第六章　深度前馈网络

深度前馈网络（deep feedforward network），也叫作 **前馈神经网络**（feedforward neural network）或者 **多层感知机**（multilayer perceptron, MLP），是典型的深度学习模型。前馈网络的目标是近似某个函数 f^*。例如，对于分类器，$y = f^*(\boldsymbol{x})$ 将输入 $\boldsymbol{x}$ 映射到一个类别 y。前馈网络定义了一个映射 $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$，并且学习参数 $\boldsymbol{\theta}$ 的值，使它能够得到最佳的函数近似。

这种模型被称为 **前向**（feedforward）的，是因为信息流过 $\boldsymbol{x}$ 的函数，流经用于定义 f 的中间计算过程，最终到达输出 $\boldsymbol{y}$。在模型的输出和模型本身之间没有 **反馈**（feedback）连接。当前馈神经网络被扩展成包含反馈连接时，它们被称为 **循环神经网络**（recurrent neural network），在第十章介绍。

前馈网络对于机器学习的从业者是极其重要的。它们是许多重要商业应用的基础。例如，用于对照片中的对象进行识别的卷积神经网络就是一种专门的前馈网络。前馈网络是通往循环网络之路的概念基石，后者在自然语言的许多应用中发挥着巨大作用。

前馈神经网络被称作 **网络**（network）是因为它们通常用许多不同函数复合在一起来表示。该模型与一个有向无环图相关联，而图描述了函数是如何复合在一起的。例如，我们有三个函数 $f^{(1)}, f^{(2)}$ 和 $f^{(3)}$ 连接在一个链上以形成 $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$。这些链式结构是神经网络中最常用的结构。在这种情况下，$f^{(1)}$ 被称为网络的 **第一层**（first layer），$f^{(2)}$ 被称为 **第二层**（second layer），以此类推。链的全长称为模型的 **深度**（depth）。正是因为这个术语才出现了 “深度学习” 这个名字。前馈网络的最后一层被称为 **输出层**（output layer）。在神经网络训练的过程中，我们让 $f(\boldsymbol{x})$ 去匹配 $f^*(\boldsymbol{x})$ 的值。训练数据为我们提供了在不同训练点上取值的、含有噪声的 $f^*(\boldsymbol{x})$ 的近似实例。每个样本 $\boldsymbol{x}$ 都伴随着一个标签 $y \approx f^*(\boldsymbol{x})$。训练样本直接指明了输出层在每一点 $\boldsymbol{x}$ 上必须做什么；它必须产生一个接近 y 的值。

但是训练数据并没有直接指明其他层应该怎么做。学习算法必须决定如何使用这些层来产生想要的输出，但是训练数据并没有说每个单独的层应该做什么。相反，学习算法必须决定如何使用这些层来最好地实现 f^* 的近似。因为训练数据并没有给出这些层中的每一层所需的输出，所以这些层被称为 **隐藏层**（hidden layer）。

最后，这些网络被称为*神经网络*是因为它们或多或少地受到神经科学的启发。网络中的每个隐藏层通常都是向量值的。这些隐藏层的维数决定了模型的 **宽度**（width）。向量的每个元素都可以被视为起到类似一个神经元的作用。除了将层想象成向量到向量的单个函数，我们也可以把层想象成由许多并行操作的 **单元**（unit）组成，每个单元表示一个向量到标量的函数。每个单元在某种意义上类似一个神经元，它接收的输入来源于许多其他的单元，并计算它自己的激活值。使用多层向量值表示的想法来源于神经科学。用于计算这些表示的函数 $f^{(i)}(\boldsymbol{x})$ 的选择，也或多或少地受到神经科学观测的指引，这些观测是关于生物神经元计算功能的。然而，现代的神经网络研究受到更多的是来自许多数学和工程学科的指引，并且神经网络的目标并不是完美地给大脑建模。我们最好将前馈神经网络想成是为了实现统计泛化而设计出的函数近似机，它偶尔从我们了解的大脑中提取灵感，但并不是大脑功能的模型。

一种理解前馈网络的方式是从线性模型开始，并考虑如何克服它的局限性。线性模型，例如逻辑回归和线性回归，是非常吸引人的，因为无论是通过闭解形式还是使用凸优化，它们都能高效且可靠地拟合。线性模型也有明显的缺陷，那就是该模型的能力被局限在线性函数里，所以它无法理解任何两个输入变量间的相互作用。

为了扩展线性模型来表示 $\boldsymbol{x}$ 的非线性函数，我们可以不把线性模型用于 $\boldsymbol{x}$ 本身，而是用在一个变换后的输入 $\phi(\boldsymbol{x})$ 上，这里 ϕ 是一个非线性变换。同样，我们可以使用第 5.7.2 节中描述的核技巧，来得到一个基于隐含地使用 ϕ 映射的非线性学习算法。我们可以认为 ϕ 提供了一组描述 $\boldsymbol{x}$ 的特征，或者认为它提供了 $\boldsymbol{x}$ 的一个新的表示。

剩下的问题就是如何选择映射 ϕ。

1. 其中一种选择是使用一个通用的 ϕ，例如无限维的 ϕ，它隐含地用在基于 RBF 核的核机器上。如果 $\phi(\boldsymbol{x})$ 具有足够高的维数，我们总是有足够的能力来拟合训练集，但是对于测试集的泛化往往不佳。非常通用的特征映射通常只基于局部光滑的原则，并且没有将足够的先验信息进行编码来解决高级问题。

2. 另一种选择是手动地设计 ϕ。在深度学习出现以前，这一直是主流的方法。这

种方法对于每个单独的任务都需要人们数十年的努力，从业者各自擅长特定的领域（如语音识别或计算机视觉），并且不同领域之间很难迁移 (transfer)。

3. 深度学习的策略是去学习 ϕ。在这种方法中，我们有一个模型 $y = f(\boldsymbol{x}; \theta, \boldsymbol{w}) = \phi(\boldsymbol{x}; \theta)^\top \boldsymbol{w}$。我们现在有两种参数：用于从一大类函数中学习 ϕ 的参数 $\boldsymbol{\theta}$，以及用于将 $\phi(\boldsymbol{x})$ 映射到所需的输出的参数 $\boldsymbol{w}$。这是深度前馈网络的一个例子，其中 ϕ 定义了一个隐藏层。这是三种方法中唯一一种放弃了训练问题的凸性的，但是利大于弊。在这种方法中，我们将表示参数化为 $\phi(\boldsymbol{x}; \boldsymbol{\theta})$，并且使用优化算法来寻找 $\boldsymbol{\theta}$，使它能够得到一个好的表示。如果我们想要的话，这种方法也可以通过使它变得高度通用以获得第一种方法的优点——我们只需使用一个非常广泛的函数族 $\phi(\boldsymbol{x}; \boldsymbol{\theta})$。这种方法也可以获得第二种方法的优点。人类专家可以将他们的知识编码进网络来帮助泛化，他们只需要设计那些他们期望能够表现优异的函数族 $\phi(\boldsymbol{x}; \boldsymbol{\theta})$ 即可。这种方法的优点是人类设计者只需要寻找正确的函数族即可，而不需要去寻找精确的函数。

这种通过学习特征来改善模型的一般化原则不仅仅适用于本章描述的前馈神经网络。它是深度学习中反复出现的主题，适用于全书描述的所有种类的模型。前馈神经网络是这个原则的应用，它学习从 $\boldsymbol{x}$ 到 $\boldsymbol{y}$ 的确定性映射并且没有反馈连接。后面出现的其他模型会把这些原则应用到学习随机映射、学习带有反馈的函数以及学习单个向量的概率分布。

本章我们先从前馈网络的一个简单例子说起。接着，我们讨论部署一个前馈网络所需的每个设计决策。首先，训练一个前馈网络至少需要做和线性模型同样多的设计决策：选择一个优化模型、代价函数以及输出单元的形式。我们先回顾这些基于梯度学习的基本知识，然后去面对那些只出现在前馈网络中的设计决策。前馈网络已经引入了隐藏层的概念，这需要我们去选择用于计算隐藏层值的**激活函数**（activation function）。我们还必须设计网络的结构，包括网络应该包含多少层、这些层应该如何连接，以及每一层包含多少单元。在深度神经网络的学习中需要计算复杂函数的梯度。我们给出**反向传播**（back propagation）算法和它的现代推广，它们可以用来高效地计算这些梯度。最后，我们以某些历史观点来结束这一章。

6.1 实例：学习 XOR

为了使前馈网络的想法更加具体，我们首先从一个可以完整工作的前馈网络说起。这个例子解决一个非常简单的任务：学习 XOR 函数。

XOR 函数（“异或” 逻辑）是两个二进制值 x_1 和 x_2 的运算。当这些二进制值中恰好有一个为 1 时，XOR 函数返回值为 1。其余情况下返回值为 0。XOR 函数提供了我们想要学习的目标函数 $y = f^*(\boldsymbol{x})$。我们的模型给出了一个函数 $y = f(\boldsymbol{x}; \boldsymbol{\theta})$ 并且我们的学习算法会不断调整参数 $\boldsymbol{\theta}$ 来使得 f 尽可能接近 f^*。

在这个简单的例子中，我们不会关心统计泛化。我们希望网络在这四个点 $\mathbb{X} = \{[0, 0]^\top, [0, 1]^\top, [1, 0]^\top, [1, 1]^\top\}$ 上表现正确。我们会用全部这四个点来训练我们的网络，唯一的挑战是拟合训练集。

我们可以把这个问题当作是回归问题，并使用均方误差损失函数。我们选择这个损失函数是为了尽可能简化本例中用到的数学。在应用领域，对于二进制数据建模时，MSE通常并不是一个合适的损失函数。更加合适的方法将在第 6.2.2.2 节中讨论。

评估整个训练集上表现的 MSE 损失函数为

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\boldsymbol{x} \in \mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}))^2. \tag{6.1}$$

我们现在必须要选择我们模型 $f(\boldsymbol{x}; \boldsymbol{\theta})$ 的形式。假设我们选择一个线性模型，$\boldsymbol{\theta}$ 包含 $\boldsymbol{w}$ 和 b，那么我们的模型被定义成

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^\top \boldsymbol{w} + b. \tag{6.2}$$

我们可以使用正规方程关于 $\boldsymbol{w}$ 和 b 最小化 $J(\boldsymbol{\theta})$，来得到一个闭式解。

解正规方程以后，我们得到 $\boldsymbol{w} = 0$ 以及 $b = \frac{1}{2}$。线性模型仅仅是在任意一点都输出 0.5。为什么会发生这种事？图 6.1 演示了线性模型为什么不能用来表示 XOR 函数。解决这个问题的其中一种方法是使用一个模型来学习一个不同的特征空间，在这个空间上线性模型能够表示这个解。

具体来说，我们这里引入一个非常简单的前馈神经网络，它有一层隐藏层并且隐藏层中包含两个单元。见图 6.2 中对该模型的解释。这个前馈网络有一个通过函数 $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ 计算得到的隐藏单元的向量 $\boldsymbol{h}$。这些隐藏单元的值随后被用作第二层的输入。第二层就是这个网络的输出层。输出层仍然只是一个线性回归模型，只不过

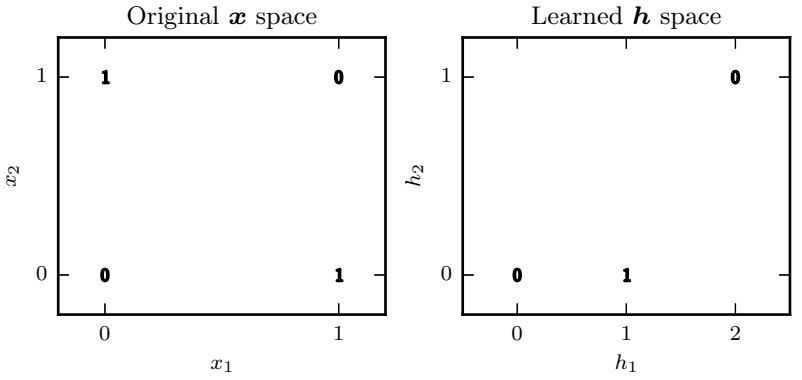

图 6.1: 通过学习一个表示来解决 XOR 问题。图上的粗体数字标明了学得的函数必须在每个点输出的值。*(左)* 直接应用于原始输入的线性模型不能实现 XOR 函数。当 $x_1 = 0$ 时，模型的输出必须随着 x_2 的增大而增大。当 $x_1 = 1$ 时，模型的输出必须随着 x_2 的增大而减小。线性模型必须对 x_2 使用固定的系数 w_2。因此，线性模型不能使用 x_1 的值来改变 x_2 的系数，从而不能解决这个问题。*(右)* 在由神经网络提取的特征表示的变换空间中，线性模型现在可以解决这个问题了。在我们的示例解决方案中，输出必须为 1 的两个点折叠到了特征空间中的单个点。换句话说，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。

