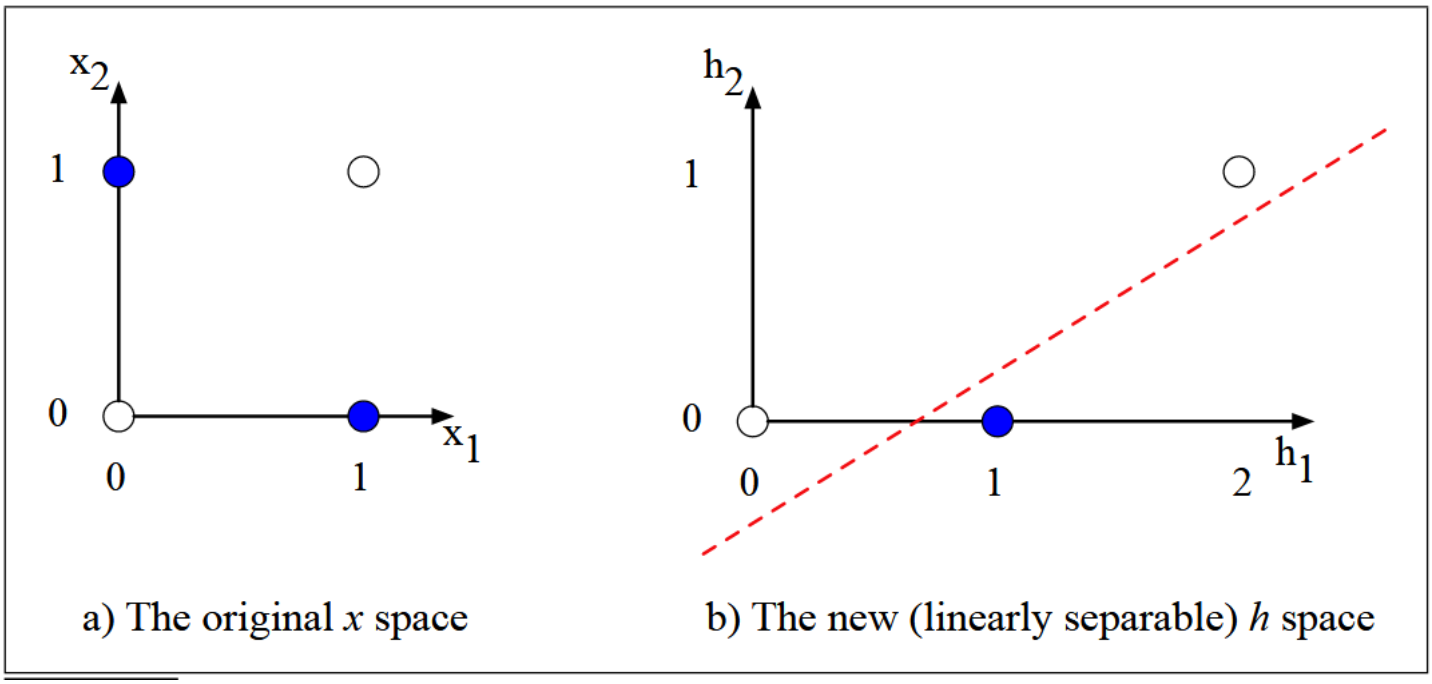

图 7.7 隐藏层形成输入的新表示。 (b) 显示了隐藏层 h 的表示与 (a) 中原始输入表示 x 的比较。请注意，输入点 [0, 1] 已与输入点 [1, 0] 折叠，从而可以线性分离 XOR 的正负情况。古德费洛等人之后。 （2016）。

在这个示例中，我们只是直接给出了图7.6中的权重。但在实际情况中，神经网络的权重是通过错误反向传播算法（将在第7.5节中介绍）自动学习的。这意味着隐藏层将学习形成有用的表示。神经网络可以自动学习输入的有用表示，这种直觉是神经网络的一个关键优势，我们将在后续章节中反复讨论这一点。

7.3 前馈神经网络

现在让我们更正式地介绍一种最简单的神经网络——**前馈网络**。前馈网络是一个多层网络，其中各层单元之间的连接没有循环；每层单元的输出传递给下一层的单元，而不会将输出传回到较低层。（在第8章中，我们将介绍具有循环结构的网络，称为**循环神经网络**。）

由于历史原因，多层网络，尤其是前馈网络，有时被称为多层感知器（multi-layer perceptrons，MLPs）；这在技术上是个误称，因为现代多层网络中的单元不是感知器（感知器是纯线性的，而现代网络由带有非线性函数如 Sigmoid 的单元组成），但这个名字一度流传开来。

简单的前馈网络有三种节点：输入单元、隐藏单元和输出单元。

图7.8展示了一个示意图。输入层 x 是一个简单的标量值向量，正如我们在图7.2中所见。

神经网络的核心是由**隐藏单元** h_i 组成的**隐藏层**，每个隐藏单元都是一个神经单元，如7.1节所述，接收加权和并应用非线性函数。在标准架构中，每层都是**全连接**的，意味着每层中的每个单元都接收来自前一层所有单元的输出，并且每对相邻层之间的每个单元都有连接。因此，每个隐藏单元都会对所有输入单元求和。

请记住，一个隐藏单元的参数是一个权重向量和一个偏置。我们通过将每个单元 i 的权重向量和偏置组合成一个权重矩阵 W 和整个隐藏层的偏置向量 b 来表示整个隐藏层的参数（见图7.8）。权重矩阵 W 中的每个元素 W_{ji} 表示从第 i 个输入单元 x_i 到第 j 个隐藏单元 h_j 的连接权重。

使用整个层的权重矩阵 W 的优点是，前馈网络的隐藏层计算可以通过简单的矩阵操作高效地完成。实际上，计算只有三个步骤：将权重矩阵与输入向量 x 相乘，加上偏置向量 b ，并应用激活函数 g （如上文定义的 Sigmoid、tanh 或 ReLU 激活函数）。

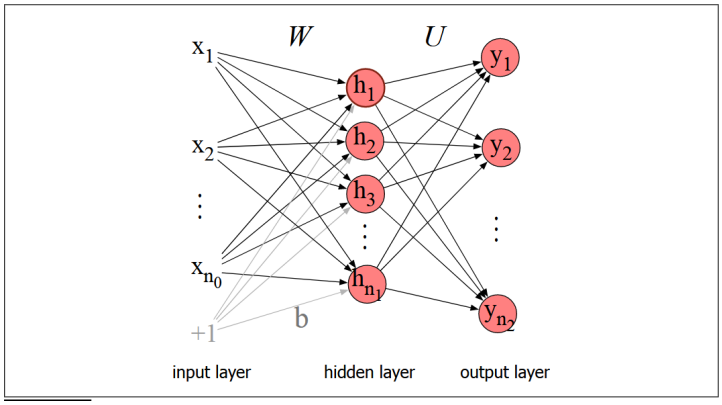

图 7.8 一个简单的2层前馈网络，有1个隐藏层、1个输出层和1个输入层（枚举层时通常不计算输入层）。

隐藏层的输出向量 h 因此如下（在这个例子中我们将使用 Sigmoid 函数 σ 作为激活函数）：

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

请注意，我们在这里对向量应用了 σ 函数，而在方程7.3中，它被应用于标量。因此，我们允许 $\sigma(\cdot)$（实际上任何激活函数 $g(\cdot)$）逐元素作用于向量，即 $g[z_1, z_2, z_3] = [g(z_1), g(z_2), g(z_3)]$。

让我们引入一些常量来表示这些向量和矩阵的维度。我们将输入层称为网络的第0层，设 n_0 表示输入的数量，因此 x 是一个维度为 n_0 的实数向量，更正式地说，$x \in \mathbb{R}^{n_0}$ ，是一个维度为 $[n_0, 1]$ 的列向量。我们称隐藏层为第1层，输出层为第2层。隐藏层的维度为 n_1 ，因此 $h \in \mathbb{R}^{n_1}$ ，同样 $b \in \mathbb{R}^{n_1}$ （因为每个隐藏单元可以有不同的偏置值）。权重矩阵 W 的维度为 $W \in \mathbb{R}^{n_1 \times n_0}$ ，即 $[n_1, n_0]$。

稍作思考，你会发现方程7.8中的矩阵乘法会计算每个 h_j 的值为 $\sigma\left(\sum_{i=1}^{n_0} W_{ji}x_i + b_j\right)$。

