7. 神经网络和神经语言模型

> “[M]achines of this character can behave in a very complicated manner when the number of units is large.”
>
> Alan Turing (1948) “Intelligent Machines”, page 6

神经网络是用于语言处理的基本计算工具，并且是一个非常古老的工具。它们之所以被称为神经元，是因为它们的起源在于麦卡洛克-皮茨神经元(McCulloch 和 Pitts，1943)，这是人类神经元的简化模型，可以作为一种计算元素，可用命题逻辑来描述。但是，语言处理中的现代用法不再汲取这些早期生物学灵感。

取而代之的是，现代的神经网络是由小型计算单元组成的网络，每个计算单元采用输入值的向量并产生单个输出值。在本章中，我们介绍了应用于分类的神经网络。我们介绍的体系结构称为**前馈网络(feedforward network)**，因为计算是从一个层(由很多单元组成)到下一个层迭代进行的。现代神经网络的使用通常称为**深度学习(deep learning)**，因为现代网络通常很深(具有许多层)。

神经网络与逻辑回归具有许多相同的数学原理。但是，神经网络比逻辑回归具有更强大的分类器，实际上，可以证明一个最小的神经网络(技术上具有单个“隐藏层”)可以学习任何功能。

神经网络分类器在另一个方面不同于逻辑回归。通过逻辑回归，我们通过基于领域知识开发多种丰富的特征模板，将回归分类器应用于许多不同的任务。在使用神经网络时，更常见的是避免大多数使用丰富的手工衍生特征，而建立以原始单词为输入并学习归纳特征的神经网络，作为学习分类过程的一部分。我们在第 6 章中看到了用于嵌入的这种表示学习的示例。非常深的网络特别擅长表示学习。因此，深度神经网络是解决大规模问题的正确工具，可以提供足够的数据来自动学习特征。

在本章中，我们将介绍前馈网络作为分类器，并将它们应用于语言建模的简单任务：为单词序列分配概率并预测即将出现的单词。在随后的章节中，我们将介绍神经模型的许多其他方面，例如**递归神经网络(recurrent neural networks，RNN)**和 **Transformer**(第 9 章)，上下文嵌入(如 **BERT**)(第 10 章)以及**编解码器(encoder-decoder)**模型和**注意力(attention)**(第 11 章)。

7.1. 单元

神经网络的构建块是单个计算单元。一个单元将一组实数值作为输入，对其进行一些计算，然后产生输出。

从本质上讲，神经单元正在对其输入进行加权求和，其中的另一个项称为偏差项(bias term)。给定一组输入 $x_1 \dots x_n$，一个单元具有一组相应的权重 $w_1 \dots w_n$ 和偏差 b，因此加权总和 z 可以表示为：

$$z = b + \sum_i w_i x_i \tag{7.1}$$

通常，使用向量符号表示此加权和会更方便； 从线性代数中回想起，向量本质上只是一个数字列表或数字数组。因此，我们将以权重向量 w，标量偏差 b 和输入向量 x 来讨论 z，并将总和替换为方便的点积：

$$z = w \cdot x + b \tag{7.2}$$

正如公式 7.2 中定义的，z 只是一个实数。

最终，神经单元将非线性函数 f 应用于 z(而不是使用 x 的线性函数 z)作为输出。我们将此功能的输出称为单元 a 的**激活(activation)**值。由于我们仅对单个单元建模，因此节点的激活实际上是网络的最终输出，通常将其称为 y。因此，值 y 定义为：

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

我们将在下面讨论三种流行的非线性函数 f( ) (sigmoid，tanh，和修正的线性 ReLU)，但从教学角度

来看，从 **sigmoid** 函数开始比较方便，因为我们在第 5 章看到过它:

$$y = \sigma(z) = \frac{1}{1+e^{-z}} \tag{7.3}$$

Sigmoid(如图 7.1 所示)具有许多优点。它将输出映射到[0,1]范围内，这对于将异常值压向 0 或 1 非常有用。而且它是可微的，正如我们在 5.8 节中看到的那样，它对于学习很方便。

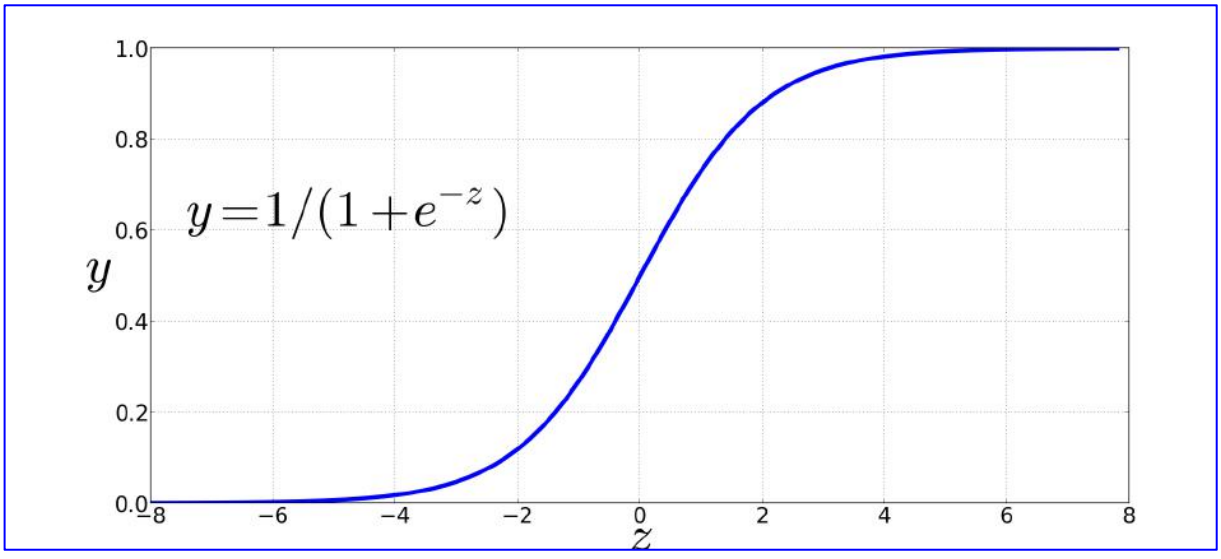

图 7-1：Sigmoid 函数

图注：Sigmoid 函数接受一个实数，并将其映射到范围[0,1]。它在 0 附近几乎是线性的，但是离群值被压向 0 或 1。

将公式 7.2 代入公式 7.3 得到一个神经单元的输出:

$$y = \sigma(w \cdot x + b) = \frac{1}{1+\exp(-(w \cdot x + b))} \tag{7.4}$$

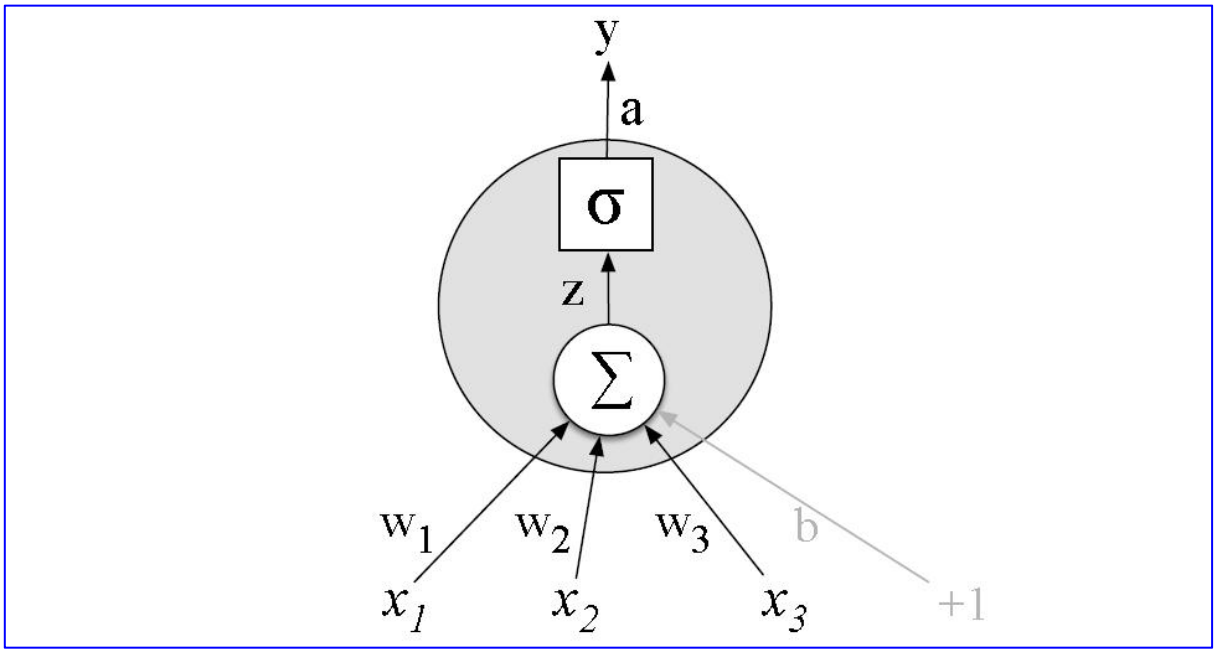

图 7-2：一个神经单元

图注：一个神经单元，取 3 个输入 x_1，x_2 和 x_3(以及一个偏差 b，我们将其表示为固定在+1 上的输入的权重)，并产生输出 y。我们包括一些方便的中间变量：总和 z 的输出和 S 型曲线 a 的输出。在这种情况下，单位 y 的输出与 a 相同，但是在更深的网络中，我们将保留 y 表示整个网络的最终输出，而保留 a 作为单个节点的激活。