现在它作用于 $\boldsymbol{h}$ 而不是 $\boldsymbol{x}$。网络现在包含链接在一起的两个函数：$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ 和 $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$，完整的模型是 $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$。

$f^{(1)}$ 应该是哪种函数？线性模型到目前为止都表现不错，让 $f^{(1)}$ 也是线性的似乎很有诱惑力。可惜的是，如果 $f^{(1)}$ 是线性的，那么前馈网络作为一个整体对于输入仍然是线性的。暂时忽略截距项，假设 $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^\top \boldsymbol{x}$ 并且 $f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^\top \boldsymbol{w}$，那么 $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{W}^\top \boldsymbol{x}$。我们可以将这个函数重新表示成 $f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{w}'$ 其中 $\boldsymbol{w}' = \boldsymbol{W}\boldsymbol{w}$。

显然，我们必须用非线性函数来描述这些特征。大多数神经网络通过仿射变换之后紧跟着一个被称为激活函数的固定非线性函数来实现这个目标，其中仿射变换由学得的参数控制。我们这里使用这种策略，定义 $\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c})$，其中 $\boldsymbol{W}$ 是线性变换的权重矩阵，$\boldsymbol{c}$ 是偏置。此前，为了描述线性回归模型，我们使用权重向量和一个标量的偏置参数来描述从输入向量到输出标量的仿射变换。现在，因为我们描述的是向量 $\boldsymbol{x}$ 到向量 $\boldsymbol{h}$ 的仿射变换，所以我们需要一整个向量的偏置参数。激活函数 g 通常选择对每个元素分别起作用的函数，有 $h_i = g(\boldsymbol{x}^\top \boldsymbol{W}_{:,i} + c_i)$。在现代神经网络

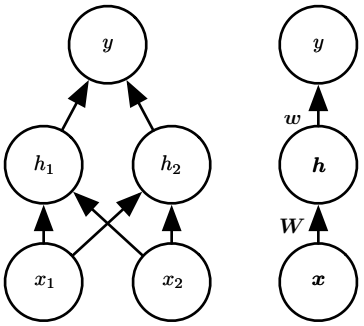

图 6.2: 使用两种不同样式绘制的前馈网络的示例。具体来说，这是我们用来解决 XOR 问题的前馈网络。它有单个隐藏层，包含两个单元。*(左)* 在这种样式中，我们将每个单元绘制为图中的一个节点。这种风格是清楚而明确的，但对于比这个例子更大的网络，它可能会消耗太多的空间。*(右)* 在这种样式中，我们将表示每一层激活的整个向量绘制为图中的一个节点。这种样式更加紧凑。有时，我们对图中的边使用参数名进行注释，这些参数是用来描述两层之间的关系的。这里，我们用矩阵 $\boldsymbol{W}$ 描述从 $\boldsymbol{x}$ 到 $\boldsymbol{h}$ 的映射，用向量 $\boldsymbol{w}$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

中，默认的推荐是使用由激活函数 $g(z) = \max\{0, z\}$ 定义的 **整流线性单元**（rectified linear unit）或者称为 ReLU (Jarrett *et al.*, 2009b; Nair and Hinton, 2010a; Glorot *et al.*, 2011a)，如图 6.3 所示。

我们现在可以指明我们的整个网络是

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b. \tag{6.3}$$

我们现在可以给出 XOR 问题的一个解。令

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \tag{6.4}$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \tag{6.5}$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \tag{6.6}$$

以及 $b = 0$。

我们现在可以了解这个模型如何处理一批输入。令 $\boldsymbol{X}$ 表示设计矩阵，它包含二

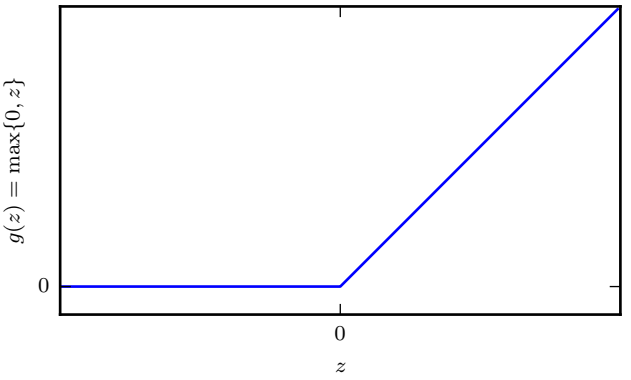

图 6.3: 整流线性激活函数。该激活函数是被推荐用于大多数前馈神经网络的默认激活函数。将此函数用于线性变换的输出将产生非线性变换。然而，函数仍然非常接近线性，在这种意义上它是具有两个线性部分的分段线性函数。由于整流线性单元几乎是线性的，因此它们保留了许多使得线性模型易于使用基于梯度的方法进行优化的属性。它们还保留了许多使得线性模型能够泛化良好的属性。计算机科学的一个通用原则是，我们可以从最小的组件构建复杂的系统。就像图灵机的内存只需要能够存储 0 或 1 的状态，我们可以从整流线性函数构建一个万能函数近似器。

进制输入空间中全部的四个点，每个样本占一行，那么矩阵表示为：

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \tag{6.7}$$

神经网络的第一步是将输入矩阵乘以第一层的权重矩阵：

$$\boldsymbol{X}\boldsymbol{W} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \tag{6.8}$$

然后，我们加上偏置向量 $\boldsymbol{c}$，得到

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.9}$$

在这个空间中，所有的样本都处在一条斜率为 1 的直线上。当我们沿着这条直线移动时，输出需要从 0 升到 1，然后再降回 0。线性模型不能实现这样一种函数。为了用 $\boldsymbol{h}$ 对每个样本求值，我们使用整流线性变换：

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.10}$$

这个变换改变了样本间的关系。它们不再处于同一条直线上了。如图 6.1 所示，它们现在处在一个可以用线性模型解决的空间上。

我们最后乘以一个权重向量 $\boldsymbol{w}$:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{6.11}$$

神经网络对这一批次中的每个样本都给出了正确的结果。

在这个例子中，我们简单地指定了解决方案，然后说明它得到的误差为零。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，所以不能像我们这里做的那样进行简单地猜解。与之相对的，基于梯度的优化算法可以找到一些参数使得产生的误差非常小。我们这里给出的 XOR 问题的解处在损失函数的全局最小点，所以梯度下降算法可以收敛到这一点。梯度下降算法还可以找到 XOR 问题一些其他的等价解。梯度下降算法的收敛点取决于参数的初始值。在实践中，梯度下降通常不会找到像我们这里给出的那种干净的、容易理解的、整数值的解。

6.2 基于梯度的学习

设计和训练神经网络与使用梯度下降训练其他任何机器学习模型并没有太大不同。在第 5.10 节中，我们描述了如何通过指定一个优化过程、代价函数和一个模型族来构建一个机器学习算法。

我们到目前为止看到的线性模型和神经网络的最大区别，在于神经网络的非线性导致大多数我们感兴趣的代价函数都变得非凸。这意味着神经网络的训练通常使

用迭代的、基于梯度的优化，仅仅使得代价函数达到一个非常小的值；而不是像用于训练线性回归模型的线性方程求解器，或者用于训练逻辑回归或 SVM 的凸优化算法那样保证全局收敛。凸优化从任何一种初始参数出发都会收敛（理论上如此——在实践中也很鲁棒但可能会遇到数值问题）。用于非凸损失函数的随机梯度下降没有这种收敛性保证，并且对参数的初始值很敏感。对于前馈神经网络，将所有的权重值初始化为小随机数是很重要的。偏置可以初始化为零或者小的正值。这种用于训练前馈神经网络以及几乎所有深度模型的迭代的基于梯度的优化算法会在第八章详细介绍，参数初始化会在第 8.4 节中具体说明。就目前而言，只需要懂得，训练算法几乎总是基于使用梯度来使得代价函数下降的各种方法即可。一些特别的算法是对梯度下降思想的改进和提纯（在第 4.3 节中介绍）还有一些更特别的，大多数是对随机梯度下降算法的改进（在第 5.9 节中介绍）。

我们当然也可以用梯度下降来训练诸如线性回归和支持向量机之类的模型，并且事实上当训练集相当大时这是很常用的。从这点来看，训练神经网络和训练其他任何模型并没有太大区别。计算梯度对于神经网络会略微复杂一些，但仍然可以很高效而精确地实现。第 6.5 节将会介绍如何用反向传播算法以及它的现代扩展算法来求得梯度。

和其他的机器学习模型一样，为了使用基于梯度的学习方法我们必须选择一个代价函数，并且我们必须选择如何表示模型的输出。现在，我们重温这些设计上的考虑，并且特别强调神经网络的情景。

6.2.1 代价函数

深度神经网络设计中的一个重要方面是代价函数的选择。幸运的是，神经网络的代价函数或多或少是和其他的参数模型例如线性模型的代价函数相同的。

在大多数情况下，我们的参数模型定义了一个分布 $p(\boldsymbol{y} \mid \boldsymbol{x};\boldsymbol{\theta})$ 并且我们简单地使用最大似然原理。这意味着我们使用训练数据和模型预测间的交叉熵作为代价函数。

有时，我们使用一个更简单的方法，不是预测 $\boldsymbol{y}$ 的完整概率分布，而是仅仅预测在给定 $\boldsymbol{x}$ 的条件下 $\boldsymbol{y}$ 的某种统计量。某些专门的损失函数允许我们来训练这些估计量的预测器。

用于训练神经网络的完整的代价函数，通常在我们这里描述的基本代价函数的

基础上结合一个正则项。我们已经在第 5.2.2 节中看到正则化应用到线性模型中的一些简单的例子。用于线性模型的权重衰减方法也直接适用于深度神经网络，而且是最流行的正则化策略之一。用于神经网络的更高级的正则化策略将在第七章中讨论。

6.2.1.1 使用最大似然学习条件分布

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

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}). \tag{6.12}$$

代价函数的具体形式随着模型而改变，取决于 $\log p_{\text{model}}$ 的具体形式。上述方程的展开形式通常会有一些项不依赖于模型的参数，我们可以舍去。例如，正如我们在第 5.1.1 节中看到的，如果 $p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{I})$，那么我们就重新得到了均方误差代价，

$$J(\theta) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})||^2 + \text{const}, \tag{6.13}$$

至少系数 $\frac{1}{2}$ 和常数项不依赖于 $\boldsymbol{\theta}$。舍弃的常数是基于高斯分布的方差，在这种情况下我们选择不把它参数化。之前，我们看到了对输出分布的最大似然估计和对线性模型均方误差的最小化之间的等价性，但事实上，这种等价性并不要求 $f(\boldsymbol{x};\boldsymbol{\theta})$ 用于预测高斯分布的均值。

使用最大似然来导出代价函数的方法的一个优势是，它减轻了为每个模型设计代价函数的负担。明确一个模型 $p(\boldsymbol{y} \mid \boldsymbol{x})$ 则自动地确定了一个代价函数 $\log p(\boldsymbol{y} \mid \boldsymbol{x})$。

贯穿神经网络设计的一个反复出现的主题是代价函数的梯度必须足够的大和具有足够的预测性，来为学习算法提供一个好的指引。饱和（变得非常平）的函数破坏了这一目标，因为它们把梯度变得非常小。这在很多情况下都会发生，因为用于产生隐藏单元或者输出单元的输出的激活函数会饱和。负的对数似然帮助我们在很多模型中避免这个问题。很多输出单元都会包含一个指数函数，这在它的变量取绝对值非常大的负值时会造成饱和。负对数似然代价函数中的对数函数消除了某些输出单元中的指数效果。我们将会在第 6.2.2 节中讨论代价函数和输出单元的选择间的相互作用。

用于实现最大似然估计的交叉熵代价函数有一个不同寻常的特性，那就是当它被应用于实践中经常遇到的模型时，它通常没有最小值。对于离散型输出变量，大

多数模型以一种特殊的形式来参数化，即它们不能表示概率零和一，但是可以无限接近。逻辑回归是其中一个例子。对于实值的输出变量，如果模型可以控制输出分布的密度（例如，通过学习高斯输出分布的方差参数），那么它可能对正确的训练集输出赋予极其高的密度，这将导致交叉熵趋向负无穷。第七章中描述的正则化技术提供了一些不同的方法来修正学习问题，使得模型不会通过这种方式来获得无限制的收益。