正如我们在7.2节中看到的，得到的值 h （不仅是隐藏的，还可以表示假设）形成了输入的表示。输出层的作用是获取这一新的表示 h 并计算最终输出。该输出可以是一个实数值，但在许多情况下，网络的目标是做出某种分类决策，因此我们将重点关注分类任务。

如果我们在做一个二元任务，例如情感分类，可能会有一个输出节点，其标量值 y 表示正向与负向情感的概率。如果我们在做多项分类任务，例如分配词性标签，可能会为每个潜在的词性标签分配一个输出节点，其输出值表示该词性的概率，并且所有输出节点的值之和必须为1。因此输出层是一个向量 y ，它给出输出节点的概率分布。

让我们看看这是如何发生的。和隐藏层一样，输出层有一个权重矩阵（我们称之为 U），但一些模型在输出层不包含偏置向量 b ，因此我们在此例中简化为省略偏置向量。权重矩阵 U 乘以其输入向量 h ，得到中间输出 z ：

$$\mathbf{z} = \mathbf{U}\mathbf{h}$$

有 n_2 个输出节点，因此 $z \in \mathbb{R}^{n_2}$ ，权重矩阵 U 的维度为 $U \in \mathbb{R}^{n_2 \times n_1}$ ，元素 U_{ij} 表示从隐藏层单元 j 到输出层单元 i 的连接权重。

然而， z 不能作为分类器的输出，因为它是一个实数向量，而分类所需的是一个概率向量。我们可以使用一种方便的函数来对实数向量进行归一化，即将其转换为编码概率分布的向量（所有数值介于0到1之间，且总和为1）：即我们在第5章中看到的 softmax 函数。更一般地，对于任何维度为 d 的向量 z ，softmax 定义如下:

$$\text{softmax}(\mathbf{z}_i) \;=\; \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{d}\exp(\mathbf{z}_j)} \; 1 \le i \le d$$

因此，给定一个向量 $\mathbf{z} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$ ，softmax 函数将其归一化为一个概率分布（此处为四舍五入后的结果）。

你可能还记得，我们在第5章中使用 softmax 从实数向量（通过求和权重与特征的乘积计算得出）中创建概率分布，这就是多项逻辑回归的版本。

这意味着我们可以将一个有单隐藏层的神经网络分类器看作是构建了一个向量 h ，它是输入的隐藏层表示，然后在网络在 h 中开发的特征上运行标准的多项逻辑回归。相比之下，在第5章中，特征主要是通过特征模板手动设计的。因此，神经网络类似于多项逻辑回归，但有以下不同：（a）有多层，因为深度神经网络相当于一层又一层的逻辑回归分类器；（b）这些中间层有许多可能的激活函数（tanh、ReLU、sigmoid），而不仅仅是 sigmoid（尽管我们为了方便仍用 σ 表示任何激活函数）；（c）网络的前几层自己引出特征表示，而不是通过特征模板形成特征。

以下是具有单隐藏层的前馈网络的最终方程，它接收一个输入向量 x ，输出一个概率分布 y ，并通过权重矩阵 W 和 U 以及偏置向量 b 参数化:

$$\begin{aligned}\mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \mathbf{y} &= \text{softmax}(\mathbf{z})\end{aligned}$$

为了记住所有变量的形状， $x \in \mathbb{R}^{n_0}$ ， $h \in \mathbb{R}^{n_1}$ ， $b \in \mathbb{R}^{n_1}$ ， $W \in \mathbb{R}^{n_1 \times n_0}$ ， $U \in \mathbb{R}^{n_2 \times n_1}$ ，输出向量 $y \in \mathbb{R}^{n_2}$ 。我们称这个网络为一个两层网络（在编号层时，我们传统上不计入输入层，但计入输出层）。因此，按照这种术语，逻辑回归是一个单层网络。

7.3.1 前馈网络的更多细节

现在让我们设置一些符号，以便更容易讨论深度超过2的深层网络。我们将使用方括号中的上标来表示层号，从输入层的0开始。因此， $\mathbf{w}^{[1]}$ 表示第一个隐藏层的权重矩阵，而 $\mathbf{b}^{[1]}$ 表示第一个隐藏层的偏置向量。n_j 表示第 j 层的单元数量。我们用 g(·) 表示激活函数，通常在中间层使用 ReLU 或 tanh，在输出层使用 softmax。我们用 $\mathbf{a}^{[i]}$ 表示第 i 层的输出，用 $\mathbf{z}^{[i]}$ 表示上一层输出、权重和偏置的组合，即 $\mathbf{W}^{[i]}\mathbf{a}^{[i-1]} + \mathbf{b}^{[i]}$ 。第0层是输入层，因此我们更一般地将输入称为 $\mathbf{a}^{[0]}$ 。

因此，我们可以将方程7.12中的2层网络重新表示如下:

$$
\begin{aligned}
\mathbf{z}^{[1]} &= \mathrm{W}^{[1]}\mathbf{a}^{[0]} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= g^{[1]}(\mathbf{z}^{[1]}) \\
\mathbf{Z}^{[2]} &= \mathrm{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \\
\mathbf{a}^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{\mathbf{y}} &= \mathbf{a}^{[2]}
\end{aligned}
$$

请注意，使用这种符号，每一层的计算方程都是相同的。给定输入向量 $\mathbf{a}^{[0]}$ ，n层前馈网络的前向计算步骤的算法如下:

$$
\begin{aligned}
&\mathbf{for} i \ \mathbf{in} \ 1, ..., \mathrm{n} \\
\mathbf{z}^{[i]} &= \mathbf{W}^{[i]}\mathbf{a}^{[i-1]} + \mathbf{b}^{[i]} \\
\mathbf{a}^{[i]} &= g^{[i]}(\mathbf{z}^{[i]}) \\
\hat{\mathbf{y}} &= \mathbf{a}^{[n]}
\end{aligned}
$$

通常为最后一组激活值（即最终 softmax 之前的激活值）起个名字是很有用的。因此，无论我们有多少层，我们一般将最后一个向量 $\mathbf{z}^{[n]}$ 中未归一化的值称为 logits（即最终 softmax 之前的分数向量，见(??)）。

非线性激活函数的必要性 我们在神经网络的每一层中使用非线性激活函数的原因之一是，如果不使用非线性激活函数，得到的网络与单层网络完全等价。让我们看看为什么这是正确的。想象一下，网络的前两层是纯线性层:

$$
\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \\
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{z}^{[1]} + \mathbf{b}^{[2]}
\end{aligned}
$$

我们可以将网络计算的函数重写如下:

$$
\begin{aligned}
\mathbf{z}^{[2]} &= \mathrm{W}^{[2]}\mathbf{z}^{[1]} + \mathbf{b}^{[2]} \\
&= \mathbf{W}^{[2]}(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]} \\
&= \mathbf{W}^{[2]}\mathbf{W}^{[1]}\mathbf{x} + \mathbf{W}^{[2]}\mathbf{b}^{[1]} + \mathbf{b}^{[2]} \\
&= \mathrm{W}'\mathbf{x} + \mathbf{b}'
\end{aligned}
$$

这种方式可以推广到任意数量的层。因此，如果没有非线性激活函数，多层网络只是单层网络具有一组不同的权重的符号变体，并且我们会失去多层网络的所有表征能力。

替换偏置单元 在描述网络时，我们经常使用稍微简化的符号，它表示完全相同的函数，而不涉及显式的偏置节点 b。相反，我们在每一层中添加一个虚拟节点 $\mathbf{a}_0$ ，其值总是1。因此，第0层（输入层）将有一个虚拟节点 $\mathbf{a}_0^{[0]} = 1$ ，第1层将有 $\mathbf{a}_0^{[1]} = 1$ ，依此类推。这个虚拟节点仍然有一个关联的权重，该权重表示偏置值 b。例如，我们不再使用类似于以下的方程:

$$
\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})
$$

而是使用下面的方程：

$$
\mathbf{h} = \sigma(\mathbf{W}\mathbf{x})
$$

但现在我们的向量 x 不再有 $\mathbf{n}_0$ 个值： $\mathbf{x} = \mathbf{x}_1, \ldots, \mathbf{x}_{n_0}$ ，而是有 $\mathbf{n}_0 + 1$ 个值，并新增了一个第0个虚拟值 $\mathbf{x}_0 = 1$: $\mathbf{x} = \mathbf{x}_0, \ldots, \mathbf{x}_{n_0}$ 。并且也不是如下计算每个 h_j :

$$\mathbf{h}_j = \sigma\left(\sum_{i=1}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i + \mathbf{b}_j\right),$$