图 7.2 显示了基本神经单元的最终示意图。在此示例中，该单元采用 3 个输入值 x_1，x_2 和 x_3，并计算一个加权和，将每个值乘以一个权重(分别为 w_1，w_2 和 w_3)，将它们加到偏差项 b，然后将累加结果通过 S 型函数传递，以得出介于 0 和 1 之间的数字的最终结果。

让我们来看一个例子，只是为了获得直觉。假设我们有一个具有以下权重向量和偏差的单元：

$$w = [0.2,0.3,0.9]$$

$$b = 0.5$$

该单元将使用以下输入向量：

x = [0.5,0.6,0.1]

结果输出 y 为：

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

实际上，sigmoid 通常不用作激活函数。图 7.3a 所示的 **tanh** 函数非常相似，但几乎总是更好。tanh 是 S 型的一种变体，范围从-1 到+1：

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{7.5}$$

最简单的激活函数，也许是最常用的激活函数，是修正的线性单元，也称为 **ReLU**，如图 7.3b 所示。若 x 为正数，则它与 x 相同；否则为 0：

$$y=\max(x,0) \tag{7.6}$$

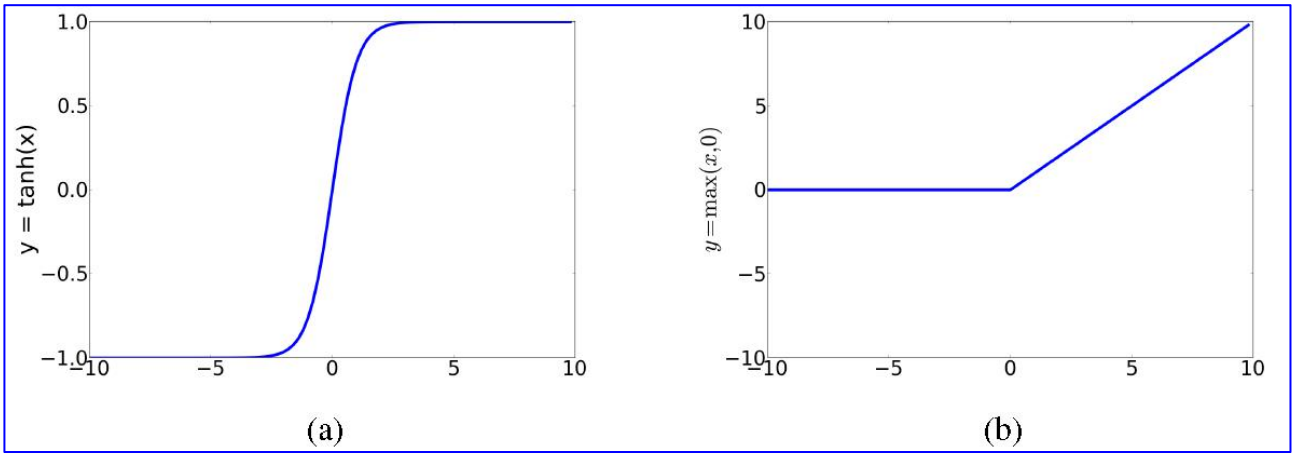

图 7-3：tanh 和 ReLU 激活函数

这些激活函数具有不同的属性，这些属性使它们对于不同的语言应用程序或网络体系结构很有用。例如，tanh 函数具有很好的特性，可以平滑地进行微分，并将离群值映射到均值。另一方面，修正函数具有很好的特性，这是由于它非常接近线性。在 S 型或 tanh 函数中，很高的 z 值会导致 y 值的**饱和(saturated)**，即非常接近 1，并且导数非常接近 0。零导数会引起学习问题，因为我们将在 7.4 节中看到，我们将通过向后传播误差信号并乘以网络各层的梯度(偏导数)来训练网络； 几乎为 0 的梯度会导致误差信号变得越来越小，直到它变得太小而无法用于训练为止，这个问题称为**消失梯度(vanishing gradient)**问题。修正函数没有这个问题，因为 ReLU 对高 z 值的导数是 1，而不是非常接近 0。

7.2.XOR 问题

在神经网络的历史早期，人们已经意识到神经网络的力量，就像激发它们的真实神经元一样，来自将这些单元组合成更大的网络。

Minsky 和 Papert(1969)证明了多层网络需求的最巧妙的证明之一，即单个神经单元无法计算其输入的某些非常简单的功能。考虑计算两个输入(例如 AND，OR 和 XOR)的基本逻辑功能的任务。提醒一下，

这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

此示例首先针对**感知器(perceptron)**显示，该感知器是一个非常简单的神经单元，具有二进制输出并且不具有非线性激活函数。感知器的输出 y 为 0 或 1，其计算方式如下(使用与公式 7.2 相同的权重 w，输入 x 和偏差 b)：

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases} \tag{7.7}$$

构建一个可计算其二元输入的逻辑 AND 和 OR 功能的感知器非常容易；图 7.4 显示了必要的权重。

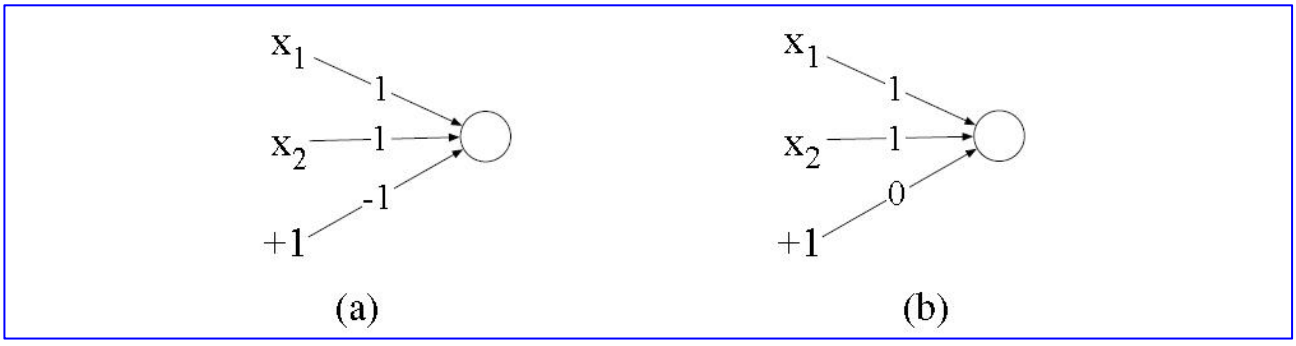

图 7-4：逻辑函数的感知器

图注：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 x_1 和 x_2，偏差显示为特殊节点，其值+1 与偏差权重 b 相乘。(a)逻辑 AND，显示权重 $w_1 = 1$ 和 $w_2 = 1$，偏差权重 $b = -1$。(b)逻辑 OR，显示权重 $w_1 = 1$ 和 $w_2 = 1$，偏差权重 $b = 0$。这些权重/偏差仅仅是实现函数的无数可能的权重和偏差集之一。

但是事实证明，不可能构建感知器来计算逻辑 XOR(异或)！(值得花点时间尝试一下！)

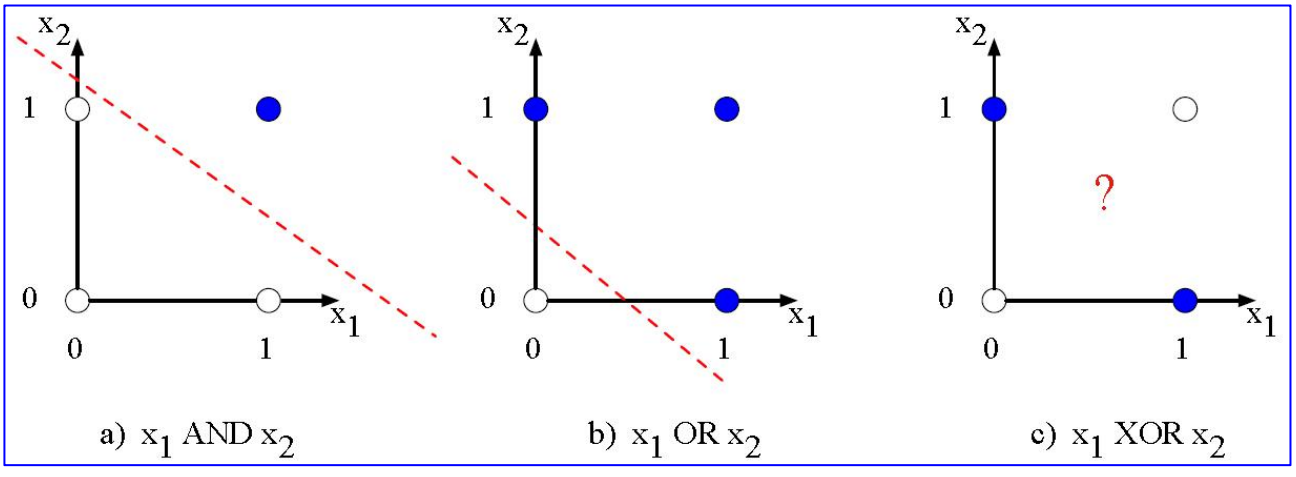

图 7-5：函数 AND，OR 和 XOR 的分类线

图注：函数 AND，OR 和 XOR 在 x 轴上表示为输入 x_1，在 y 轴上表示为输入 x_2。实心圆圈表示感知器输出为 1，白色圆圈感知器输出为 0。无法绘制正确区分 XOR 的两个类别的线。图样来自 Russell 和 Norvig(2002)。

重要结果背后的直觉取决于对感知器是线性分类器的理解。对于二维输入 x_1 和 x_2，感知方程 $w_1x_1 + w_2x_2 + b = 0$ 是直线的方程(我们可以通过将其放入标准线性格式中来看到它：$x_2 = (-w_1/w_2)x_1 + (-b/w_2)$)。该线充当二维空间中的**决策边界(decision boundary)**，在该二维空间中，将输出 0 分配给该线一侧的所有输入点，将输出 1 分配给该线另一侧的所有输入点。如果我们有 2 个以上的输入，则决策边界将变为超平面而不是直线，但是想法是相同的，将空间分为两类。