6.2.1.2 学习条件统计量

有时我们并不是想学习一个完整的概率分布 $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$，而仅仅是想学习在给定 $\boldsymbol{x}$ 时 $\boldsymbol{y}$ 的某个条件统计量。

例如，我们可能有一个预测器 $f(\boldsymbol{x}; \boldsymbol{\theta})$，我们想用它来预测 $\boldsymbol{y}$ 的均值。如果我们使用一个足够强大的神经网络，我们可以认为这个神经网络能够表示一大类函数中的任何一个函数 f，这个类仅仅被一些特征所限制，例如连续性和有界，而不是具有特殊的参数形式。从这个角度来看，我们可以把代价函数看作是一个**泛函**（functional）而不仅仅是一个函数。泛函是函数到实数的映射。我们因此可以将学习看作是选择一个函数而不仅仅是选择一组参数。我们可以设计代价泛函在我们想要的某些特殊函数处取得最小值。例如，我们可以设计一个代价泛函，使它的最小值处于一个特殊的函数上，这个函数将 $\boldsymbol{x}$ 映射到给定 $\boldsymbol{x}$ 时 $\boldsymbol{y}$ 的期望值。对函数求解优化问题需要用到**变分法**（calculus of variations）这个数学工具，我们将在第 19.4.2 节中讨论。理解变分法对于理解本章的内容不是必要的。目前，只需要知道变分法可以被用来导出下面的两个结果。

我们使用变分法导出的第一个结果是解优化问题

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y} \sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||^2 \tag{6.14}$$

得到

$$f^*(\boldsymbol{x}) = \mathbb{E}_{\mathbf{y} \sim p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x})}[\boldsymbol{y}], \tag{6.15}$$

要求这个函数处在我们要优化的类里。换句话说，如果我们能够用无穷多的、来源于真实的数据生成分布的样本进行训练，最小化均方误差代价函数将得到一个函数，它可以用来对每个 $\boldsymbol{x}$ 的值预测出 $\boldsymbol{y}$ 的均值。

不同的代价函数给出不同的统计量。第二个使用变分法得到的结果是

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||_1 \tag{6.16}$$

将得到一个函数可以对每个 $\boldsymbol{x}$ 预测 $\boldsymbol{y}$ 取值的中位数，只要这个函数在我们要优化的函数族里。这个代价函数通常被称为 **平均绝对误差**（mean absolute error）。

可惜的是，均方误差和平均绝对误差在使用基于梯度的优化方法时往往成效不佳。一些饱和的输出单元当结合这些代价函数时会产生非常小的梯度。这就是为什么交叉熵代价函数比均方误差或者平均绝对误差更受欢迎的原因之一了，即使是在没必要估计整个 $p(\boldsymbol{y} \mid \boldsymbol{x})$ 分布时。

6.2.2 输出单元

代价函数的选择与输出单元的选择紧密相关。大多数时候，我们简单地使用数据分布和模型分布间的交叉熵。选择如何表示输出决定了交叉熵函数的形式。

任何可用作输出的神经网络单元，也可以被用作隐藏单元。这里，我们着重讨论将这些单元用作模型输出时的情况，不过原则上它们也可以在内部使用。我们将在第 6.3 节中重温这些单元，并且给出当它们被用作隐藏单元时一些额外的细节。

在本节中，我们假设前馈网络提供了一组定义为 $\boldsymbol{h} = f(\boldsymbol{x};\boldsymbol{\theta})$ 的隐藏特征。输出层的作用是随后对这些特征进行一些额外的变换来完成整个网络必须完成的任务。

6.2.2.1 用于高斯输出分布的线性单元

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

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

线性输出层经常被用来产生条件高斯分布的均值：

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I}). \tag{6.17}$$

最大化其对数似然此时等价于最小化均方误差。

最大似然框架也使得学习高斯分布的协方差矩阵更加容易，或更容易地使高斯分布的协方差矩阵作为输入的函数。然而，对于所有输入，协方差矩阵都必须被限

定成一个正定矩阵。线性输出层很难满足这种限定，所以通常使用其他的输出单元来对协方差参数化。对协方差建模的方法将在第 6.2.2.4 节中简要介绍。

因为线性模型不会饱和，所以它们易于采用基于梯度的优化算法，甚至可以使用其他多种优化算法。

6.2.2.2 用于 Bernoulli 输出分布的 sigmoid 单元

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

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

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

为满足该约束条件需要一些细致的设计工作。假设我们打算使用线性单元，并且通过阈值来限制它成为一个有效的概率：

$$P(y=1 \mid \boldsymbol{x}) = \max\left\{0, \min\{1, \boldsymbol{w}^\top \boldsymbol{h} + b\}\right\}. \tag{6.18}$$

这的确定义了一个有效的条件概率分布，但我们无法使用梯度下降来高效地训练它。当 $\boldsymbol{w}^\top \boldsymbol{h} + b$ 处于单位区间外时，模型的输出对其参数的梯度都将为 $\mathbf{0}$。梯度为 $\mathbf{0}$ 通常是有问题的，因为学习算法对于如何改善相应的参数不再具有指导意义。

相反，最好是使用一种新的方法来保证无论何时模型给出了错误的答案时，总能有一个较大的梯度。这种方法是基于使用 sigmoid 输出单元结合最大似然来实现的。

sigmoid 输出单元定义为

$$\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right), \tag{6.19}$$

这里 σ 是第 3.10 节中介绍的 logistic sigmoid 函数。

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

我们暂时忽略对于 $\boldsymbol{x}$ 的依赖性，只讨论如何用 z 的值来定义 y 的概率分布。sigmoid 可以通过构造一个非归一化（和不为 1）的概率分布 $\tilde{P}(y)$ 来得到。我们可以随后除以一个合适的常数来得到有效的概率分布。如果我们假定非归一化的对数

概率对 y 和 z 是线性的，可以对它取指数来得到非归一化的概率。我们然后对它归一化，可以发现这服从 Bernoulli 分布，该分布受 z 的 sigmoid 变换控制：

$$\log \tilde{P}(y) = yz, \tag{6.20}$$

$$\tilde{P}(y) = \exp(yz), \tag{6.21}$$

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z)}, \tag{6.22}$$

$$P(y) = \sigma((2y-1)z). \tag{6.23}$$

基于指数和归一化的概率分布在统计建模的文献中很常见。用于定义这种二值型变量分布的变量 z 被称为 **分对数**（logit）。

这种在对数空间里预测概率的方法可以很自然地使用最大似然学习。因为用于最大似然的代价函数是 $-\log P(y \mid \boldsymbol{x})$，代价函数中的 log 抵消了 sigmoid 中的 exp。如果没有这个效果，sigmoid 的饱和性会阻止基于梯度的学习做出好的改进。我们使用最大似然来学习一个由 sigmoid 参数化的 Bernoulli 分布，它的损失函数为

$$J(\boldsymbol{\theta}) = -\log P(y \mid \boldsymbol{x}) \tag{6.24}$$

$$= -\log \sigma((2y-1)z) \tag{6.25}$$

$$= \zeta((1-2y)z). \tag{6.26}$$

这个推导使用了第 3.10 节中的一些性质。通过将损失函数写成 softplus 函数的形式，我们可以看到它仅仅在 $(1-2y)z$ 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，softplus 函数的变量 $(1-2y)z$ 可以简化为 $|z|$。当 $|z|$ 变得很大并且 z 的符号错误时，softplus 函数渐近地趋向于它的变量 $|z|$。对 z 求导则渐近地趋向于 $\text{sign}(z)$，所以，对于极限情况下极度不正确的 z，softplus 函数完全不会收缩梯度。这个性质很有用，因为它意味着基于梯度的学习可以很快地改正错误的 z。

当我们使用其他的损失函数，例如均方误差之类的，损失函数会在 $\sigma(z)$ 饱和时饱和。sigmoid 激活函数在 z 取非常小的负值时会饱和到 0，当 z 取非常大的正值时会饱和到 1。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。因此，最大似然几乎总是训练 sigmoid 输出单元的优选方法。

理论上，sigmoid 的对数总是确定和有限的，因为 sigmoid 的返回值总是被限制在开区间 $(0,1)$ 上，而不是使用整个闭区间 $[0,1]$ 的有效概率。在软件实现时，为了避免数值问题，最好将负的对数似然写作 z 的函数，而不是 $\hat{y} = \sigma(z)$ 的函数。如果 sigmoid 函数下溢到零，那么之后对 $\hat{y}$ 取对数会得到负无穷。

6.2.2.3 用于 Multinoulli 输出分布的 softmax 单元

任何时候当我们想要表示一个具有 n 个可能取值的离散型随机变量的分布时，我们都可以使用 softmax 函数。它可以看作是 sigmoid 函数的扩展，其中 sigmoid 函数用来表示二值型变量的分布。

softmax 函数最常用作分类器的输出，来表示 n 个不同类上的概率分布。比较少见的是，softmax 函数可以在模型内部使用，例如如果我们想要在某个内部变量的 n 个不同选项中进行选择。

在二值型变量的情况下，我们希望计算一个单独的数

$$\hat{y} = P(y = 1 \mid \boldsymbol{x}). \tag{6.27}$$

因为这个数需要处在 0 和 1 之间，并且我们想要让这个数的对数可以很好地用于对数似然的基于梯度的优化，我们选择去预测另外一个数 $z = \log \hat{P}(y = 1 \mid \boldsymbol{x})$。对其指数化和归一化，我们就得到了一个由 sigmoid 函数控制的 Bernoulli 分布。

为了推广到具有 n 个值的离散型变量的情况，我们现在需要创造一个向量 $\hat{\boldsymbol{y}}$，它的每个元素是 $\hat{y}_i = P(y = i \mid \boldsymbol{x})$。我们不仅要求每个 $\hat{y}_i$ 元素介于 0 和 1 之间，还要使得整个向量的和为 1，使得它表示一个有效的概率分布。用于 Bernoulli 分布的方法同样可以推广到 Multinoulli 分布。首先，线性层预测了未归一化的对数概率：

$$\boldsymbol{z} = \boldsymbol{W}^{\top}\boldsymbol{h} + \boldsymbol{b}, \tag{6.28}$$

其中 $z_i = \log \hat{P}(y = i \mid \boldsymbol{x})$。softmax 函数然后可以对 z 指数化和归一化来获得需要的 $\hat{\boldsymbol{y}}$。最终，softmax 函数的形式为

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \tag{6.29}$$

和 logistic sigmoid一样，当使用最大化对数似然训练 softmax 来输出目标值 y 时，使用指数函数工作地非常好。这种情况下，我们想要最大化 $\log P(\text{y} = i; \boldsymbol{z}) =$

$\log \text{softmax}(\boldsymbol{z})_i$。将 softmax 定义成指数的形式是很自然的因为对数似然中的 log 可以抵消 softmax 中的 exp：

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j). \tag{6.30}$$

式 (6.30) 中的第一项表示输入 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 对式 (6.30) 的第二项的贡献很小，学习依然可以进行。当最大化对数似然时，第一项鼓励 z_i 被推高，而第二项则鼓励所有的 $\boldsymbol{z}$ 被压低。为了对第二项 $\log \sum_j \exp(z_j)$ 有一个直观的理解，注意到这一项可以大致近似为 $\max_j z_j$。这种近似是基于对任何明显小于 $\max_j z_j$ 的 z_k，$\exp(z_k)$ 都是不重要的。我们能从这种近似中得到的直觉是，负对数似然代价函数总是强烈地惩罚最活跃的不正确预测。如果正确答案已经具有了 softmax 的最大输入，那么 $-z_i$ 项和 $\log \sum_j \exp(z_j) \approx \max_j z_j = z_i$ 项将大致抵消。这个样本对于整体训练代价贡献很小，这个代价主要由其他未被正确分类的样本产生。

到目前为止我们只讨论了一个例子。总体来说，未正则化的最大似然会驱动模型去学习一些参数，而这些参数会驱动 softmax 函数来预测在训练集中观察到的每个结果的比率：

$$\text{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))_i \approx \frac{\sum_{j=1}^{m} \mathbf{1}_{y^{(j)}=i, \boldsymbol{x}^{(j)}=\boldsymbol{x}}}{\sum_{j=1}^{m} \mathbf{1}_{\boldsymbol{x}^{(j)}=\boldsymbol{x}}}. \tag{6.31}$$