而是计算：

$$\mathbf{h}_j = \sigma\left(\sum_{i=0}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i\right),$$

其中 $\mathbf{W}_{j0}$ 代替了 b_j 。图7.9对该情况做了可视化。

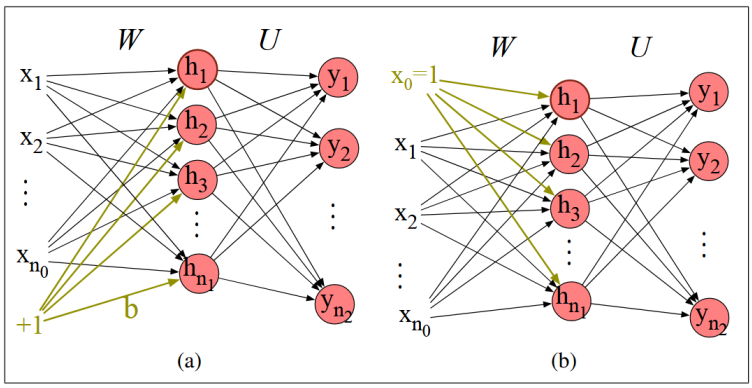

图 7.9 将偏置节点（a）替换为 x0 (b)。

在第7.5节讲解学习算法时，我们会继续显示偏置为 b，但在书的其余部分中，我们将转向这种没有显式偏置项的简化符号。

7.4 前馈网络在NLP中的应用：分类

让我们看看如何将前馈网络应用于NLP任务！在本节中，我们将关注分类任务，比如情感分析；在下一节中，我们会介绍神经语言模型。

我们从一个简单的两层情感分类器开始。你可以想象将第5章中的逻辑回归分类器（对应于一个一层网络）加上一层隐藏层。输入元素 x_i 可以是标量特征，比如图中的那些，例如， x_1 = 词汇出现在文档中的次数， $x2$ = 积极词典中的词出现在文档中的次数， $x3$ = 如果“no”出现在文档中则为1，等等。而输出层 $\hat{\mathbf{y}}$ 可以有两个节点（分别对应积极和消极），或者3个节点（积极、消极、中性），在这种情况下， $\hat{\mathbf{y}}_1$ 是对积极情感的概率估计， $\hat{\mathbf{y}}_2$ 是对消极情感的概率估计， $\hat{\mathbf{y}}_3$ 是对中性情感的概率估计。最终的方程就是我们在前面看到的两层网络的形式（我们仍将使用σ来表示任何非线性函数，不管是Sigmoid、ReLU还是其他）。

$$\begin{aligned}
\mathbf{x} &= [\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_N] \quad (\text{each } \mathbf{x}_i \text{ is a hand-designed feature}) \\
\mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \\
\mathbf{z} &= \mathbf{U}\mathbf{h} \\
\hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})
\end{aligned}$$

图7.10展示了该架构的草图。正如我们之前提到的，为我们的逻辑回归分类器添加隐藏层允许网络表示特征之间的非线性交互。仅此一点可能就能给我们带来一个更好的情感分类器。

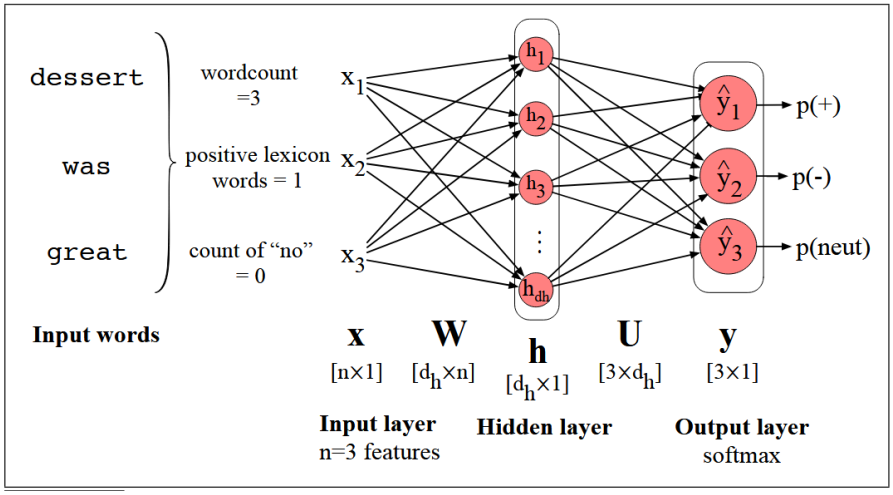

图 7.10 使用输入文本的传统手工构建特征进行前馈网络情感分析。

然而，大多数神经网络在NLP中的应用是有所不同的。与其使用人为设计的特征作为分类器的输入，我们利用深度学习从数据中学习特征的能力，通过将单词表示为嵌入，如我们在第6章中看到的word2vec或GloVe嵌入。有多种方式可以表示分类任务的输入。一个简单的基线方法是对输入中所有单词的嵌入应用某种**池化(pooling)**函数。例如，对于一个包含 n 个输入单词/标记的文本 $w_1, ..., w_n$ ，我们可以将 n 个嵌入 $e(w_1), ..., e(w_n)$ （每个嵌入维度为 d ）转换为一个同样维度为 d 的单一嵌入，通过将这些嵌入相加，或者通过计算它们的平均值（相加后再除以 n ）：

$$\mathbf{x}_{mean} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{e}(w_i)$$

还有许多其他的选项，比如取逐元素最大值。 n 个向量的逐元素最大值是一个新的向量，其第 k 个元素是所有 n 个向量中第 k 个元素的最大值。以下是这个假设使用均值池化(mean pooling)的分类器的方程;图7.11展示了该架构的草图:

$$\begin{aligned}\mathbf{x} &= \text{mean}(\mathbf{e}(w_1), \mathbf{e}(w_2), \ldots, \mathbf{e}(w_n))\\ \mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})\\ \mathbf{z} &= \mathbf{U}\mathbf{h}\\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})\end{aligned}$$

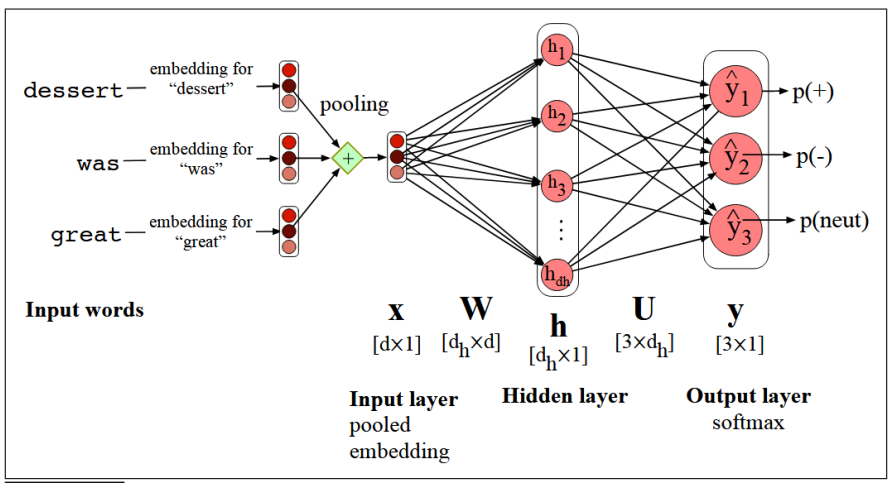

图 7.11 使用输入单词的池化嵌入进行前馈网络情感分析。

尽管方程7.21展示了如何对单个示例 x 进行分类，但实际上我们希望能够高效地对整个测试集的 m 个示例进行分类。我们通过向量化的方式来完成这个任务，正如我们在逻辑回归中看到的那样；与其使用for循环遍历每个示例，我们将使用矩阵乘法一次性完成整个测试集的计算。首先，我们将每个输入 x 的输入特征向量打包成一个输入矩阵 X ，每行 i 是一个行向量，由输入示例 $x^{(i)}$ 的池化嵌入组成（即向量 $x^{(i)}$ ）。如果池化后的输入嵌入的维度为 d ，则 X 的形状为 $[m \times d]$ 。