图 7.5 显示了可能的逻辑输入(00、01、10 和 11)，以及由一个可能的一组参数组成的与门(AND)、或门(OR)的分类线。注意，根本没有办法画出将异或门(XOR)的正例(01 和 10)与负例(00 和 11)分开的线。我们说 XOR 不是**线性可分(linearly separable)**函数。当然，我们可以用曲线或其他函数绘制边界，但不能绘制一条直线。

7.2.1. 解决方案：神经网络

虽然 XOR 函数不能由单个感知器计算，但可以由单元的分层网络计算。我们来看看 Goodfellow 等人(2016)的示例。使用两层基于 ReLU 的单元计算 XOR。图 7.6 显示了一个由两层神经单元处理输入的图形。中间层(称为 h)具有两个单元，对于正确计算 XOR 函数的每个 ReLU，显示了一组权重和偏差。

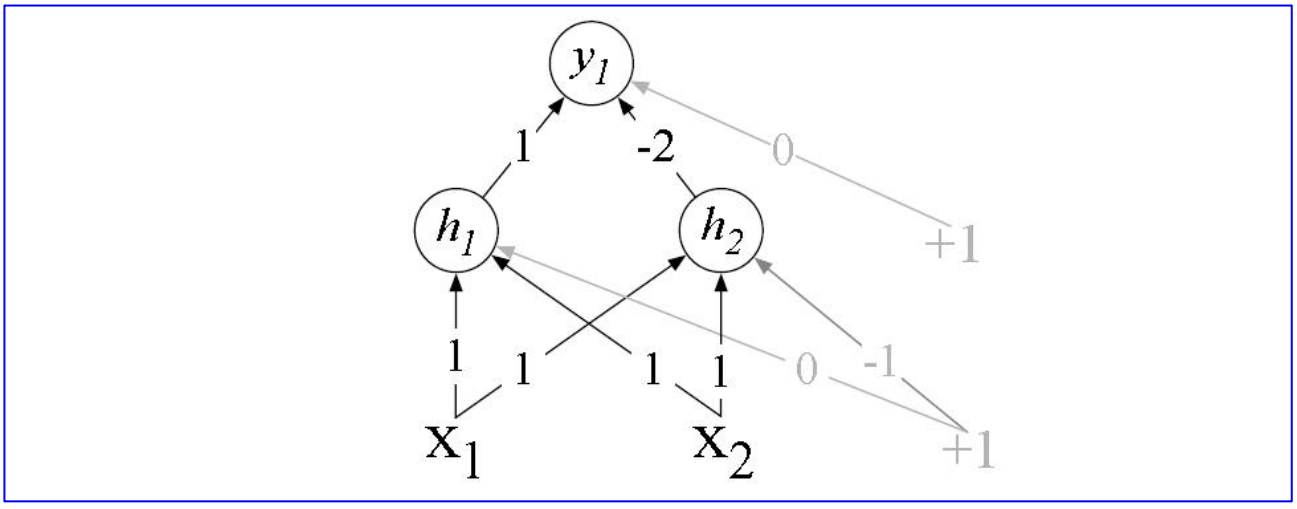

图 7-6：Goodfellow 等人的 XOR 解决方案

图注：Goodfellow 等人(2016)的 XOR 解决方案。共有三个 ReLU 单元，分为两层；称之为 h_1、h_2(h 代表隐藏层)和 y_1。如前所述，箭头上的数字表示每个单元的权重 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。

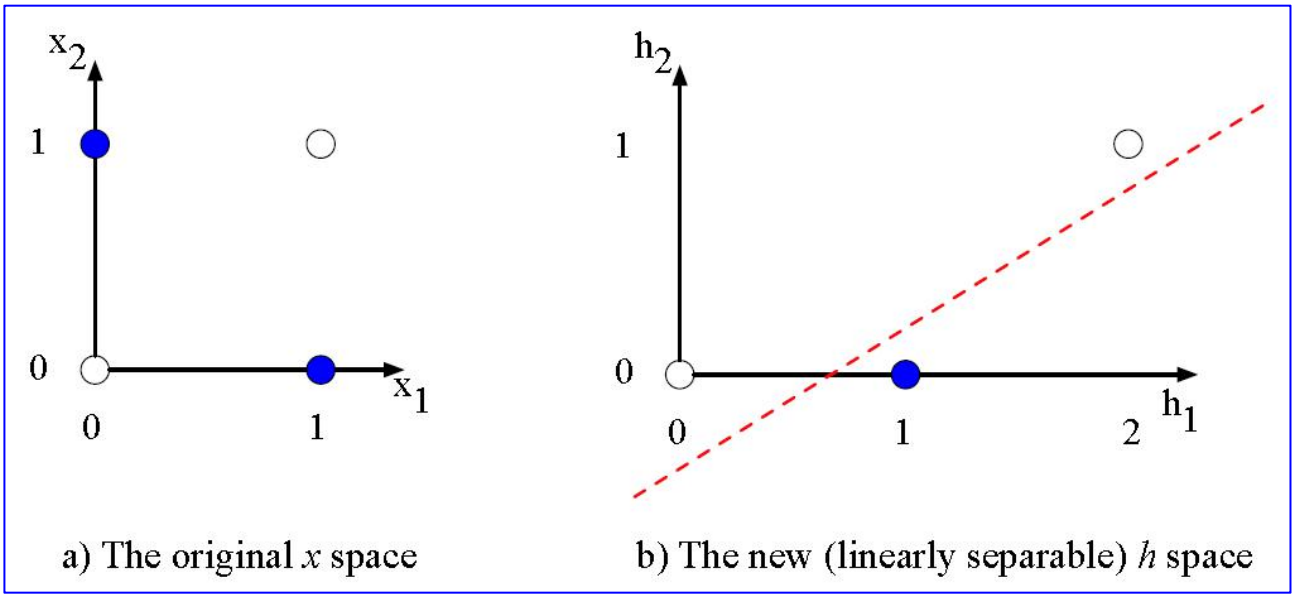

图 7-7：隐含层形成输入的新表示

图注：图(a)显示了原始输入 x 的表示，图(b)显示了隐藏层 h 的表示。注意，输入点[0 1]已经与输入点[1 0]折叠，这使得线性分离异或(XOR)的正负情况成为可能。根据 Goodfellow 等人(2016)。

让我们来看看输入 x = [0 0]会发生什么。如果我们将每个输入值乘以适当的权重，再求和，然后加上偏差 b，我们得出矢量[0 -1]，然后应用修正的线性变换将 h 层的输出设为[0 0] 。现在，我们再次乘以权重、求和并加上偏差(在这种情况下为 0)，得出的值为 0。读者应该对其余 3 个可能的输入对进行计算，以查看输入[0 1]和[1 0]的 y 值为 1，输入[0 0]和[1 1]的 y 值为 0。

查看中间结果(两个隐藏节点 h_1 和 h_2 的输出)也很有启发。在上一段中，我们证明了输入 x = [0 0]的 h 矢量为[0 0]。图 7.7b 显示了所有 4 个输入的 h 层的值。请注意，两个输入点 x = [0 1]和 x = [1 0](XOR 输出= 1 的两种情况)的隐藏表示被合并到单个点 h = [1 0]。通过合并，可以很容易地线性区分异或(XOR)的正例和负例。换句话说，我们可以将网络的隐藏层视为输入的表示形式。

在此示例中，我们仅在图 7.6 中规定了权重。但是对于真实的例子，将使用第 7.4 节介绍的误差反向传播算法自动学习神经网络的权重。这意味着隐藏层将学习形成有用的表示。这种直觉，即神经网络可以自动学习输入的有用表示形式，是它们的主要优势之一，这一点我们将在以后的章节中反复讨论。

请注意，解决 XOR 问题的方法需要具有非线性激活函数的单元网络。由简单线性(感知器)单元组成的网络无法解决 XOR 问题。这是因为，由多层纯线性单元组成的一个网络，始终可以被简化为(例如,显示在计算上相同)一个单层线性单元(具有适当权重)，而且我们已经显示(视觉上,在图 7.5 中)了单个单元不能解决 XOR 问题。

7.3. 前馈神经网络

现在，让我们简单地介绍一下最简单的神经网络，即**前馈网络(feedforward network)**。前馈网络是一个多层网络，其中的单元之间没有循环连接。每层单元的输出将传递到下一个较高层的单元，而没有输出被传递回到较低层。(在第 9 章中，我们将介绍具有循环的网络，称为循环神经网络。)

出于历史原因，多层网络，尤其是前馈网络，有时也称为**多层感知器(multi-layer perceptrons，MLP)**。这是一种技术上的误称，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，但是现代网络是由具有非线性的单元(如 S 型)组成的)，但在某种程度上，这个名称就固定下来了。