因为最大似然是一致的估计量，所以只要模型族能够表示训练的分布，这就能保证发生。在实践中，有限的模型能力和不完美的优化将意味着模型只能近似这些比率。

除了对数似然之外的许多目标函数对 softmax 函数不起作用。具体来说，那些不使用对数来抵消 softmax 中的指数的目标函数，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。特别是，平方误差对于 softmax 单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，也不能训练模型改变其输出 (Bridle, 1990)。要理解为什么这些损失函数可能失败，我们需要检查 softmax 函数本身。

像 sigmoid 一样，softmax 激活函数可能会饱和。sigmoid 函数具有单个输出，当它的输入极端负或者极端正时会饱和。对于 softmax 的情况，它有多个输出值。当输入值之间的差异变得极端时，这些输出值可能饱和。当 softmax 饱和时，基于 softmax 的许多代价函数也饱和，除非它们能够转化饱和的激活函数。

为了说明 softmax 函数对于输入之间差异的响应，观察到当对所有的输入都加

上一个相同常数时 softmax 的输出不变：

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} + c). \tag{6.32}$$

使用这个性质，我们可以导出一个数值方法稳定的 softmax 函数的变体：

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} - \max_i z_i). \tag{6.33}$$

变换后的形式允许我们在对 softmax 函数求值时只有很小的数值误差，即使是当 z 包含极正或者极负的数时。观察 softmax 数值稳定的变体，可以看到 softmax 函数由它的变量偏离 $\max_i z_i$ 的量来驱动。

当其中一个输入是最大（$z_i = \max_i z_i$）并且 z_i 远大于其他的输入时，相应的输出 $\text{softmax}(\boldsymbol{z})_i$ 会饱和到 1。当 z_i 不是最大值并且最大值非常大时，相应的输出 $\text{softmax}(\boldsymbol{z})_i$ 也会饱和到 0。这是 sigmoid 单元饱和方式的一般化，并且如果损失函数不被设计成对其进行补偿，那么也会造成类似的学习困难。

softmax 函数的变量 $\boldsymbol{z}$ 可以通过两种方式产生。最常见的是简单地使神经网络较早的层输出 $\boldsymbol{z}$ 的每个元素，就像先前描述的使用线性层 $\boldsymbol{z} = W^\top \boldsymbol{h} + \boldsymbol{b}$。虽然很直观，但这种方法是对分布的过度参数化。$n$ 个输出总和必须为 1 的约束意味着只有 $n-1$ 个参数是必要的；第 n 个概率值可以通过 1 减去前面 $n-1$ 个概率来获得。因此，我们可以强制要求 $\boldsymbol{z}$ 的一个元素是固定的。例如，我们可以要求 $z_n = 0$。事实上，这正是 sigmoid 单元所做的。定义 $P(y = 1 \mid \boldsymbol{x}) = \sigma(z)$ 等价于用二维的 $\boldsymbol{z}$ 以及 $z_1 = 0$ 来定义 $P(y = 1 \mid \boldsymbol{x}) = \text{softmax}(\boldsymbol{z})_1$。无论是 $n-1$ 个变量还是 n 个变量的方法，都描述了相同的概率分布，但会产生不同的学习机制。在实践中，无论是过度参数化的版本还是限制的版本都很少有差别，并且实现过度参数化的版本更为简单。

从神经科学的角度看，有趣的是认为 softmax 是一种在参与其中的单元之间形成竞争的方式：softmax 输出总是和为 1，所以一个单元的值增加必然对应着其他单元值的减少。这与被认为存在于皮质中相邻神经元间的侧抑制类似。在极端情况下（当最大的 a_i 和其他的在幅度上差异很大时），它变成了 **赢者通吃**（winner-take-all）的形式（其中一个输出接近 1，其他的接近 0）。

“softmax’’ 的名称可能会让人产生困惑。这个函数更接近于 argmax 函数而不是 max 函数。“soft’’ 这个术语来源于 softmax 函数是连续可微的。“argmax’’ 函数的结果表示为一个one-hot向量（只有一个元素为 1，其余元素都为 0 的向量），不是连续和可微的。softmax 函数因此提供了 argmax 的 ‘‘软化’’ 版本。max 函数相应的软化版本是 $\text{softmax}(\boldsymbol{z})^\top \boldsymbol{z}$。可能最好是把 softmax 函数称为 “softargmax’’，但当前名称

已经是一个根深蒂固的习惯了。

6.2.2.4 其他的输出类型

之前描述的线性、sigmoid 和 softmax 输出单元是最常见的。神经网络可以推广到我们希望的几乎任何种类的输出层。最大似然原则给如何为几乎任何种类的输出层设计一个好的代价函数提供了指导。

一般的，如果我们定义了一个条件分布 $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$，最大似然原则建议我们使用 $-\log p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ 作为代价函数。

一般来说，我们可以认为神经网络表示函数 $f(\boldsymbol{x}; \boldsymbol{\theta})$。这个函数的输出不是对 $\boldsymbol{y}$ 值的直接预测。相反，$f(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{\omega}$ 提供了 y 分布的参数。我们的损失函数就可以表示成 $-\log p(\mathbf{y}; \boldsymbol{\omega}(\boldsymbol{x}))$。

例如，我们想要学习在给定 $\mathbf{x}$ 时，$\mathbf{y}$ 的条件高斯分布的方差。简单情况下，方差 σ^2 是一个常数，此时有一个解析表达式，这是因为方差的最大似然估计量仅仅是观测值 $\mathbf{y}$ 与它们的期望值的差值的平方平均。一种计算上代价更加高但是不需要写特殊情况代码的方法是简单地将方差作为分布 $p(\mathbf{y} \mid \boldsymbol{x})$ 的其中一个属性，这个分布由 $\boldsymbol{\omega} = f(\boldsymbol{x}; \boldsymbol{\theta})$ 控制。负对数似然 $-\log p(\boldsymbol{y}; \boldsymbol{\omega}(\boldsymbol{x}))$ 将为代价函数提供一个必要的合适项来使我们的优化过程可以逐渐地学到方差。在标准差不依赖于输入的简单情况下，我们可以在网络中创建一个直接复制到 $\boldsymbol{\omega}$ 中的新参数。这个新参数可以是 σ 本身，或者可以是表示 σ^2 的参数 v，或者可以是表示 $\frac{1}{\sigma^2}$ 的参数 β，取决于我们怎样对分布参数化。我们可能希望模型对不同的 $\mathbf{x}$ 值预测出 $\mathbf{y}$ 不同的方差。这被称为 **异方差**（heteroscedastic）模型。在异方差情况下，我们简单地把方差指定为 $f(\mathbf{x}; \boldsymbol{\theta})$ 其中一个输出值。实现它的典型方法是使用精度而不是方差来表示高斯分布，就像式 (3.22) 所描述的。在多维变量的情况下，最常见的是使用一个对角精度矩阵

$$\operatorname{diag}(\boldsymbol{\beta}). \tag{6.34}$$

这个公式适用于梯度下降，因为由 $\boldsymbol{\beta}$ 参数化的高斯分布的对数似然的公式仅涉及 β_i 的乘法和 $\log \beta_i$ 的加法。乘法、加法和对数运算的梯度表现良好。相比之下，如果我们用方差来参数化输出，我们需要用到除法。除法函数在零附近会变得任意陡峭。虽然大梯度可以帮助学习，但任意大的梯度通常导致不稳定。如果我们用标准差来参数化输出，对数似然仍然会涉及除法，并且还将涉及平方。通过平方运算的梯度可能在零附近消失，这使得学习被平方的参数变得困难。无论我们使用的是标准差，

方差还是精度，我们必须确保高斯分布的协方差矩阵是正定的。因为精度矩阵的特征值是协方差矩阵特征值的倒数，所以这等价于确保精度矩阵是正定的。如果我们使用对角矩阵，或者是一个常数乘以单位矩阵[1]，那么我们需要对模型输出强加的唯一条件是它的元素都为正。如果我们假设 $\boldsymbol{a}$ 是用于确定对角精度的模型的原始激活，那么可以用 softplus 函数来获得正的精度向量：$\boldsymbol{\beta} = \boldsymbol{\zeta}(\boldsymbol{a})$。这种相同的策略对于方差或标准差同样适用，也适用于常数乘以单位阵的情况。

学习一个比对角矩阵具有更丰富结构的协方差或者精度矩阵是很少见的。如果协方差矩阵是满的和有条件的，那么参数化的选择就必须要保证预测的协方差矩阵是正定的。这可以通过写成 $\boldsymbol{\Sigma}(\boldsymbol{x}) = \boldsymbol{B}(\boldsymbol{x})\boldsymbol{B}^\top(\boldsymbol{x})$ 来实现，这里 $\boldsymbol{B}$ 是一个无约束的方阵。如果矩阵是满秩的，那么一个实际问题是计算似然的代价是很高的，计算一个 $d \times d$ 的矩阵的行列式或者 $\boldsymbol{\Sigma}(\boldsymbol{x})$ 的逆（或者等价地并且更常用地，对它特征值分解或者 $\boldsymbol{B}(\boldsymbol{x})$ 的特征值分解）需要 $O(d^3)$ 的计算量。

我们经常想要执行多峰回归 (multimodal regression)，即预测条件分布 $p(\boldsymbol{y} \mid \boldsymbol{x})$ 的实值，该条件分布对于相同的 $\boldsymbol{x}$ 值在 $\boldsymbol{y}$ 空间中有多个不同的峰值。在这种情况下，高斯混合是输出的自然表示 (Jacobs *et al.*, 1991; Bishop, 1994)。将高斯混合作为其输出的神经网络通常被称为 **混合密度网络**（mixture density network）。具有 n 个分量的高斯混合输出由下面的条件分布定义：

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \sum_{i=1}^{n} p(\mathrm{c} = i \mid \boldsymbol{x}) \mathcal{N}(\boldsymbol{y}; \boldsymbol{\mu}^{(i)}(\boldsymbol{x}), \boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})). \tag{6.35}$$

神经网络必须有三个输出：定义 $p(\mathrm{c} = i \mid \boldsymbol{x})$ 的向量，对所有的 i 给出 $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$ 的矩阵，以及对所有的 i 给出 $\boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})$ 的张量。这些输出必须满足不同的约束：

1. 混合组件 $p(\mathrm{c} = i \mid \boldsymbol{x})$：它们由潜变量[2] c 关联着，在 n 个不同组件上形成 Multinoulli 分布。这个分布通常可以由 n 维向量的 softmax 来获得，以确保这些输出是正的并且和为 1。

2. 均值 $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$：它们指明了与第 i 个高斯组件相关联的中心或者均值，并且是无约束的（通常对于这些输出单元完全没有非线性）。如果 $\mathbf{y}$ 是个 d 维向量，那

[1]译者注：这里原文是 “If we use a diagonal matrix, or a scalar times the diagonal matrix...” 即 “如果我们使用对角矩阵，或者是一个标量乘以对角矩阵...”，但一个标量乘以对角矩阵和对角矩阵没区别，结合上下文可以看出，这里原作者误把 “identity” 写成了 “diagonal matrix”，因此这里采用 “常数乘以单位矩阵” 的译法。

[2]我们之所以认为 c 是潜在的，是因为我们不能直接在数据中观测到它：给定输入 $\mathbf{x}$ 和目标 $\mathbf{y}$，不可能确切地知道是哪个高斯组件产生 $\mathbf{y}$，但我们可以想象 $\mathbf{y}$ 是通过选择其中一个来产生的，并且将那个未被观测到的选择作为随机变量。

么网络必须输出一个由 n 个这种 d 维向量组成的 $n \times d$ 的矩阵。用最大似然来学习这些均值要比学习只有一个输出模式的分布的均值稍稍复杂一些。我们只想更新那个真正产生观测数据的组件的均值。在实践中，我们并不知道是哪个组件产生了观测数据。负对数似然表达式将每个样本对每个组件的贡献进行赋权，权重的大小由相应的组件产生这个样本的概率来决定。

3. 协方差 $\boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})$：它们指明了每个组件 i 的协方差矩阵。和学习单个高斯组件时一样，我们通常使用对角矩阵来避免计算行列式。和学习混合均值时一样，最大似然是很复杂的，它需要将每个点的部分责任分配给每个混合组件。如果给定了混合模型的正确的负对数似然，梯度下降将自动地遵循正确的过程。