接下来我们需要稍微修改方程7.21。 X 的形状是 $[m \times d]$ ，W的形状是 $[d_h \times d]$ ，所以我们必须重新排列 X 和 W 的乘法顺序，并对 W 进行转置，使它们能够正确相乘，得到一个形状为 $[m \times d_h]$ 的矩阵 H 。方程7.21中的偏置向量 b 的形状为 $[1 \times d_h]$ ，现在必须复制为一个形状为 $[m \times d_h]$ 的矩阵。我们也需要类似地重新排列下一步，并对 U 进行转置。最后，我们的输出矩阵 $\hat{\mathbf{Y}}$ 的形状将是 $[m \times 3]$ （或更一般地为 $[m \times d_o]$ ，其中 d_o 是输出类别的数量），每行 i 是输出向量 $\hat{\mathbf{y}}^{(i)}$ 。对于整个测试集计算输出类别分布：

$$\begin{aligned}\mathcal{H} &= \sigma(\mathbf{X}\mathbf{W}^{\mathsf{T}} + \mathbf{b}) \\ \mathbf{Z} &= \mathbf{H}\mathbf{U}^{\mathsf{T}} \\ \hat{\mathbf{Y}} &= \ \mathrm{softmax}(\mathbf{Z})\end{aligned}$$

使用word2vec或GloVe嵌入作为输入表示的想法——更广泛地说，依赖于另一个算法已经为我们的输入单词学习了嵌入表示的想法——被称为**预训练（pretraining）**。使用预训练的嵌入表示（无论是像word2vec这样简单的静态词嵌入，还是我们将在第11章介绍的更强大的上下文嵌入）是深度学习的核心思想之一。（不过，也可以在NLP任务中训练词嵌入；我们将在第7.7节的神经语言模型任务中讨论如何实现这一点。）

7.5 训练神经网络

前馈神经网络是一种监督学习的实例，其中我们知道每个观察值 x 对应的正确输出 y 。系统预测的 $\hat{\mathbf{y}}$ 是系统对真实 y 的估计。训练过程的目标是学习每一层 i 的参数 $\mathbf{w}^{[i]}$ 和 $\mathbf{b}^{[i]}$ ，使得每个训练观察值的 $\hat{\mathbf{y}}$ 尽可能接近真实的 y 。

通常，我们通过第5章中介绍的逻辑回归方法来实现这一目标，因此读者在继续之前应熟悉该章节的内容。

首先，我们需要一个**损失函数(loss function)**来衡量系统输出与标准输出之间的距离，通常使用逻辑回归中使用的**交叉熵损失函数(cross-entropy loss)**。

其次，为了找到能最小化该损失函数的参数，我们将使用第5章介绍的梯度下降优化算法。

第三，梯度下降需要知道损失函数的**梯度(gradient)**，即包含损失函数对每个参数的偏导数的向量。在逻辑回归中，对于每个观察值，我们可以直接计算损失函数对单个 w 或 b 的导数。但对于有数百万参数且有多个层次的神经网络，很难看到如何在损失附加到后面的某个层时，计算第1层中某个权重的偏导数。我们如何将损失分摊到所有这些中间层？答案是称为误差**反向传播(error backpropagation)**或**反向微分(backward differentiation)**的算法。

7.5.1 损失函数

神经网络中使用的交叉熵损失与我们在逻辑回归中看到的损失是相同的。如果神经网络用作二分类器，并且在最后一层使用Sigmoid函数，则损失函数与我们在方程??中看到的逻辑回归损失相同。

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

如果我们使用网络对3个或更多类别进行分类，那么损失函数与我们在第5章中看到的多项式回归损失完全相同。为了方便起见，我们在这里简要总结一下解释。首先，当我们有超过2个类别时，我们需要将 y 和 $\hat{y}$ 表示为向量。假设我们进行的是**硬分类(hard classification)**，其中只有一个类别是正确的。那么，真实标签 y 是一个有 K 个元素的向量，每个元素对应一个类别，若正确的类别为 c ，则 y_c = 1，其余元素为0。回想一下，这种一个值为1其余为0的向量称为独热向量。我们的分类器将生成一个估计向量 $\hat{y}$ ，该向量有 K 个元素，每个元素 $\hat{y}_k$ 表示估计的概率 $p(y_k = 1|x)$。

单个示例 x 的损失函数是 K 个输出类别的对数之和的负值，每个类别的概率 y_k 作为权重:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k$$

我们可以进一步简化这个方程；首先使用函数 $\mathbb{I}\{\}$ 来重写方程，该函数在括号中的条件为真时返回1，否则返回0。这使得方程7.24中的和的各项将为0，除了对应于正确类别的项， $y_k = 1$:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) \;=\; -\sum_{k=1}^{K} \mathbb{I}\{\mathbf{y}_k = 1\} \log \hat{\mathbf{y}}_k$$

换句话说，交叉熵损失只是与正确类别对应的输出概率的负对数，因此我们也称其为**负对数似然损失(negative log likelihood loss)**:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) \;=\; -\log \hat{\mathbf{y}}_c \quad (\text{where } c \text{ is the correct class})$$

将方程7.9中的Softmax公式代入，并且 K 为类别的数量:

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) \;=\; -\log \frac{\exp(\mathbf{z}_c)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)} \quad (\text{where } c \text{ is the correct class})$$

7.5.2 计算梯度

我们如何计算该损失函数的梯度？计算梯度需要损失函数对每个参数的偏导数。对于一个具有一个权重层和Sigmoid输出（即逻辑回归）的网络，我们可以简单地使用我们在方程7.27中使用的逻辑回归的导数（在第??节中推导）:

$$\begin{aligned}\frac{\partial L_{CE}(\hat{\mathbf{y}}, \mathbf{y})}{\partial w_j} &= (\hat{y} - y)\mathbf{x}_j \\ &= (\sigma(\mathbf{w} \cdot \mathbf{x} + b) - y)\mathbf{x}_j\end{aligned}$$

或者，对于一个具有一个权重层和Softmax输出（即多项式逻辑回归）的网络，我们可以使用Softmax损失导数，针对特定权重w_k和输入x_i:

$$
\begin{aligned}
\frac{\partial L_{\mathrm{CE}}(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{w}_{k,i}} &= -(\mathbf{y}_k - \hat{\mathbf{y}}_k)\mathbf{x}_i \\
&= -(\mathbf{y}_k - p(\mathbf{y}_k = 1|\mathbf{x}))\mathbf{x}_i \\
&= -\left(\mathbf{y}_k - \frac{\exp\left(\mathbf{w_k} \cdot \mathbf{x} + b_k\right)}{\sum_{j=1}^{K} \exp\left(\mathbf{w_j} \cdot \mathbf{x} + b_j\right)}\right)\mathbf{x}_i
\end{aligned}
$$

但这些导数只给出了最后一层权重的正确更新！对于深层网络，计算每个权重的梯度要复杂得多，因为我们需要计算前面的网络层中的权重参数的导数，但损失只在网络的最后一层计算。

解决计算这个梯度问题的方法是称为误差反向传播或反向传播（backprop）的算法（Rumelhart等人，1986年）。虽然反向传播是专门为神经网络发明的，但它实际上与称为**反向微分（backward differentiation）**的更一般的过程相同，反向微分依赖于**计算图（computation graphs）**的概念。让我们在下一小节中看看它是如何工作的。

7.5.3 计算图

计算图是对计算数学表达式过程的表示，其中计算被分解为独立的操作，每个操作作为图中的一个节点进行建模。

考虑计算函数 $L(a, b, c) = c(a + 2b)$。如果我们将每个组成部分的加法和乘法操作显式表示出来，并为中间输出添加名称（d 和 e），则生成的计算序列如下:

$$
\begin{aligned}
d &= 2 * b \\
e &= a + d \\
L &= c * e
\end{aligned}
$$

我们现在可以将其表示为一个图，每个操作为一个节点，带有有向边显示每个操作的输出作为下一个操作的输入，如图7.12所示。计算图的最简单用途是用给定的输入来计算函数的值。在图中，我们假设输入 $a = 3$，$b = 1$，$c = -2$，并显示了前向传播的结果 $L(3, 1, -2) = -10$。在计算图的前向传播中，我们从左到右应用每个操作，将每次计算的输出作为输入传递给下一个节点。