简单前馈网络具有三种节点：输入单元，隐藏单元和输出单元。图 7.8 示出了一个图片。

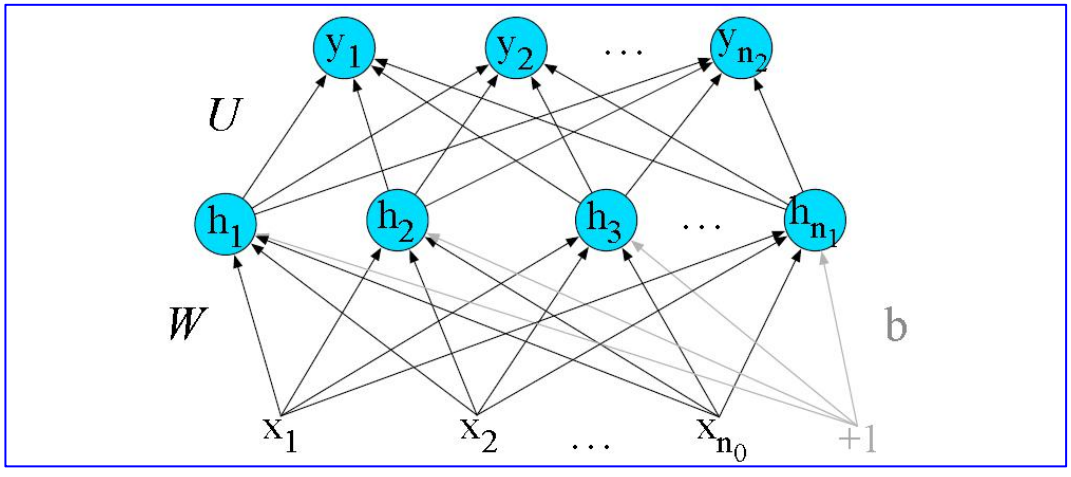

图 7-8：一个简单的 2 层前馈网络

图注：具有一个隐藏层，一个输出层和一个输入层(枚举层时通常不计算输入层)。

输入单元只是标量值，如图 7.2 所示。

神经网络的核心是由隐藏单元组成的隐藏层，每个隐藏单元都是第 7.1 节所述的神经单元，对输入进行加权求和，然后应用非线性。在标准体系结构中，每一层都是**全连接(full connected)**，这意味着每一层中的每个单元都将前一层中所有单元的输出作为输入，并且来自两个相邻层的每对单元之间都有一个连接。因此，每个隐藏单元对所有输入单元进行累加求和。

回想一下，单个隐藏单元的参数为 w(权重向量)和 b(偏差标量)。我们通过将每个单位 i 的权重向量 w_i 和偏差标量 b_i 组合为单个权重矩阵 W 和整个层的单个偏差向量 b 来表示整个隐藏层的参数(见图 7.8)。权重矩阵 W 的每个元素 W_{ji} 代表从第 i 个输入单元 x_i 到第 j 个隐藏单元 h_j 的连接的权重。

对整个层的权重使用单个矩阵 W 的优点是，现在可以通过简单的矩阵运算非常有效地完成前馈网络的隐藏层计算。实际上，该计算仅包括三个步骤：将权重矩阵与输入向量 x 相乘，加上偏差向量 b，然后应用激活函数 g(例如上面定义的 S 型，tanh 或 ReLU 激活函数)。

因此，使用 S 形函数 σ，隐藏层的输出矢量 h 如下：

$$h = \sigma(Wx+b) \tag{7.8}$$

注意，我们在这里对一个向量应用σ函数，而在公式 7.3 中，它是对一个标量应用的。因此，我们允许σ(·)，以及实际上任何激活函数 g(·)适用于向量元素，因此 $g[z_1,z_2,z_3] = [g(z_1),g(z_2),g(z_3)]$。

让我们介绍一些常数来表示这些向量和矩阵的维数。我们将输入层称为网络的第 0 层，并用 n_0 表示输入的数量，因此 x 是维度为 n_0 的实数的矢量，或者更正式地说 $x \in R^{n_0}$是维数$[n_0,1]$的列向量。我们将隐藏层称为第 1 层，将输出层称为第 2 层。该隐藏层的维数为 n_1，因此 $h \in R^{n_1}$和 $b \in R^{n_1}$(因为每个隐藏单元可以采用不同的偏差值)，并且权重矩阵 W 具有维数 $W \in R^{n_1 \times n_0}$，即$[n_1,n_0]$。

花点时间说服自己，在公式 7.8 中的矩阵乘法将计算每个 h_j 的值为$\sigma(\sum_{i=0}^{n_0} w_{ji}x_i + b_j)$。

正如我们在 7.2 节中所看到的，结果值 h(对于隐藏的还是对于假设的)形成了输入的表示。输出层的作用是采用新的表示形式 h 并计算最终输出。该输出可能是实数值，但是在许多情况下，网络的目标是做出某种分类决策，因此我们将重点介绍分类情况。

如果我们正在执行情感分类之类的二元任务，那么我们可能只有一个输出节点，其值 y 是正面情感与负面情感的概率。如果我们正在进行多元分类，例如分配词类标签，则每个潜在的词类可能有一个输出节点，其输出值是该词类的概率，并且所有输出节点的值之和必须为 1。因此，输出层给出了横跨(across)输出节点的概率分布。

让我们看看这是怎么发生的。与隐藏层一样，输出层具有权重矩阵(简称为 U)，但是某些模型在输出层中不包含偏差矢量 b，因此在此示例中，我们将通过消除偏差矢量来简化操作。权重矩阵乘以其输入向量(h)以产生中间输出 z。

$$z = Uh$$

有 n_2 个输出节点，因此 $z \in R^{n2}$，权重矩阵 U 具有维度 $U \in R^{n_2 \times n_1}$，元素 U_{ij} 是从隐藏层的单元 j 到输出层的单元 i 的权重。

但是，z 不能作为分类器的输出，因为 z 是实数值的向量，而我们需要的分类是概率的向量。有一个方便的函数可以对实值向量进行归一化，即将其归一化为将其编码为概率分布的向量(所有数字都在 0 到 1 之间，并且总和为 1)：softmax 函数(第 5 章)。对于向量为 d 的向量，softmax 定义为：

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} \quad 1 \le i \le d \tag{7.9}$$

因此，例如给定向量 z = [0.6，1.1，-1.5，1.2，3.2，-1.1]，softmax(z)为[0.055，0.090，0.0067，0.10，0.74，0.010]。

您可能还记得，在第 5 章的逻辑回归中，softmax 正是根据实值数字(从权重乘以特征求和而来)的向量来创建概率分布的方法。

这意味着我们可以将具有一个隐藏层的神经网络分类器视为构建向量 h，该向量 h 是输入的隐藏层表示，然后对网络在 h 中发展的特征进行标准逻辑回归。相比之下，在第 5 章中，主要通过特征模板手动设计功能。因此，神经网络类似逻辑回归，但是有两点不同：(a)有许多层，因为深度神经网络就像是逻辑回归分类器的一层又一层；(b)不是通过特征模板形成特征，而是网络的前几层自己引导特征表示。

以下是具有单个隐藏层的前馈网络的最终方程式，该方程式采用输入向量 x，输出概率分布 y，并由权重矩阵 W 和 U 以及偏置向量 b 进行参数化：

$$\begin{aligned} h &= \sigma(Wx+b) \\ z &= Uh \\ y &= \text{softmax}(z) \end{aligned} \tag{7.10}$$

我们将此网络称为 2 层网络(传统上，在对层进行编号时，我们不对输入层进行计数，而对输出层进行计数)。因此，根据此术语，逻辑回归是一个 1 层网络。

现在，让我们设置一些符号，以便更轻松地讨论深度大于 2 的更深层网络。我们将在方括号中使用上标来表示层号，输入层的编号从 0 开始。因此，$W^{[1]}$表示(第一)隐藏层的权重矩阵，$b^{[1]}$表示(第一)隐藏层的偏差矢量。n_j 表示第 j 层的单元数。我们将使用 g(·)代表激活函数，中间层往往是 ReLU 或 tanh，输出层

通常为 softmax。我们将使用 $a^{[i]}$表示第 i 层的输出，使用 $z^{[i]}$表示权重和偏差 $W^{[i]}\ a^{[i-1]}+ b^{[i]}$的组合。第 0 层用于输入，因此我们通常将输入 x 称为 $a^{[0]}$。

因此，我们可以将公式 7.10 中的 2 层网络重新表示为:

$$\begin{aligned} z^{[1]} &= W^{[1]}a^{[0]} + b^{[1]} \\ a^{[1]} &= g^{[1]}(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= g^{[2]}(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned} \tag{7.11}$$

注意，使用这种符号，每一层的计算公式都是相同的。给定输入向量 $a^{[0]}$，在 n 层前馈网络中计算前向步长的算法因此很简单：

for i **in** 1..n

$z^{[i]} = W^{[i]}\ a^{[i-1]} + b^{[i]}$

$a^{[i]} = g^{[i]}(z^{[i]})$

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

激活函数 g(·)通常在最后一层是不同的。因此，对于多元分类，$g^{[2]}$可能是 softmax；对于二进制分类，$g^{[2]}$可能是 sigmoid。而 ReLU 或 tanh 可能是内部层的激活函数 g(·)。

更换偏置单元 在描述网络时，我们经常会使用略微简化的符号表示完全相同的函数，而无需引用显式偏置节点 b。相反，我们向每个值始终为 1 的层添加一个虚拟节点 a_0。因此，输入层 0 层将具有一个虚拟节点$a_0^{[0]}= 1$，第 1 层将具有$a_0^{[1]}= 1$，并且以此类推。该虚拟节点仍然具有关联的权重，并且该权重代表偏差值 b。例如：

不使用以下公式：

$$h = \sigma(Wx+b) \tag{7.12}$$

而使用以下公式：

$$h = \sigma(Wx) \tag{7.13}$$

但是现在，我们的向量 x 不再具有 n 个值：$x = x_1, \ldots, x_n$，而是 n+1 个值，其中第 0 个虚拟值 $x_0 = 1$：$x = x_0, \ldots, x_n$。