有报告说基于梯度的优化方法对于混合条件高斯（作为神经网络的输出）可能是不可靠的，部分是因为涉及到除法（除以方差）可能是数值不稳定的（当某个方差对于特定的实例变得非常小时，会导致非常大的梯度）。一种解决方法是**梯度截断**（clip gradient）（见第 10.11.1 节），另外一种是启发式缩放梯度 (Murray and Larochelle, 2014)。

高斯混合输出在语音生成模型 (Schuster, 1999) 和物理运动 (Graves, 2013) 中特别有效。混合密度策略为网络提供了一种方法来表示多种输出模式，并且控制输出的方差，这对于在这些实数域中获得高质量的结果是至关重要的。混合密度网络的一个实例如图 6.4 所示。

一般的，我们可能希望继续对包含更多变量的、更大的向量 $\boldsymbol{y}$ 来建模，并在这些输出变量上施加更多更丰富的结构。例如，我们可能希望神经网络输出字符序列形成一个句子。在这些情况下，我们可以继续使用最大似然原理应用到我们的模型 $p(\boldsymbol{y};\boldsymbol{\omega}(\boldsymbol{x}))$ 上，但我们用来描述 $\boldsymbol{y}$ 的模型会变得非常复杂，超出了本章的范畴。第十章描述了如何使用循环神经网络来定义这种序列上的模型，第三部分描述了对任意概率分布进行建模的高级技术。

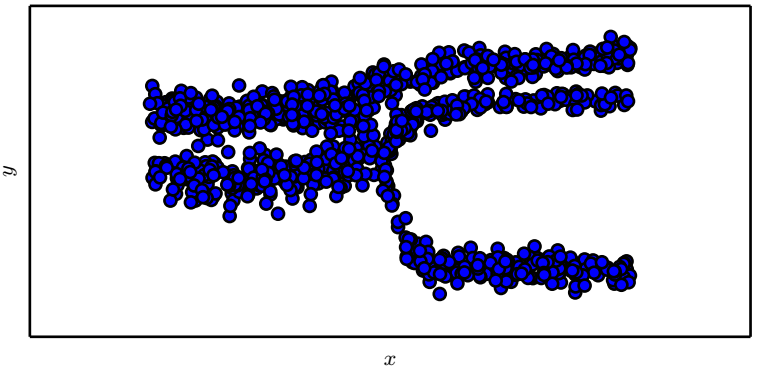

图 6.4: 从具有混合密度输出层的神经网络中抽取的样本。输入 x 从均匀分布中采样，输出 y 从 $p_{\text{model}}(y \mid x)$ 中采样。神经网络能够学习从输入到输出分布的参数的非线性映射。这些参数包括控制三个组件中的哪一个将产生输出的概率，以及每个组件各自的参数。每个混合组件都是高斯分布，具有预测的均值和方差。输出分布的这些方面都能够相对输入 x 变化，并且以非线性的方式改变。

6.3 隐藏单元

到目前为止，我们集中讨论了神经网络的设计选择，这对于使用基于梯度的优化方法来训练的大多数参数化机器学习模型都是通用的。现在我们转向一个前馈神经网络独有的问题：该如何选择隐藏单元的类型，这些隐藏单元用在模型的隐藏层中。

隐藏单元的设计是一个非常活跃的研究领域，并且还没有许多明确的指导性理论原则。

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

这里列出的一些隐藏单元可能并不是在所有的输入点上都是可微的。例如，整流线性单元 $g(z) = \max\{0, z\}$ 在 $z = 0$ 处不可微。这似乎使得 g 对于基于梯度的学习算法无效。在实践中，梯度下降对这些机器学习模型仍然表现得足够好。部分原因

是神经网络训练算法通常不会达到代价函数的局部最小值，而是仅仅显著地减小它的值，如图 4.3 所示。这些想法会在第八章中进一步描述。因为我们不再期望训练能够实际到达梯度为 $\mathbf{0}$ 的点，所以代价函数的最小值对应于梯度未定义的点是可以接受的。不可微的隐藏单元通常只在少数点上不可微。一般来说，函数 $g(z)$ 具有左导数和右导数，左导数定义为紧邻在 z 左边的函数的斜率，右导数定义为紧邻在 z 右边的函数的斜率。只有当函数在 z 处的左导数和右导数都有定义并且相等时，函数在 z 点处才是可微的。神经网络中用到的函数通常对左导数和右导数都有定义。在 $g(z) = \max\{0, z\}$ 的情况下，在 $z = 0$ 处的左导数是 0，右导数是 1。神经网络训练的软件实现通常返回左导数或右导数的其中一个，而不是报告导数未定义或产生一个错误。这可以通过观察到在数字计算机上基于梯度的优化总是会受到数值误差的影响来启发式地给出理由。当一个函数被要求计算 $g(0)$ 时，底层值真正为 0 是不太可能的。相对的，它可能是被舍入为 0 的一个小量 ϵ。在某些情况下，理论上有更好的理由，但这些通常对神经网络训练并不适用。重要的是，在实践中，我们可以放心地忽略下面描述的隐藏单元激活函数的不可微性。

除非另有说明，大多数的隐藏单元都可以描述为接受输入向量 $\boldsymbol{x}$，计算仿射变换 $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$，然后使用一个逐元素的非线性函数 $g(\boldsymbol{z})$。大多数隐藏单元的区别仅仅在于激活函数 $g(\boldsymbol{z})$ 的形式。

6.3.1 整流线性单元及其扩展

整流线性单元使用激活函数 $g(z) = \max\{0, z\}$。

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

整流线性单元通常作用于仿射变换之上：

$$\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}). \tag{6.36}$$

当初始化仿射变换的参数时，可以将 $\boldsymbol{b}$ 的所有元素设置成一个小的正值，例如 0.1。这使得整流线性单元很可能初始时就对训练集中的大多数输入呈现激活状态，并且

允许导数通过。

有很多整流线性单元的扩展存在。大多数这些扩展的表现比得上整流线性单元，并且偶尔表现得更好。

整流线性单元的一个缺陷是它们不能通过基于梯度的方法学习那些使它们激活为零的样本。整流线性单元的各种扩展保证了它们能在各个位置都接收到梯度。

整流线性单元的三个扩展基于当 $z_i < 0$ 时使用一个非零的斜率 α_i：$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$。**绝对值整流**（absolute value rectification）固定 $\alpha_i = -1$ 来得到 $g(z) = |z|$。它用于图像中的对象识别 (Jarrett *et al.*, 2009a)，其中寻找在输入照明极性反转下不变的特征是有意义的。整流线性单元的其他扩展比这应用地更广泛。**渗漏整流线性单元**（Leaky ReLU）(Maas *et al.*, 2013) 将 α_i 固定成一个类似 0.01 的小值，**参数化整流线性单元**（parametric ReLU）或者 **PReLU** 将 α_i 作为学习的参数 (He *et al.*, 2015)。

maxout 单元（maxout unit）(Goodfellow *et al.*, 2013a) 进一步扩展了整流线性单元。maxout 单元将 $\boldsymbol{z}$ 划分为每组具有 k 个值的组，而不是使用作用于每个元素的函数 $g(z)$。每个maxout 单元则输出每组中的最大元素：

$$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j \tag{6.37}$$

这里 $\mathbb{G}^{(i)}$ 是组 i 的输入索引集 $\{(i-1)k+1, \ldots, ik\}$。这提供了一种方法来学习对输入 $\boldsymbol{x}$ 空间中多个方向响应的分段线性函数。

maxout 单元可以学习具有多达 k 段的分段线性的凸函数。maxout 单元因此可以视为学习激活函数本身而不仅仅是单元之间的关系。使用足够大的 k，maxout 单元可以以任意的精确度来近似任何凸函数。特别地，具有两块的 maxout 层可以学习实现和传统层相同的输入 $\boldsymbol{x}$ 的函数，这些传统层可以使用整流线性激活函数、绝对值整流、渗漏整流线性单元 或参数化整流线性单元，或者可以学习实现与这些都不同的函数。maxout 层的参数化当然也将与这些层不同，所以即使是 maxout 学习去实现和其他种类的层相同的 $\boldsymbol{x}$ 的函数这种情况下，学习的机理也是不一样的。

每个 maxout 单元现在由 k 个权重向量来参数化，而不仅仅是一个，所以 maxout 单元通常比整流线性单元需要更多的正则化。如果训练集很大并且每个单元的块数保持很低的话，它们可以在没有正则化的情况下工作得不错 (Cai *et al.*, 2013)。

maxout 单元还有一些其他的优点。在某些情况下，要求更少的参数可以获得一些统计和计算上的优点。具体来说，如果由 n 个不同的线性过滤器描述的特征可以

在不损失信息的情况下，用每一组 k 个特征的最大值来概括的话，那么下一层可以获得 k 倍更少的权重数。

因为每个单元由多个过滤器驱动，maxout 单元具有一些冗余来帮助它们抵抗一种被称为 **灾难遗忘**（catastrophic forgetting）的现象，这个现象是说神经网络忘记了如何执行它们过去训练的任务 (Goodfellow *et al.*, 2014a)。

整流线性单元和它们的这些扩展都是基于一个原则，那就是如果它们的行为更接近线性，那么模型更容易优化。使用线性行为更容易优化的一般性原则同样也适用于除深度线性网络以外的情景。循环网络可以从序列中学习并产生状态和输出的序列。当训练它们时，需要通过一些时间步来传播信息，当其中包含一些线性计算（具有大小接近 1 的某些方向导数）时，这会更容易。作为性能最好的循环网络结构之一，LSTM 通过求和在时间上传播信息，这是一种特别直观的线性激活。它将在第 10.10 节中进一步讨论。

6.3.2 logistic sigmoid与双曲正切函数

在引入整流线性单元之前，大多数神经网络使用 logistic sigmoid 激活函数

$$g(z) = \sigma(z) \tag{6.38}$$

或者是双曲正切激活函数

$$g(z) = \tanh(z). \tag{6.39}$$

这些激活函数紧密相关，因为 $\tanh(z) = 2\sigma(2z) - 1$。

我们已经看过 sigmoid 单元作为输出单元用来预测二值型变量取值为 1 的概率。与分段线性单元不同，sigmoid 单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，它们饱和到一个高值，当 z 取绝对值很大的负值时，它们饱和到一个低值，并且仅仅当 z 接近 0 时它们才对输入强烈敏感。sigmoid 单元的广泛饱和性会使得基于梯度的学习变得非常困难。因为这个原因，现在不鼓励将它们用作前馈网络中的隐藏单元。当使用一个合适的代价函数来抵消 sigmoid 的饱和性时，它们作为输出单元可以与基于梯度的学习相兼容。

当必须要使用 sigmoid 激活函数时，双曲正切激活函数通常要比 logistic sigmoid 函数表现更好。在 $\tanh(0) = 0$ 而 $\sigma(0) = \frac{1}{2}$ 的意义上，它更像是单位函数。因为 tanh 在 0 附近与单位函数类似，训练深层神经网络 $\hat{y} = \boldsymbol{w}^\top \tanh(\boldsymbol{U}^\top \tanh(\boldsymbol{V}^\top \boldsymbol{x}))$

类似于训练一个线性模型 $\hat{y} = \boldsymbol{w}^\top \boldsymbol{U}^\top \boldsymbol{V}^\top \boldsymbol{x}$，只要网络的激活能够被保持地很小。这使得训练 tanh 网络更加容易。

sigmoid 激活函数在除了前馈网络以外的情景中更为常见。循环网络、许多概率模型以及一些自编码器有一些额外的要求使得它们不能使用分段线性激活函数，并且使得 sigmoid 单元更具有吸引力，尽管它存在饱和性的问题。

6.3.3 其他隐藏单元

也存在许多其他种类的隐藏单元，但它们并不常用。

一般来说，很多种类的可微函数都表现得很好。许多未发布的激活函数与流行的激活函数表现得一样好。为了提供一个具体的例子，作者在 MNIST 数据集上使用 $\boldsymbol{h} = \cos(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ 测试了一个前馈网络，并获得了小于 1% 的误差率，这可以与更为传统的激活函数获得的结果相媲美。在新技术的研究和开发期间，通常会测试许多不同的激活函数，并且会发现许多标准方法的变体表现非常好。这意味着，通常新的隐藏单元类型只有在被明确证明能够提供显著改进时才会被发布。新的隐藏单元类型如果与已有的隐藏单元表现大致相当的话，那么它们是非常常见的，不会引起别人的兴趣。