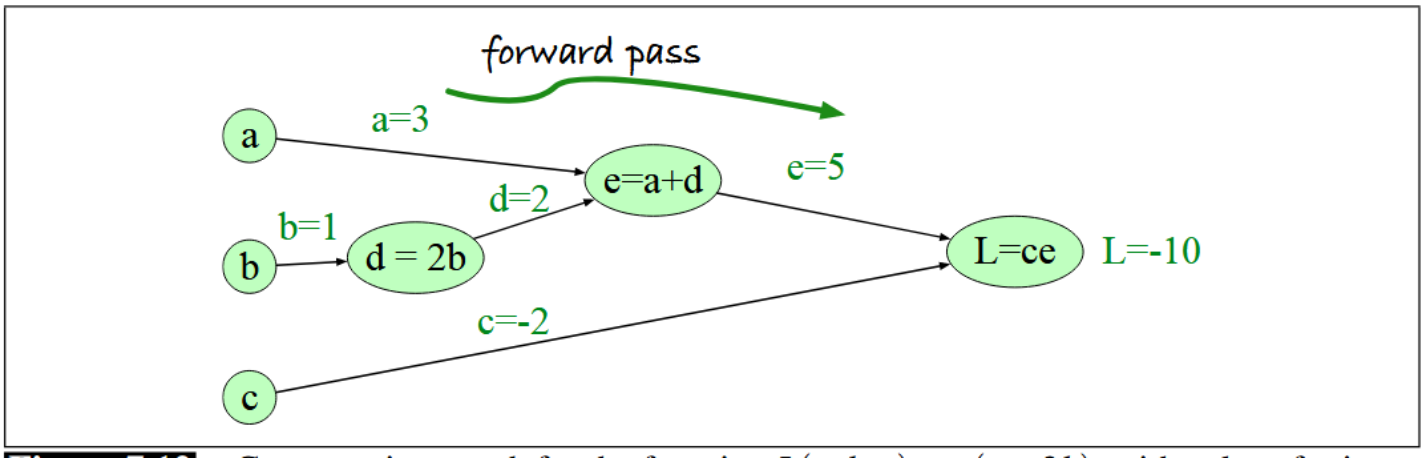

图 7.12 函数 L(a, b, c) = c(a+2b) 的计算图，输入节点的值为 a = 3, b = 1, c = −2，显示了 L 的前向传递计算。

7.5.4 计算图上的反向微分

计算图的重要性在于反向传播，它用于计算我们在权重更新中所需的导数。在这个例子中，我们的目标是计算输出函数L对每个输入变量的导数，即 $\partial L/\partial a$ ， $\partial L/\partial b$ 和 $\partial L/\partial c$ 。导数 $\partial L/\partial a$ 告诉我们a的微小变化如何影响L。

反向微分利用了微积分中的链式法则，因此让我们回顾一下。假设我们正在计算复合函数 $f(x) = u(v(x))$ 的导数， $f(x)$ 的导数是 $u(x)$ 对 $v(x)$ 的导数乘以 $v(x)$ 对 x 的导数:

$$\frac{df}{dx} = \frac{du}{d\nu} \cdot \frac{d\nu}{dx}$$

链式法则可以扩展到超过两个函数。如果计算复合函数 $f(x) = u(v(w(x)))$ 的导数，那么 $f(x)$ 的导数如下所示:

$$\frac{df}{dx} = \frac{du}{d\nu} \cdot \frac{d\nu}{dw} \cdot \frac{dw}{dx}$$

反向微分的直觉是将梯度从最终节点传回图中的所有节点。图7.13展示了在节点e上进行的部分反向计算。每个节点接收从其右侧父节点传入的上游梯度，并对每个输入计算局部梯度（输出对输入的导数），并利用链式法则将这两者相乘，计算出向前一个节点传递的下游梯度。

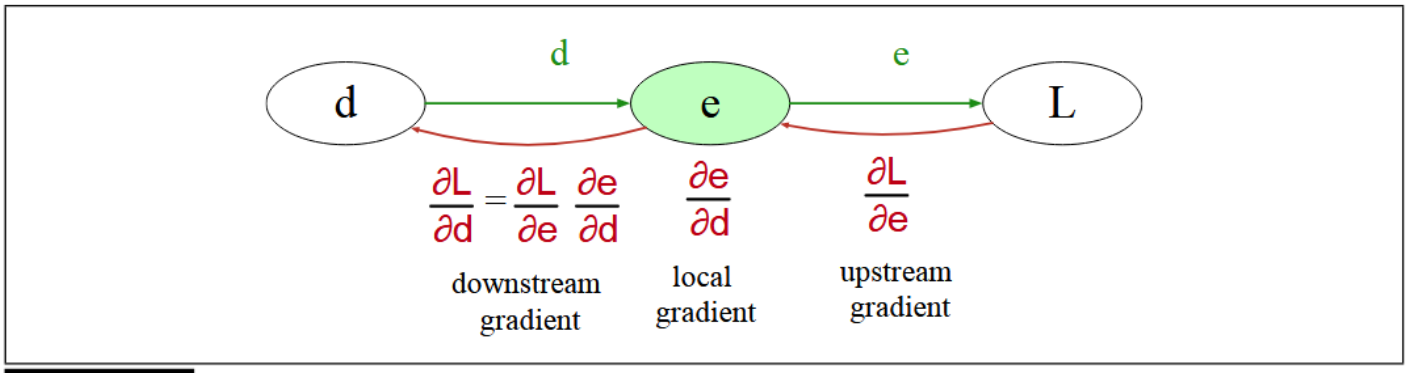

图 7.13 每个节点（如此处的 e）采用上游梯度，将其乘以本地梯度（其输出相对于其输入的梯度），并使用链式法则计算要传递到先前节点的下游梯度。如果一个节点有多个输入，则它可能有多个局部梯度。

现在让我们计算所需的三个导数。由于在计算图中 $L = ce$ ，我们可以直接计算导数 $\partial L/\partial c$:

$$\frac{\partial L}{\partial c} = e$$

对于其他两个导数，我们需要使用链式法则:

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

因此，方程7.32和7.31需要五个中间导数： $\partial L/\partial e$ ， $\partial L/\partial c$ ， $\partial e/\partial a$ ， $\partial e/\partial d$ 和 $\partial d/\partial b$ ，具体如下（利用求和的导数等于各项导数之和）:

$$L = ce: \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$
$$e = a + d: \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$
$$d = 2b: \quad \frac{\partial d}{\partial b} = 2$$

在反向传播过程中，我们沿着图中的每条边从右到左计算这些偏导数，使用链式法则正如我们之前所做的那样。因此，我们从节点L开始计算下游梯度，即 $\partial L/\partial e$ 和 $\partial L/\partial c$。对于节点e，我们将上游梯度 $\partial L/\partial e$ 乘以局部梯度（输出对输入的导数） $\partial e/\partial d$，得到传回节点 d 的输出 $\partial L/\partial d$。依此类推，直到我们将图中的所有输入变量标注完毕。前向传播已经方便地计算出了我们需要的中间变量的值（如d和e），这些可用于计算导数。图7.14展示了反向传播。