不是像下面那样计算每个 h_j：

$$h_j = \sigma\left(\sum_{i=1}^{n_0} w_{ji}x_i + b_j\right), \tag{7.14}$$

而是改用：

$$\sigma\left(\sum_{i=0}^{n_0} w_{ji}x_i\right), \tag{7.15}$$

其中值 W_{j0} 代替了 b_j。图 7.9 显示了可视化。

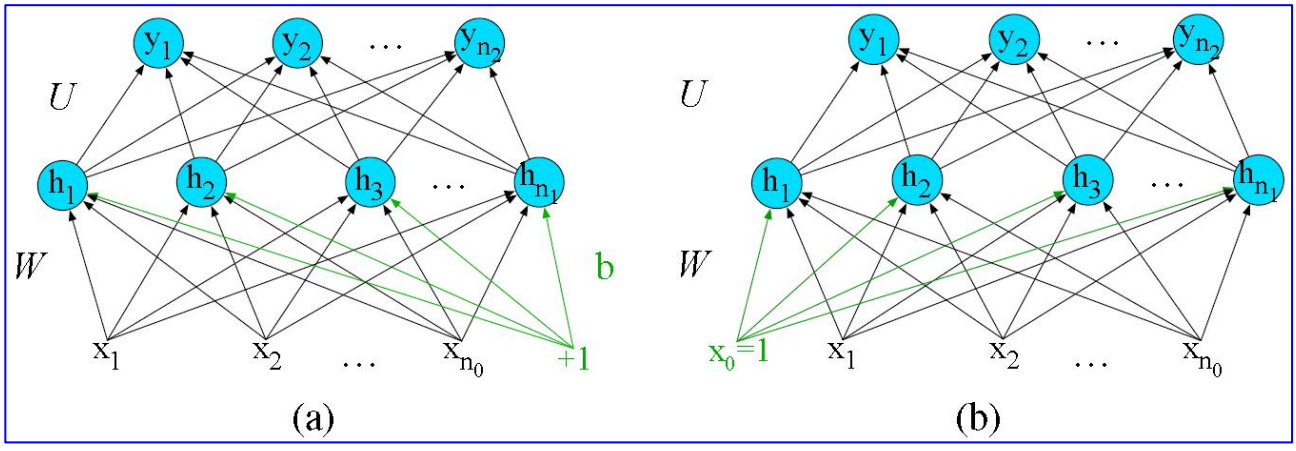

图 7-9：用 x_0 替换偏置节点 b

在下一节中，我们将继续在学习示例中将偏差显示为 b，但是在本书的其余部分中，我们将切换为这种简化的表示法，而没有明确的偏差项。

7.4. 训练神经网络

前馈神经网络是有监督机器学习的一个实例，其中我们知道每个观测值 x 的正确输出 y。系统通过公式 7.11 产生的结果是ŷ，即系统对真实 y 的估计。训练过程的目标是学习每个层 i 的参数 $W^{[i]}$和 $b^{[i]}$，使每个训练观察的ŷ尽可能接近真实 y。

一般来说，我们都是通过在第 5 章中介绍的逻辑回归方法来完成这一切的，所以读者在继续之前应该熟悉那一章。

首先，我们需要一个**损失函数(loss function)**来对系统输出和**黄金输出(gold output)**之间的距离进行建模，通常使用用于逻辑回归的损失函数，即**交叉熵损失(cross-entropy loss)**。

其次，为了找到使损失函数最小的参数，我们将使用第 5 章中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，该向量包含损失函数相对于每个参数的偏导数。这是神经网络的学习比逻辑回归更复杂的部分。在逻辑回归中，对于每个观察值，我们都可以直接计算损失函数相对于单个 w 或 b 的导数。但是对于神经网络来说，在许多层中都有数百万个参数，当损失附加到后面的层时，如何计算第一层中某些权重的偏导数要困难得多。我们如何分摊所有这些中间层的损失？

答案是称为**误差反向传播(error backpropagation)**或**逆向微分(reverse differentiation)**的算法。

7.4.1. 损失函数

在神经网络中使用的交叉熵损失和我们在逻辑回归中看到的是一样的。事实上，如果将神经网络用作二进制分类器，将 sigmoid 置于最后一层，则损失函数与我们在公式 5.11 中看到的逻辑回归完全相同:

$$L_{CE}(\hat{y},y) = -\log p(y|x) = -[y\log\hat{y} + (1-y)\log(1-\hat{y})] \tag{7.16}$$

如果神经网络被用作多元分类器怎么办？令 y 为代表真实输出概率分布的 C 类上的向量。这里的交叉熵损失是：

$$L_{CE}(\hat{y},y) = -\sum_{i=1}^{C} y_i \log\hat{y}_i \tag{7.17}$$

我们可以进一步简化该方程式。假设这是一项**硬分类(hard classification)**任务，这意味着只有一个类别是正确的类别，并且每个类别的 y 中都有一个输出单元。如果真实类别是 i，则 y 是一个向量，其中 $y_i = 1$ 且 $y_j = 0 \ \ \forall j = i$。像这样的一个向量，其中只有一个值为 1，其余值均为 0，被称为**独热(one-hot)**向量。除真实类别的项之外，公式 7.17 中的各项总和将为 0，即：

$$\begin{aligned} L_{CE}(\hat{y},y) &= -\sum_{k=1}^{K} \mathbb{1}\{y=k\}\log\hat{y}_i \\ &= -\sum_{k=1}^{K} \mathbb{1}\{y=k\}\log\hat{p}(y=k|x) \\ &= -\sum_{k=1}^{K} \mathbb{1}\{y=k\}\log\frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \end{aligned} \tag{7.18}$$

因此，交叉熵损失只是对应于正确类的输出概率的对数，因此我们也称其为**负对数似然损失(negative log likelihood loss)**:

$$L_{CE}(\hat{y},y) = -\log\hat{y}_i, \quad \text{(where } i \text{ is the correct class)} \tag{7.19}$$

插入公式 7.9 中的 softmax 公式，其中 K 为类别的数量：

$$L_{CE}(\hat{y},y) = -\log \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{(where } i \text{ is the correct class)} \tag{7.20}$$

7.4.2. 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相对于每个参数的偏导数。对于只有一个权重层和 S 形输出的网络(这就是逻辑回归)，我们可以简单地使用等式 7.21(并在 5.8 节中得出)中用于逻辑回归的损失导数：

$$\begin{aligned} \frac{\partial L_{CE}(w,b)}{\partial w_j} &= (\hat{y}-y)x_j \\ &= (\sigma(w\cdot x+b)-y)x_j \end{aligned} \tag{7.21}$$

或者对于一个有一个隐含层和 softmax 输出的网络，我们可以使用公式 5.37 中 softmax 损失的导数:

$$\begin{aligned} \frac{\partial L_{CE}}{\partial w_k} &= (\mathbb{1}\{y=k\}-p(y=k|x))x_k \\ &= \left(\mathbb{1}\{y=k\}-\frac{\exp(w_k\cdot x+b_k)}{\sum_{j=1}^{K}\exp(w_j\cdot x+b_j)}\right)x_k \end{aligned} \tag{7.22}$$

但是，这些导数仅对一个权重层(最后一个权重层)提供正确的更新！ 对于深度网络，计算每个权重的梯度要复杂得多，因为我们计算的是权重参数的导数，这些参数出现在网络较早(very early)的层，即便损失只在网络的较后(very end)端计算。

解决此梯度的方法是一种称为**误差反向传**的算法(Rumelhart 等人，1986)。虽然反向传播是专门为神经网络发明的，但事实证明它与一种称为逆向微分的更通用的过程是一样的，后者取决于**计算图**的概念。

7.4.3. 计算图

计算图是计算数学表达式的过程表示，其中的计算被分解成独立的操作，每个操作被建模为图中的一个节点。

考虑计算函数 L(a，b，c)= c(a + 2b)。如果我们明确每个分量的加法和乘法运算，并为中间输出添加名称(d 和 e)，则所得的一系列计算为：

d = 2*b
e = a+d
L = c*e

现在我们可以用一个图来表示它，用节点表示每个操作，有向边表示每个操作的输出作为下一个操作的输入，如图 7.10 所示。计算图最简单的用法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，我们已经展示了向前传递的结果，计算结果 L(3,1，-2) = -10。在计算图的向前传递过程中，我们从左到右应用每个操作，将每个计算的输出作为下一个节点的输入来传递。

7.4.4. 计算图上的逆向微分

计算图的重要性来自向后传递，它用于计算权重更新所需的导数。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∂L/∂a，∂L/∂b，∂L/∂c。导数∂L/∂a，告诉我们 a 中的一个小变化对 L 的影响有多大。

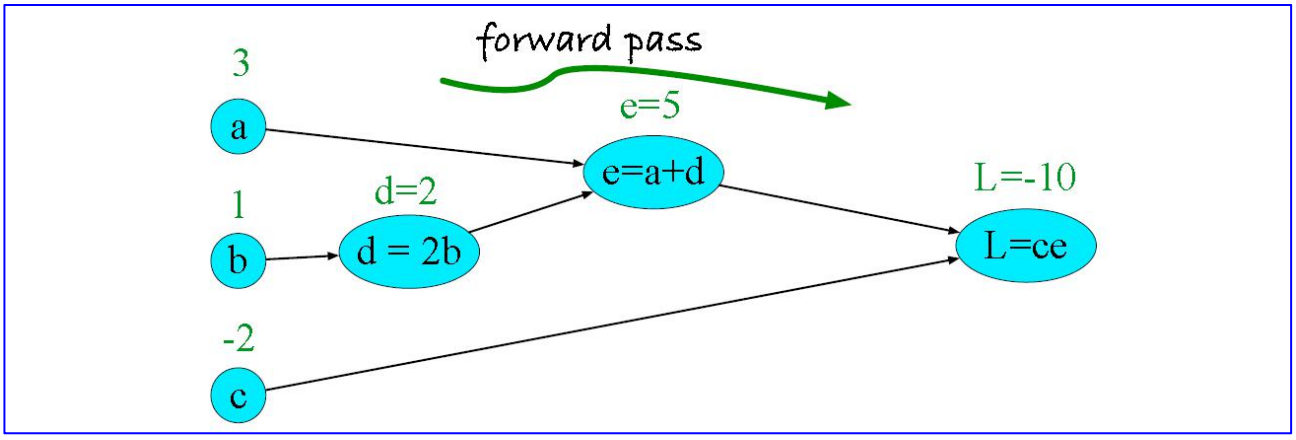

图 7-10：函数 L(a,b, c) = c(a+2b)的正向传递计算图

图注：输入节点 a = 3，b = 1，c = - 2，表示 L 的正向传递计算。

逆向微分在微积分中利用了链式规则。假设我们计算一个复合函数 f(x) = u(v(x))的导数。f(x)的导数等于 u(x)对 v(x)的导数乘以 v(x)对 x 的导数：