列出文献中出现的所有隐藏单元类型是不切实际的。我们只对一些特别有用和独特的类型进行强调。

其中一种是完全没有激活函数 $g(z)$。也可以认为这是使用单位函数作为激活函数的情况。我们已经看过线性单元可以用作神经网络的输出。它也可以用作隐藏单元。如果神经网络的每一层都仅由线性变换组成，那么网络作为一个整体也将是线性的。然而，神经网络的一些层是纯线性也是可以接受的。考虑具有 n 个输入和 p 个输出的神经网络层 $\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b})$。我们可以用两层来代替它，一层使用权重矩阵 $\boldsymbol{U}$，另一层使用权重矩阵 $\boldsymbol{V}$。如果第一层没有激活函数，那么我们对基于 $\boldsymbol{W}$ 的原始层的权重矩阵进行因式分解。分解方法是计算 $\boldsymbol{h} = g(\boldsymbol{V}^\top \boldsymbol{U}^\top \boldsymbol{x} + \boldsymbol{b})$。如果 U 产生了 q 个输出，那么 $\boldsymbol{U}$ 和 V 一起仅包含 $(n+p)q$ 个参数，而 $\boldsymbol{W}$ 包含 np 个参数。如果 q 很小，这可以在很大程度上节省参数。这是以将线性变换约束为低秩的代价来实现的，但这些低秩关系往往是足够的。线性隐藏单元因此提供了一种减少网络中参数数量的有效方法。

softmax 单元是另外一种经常用作输出的单元（如第 6.2.2.3 节中所描述的），但

有时也可以用作隐藏单元。softmax 单元很自然地表示具有 k 个可能值的离散型随机变量的概率分布，所以它们可以用作一种开关。这些类型的隐藏单元通常仅用于明确地学习操作内存的高级结构中，将在第 10.12 节中描述。

其他一些常见的隐藏单元类型包括：

- **径向基函数**（radial basis function, RBF）：$h_i = \exp\left(-\frac{1}{\sigma_i^2}||\boldsymbol{W}_{:,i} - \boldsymbol{x}||^2\right)$。这个函数在 $\boldsymbol{x}$ 接近模板 $\boldsymbol{W}_{:,i}$ 时更加活跃。因为它对大部分 $\boldsymbol{x}$ 都饱和到 0，因此很难优化。

- **softplus**函数：$g(a) = \zeta(a) = \log(1 + e^a)$。这是整流线性单元的平滑版本，由 Dugas *et al.* (2001) 引入用于函数近似，由 Nair and Hinton (2010a) 引入用于无向概率模型的条件分布。Glorot *et al.* (2011a) 比较了 softplus 和整流线性单元，发现后者的结果更好。通常不鼓励使用 softplus 函数。softplus 表明隐藏单元类型的性能可能是非常反直觉的——因为它处处可导或者因为它不完全饱和，人们可能希望它具有优于整流线性单元的点，但根据经验来看，它并没有。

- **硬双曲正切函数**（hard tanh）：它的形状和 tanh 以及整流线性单元类似，但是不同于后者，它是有界的，$g(a) = \max(-1, \min(1, a))$。它由 Collobert (2004) 引入。

隐藏单元的设计仍然是一个活跃的研究领域，许多有用的隐藏单元类型仍有待发现。

6.4 架构设计

神经网络设计的另一个关键点是确定它的架构。**架构**（architecture）一词是指网络的整体结构：它应该具有多少单元，以及这些单元应该如何连接。

大多数神经网络被组织成称为层的单元组。大多数神经网络架构将这些层布置成链式结构，其中每一层都是前一层的函数。在这种结构中，第一层由下式给出：

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right); \tag{6.40}$$

第二层由

$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right); \tag{6.41}$$

给出，以此类推。

在这些链式架构中，主要的架构考虑是选择网络的深度和每一层的宽度。我们将会看到，即使只有一个隐藏层的网络也足够适应训练集。更深层的网络通常能够对每一层使用更少的单元数和更少的参数，并且经常容易泛化到测试集，但是通常也更难以优化。对于一个具体的任务，理想的网络架构必须通过实验，观测在验证集上的误差来找到。

6.4.1 万能近似性质和深度

线性模型，通过矩阵乘法将特征映射到输出，顾名思义，仅能表示线性函数。它具有易于训练的优点，因为当使用线性模型时，许多损失函数会导出凸优化问题。可惜的是，我们经常希望我们的系统学习非线性函数。

乍一看，我们可能认为学习非线性函数需要为我们想要学习的那种非线性专门设计一类模型族。幸运的是，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）(Hornik *et al.*, 1989; Cybenko, 1989) 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任何一种 "挤压" 性质的激活函数（例如logistic sigmoid激活函数）的隐藏层，只要给予网络足够数量的隐藏单元，它可以以任意的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。前馈网络的导数也可以任意好地来近似函数的导数 (Hornik *et al.*, 1990)。Borel 可测的概念超出了本书的范畴；对于我们想要实现的目标，只需要知道定义在 $\mathbb{R}^n$ 的有界闭集上的任意连续函数是 Borel 可测的，因此可以用神经网络来近似。神经网络也可以近似从任何有限维离散空间映射到另一个的任意函数。虽然原始定理最初以具有特殊激活函数的单元的形式来描述，这个激活函数当变量取绝对值非常大的正值和负值时都会饱和，万能近似定理也已经被证明对于更广泛类别的激活函数也是适用的，其中就包括现在常用的整流线性单元 (Leshno *et al.*, 1993)。

万能近似定理意味着无论我们试图学习什么函数，我们知道一个大的 MLP 一定能够表示这个函数。然而，我们不能保证训练算法能够学得这个函数。即使 MLP 能够表示该函数，学习也可能因两个不同的原因而失败。首先，用于训练的优化算法可能找不到用于期望函数的参数值。其次，训练算法可能由于过拟合而选择了错误的函数。回忆第 5.2.1 节中的 "没有免费的午餐" 定理，说明了没有普遍优越的机器学习算法。前馈网络提供了表示函数的万能系统，在这种意义上，给定一个函数，

存在一个前馈网络能够近似该函数。不存在万能的过程既能够验证训练集上的特殊样本，又能够选择一个函数来扩展到训练集上没有的点。

万能近似定理说明了，存在一个足够大的网络能够达到我们所希望的任意精度，但是定理并没有说这个网络有多大。Barron (1993) 提供了单层网络近似一大类函数所需大小的一些界。不幸的是，在最坏情况下，可能需要指数数量的隐藏单元（可能一个隐藏单元对应着一个需要区分的输入配置）。这在二进制值的情况下很容易看到：向量 $\boldsymbol{v} \in \{0,1\}^n$ 上的可能的二值型函数的数量是 2^{2^n}，并且选择一个这样的函数需要 2^n 位，这通常需要 $O(2^n)$ 的自由度。

总之，具有单层的前馈网络足以表示任何函数，但是网络层可能大得不可实现，并且可能无法正确地学习和泛化。在很多情况下，使用更深的模型能够减少表示期望函数所需的单元的数量，并且可以减少泛化误差。

存在一些函数族能够在网络的深度大于某个值 d 时被高效地近似，而当深度被限制到小于或等于 d 时需要一个远远大于之前的模型。在很多情况下，浅层模型所需的隐藏单元的数量是 n 的指数级。这个结果最初被证明是在那些不与连续可微的神经网络类似的机器学习模型中出现，但现在已经扩展到了这些模型。第一个结果是关于逻辑门电路的 (Håstad, 1986)。后来的工作将这些结果扩展到了具有非负权重的线性阈值单元 (Håstad and Goldmann, 1991; Hajnal *et al.*, 1993)，然后扩展到了具有连续值激活的网络 (Maass, 1992; Maass *et al.*, 1994)。许多现代神经网络使用整流线性单元。Leshno *et al.* (1993) 证明带有一大类非多项式激活函数族的浅层网络，包括整流线性单元，具有万能的近似性质，但是这些结果并没有强调深度或效率的问题——它们仅指出足够宽的整流网络能够表示任意函数。Montufar *et al.* (2014) 指出一些用深度整流网络表示的函数可能需要浅层网络（一个隐藏层）指数级的隐藏单元才能表示。更确切的说，他们说明分段线性网络（可以通过整流非线性或 maxout 单元获得）可以表示区域的数量是网络深度的指数级的函数。图 6.5 解释了带有绝对值整流的网络是如何创建函数的镜像图像的，这些函数在某些隐藏单元的顶部计算，作用于隐藏单元的输入。每个隐藏单元指定在哪里折叠输入空间，来创造镜像响应（在绝对值非线性的两侧）。通过组合这些折叠操作，我们获得指数级的分段线性区域，他们可以概括所有种类的规则模式（例如，重复）。

Montufar *et al.* (2014) 的主要定理指出，具有 d 个输入、深度为 l、每个隐藏

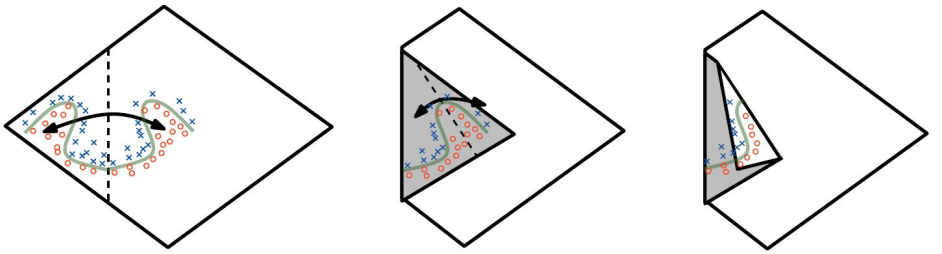

图 6.5: 关于更深的整流网络具有指数优势的一个直观的几何解释，来自 Montufar *et al.* (2014)。*(左)*绝对值整流单元对其输入中的每对镜像点有相同的输出。镜像的对称轴由单元的权重和偏置定义的超平面给出。在该单元顶部计算的函数（绿色决策面）将是横跨该对称轴的更简单模式的一个镜像。*(中)* 该函数可以通过折叠对称轴周围的空间来得到。*(右)* 另一个重复模式可以在第一个的顶部折叠（由另一个下游单元）以获得另外的对称性（现在重复四次，使用了两个隐藏层）。经 Montufar *et al.* (2014) 许可改编此图。

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

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right), \tag{6.42}$$

意味着，这是深度 l 的指数级。在每个单元具有 k 个过滤器的 maxout 网络中，线性区域的数量是

$$O\left(k^{(l-1)+d}\right). \tag{6.43}$$

当然，我们不能保证在机器学习（特别是 AI）的应用中我们想要学得的函数类型享有这样的属性。

我们还可能出于统计原因来选择深度模型。任何时候，当我们选择一个特定的机器学习算法时，我们隐含地陈述了一些先验，这些先验是关于算法应该学得什么样的函数的。选择深度模型默许了一个非常普遍的信念，那就是我们想要学得的函数应该涉及几个更加简单的函数的组合。这可以从表示学习的观点来解释，我们相信学习的问题包含发现一组潜在的变差因素，它们可以根据其他更简单的潜在的变差因素来描述。或者，我们可以将深度结构的使用解释为另一种信念，那就是我们想要学得的函数是包含多个步骤的计算机程序，其中每个步骤使用前一步骤的输出。这些中间输出不一定是变差因素，而是可以类似于网络用来组织其内部处理的计数器或指针。根据经验，更深的模型似乎确实在广泛的任务中泛化得更好 (Bengio *et al.*, 2007b;

Erhan *et al.*, 2009; Bengio, 2009; Mesnil *et al.*, 2011; Ciresan *et al.*, 2012; Krizhevsky *et al.*, 2012a; Sermanet *et al.*, 2013; Farabet *et al.*, 2013; Couprie *et al.*, 2013; Kahou *et al.*, 2013; Goodfellow *et al.*, 2014d; Szegedy *et al.*, 2014a)。图 6.6 和图 6.7 展示了一些实验结果的例子。这表明使用深层架构确实在模型学习的函数空间上表示了一个有用的先验。