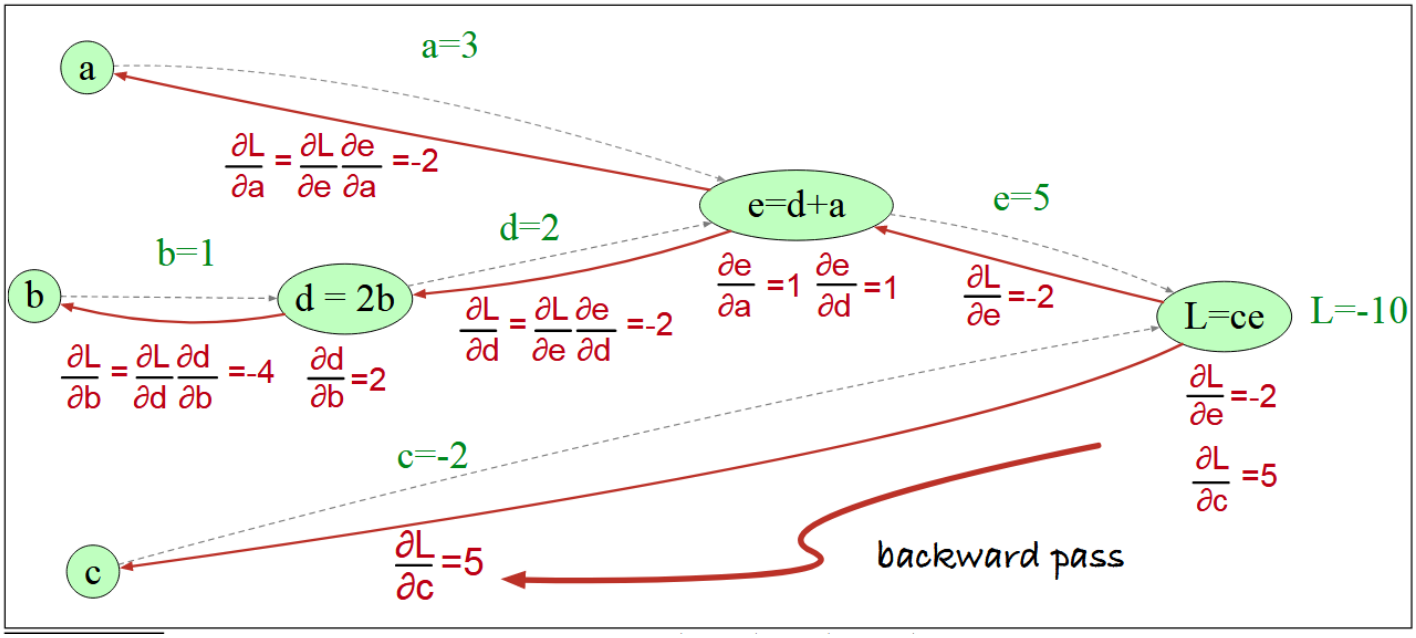

图 7.14 函数 L(a, b, c) = c(a + 2b) 的计算图，显示 ∂ L ∂a、∂L ∂ b 和 ∂ L ∂c 的后向传递计算。

神经网络的反向微分

当然，实际神经网络的计算图要复杂得多。图7.15展示了一个两层神经网络的示例计算图，假设 $n0 = 2$， $n1 = 2$， $n2 = 1$，并假设是二分类任务，因此为了简化使用了Sigmoid输出单元。该计算图计算的函数如下所示:

$$\begin{aligned}
\mathbf{z}^{[1]} &= -\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= \mathrm{ReLU}(\mathbf{z}^{[1]}) \\
z^{[2]} &= \mathrm{W}^{[2]}\mathbf{a}^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}$$

对于反向传播，我们还需要计算损失L。方程7.23中二分类Sigmoid输出的损失函数如下所示:

$$L_{CE}(\hat{y}, y) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

我们的输出 $\hat{y} = a^{[2]}$，因此我们可以将其重新表述如下:

$$L_{CE}(a^{[2]}, y) = -\left[y \log a^{[2]} + (1-y)\log(1-a^{[2]})\right]$$

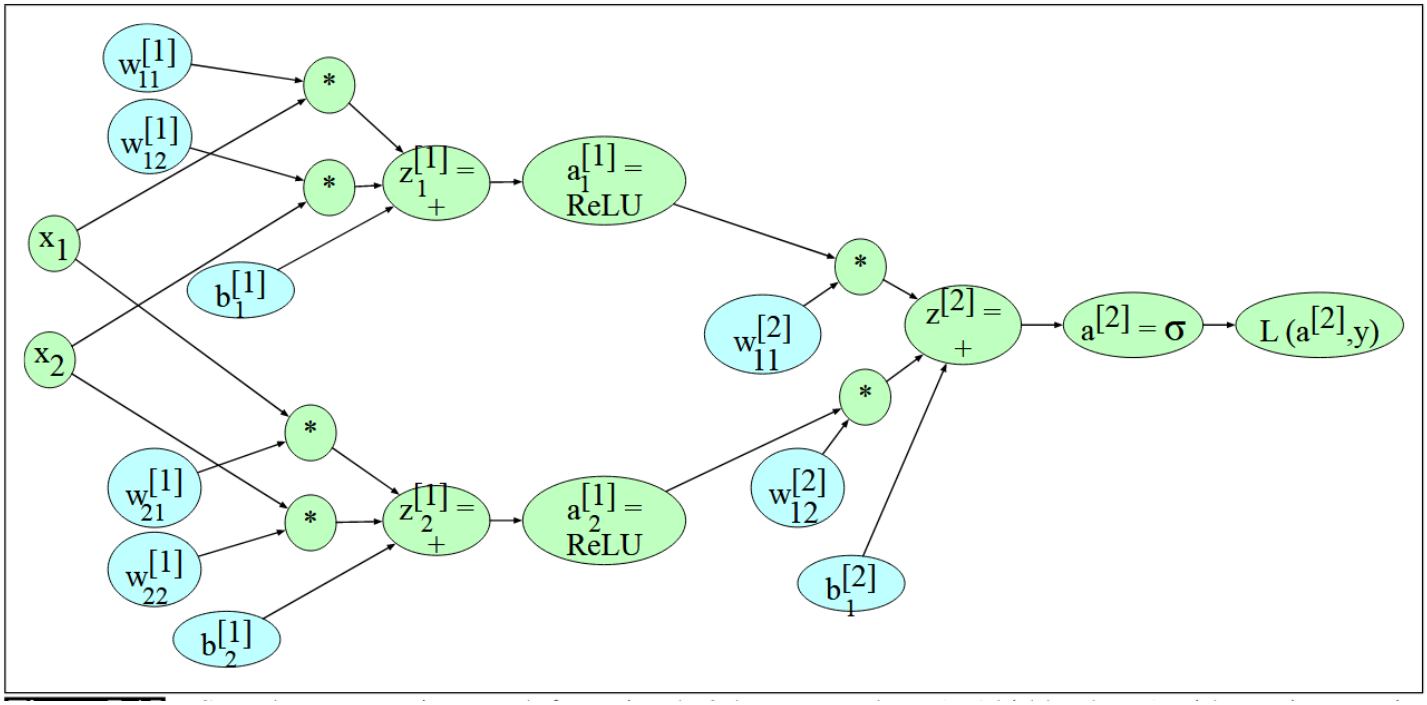

图 7.15 具有两个输入单元和 2 个隐藏单元的简单 2 层神经网络（= 1 个隐藏层）的示例计算图。我们对符号进行了一些调整，以避免节点中出现长方程，只需提及正在计算的函数以及生成的变量名称。因此，节点 w[1] 11 右侧的 * 表示 w[1] 11 要乘以 x1，而节点 z[1] = + 表示 z[1] 的值是通过对馈入其中的三个节点（两个乘积和偏差项 b[1] i ）。

需要更新的权重（我们需要知道损失函数的偏导数的权重）用青色表示。为了进行反向传播，我们需要知道图中所有函数的导数。我们已经在第??节中看到Sigmoid σ 的导数：

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

我们还需要每个其他激活函数的导数。Tanh的导数如下所示。

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

ReLU的导数如下所示：

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

我们将从计算开始，计算损失函数 L 对 z 的导数，即 $\partial L/\partial z$ （将其余部分留给读者作为练习）。根据链式法则:

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

因此让我们首先计算 $\dfrac{\partial L}{\partial a^{[2]}}$ ，其导数来自方程7.35:

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

接下来，根据下面Sigmoid的导数：

$$
\frac{\partial a^{[2]}}{\partial z} = a^{[2]}(1-a^{[2]})
$$

最后，我们可以使用链式法则：

$$
\begin{aligned}
\frac{\partial L}{\partial z} &= -\frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z} \\
&= -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}}\right) a^{[2]}(1-a^{[2]}) \\
&= -a^{[2]} - y
\end{aligned}
$$

继续反向计算梯度（接下来通过 $b_1^{[2]}$ 上的梯度以及两个乘积节点，依次传递回所有青色节点），这部分留给读者作为练习。

7.5.5 关于学习的更多细节

神经网络中的优化是一个非凸优化问题，比逻辑回归更复杂，出于这个原因以及其他原因，有许多关于成功学习的最佳实践。

对于逻辑回归，我们可以使用所有权重和偏置为0来初始化梯度下降。然而，在神经网络中，我们需要用小的随机数来初始化权重。同时，将输入值归一化为零均值和单位方差也是有帮助的。

各种形式的正则化被用来防止过拟合。其中最重要的一种是**dropout**（随机失活）：在训练期间随机丢弃网络中的一些单元及其连接（Hinton等人，2012年，Srivastava等人，2014年）。在训练的每次迭代（每次更新参数时，比如使用小批量梯度下降时的每个小批量），我们都会选择一个概率p，并对每个单元以概率p将其输出替换为零（并重新归一化该层其余部分的输出）。