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx} \tag{7.23}$$

链式规则扩展到两个以上的函数。如果计算复合函数 f(x) = u(v(w(x)))的导数，则 f(x)的导数为:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx} \tag{7.24}$$

现在我们来计算我们需要的三个导数。因为在计算图 L = ce 中，我们可以直接计算 ∂ L/ ∂ c 的导数:

$$\frac{\partial L}{\partial c} = e \tag{7.25}$$

对于其他两个，我们需要使用链式规则:

$$\begin{aligned} \frac{\partial L}{\partial a} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} \end{aligned} \tag{7.26}$$

公式 7.26 因此需要 5 个中间导数：∂L/∂e、∂L/∂c、∂e/∂a、∂e/∂d、∂d/∂b，它们如下(利用"和的导数是导数的和"这一事实):

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

在反向过程中，我们沿着图的每条边从右到左计算这些偏导数，将必要的偏导数相乘，得到我们需要的最终导数。因此，我们首先用∂L/∂L = 1 来标注最后一个节点。向左移动，然后计算∂L/∂c 和∂L/∂e，以此类推，直到我们把图标注到输入变量为止。正向传递已经很方便地计算出了我们需要的正向中间变量(如 d 和 e)的值来计算这些导数。图 7.11 显示了反向传播。在每个节点上，我们需要计算关于父节点的局部偏

导数，将它乘以从父节点传递下来的偏导数，然后传递给子节点。

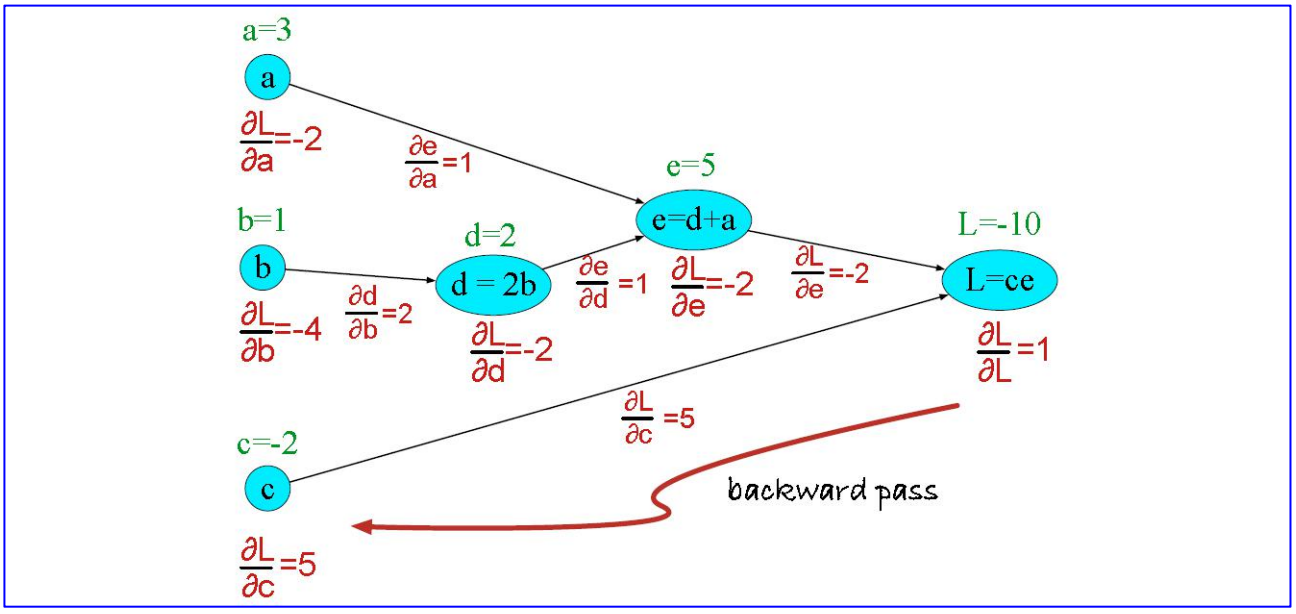

图 7-11：函数 L(a,b, c) = c(a+2b)的反向传递计算图

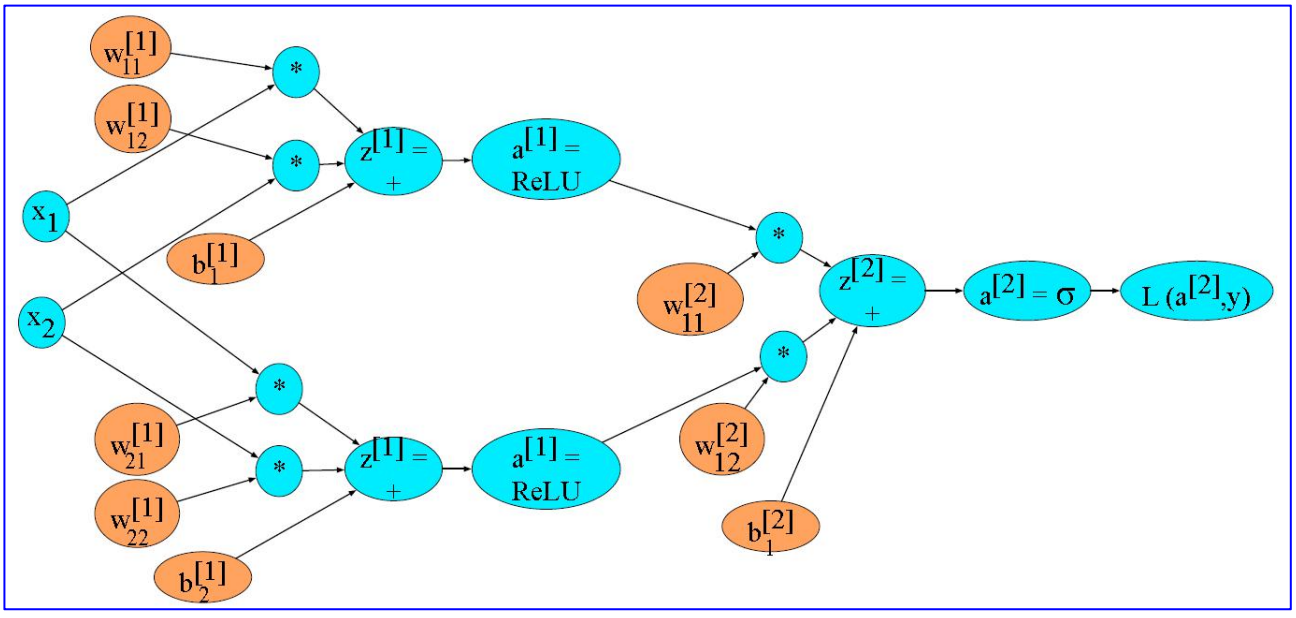

图 7-12：简单 2 层神经网络的计算图

图注：具有两个输入维度和两个隐藏维度的简单 2 层神经网络(1 个隐含层)的计算示例图。

神经网络的逆向微分

当然，真实神经网络的计算图要复杂得多。

图 7.12 显示了一个 $n_0=2, n_1=2, n_2=1$ 的两层神经网络的示例计算图，假设是二分类，因此为了简单起见使用了 sigmoid 输出单元。计算图计算的函数为:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \text{ReLU}(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$
$$\hat{y} = a^{[2]} \tag{7.27}$$

需要更新的权重(我们需要知道损失函数的偏导数)用橙色表示。为了进行反向传递，我们需要知道图

中所有函数的导数。我们已经在 5.8 节看到了 sigmoid σ的导数:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)) \tag{7.28}$$

我们还需要其他激活函数的导数。tanh 的导数为:

$$\frac{d\tanh(z)}{dz} = 1-\tanh^2(z) \tag{7.29}$$

ReLU 的导数是：

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad x<0 \\ 1 & for \quad x \geq 0 \end{cases} \tag{7.30}$$

7.4.5. 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，并且由于这个和其他原因，有许多成功学习的最佳实践。

对于逻辑回归，我们可以初始化梯度下降，使所有的权值和偏差都为 0。而在神经网络中，我们需要用小的随机数初始化权值。将输入值归一化使其均值和单位方差为 0 也很有帮助。

使用各种形式的**正则化**来防止**过拟合**。最重要的一项是**丢弃(dropout)**：在训练期间从网络中随机丢弃一些单元及其连接(Hinton 等人 2012； Srivastava 等人 2014)。**超参数(hyperparameters)**的调整也很重要。神经网络的参数是权重 W 和偏差 b；这些是通过梯度下降来学习的。超参数是算法设计者选择的东西。最佳值是在 devset 上调整的，而不是通过训练集上的梯度下降学习来调整的。超参数包括学习率 η、小批量尺寸、模型体系结构(层数、每层隐藏节点数、激活函数的选择)、如何进行正则化等。梯度下降本身也有许多架构变体，例如 Adam(Kingma 和 Ba，2015)。

最后，大多数现代神经网络都是使用计算图形形式来构建的，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容易和自然。PyTorch (Paszke 等人，2017)和 TensorFlow (Abadi 等人，2015)是其中最受欢迎的两种。感兴趣的读者应该查阅神经网络教科书以了解更多的细节;在本章的最后给出了一些建议。