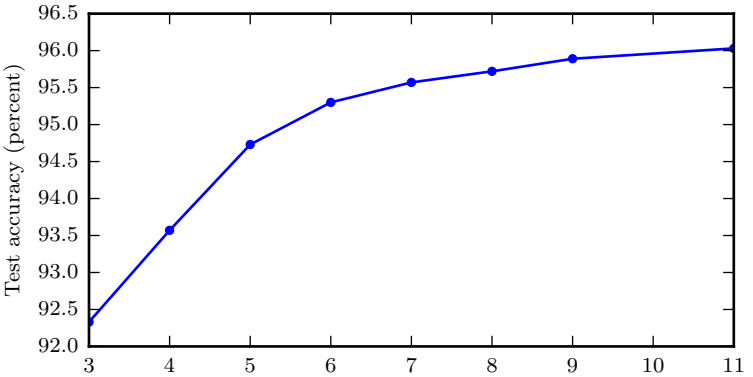

图 6.6: 深度的影响。实验结果表明，当从地址照片转录多位数字时，更深层的网络能够更好地泛化。数据来自 Goodfellow *et al.* (2014d)。测试集上的准确率随着深度的增加而不断增加。图 6.7 给出了一个对照实验，它说明了对模型尺寸其他方面的增加并不能产生相同的效果。

6.4.2 其他架构上的考虑

目前为止，我们都将神经网络描述成层的简单链式结构，主要的考虑因素是网络的深度和每层的宽度。在实践中，神经网络显示出相当的多样性。

许多神经网络架构已经被开发用于特定的任务。用于计算机视觉的卷积神经网络的特殊架构将在第九章中介绍。前馈网络也可以推广到用于序列处理的循环神经网络，但有它们自己的架构考虑，将在第十章中介绍。

一般的，层不需要连接在链中，尽管这是最常见的做法。许多架构构建了一个主链，但随后又添加了额外的架构特性，例如从层 i 到层 $i+2$ 或者更高层的跳跃连接。这些跳跃连接使得梯度更容易从输出层流向更接近输入的层。

架构设计考虑的另外一个关键点是如何将层与层之间连接起来。默认的神经网络层采用矩阵 $\boldsymbol{W}$ 描述的线性变换，每个输入单元连接到每个输出单元。在之后章节

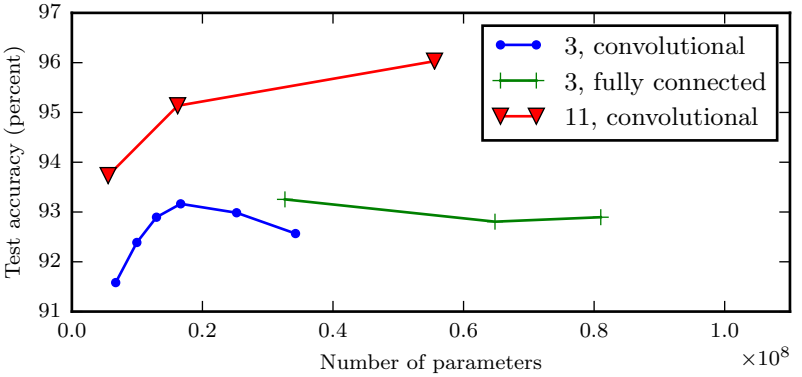

图 6.7: 参数数量的影响。更深的模型往往表现更好。这不仅仅是因为模型更大。Goodfellow *et al.* (2014d) 的这项实验表明，增加卷积网络层中参数的数量，但是不增加它们的深度，在提升测试集性能方面几乎没有效果，如此图所示。图例标明了用于画出每条曲线的网络深度，以及曲线表示的是卷积层还是全连接层的大小变化。我们可以观察到，在这种情况下，浅层模型在参数数量达到 2000 万时就过拟合，而深层模型在参数数量超过 6000 万时仍然表现良好。这表明，使用深层模型表达出了对模型可以学习的函数空间的有用偏好。具体来说，它表达了一种信念，即该函数应该由许多更简单的函数复合在一起而得到。这可能导致学习由更简单的表示所组成的表示（例如，由边所定义的角）或者学习具有顺序依赖步骤的程序（例如，首先定位一组对象，然后分割它们，之后识别它们）。

中的许多专用网络具有较少的连接，使得输入层中的每个单元仅连接到输出层单元的一个小子集。这些用于减少连接数量的策略减少了参数的数量以及用于评估网络的计算量，但通常高度依赖于问题。例如，第九章描述的卷积神经网络使用对于计算机视觉问题非常有效的稀疏连接的专用模式。在这一章中，很难对通用神经网络的架构给出更多具体的建议。我们在随后的章节中介绍一些特殊的架构策略，可以在不同的领域工作良好。

6.5 反向传播和其他的微分算法

当我们使用前馈神经网络接收输入 $\boldsymbol{x}$ 并产生输出 $\hat{\boldsymbol{y}}$ 时，信息通过网络向前流动。输入 $\boldsymbol{x}$ 提供初始信息，然后传播到每一层的隐藏单元，最终产生输出 $\hat{\boldsymbol{y}}$。这称之为 **前向传播**（forward propagation）。在训练过程中，前向传播可以持续向前直到它产生一个标量代价函数 $J(\boldsymbol{\theta})$。**反向传播**（back propagation）算法 (Rumelhart

et al., 1986c)，经常简称为**backprop**，允许来自代价函数的信息通过网络向后流动，以便计算梯度。

计算梯度的解析表达式是很直观的，但是数值化地求解这样的表达式在计算上的代价可能很大。反向传播算法使用简单和廉价的程序来实现这个目标。

反向传播这个术语经常被误解为用于多层神经网络的整个学习算法。实际上，反向传播仅指用于计算梯度的方法，而另一种算法，例如随机梯度下降，使用该梯度来进行学习。此外，反向传播经常被误解为仅适用于多层神经网络，但是原则上它可以计算任何函数的导数（对于一些函数，正确的响应是报告函数的导数是未定义的）。特别地，我们会描述如何计算一个任意函数 f 的梯度 $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$，其中 $\boldsymbol{x}$ 是一组变量，我们需要它们的导数，而 $\boldsymbol{y}$ 是函数的另外一组输入变量，但我们并不需要它们的导数。在学习算法中，我们最常需要的梯度是代价函数关于参数的梯度，即 $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$。许多机器学习任务需要计算其他导数，来作为学习过程的一部分，或者用来分析学得的模型。反向传播算法也适用于这些任务，不局限于计算代价函数关于参数的梯度。通过在网络中传播信息来计算导数的想法非常普遍，它还可以用于计算诸如多输出函数 f 的 Jacobian 的值。我们这里描述的是最常用的情况，其中 f 只有单个输出。

6.5.1 计算图

目前为止，我们已经用相对非正式的图形语言讨论了神经网络。为了更精确地描述反向传播算法，使用更精确的**计算图**（computational graph）语言是很有帮助的。

将计算形式化为图形的方法有很多。

这里，我们使用图中的每一个节点来表示一个变量。变量可以是标量、向量、矩阵、张量、或者甚至是另一类型的变量。

为了形式化我们的图形，我们还需引入**操作**（operation）这一概念。操作是指一个或多个变量的简单函数。我们的图形语言伴随着一组被允许的操作。我们可以通过将多个操作复合在一起来描述更为复杂的函数。

不失一般性，我们定义一个操作仅返回单个输出变量。这并没有失去一般性，是因为输出变量可以有多个条目，例如向量。反向传播的软件实现通常支持具有多个输出的操作，但是我们在描述中避免这种情况，因为它引入了对概念理解不重要的

许多额外细节。

如果变量 y 是变量 x 通过一个操作计算得到的，那么我们画一条从 x 到 y 的有向边。我们有时用操作的名称来注释输出的节点，当上下文很明确时，有时也会省略这个标注。

计算图的实例可以参考图 6.8 。

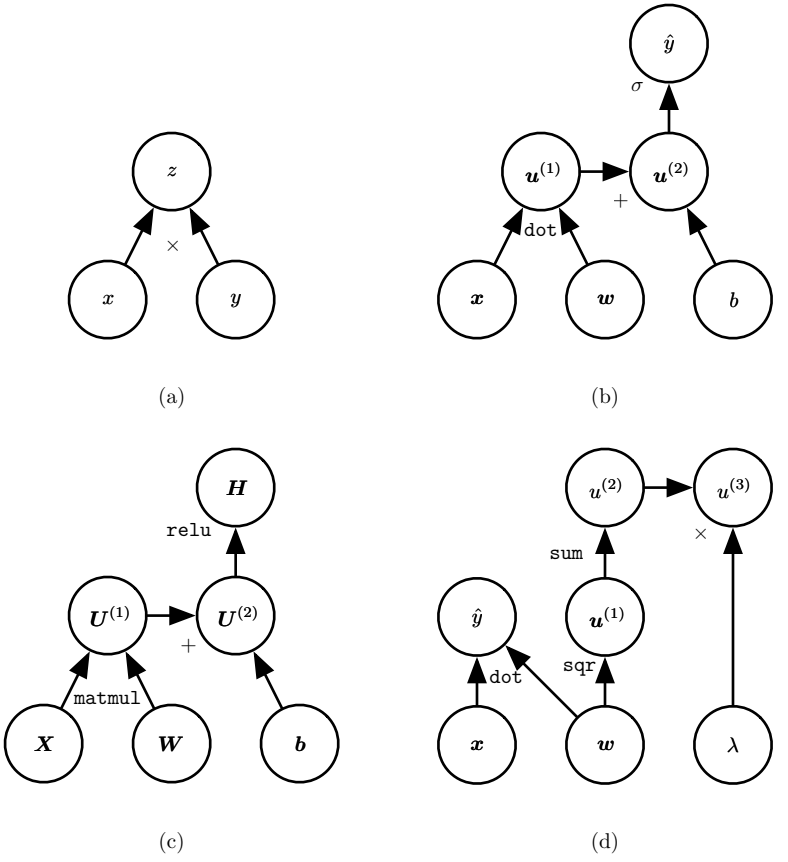

图 6.8: 一些计算图的示例。*(a)* 使用 $\times$ 操作计算 $z = xy$ 的图。*(b)* 用于逻辑回归预测 $\hat{y} = \sigma(\boldsymbol{x}^\top \boldsymbol{w} + b)$ 的图。一些中间表达式在代数表达式中没有名称，但在图形中却需要。我们简单地将第 i 个这样的变量命名为 $\boldsymbol{u}^{(i)}$。*(c)* 表达式 $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}$ 的计算图，在给定包含小批量输入数据的设计矩阵 $\boldsymbol{X}$ 时，它计算整流线性单元激活的设计矩阵 $\boldsymbol{H}$。*(d)* 示例 a-c 对每个变量最多只实施一个操作，但是对变量实施多个操作也是可能的。这里我们展示一个计算图，它对线性回归模型的权重 $\boldsymbol{w}$ 实施多个操作。这个权重不仅用于预测 $\hat{y}$，也用于权重衰减罚项 $\lambda \sum_i w_i^2$。

6.5.2 微积分中的链式法则

微积分中的链式法则（为了不与概率中的链式法则相混淆）用于计算复合函数的导数。反向传播是一种计算链式法则的算法，使用高效的特定运算顺序。

设 x 是实数，f 和 g 是从实数映射到实数的函数。假设 $y = g(x)$ 并且 $z = f(g(x)) = f(y)$。那么链式法则是说

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{6.44}$$

我们可以将这种标量情况进行扩展。假设 $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$，$g$ 是从 $\mathbb{R}^m$ 到 $\mathbb{R}^n$ 的映射，f 是从 $\mathbb{R}^n$ 到 $\mathbb{R}$ 的映射。如果 $\boldsymbol{y} = g(\boldsymbol{x})$ 并且 $z = f(\boldsymbol{y})$，那么

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{6.45}$$

使用向量记法，可以等价地写成

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z, \tag{6.46}$$

这里 $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ 是 g 的 $n \times m$ 的 Jacobian 矩阵。

从这里我们看到，变量 $\boldsymbol{x}$ 的梯度可以通过 Jacobian 矩阵 $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ 和梯度 $\nabla_{\boldsymbol{y}} z$ 相乘来得到。反向传播算法由图中每一个这样的 Jacobian 梯度的乘积操作所组成。

通常我们将反向传播算法应用于任意维度的张量，而不仅仅用于向量。从概念上讲，这与使用向量的反向传播完全相同。唯一的区别是如何将数字排列成网格以形成张量。我们可以想象，在我们运行反向传播之前，将每个张量变平为一个向量，计算一个向量值梯度，然后将该梯度重新构造成一个张量。从这种重新排列的观点上看，反向传播仍然只是将 Jacobian 乘以梯度。