超参数的调整也非常重要。神经网络的参数是权重 W 和偏置 b ；这些通过梯度下降进行学习。超参数是由算法设计者选择的；最佳值是在验证集上调试的，而不是通过训练集上的梯度下降学习得到的。超参数包括学习率 η 、小批量的大小、模型架构（层数、每层隐藏节点的数量、激活函数的选择）、如何正则化等。梯度下降本身也有许多结构变体，如Adam算法（Kingma和Ba，2015年）。

最后，大多数现代神经网络都是使用计算图形式构建的，这使得在基于向量的GPU（图形处理单元）上进行梯度计算和并行化变得简单而自然。PyTorch（Paszke等人，2017年）和TensorFlow（Abadi等人，2015年）是其中最流行的两个框架。感兴趣的读者应参考神经网络的教科书获取更多详细信息，本章结尾提供了一些推荐书目。

7.6 前馈神经语言建模

作为前馈网络的第二个应用，让我们来看看语言建模：从前面的单词预测接下来的单词。基于我们将在第9章看到的transformer架构的神经语言建模，是现代NLP的基础算法。在本节和下一节中，我们将介绍一种更简单的前馈网络神经语言模型版本，这是由Bengio等人（2003年）首次提出的算法。前馈语言模型引入了神经语言建模中的许多重要概念，而我们将在第8章和第9章描述更强大的模型时再次讨论这些概念。

神经语言模型相比于第3章中的n元语言模型有许多优势。与n元模型相比，神经语言模型可以处理更长的历史上下文，能够更好地在相似的单词上下文中进行泛化，并且在单词预测上更准确。然而，神经网络语言模型更加复杂，训练速度较慢，能耗较高，并且不如n元模型易于解释，因此对于一些较小的任务，n元语言模型仍然是合适的工具。

前馈神经语言模型（LM）是一个前馈网络，它在时间t接受前几个单词（wt－1, wt－2, 等等）的表示作为输入，并输出一个可能的下一个单词的概率分布。因此，像n元语言模型一样，前馈神经语言模型通过基于前N－1个单词的近似来估计给定整个先前上下文的单词概率P(wt |w1:t－1):

$$P(w_t|w_1,\ldots,w_{t-1}) \approx P(w_t|w_{t-N+1},\ldots,w_{t-1})$$

在下面的示例中，我们将使用一个4元模型，因此我们会展示一个神经网络来估计概率 $P(w_t|w_1,\ldots,w_{t-1})$。

神经语言模型通过它们的嵌入表示这些前面的单词上下文，而不是像n元语言模型那样仅仅使用单词的标识。使用嵌入允许神经语言模型在未见过的数据上进行更好的泛化。例如，假设我们在训练中见过这个句子：

I have to make sure that the cat gets fed.

但是从未见过“gets fed”出现在“dog”之后。我们的测试集中有“I forgot to make sure that the dog gets”的前缀。接下来的单词是什么？n元语言模型会在“that the cat gets”之后预测“fed”，但不会在“that the dog gets”之后预测。然而，神经语言模型，知道“cat”和“dog”有相似的嵌入，就能从“cat”的上下文中泛化，并且即使看到“dog”也会给“fed”赋予足够高的概率。

7.6.1 神经语言模型中的前向推理

让我们走一遍神经语言模型中的**前向推理**或**解码过程**。前向推理的任务是，给定一个输入，对网络进行前向传递以生成一个可能输出的概率分布，在这种情况下是预测下一个单词。

首先，我们将之前的 N 个单词每个表示为一个长度为 $|V|$ 的独热向量，即词汇表中每个单词占据一个维度。独热向量是一个向量，其中一个元素等于1（对应于该单词在词汇表中的索引），而其他所有元素都为零。因此，假设单词“toothpaste”是 V_5（即词汇表中的索引为5），则 $x_5 = 1$，且对于所有 $i \neq 5$，$xi = 0$，正如下图所示:

$$\begin{array}{cccccccccccc} [0 & 0 & 0 & 0 & 1 & 0 & 0 & \ldots & 0 & 0 & 0 & 0] \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & \ldots & \ldots & \ldots & \ldots & |V| \end{array}$$

前馈神经语言模型（如图7.17所示）使用一个可以查看过去N个单词的移动窗口。我们将N设为3，因此单词wt－1、wt－2和wt－3分别表示为独热向量。然后，我们将这些独热向量与嵌入矩阵E相乘。嵌入权重矩阵E为每个单词都有一列，每列为一个d维的列向量，因此其维度为 $d \times |V|$ 。与仅有一个非零元素 $x_i = 1$ 的独热向量相乘时，会选择出对应单词 i 的相关列向量，生成单词 i 的嵌入，如图7.16所示。

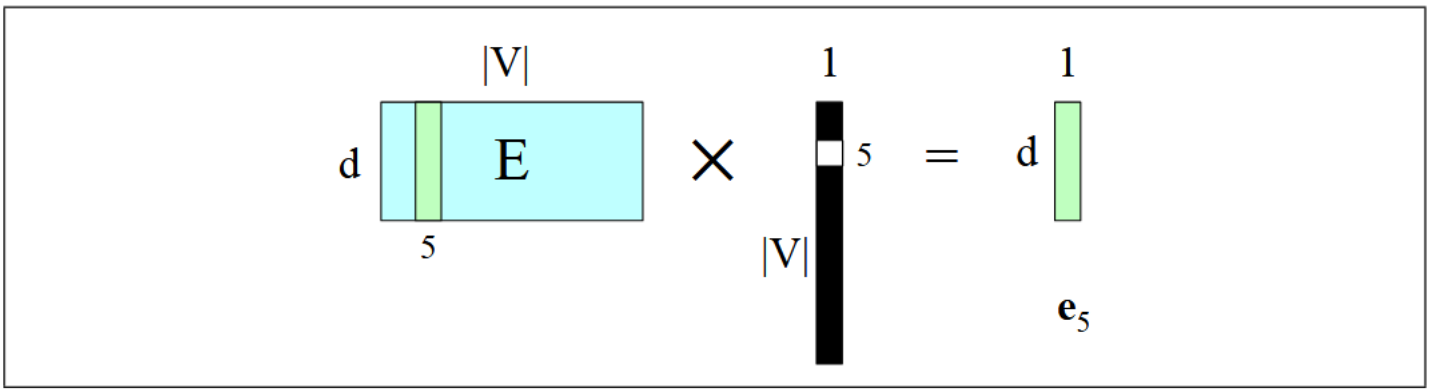

图 7.16 通过将嵌入矩阵 E 与索引 5 中值为 1 的 one-hot 向量相乘来选择单词 V5 的嵌入向量。

生成的3个嵌入向量被**连接**在一起，形成嵌入层 e 。接下来是一个隐藏层和一个输出层，输出层的Softmax会生成单词的概率分布。例如，输出节点42的值 $y_4 2$ 是下一个单词 w_t 为 V_{42} 的概率， V_{42} 是词汇表中索引为42的单词（在我们的例子中是“fish”这个单词）。

以下是我们的小示例中的详细算法步骤：

1. 从E中选择三个嵌入：给定前面三个单词，我们查找它们的索引，创建3个独热向量，然后分别与嵌入矩阵E相乘。考虑 w_{t-3} ，单词“for”（索引为35）的独热向量与嵌入矩阵E相乘，生成第一个隐藏层的第一部分，即嵌入层。由于输入矩阵E的每一列是一个单词的嵌入，输入是单词 V_i 的独热列向量 x_i ，因此输入 w 的嵌入层为 $E_{x_i} = e_i$ ，即单词 i 的嵌入。我们现在将三个上下文单词的嵌入连接起来，生成嵌入层e。
2. 与W相乘：我们将e与W相乘（并加上b），然后通过ReLU（或其他）激活函数，得到隐藏层h。
3. 与U相乘：h与U相乘。
4. 应用Softmax：在Softmax之后，输出层中的每个节点i估计下一个单词 w_t 为i的概率，即 $$P(w_t=i|w_{t-1},w_{t-2},w_{t-3})$$。

总结一下，窗口大小为3的神经语言模型的方程如下，输入上下文单词的独热向量:

$$
\begin{aligned}
\mathrm{e} &= [\mathrm{Ex}_{\mathrm{t}-3}; \mathrm{Ex}_{\mathrm{t}-2}; \mathrm{Ex}_{\mathrm{t}-1}] \\
\mathrm{h} &= \sigma(\mathrm{We} + \mathrm{b}) \\
\mathbf{z} &= \mathbf{Uh} \\
\hat{\mathbf{y}} &= \mathrm{softmax}(\mathbf{z})
\end{aligned}
$$

注意，我们通过将三个上下文向量的嵌入连接起来形成嵌入层e；我们通常使用分号表示向量的连接.