7.5. 神经语言模型

作为神经网络的第一个应用，让我们考虑语言建模：根据先前的单词上下文预测即将到来的单词。

基于神经网络的语言模型比第三章的 n-gram 语言模型有很多优点。其中之一是神经语言模型不需要平滑，它们可以处理更长的历史，它们可以在类似单词的上下文中进行概括。对于给定大小的训练集，神经语言模型比 n-gram 语言模型具有更高的预测精度。此外，神经语言模型是我们将介绍的用于机器翻译、对话和语言生成等任务的许多模型的基础。

另一方面，这种改进的性能是有代价的:神经网络语言模型训练起来比传统语言模型慢得多，因此对于许多任务来说，n-gram 语言模型仍然是正确的工具。

在本章中，我们将描述简单的前馈神经语言模型，这是本吉奥(Bengio 等人，2003)首先介绍的。现代神经语言模型通常不是前馈的，而是循环的，使用我们将在第 9 章中介绍的技术。

前馈神经 LM 是一种标准前馈网络，它以时间 t 处的一些先前单词(w_{t-1}，w_{t-2} 等)的表示作为输入，并输出可能的下一个单词的概率分布。因此，就像 n-gram 的 LM 一样，前馈神经 LM 通过基于 N 个先前单词进行近似，从而在给定整个现有上下文 $P(w_t \mid w_{1:t-1})$的情况下近似单词的概率：

$$P(w_t|w_1,\ldots,w_{t-1}) \approx P(w_t|w_{t-N+1},\ldots,w_{t-1}) \tag{7.31}$$

在下面的例子中，我们将使用一个 4-gram 的例子，展示一个网络来估计概率 $P(w_t = i|w_{t-1},w_{t-2},w_{t-3})$。

7.5.1. 嵌入

在神经语言模型中，先前上下文是由前一个单词的嵌入来表示的。将先前上下文表示为嵌入，而不是像 n-gram 语言模型中使用的精确词汇，这允许神经语言模型比 n-gram 语言模型更好地泛化到不可见的数据。例如，假设我们在训练中见过这样的句子:

I have to make sure that the cat gets fed.

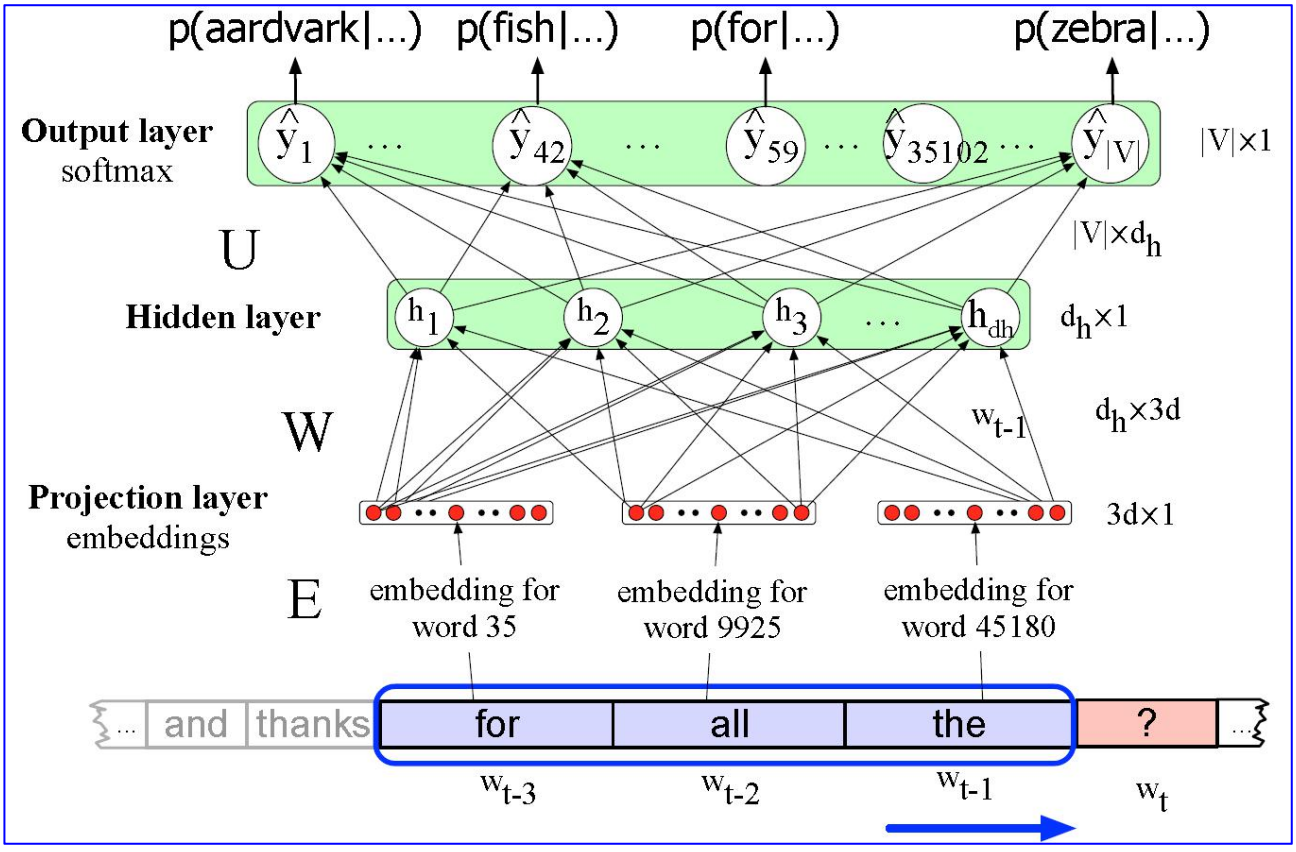

图 7-13：前馈神经语言模型的简化视图

图注：在文本中移动的前馈神经语言模型的简化视图。在每个时间步长 t，网络获取 3 个上下文词，将每个上下文词转换为 d 维嵌入，并将 3 个嵌入拼接在一起，以获得网络的 $1\times N_d$ 单元输入层 x。首先将这些单元乘以权重矩阵 W，其次逐个元素应用激活函数以生成隐藏层 h，然后将其乘以另一个权重矩阵 U。最后，softmax 输出层在每个节点 i 预测以下概率：下一个单词 w_t 将是词汇表 V 中的单词 V_i。(此图片是简化的，因为它假定我们只是在嵌入字典 E 中查找每个单词的 d 维嵌入向量，并通过诸如 word2vec 之类的算法预先计算得出。)

但从未见过“dog”之后的“gets fed”。我们的测试集带有前缀“I forgot to make sure that the dog gets”。下一个词是什么？一个 n-gram 语言模型将在“that the cat gets”之后预测“fed”，而不是在“that the dog gets”之后预测“fed”。但是神经 LM，知道“cat”和“dog”具有相似的嵌入，即使在看到“dog”之后，也能够从“cat”上下文中泛化出足够高的概率给“fed”。

让我们看看这在实践中是如何工作的。现在，假设我们已经有一个嵌入字典 E，它为词汇表 V 中的每个单词提供了嵌入。

图 7.13 是简化的前馈神经语言模型(N = 3)的示意图。我们在时间 t 有一个移动窗口，带有一个嵌入向量，表示三个先前单词(单词 w_{t-1}，w_{t-2} 和 w_{t-3})中的每个单词。这 3 个向量被拼接在一起以产生 x，即神经网络的输入层，其输出是 softmax，其概率分布在单词上。因此，y_{42}(输出节点 42 的值)是下一个单词 w_t 为 V_{42}(词汇表 V 中索引号为 42 的单词)的概率。

图 7.13 所示的模型是足够的，假设我们已经通过类似第 6 章的 word2vec 方法分别学习了嵌入。

如果有一种算法已经学习了输入词的嵌入表示，那么依赖于这种算法的训练就这被称为**预训练(pretraining)**。如果这些预训练的嵌入对您的目的足够了，那么这就是您所需要的。

但是，通常我们想在训练网络的同时学习嵌入。当为网络设计的任务(情感分类，翻译或解析)对构成良好表示的内容施加了严格的约束时，这是正确的。

因此，让我们展示一种允许学习嵌入的架构。为此，我们将在网络上添加一个额外的层，然后将误差一直传播到嵌入向量，首先从具有随机值的嵌入开始，然后逐渐向合理的表示形式发展。

为了使它在输入层起作用，而不是预先训练的嵌入，我们将把 N 个先前的词表示为长度为|V|的独热向量，即，词汇表中每个词的维度为 1。在独热向量中，只有一个等于 1 的元素(在维度上对应于该单词在词汇表中的索引位置)，而所有其他元素均为 0。

假设“toothpaste”是词汇表里的索引 5，$x_5 = 1$，$x_i = 0 \ \forall i \neq 5$，如下图所示:

[0 0 0 0 1 0 0 ... 0 0 0 0]
1 2 3 4 5 6 7 ... ... |V|

图 7.14 显示了 LM 训练中学习嵌入所需的额外层数。在这里，N=3 个上下文单词被表示为 3 个独热向量，通过嵌入矩阵 E 的 3 个实例完全拼接到嵌入层。