7.7 训练神经语言模型

无论是我们在这里描述的简单前馈语言模型，还是第9章中更强大的transformer语言模型，训练神经语言模型的高层直觉都是第6章中我们在学习词表示时看到的自我训练或**自监督(self-supervision)**的理念。在语言建模的自我训练中，我们将一篇文本作为训练材料，并在每个时间步t要求模型预测下一个单词。起初，模型在这个任务上表现会很差，但由于我们在每种情况下都知道正确答案（即文本中的下一个单词！），我们可以轻松地训练它，使其在预测正确的下一个单词时表现得更好。我们称这样的模型为自监督模型，因为我们不需要为数据添加任何特殊的标签；自然的单词序列就是它自己的监督！我们只需训练模型，最小化在训练序列中预测真实下一个单词时的误差。

在实际操作中，训练模型意味着设定参数 $\theta = E, W, U, b$ 。对于某些任务，可以将嵌入层E固定为初始的word2vec值。固定意味着我们使用word2vec或其他预训练算法计算初始嵌入矩阵E，然后在训练语言模型时保持其不变，只修改W, U和b，即我们不在语言模型训练期间更新E。然而，通常我们希望在训练网络的同时学习嵌入。这在网络设计的任务（如情感分类、翻译或句法分析）对单词表示的要求很高时非常有用。

让我们看看如何训练整个模型，包括E，即设定所有参数 $\theta = E, W, U, b$ 。我们将通过梯度下降来实现这一点（图??），使用计算图上的误差反向传播来计算梯度。因此，训练不仅设定了网络的权重W和U，同时在我们预测接下来的单词时，我们也在为每个单词学习最适合预测下一个单词的嵌入E。

图7.18展示了窗口大小为N=3的上下文单词的设置。输入x由3个独热向量组成，通过嵌入矩阵E的3个实例完全连接到嵌入层。我们不希望为将每个前面3个单词映射到投影层学习单独的权重矩阵，而是希望这三个单词共享同一个嵌入词典E。因为随着时间的推移，许多不同的单词会出现在wt－2或wt－1位置，我们只希望某个单词无论出现在什么上下文位置时，都由一个向量表示。回想一下，嵌入权重矩阵E为每个单词都有一列，每列是一个d维列向量，因此其维度为d × |V|。

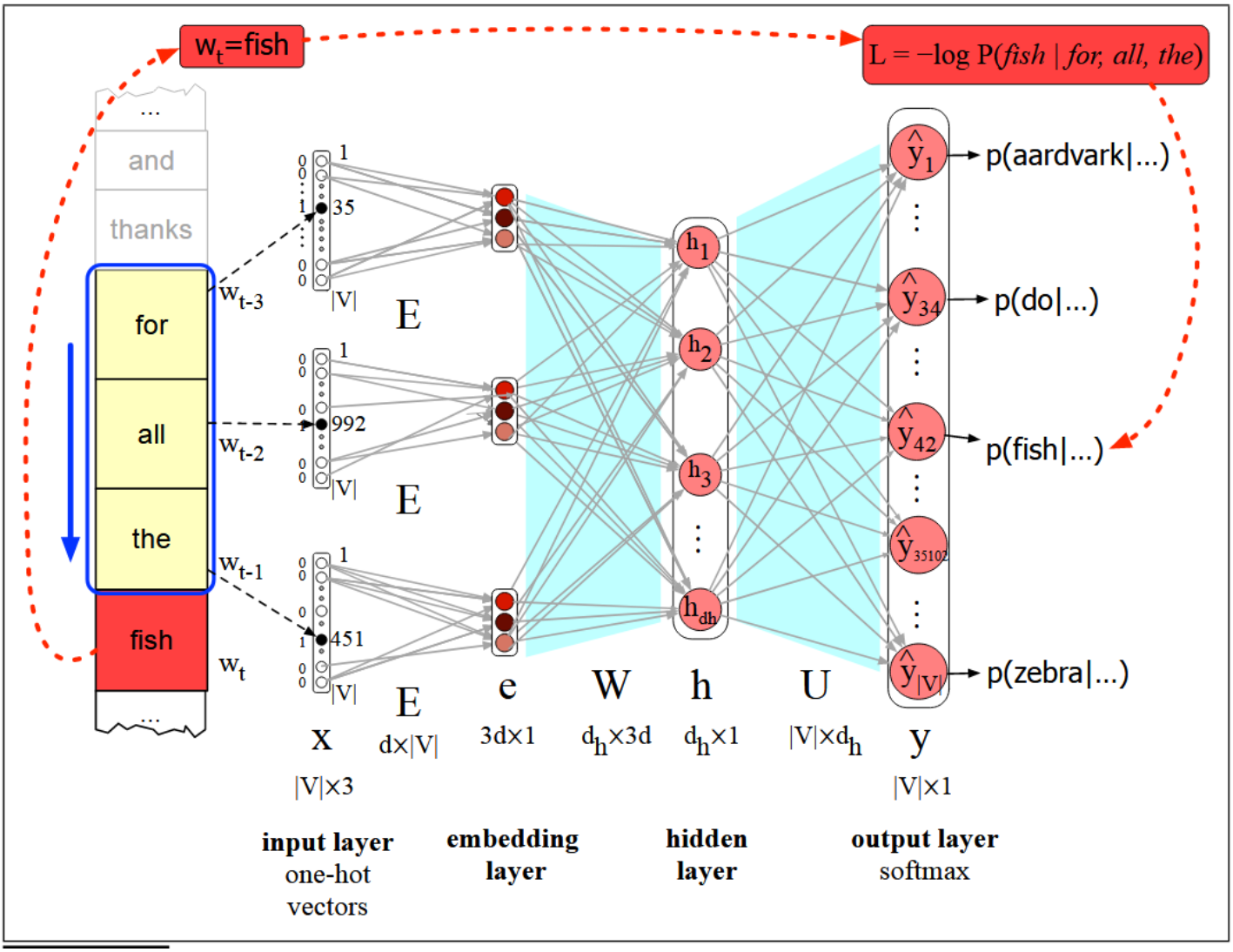

图 7.18 包括嵌入在内的参数全部进行学习。同样，嵌入矩阵 E 在 3 个上下文单词之间共享。

通常训练通过输入一段很长的文本开始，将所有句子连接起来，使用随机权重初始化，然后在文本中迭代预测每个单词wt。在每个单词wt处，我们使用交叉熵（负对数似然）损失。回想一下，这个损失的一般形式（重复方程7.25）如下：

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i, \quad \text{(where } i \text{ is the correct class)}$$

对于语言建模，类别是词汇表中的单词，因此 $\hat{y}_i$ 表示模型赋予正确下一个单词wt的概率：

$$L_{\text{CE}} = -\log p(w_t|w_{t-1}, ..., w_{t-n+1})$$

从步骤s到s + 1的随机梯度下降的参数更新可以在任何标准的神经网络框架中计算出来，这样就可以通过θ = E, W, U, b进行反向传播：

$$\boldsymbol{\theta}^{s+1} = \boldsymbol{\theta}^s - \boldsymbol{\eta}\frac{\partial\left[-\log p(w_t|w_{t-1}, ..., w_{t-n+1})\right]}{\partial \boldsymbol{\theta}}$$

通过训练参数以最小化损失，我们不仅得到了一个用于语言建模的算法（即单词预测器），还得到了可以用于其他任务的全新嵌入集E。

7.8 总结

- 神经网络由神经单元构建，这些单元最初受生物神经元启发，但现在只是一个抽象的计算装置。

- 每个神经单元通过一个权重向量对输入值进行相乘，加上一个偏置，然后应用非线性激活函数，如Sigmoid、tanh或修正线性单元（ReLU）。
- 在一个全连接的前馈网络中，第i层的每个单元都与第i+1层的每个单元相连，没有循环。
- 神经网络的强大之处在于，早期层能够学习表示，这些表示可以被网络的后续层利用。
- 神经网络通过像梯度下降这样的优化算法进行训练。
- 误差反向传播，即在计算图上进行的反向微分，被用于计算网络的损失函数梯度。
- 神经语言模型使用神经网络作为概率分类器，根据前n个单词计算下一个单词的概率。
- 神经语言模型可以使用预训练的嵌入，也可以在语言建模过程中从头开始学习嵌入。