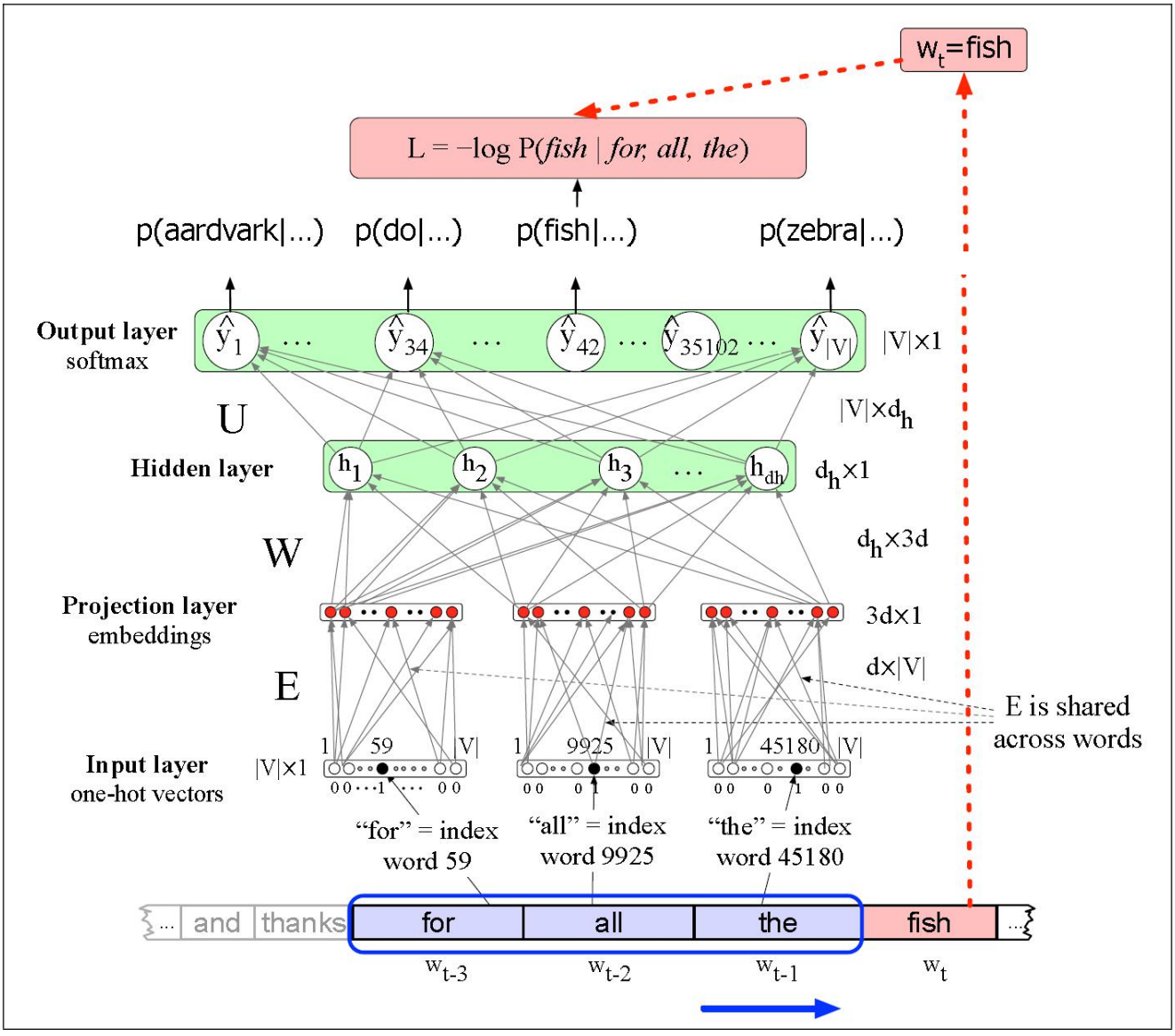

图 7-14：学习嵌入的所有方法

图注：请注意，嵌入矩阵 E 在三个上下文单词之间共享。

请注意，我们不想学习用于将前面的 3 个单词中的每个单词映射到投影层的单独的权重矩阵，我们希望在这三个单词之间共享一个嵌入字典 E。这是因为随着时间的流逝，许多不同的单词将以 w_{t-2} 或 w_{t-1} 的形式出现，而我们只想用一个矢量表示每个单词，无论它出现在哪个上下文位置。因此，嵌入权重矩阵 E 的每个单词都有一列，每个列向量的维数均为 d，因此维数为 d×|V|。

让我们看一下图 7.14 的正向传递。

1.**从 E 中选择三个嵌入**：给定先前三个词，我们查找它们的索引，创建 3 个独热向量，然后将每个独热向量乘以嵌入矩阵 E。考虑 w_{t-3}。将“the”(索引 35)的独热向量乘以嵌入矩阵 E，得到第一隐藏层的第一部分，即**投影层(projection layer)**。由于输入矩阵 E 的每一行只是一个单词的嵌入，而输入是单词 V_i 的单列向量 x_i，因此输入 w 的投影层将为 $Ex_i = e_i$，即单词 i 的嵌入。然后，我们将上下文单词的三个嵌入拼接起来。

2. **乘以 W**:我们现在乘以 W(并添加 b)并通过 ReLU(或其他)激活函数得到隐藏层 h。

3. **乘以 U**:现在 h 乘以 U。

4. **应用 softmax**:在 softmax 之后，输出层的每个节点 i 估算概率 $P(w_t=i|w_{t-1},w_{t-2},w_{t-3})$。

总而言之，如果我们用 e 表示投影层，该投影层是通过将三个上下文向量的三个嵌入拼接起来而形成的，则神经语言模型的等式变为：

$$e = (Ex_1, Ex_2, ..., Ex) \tag{7.32}$$

$$h = \sigma(We + b) \tag{7.33}$$

$$z = Uh \tag{7.34}$$

$$\hat{y} = \text{softmax}(z) \tag{7.35}$$

7.5.2. 训练神经语言模型

为了训练模型，即设置所有参数θ = E,W,U,b，我们做梯度下降(图 5.5)，在计算图上使用误差反向传播来计算梯度。因此，训练不仅设置了网络的权值 W 和 U，而且当我们预测即将到来的单词时，我们学习每个单词的嵌入 E，它能最好地预测即将到来的单词。

一般来说，训练的过程是将一个很长的文本作为输入，拼接所有的句子，从随机权重开始，然后迭代地在文本中移动，预测每个单词的 w_t。对每个单词 w_t，我们使用交叉熵(负对数似然)损失。回想一下这个的一般形式(从公式 7.19 中重复得到)是:

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i, \quad \text{(where } i \text{ is the correct class)} \tag{7.36}$$

对于语言建模，类是词汇表中的单词，所以$\hat{y}_i$ 在这里表示模型赋给正确的下一个单词 w_t 的概率:

$$L_{\text{CE}} = -\log p(w_t|w_{t-1}, ..., w_{t-n+1}) \tag{7.37}$$

将此损失从步骤 s 到 s+1 的随机梯度下降的参数更新为:

$$\theta^{s+1} = \theta^s - \eta \frac{\partial -\log p(w_t|w_{t-1}, ..., w_{t-n+1})}{\partial \theta} \tag{7.38}$$

这个梯度可以在任何标准的神经网络框架中计算，然后通过θ = E,W,U,b 反向传播。

训练参数以使损失最小化将不仅产生了语言建模算法(单词预测器)，而且还产生了一组新的嵌入 E，可用作其他任务的单词表示。

7.6. 总结

•神经网络由神经单元构建而成，最初灵感来自人类**神经单元**，但现在只是一个抽象的计算设备。

•每个神经单元将输入值乘以一个权重向量，添加一个偏差，然后应用一个非线性激活函数，如 sigmoid, tanh，或矫正线性(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，不存在循环。

•神经网络的力量来自于早期层学习表示的能力，这些表示可以被网络的后续层利用。

•神经网络通过**梯度下降**等优化算法进行训练。

•**误差反向传播**，**计算图**上的逆向微分，用于计算网络损失函数的梯度。

•**神经语言模型**使用神经网络作为概率分类器，计算给定前 n 个单词的下一个单词的概率。

•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零开始学习嵌入。

7.7. 文献和历史说明

神经网络起源于 20 世纪 40 年代的 McCulloch-Pitts 神经元(McCulloch 和 Pitts, 1943)，这是人类神经元的一种简化模型，可以用命题逻辑来描述。到 20 世纪 50 年代末和 60 年代初，一些实验室(包括康奈尔大学的弗兰克 • 罗森布拉特和斯坦福大学的伯纳德 • Widrow)发展了神经网络的研究;这一阶段见证了感知机的发展(Rosenblatt, 1958)，以及阈值转变为偏差，一种我们仍在使用的符号(Widrow 和 Hoff, 1960)。

当单个感知器单元无法模拟像 XOR 这样简单的功能时，神经网络领域就衰落了(Minsky 和 Papert, 1969)。虽然在接下来的 20 年里仍有少量工作在进行，但直到 20 世纪 80 年代，当建立更深入的网络(如误差反向传播)的实用工具变得广泛时，该领域才出现了重大复兴(Rumelhart 等人，1986)。在 1980 年代各种神经网络和相关架构开发，特别是应用于心理学和认知科学(Rumelhart 和 McClelland 1986b, McClelland 和 Elman 1986, Rumelhart 和 McClelland 1986a, Elman 1990),**联结主义(connectionist)**一词或并行分布式处理被经常使用(Feldman 和 Ballard 1982, Smolensky 1988)。这一时期发展起来的许多原则和技术是现代工作的基础，包括分布式表示(Hinton, 1986)、循环网络(Elman, 1990)和合成性张量的使用(Smolensky, 1990)。

到 20 世纪 90 年代，更大的神经网络也开始应用于许多实际的语言处理任务，如手写识别(LeCun 等人，1989)和语音识别(Morgan 和 Bourlard，1990)。到 21 世纪初，计算机硬件的改进以及优化和训练技术的进步，使训练更大、更深层次的网络成为可能，从而产生了现代术语“深度学习”(Hinton 等人 2006, Bengio 等人 2007)。我们将在第 9 章和第 26 章讨论更多相关的历史。

关于这个问题有许多优秀的书籍。Goldberg(2017)对自然语言处理的神经网络进行了卓越的研究。对于一般的神经网络，请参阅 Goodfellow 等人(2016)和 Nielsen(2015)。

7.8. 练习

(无)