为了表示值 z 关于张量 $\mathbf{X}$ 的梯度，我们记为 $\nabla_{\mathbf{X}} z$，就像 $\mathbf{X}$ 是向量一样。$\mathbf{X}$ 的索引现在有多个坐标——例如，一个 3 维的张量由三个坐标索引。我们可以通过使用单个变量 i 来表示完整的索引元组，从而完全抽象出来。对所有可能的元组 i，$(\nabla_{\mathbf{X}} z)_i$ 给出 $\frac{\partial z}{\partial \mathrm{X}_i}$。这与向量中索引的方式完全一致，$(\nabla_{\boldsymbol{x}} z)_i$ 给出 $\frac{\partial z}{\partial x_i}$。使用这种记法，我们可以写出适用于张量的链式法则。如果 $\mathbf{Y} = g(\mathbf{X})$ 并且 $z = f(\mathbf{Y})$，那么

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} \mathrm{Y}_j)\frac{\partial z}{\partial \mathrm{Y}_j}. \tag{6.47}$$

6.5.3 递归地使用链式法则来实现反向传播

使用链式规则，我们可以直接写出某个标量关于计算图中任何产生该标量的节点的梯度的代数表达式。然而，实际在计算机中计算该表达式时会引入一些额外的考虑。

具体来说，许多子表达式可能在梯度的整个表达式中重复若干次。任何计算梯度的程序都需要选择是存储这些子表达式还是重新计算它们几次。图 6.9 给出了一个例子来说明这些重复的子表达式是如何出现的。在某些情况下，计算两次相同的子表达式纯粹是浪费。在复杂图中，可能存在指数多的这种计算上的浪费，使得简单的链式法则不可实现。在其他情况下，计算两次相同的子表达式可能是以较高的运行时间为代价来减少内存开销的有效手段。

我们首先给出一个版本的反向传播算法，它指明了梯度的直接计算方式（算法 6.2 以及相关的正向计算的算法 6.1 ），按照它实际完成的顺序并且递归地使用链式法则。我们可以直接执行这些计算或者将算法的描述视为用于计算反向传播的计算图的符号表示。然而，这些公式并没有明确地操作和构造用于计算梯度的符号图。这些公式将在后面的第 6.5.6 节和算法 6.5 中给出，其中我们还推广到了包含任意张量的节点。

首先考虑描述如何计算单个标量 $u^{(n)}$（例如训练样本上的损失函数）的计算图。我们想要计算这个标量对 n_i 个输入节点 $u^{(1)}$ 到 $u^{(n_i)}$ 的梯度。换句话说，我们希望对所有的 $i \in \{1, 2, \ldots, n_i\}$ 计算 $\frac{\partial u^{(n)}}{\partial u^{(i)}}$。在使用反向传播计算梯度来实现参数的梯度下降时，$u^{(n)}$ 将对应单个或者小批量实例的代价函数，而 $u^{(1)}$ 到 $u^{(n_i)}$ 则对应于模型的参数。

我们假设图的节点已经以一种特殊的方式被排序，使得我们可以一个接一个地计算他们的输出，从 $u^{(n_i+1)}$ 开始，一直上升到 $u^{(n)}$。如算法 6.1 中所定义的，每个节点 $u^{(i)}$ 与操作 $f^{(i)}$ 相关联，并且通过对以下函数求值来得到

$$u^{(i)} = f(\mathbb{A}^{(i)}), \tag{6.48}$$

其中 $\mathbb{A}^{(i)}$ 是 $u^{(i)}$ 所有父节点的集合。

该算法详细说明了前向传播的计算，我们可以将其放入图 $\mathcal{G}$ 中。为了执行反向传播，我们可以构造一个依赖于 $\mathcal{G}$ 并添加额外一组节点的计算图。这形成了一个子图 $\mathcal{B}$，它的每个节点都是 $\mathcal{G}$ 的节点。$\mathcal{B}$ 中的计算和 $\mathcal{G}$ 中的计算顺序完全相反，而且 $\mathcal{B}$ 中的每个节点计算导数 $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ 与前向图中的节点 $u^{(i)}$ 相关联。这通过对标量输出

算法 6.1 计算将 n_i 个输入 $u^{(1)}$ 到 $u^{(n_i)}$ 映射到一个输出 $u^{(n)}$ 的程序。这定义了一个计算图，其中每个节点通过将函数 $f^{(i)}$ 应用到变量集合 $\mathbb{A}^{(i)}$ 上来计算 $u^{(i)}$ 的值，$\mathbb{A}^{(i)}$ 包含先前节点 $u^{(j)}$ 的值满足 $j < i$ 且 $j \in Pa(u^{(i)})$。计算图的输入是向量 $\boldsymbol{x}$，并且被分配给前 n_i 个节点 $u^{(1)}$ 到 $u^{(n_i)}$。计算图的输出可以从最后一个（输出）节点 $u^{(n)}$ 读出。

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

$u^{(n)}$ 使用链式法则来完成：

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}} \tag{6.49}$$

这在算法 6.2 中详细说明。子图 $\mathcal{B}$ 恰好包含每一条对应着 $\mathcal{G}$ 中从节点 $u^{(j)}$ 到节点 $u^{(i)}$ 的边。从 $u^{(j)}$ 到 $u^{(i)}$ 的边对应着计算 $\frac{\partial u^{(i)}}{\partial u^{(j)}}$。另外，对于每个节点都要执行一个内积，内积的一个因子是对于 u^j 子节点 $u^{(i)}$ 的已经计算的梯度，另一个因子是对于相同子节点 $u^{(i)}$ 的偏导数 $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ 组成的向量。总而言之，执行反向传播所需的计算量与 $\mathcal{G}$ 中的边的数量成比例，其中每条边的计算包括计算偏导数（节点关于它的一个父节点的偏导数）以及执行一次乘法和一次加法。下面，我们将此分析推广到张量值节点，这只是在同一节点中对多个标量值进行分组并能够更高效地实现。

反向传播算法被设计为减少公共子表达式的数量而不考虑存储的开销。具体来说，它大约对图中的每个节点执行一个 Jacobian 乘积。这可以从算法 6.2 中看出，反向传播算法访问了图中的节点 $u^{(j)}$ 到节点 $u^{(i)}$ 的每条边一次，以获得相关的偏导数 $\frac{\partial u^{(i)}}{\partial u^{(j)}}$。反向传播因此避免了重复子表达式的指数爆炸。然而，其他算法可能通过对计算图进行简化来避免更多的子表达式，或者也可能通过重新计算而不是存储这些子表达式来节省内存。我们将在描述完反向传播算法本身后再重新审视这些想法。

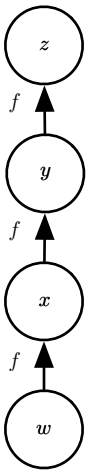

图 6.9: 计算梯度时导致重复子表达式的计算图。令 $w \in \mathbb{R}$ 为图的输入。我们对链中的每一步使用相同的操作函数 $f : \mathbb{R} \to \mathbb{R}$，这样 $x = f(w), y = f(x), z = f(y)$。为了计算 $\frac{\partial z}{\partial w}$，我们应用式 (6.44) 得到：

$$\frac{\partial z}{\partial w} \tag{6.50}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \tag{6.51}$$

$$= f'(y)f'(x)f'(w) \tag{6.52}$$

$$= f'(f(f(w)))f'(f(w))f'(w). \tag{6.53}$$

式 (6.52) 建议我们采用的实现方式是，仅计算 $f(w)$ 的值一次并将它存储在变量 x 中。这是反向传播算法所采用的方法。式 (6.53) 提出了一种替代方法，其中子表达式 $f(w)$ 出现了不止一次。在替代方法中，每次只在需要时重新计算 $f(w)$。当存储这些表达式的值所需的存储较少时，式 (6.52) 的反向传播方法显然是较优的，因为它减少了运行时间。然而，式 (6.53) 也是链式法则的有效实现，并且当存储受限时它是有用的。

6.5.4 全连接 MLP 中的反向传播计算

为了阐明反向传播的上述定义，让我们考虑一个与全连接的多层 MLP 相关联的特定图。

算法 6.3 首先给出了前向传播，它将参数映射到与单个训练样本（输入，目标）$(\boldsymbol{x}, \boldsymbol{y})$ 相关联的监督损失函数 $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$，其中 $\hat{\boldsymbol{y}}$ 是当 $\boldsymbol{x}$ 提供输入时神经网络的输出。

算法 6.4 随后说明了将反向传播应用于该图所需的相关计算。

算法 6.2 反向传播算法的简化版本，用于计算 $u^{(n)}$ 关于图中变量的导数。这个示例旨在通过演示所有变量都是标量的简化情况来进一步理解反向传播算法，这里我们希望计算关于 $u^{(1)}, \ldots, u^{(n_i)}$ 的导数。这个简化版本计算了关于图中所有节点的导数。假定与每条边相关联的偏导数计算需要恒定的时间的话，该算法的计算成本与图中边的数量成比例。这与前向传播的计算次数具有相同的阶。每个 $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ 是 $u^{(i)}$ 的父节点 $u^{(j)}$ 的函数，从而将前向图的节点链接到反向传播图中添加的节点。

运行前向传播 (对于此例是算法 6.1) 获得网络的激活。

初始化 `grad_table`，用于存储计算好的导数的数据结构。`grad_table`$[u^{(i)}]$ 将存储 $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ 计算好的值。

`grad_table`$[u^{(n)}] \leftarrow 1$

for $j = n - 1$ down to 1 **do**

下一行使用存储的值计算 $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$：

`grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return {`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i$}

算法 6.3 和算法 6.4 是简单而直观的演示。然而，它们专门针对特定的问题。

现在的软件实现基于之后第 6.5.6 节中描述的一般形式的反向传播，它可以通过显式地操作表示符号计算的数据结构，来适应任何计算图。

6.5.5 符号到符号的导数

代数表达式和计算图都对 **符号**（symbol）或不具有特定值的变量进行操作。这些代数或者基于图的表达式被称为 **符号表示**（symbolic representation）。当我们实际使用或者训练神经网络时，我们必须给这些符号赋特定的值。我们用一个特定的 **数值**（numeric value）来替代网络的符号输入 $\boldsymbol{x}$，例如 $[1.2, 3, 765, -1.8]^\top$。

一些反向传播的方法采用计算图和一组用于图的输入的数值，然后返回在这些输入值处梯度的一组数值。我们将这种方法称为 **符号到数值**的微分。这种方法用在诸如 Torch (Collobert *et al.*, 2011b) 和 Caffe (Jia, 2013) 之类的库中。

另一种方法是采用计算图以及添加一些额外的节点到计算图中，这些额外的节点提供了我们所需导数的符号描述。这是 Theano (Bergstra *et al.*, 2010b; Bastien

算法 6.3 典型深度神经网络中的前向传播和代价函数的计算。损失函数 $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ 取决于输出 $\hat{\boldsymbol{y}}$ 和目标 $\boldsymbol{y}$（参考第 6.2.1.1 节中损失函数的示例）。为了获得总代价 J，损失函数可以加上正则项 $\Omega(\theta)$，其中 θ 包含所有参数（权重和偏置）。算法 6.4 说明了如何计算 J 关于参数 $\boldsymbol{W}$ 和 $\boldsymbol{b}$ 的梯度。为简单起见，该演示仅使用单个输入样本 $\boldsymbol{x}$。实际应用应该使用小批量。请参考第 6.5.7 节以获得更加真实的演示。

Require: 网络深度，l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$，模型的权重矩阵
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$，模型的偏置参数
Require: $\boldsymbol{x}$，程序的输入
Require: $\boldsymbol{y}$，目标输出

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

et al., 2012b) 和 TensorFlow (Abadi *et al.*, 2015) 所采用的方法。图 6.10 给出了该方法如何工作的一个例子。这种方法的主要优点是导数可以使用与原始表达式相同的语言来描述。因为导数只是另外一张计算图，我们可以再次运行反向传播，对导数再进行求导就能得到更高阶的导数。高阶导数的计算在第 6.5.10 节中描述。

我们将使用后一种方法，并且使用构造导数的计算图的方法来描述反向传播算法。图的任意子集之后都可以使用特定的数值来求值。这允许我们避免精确地指明每个操作应该在何时计算。相反，通用的图计算引擎只要当一个节点的父节点的值都可用时就可以进行求值。

基于符号到符号的方法的描述包含了符号到数值的方法。符号到数值的方法可以理解为执行了与符号到符号的方法中构建图的过程中完全相同的计算。关键的区别是符号到数值的方法不会显示出计算图。

算法 6.4 深度神经网络中算法 6.3 的反向计算，它不止使用了输入 $\boldsymbol{x}$ 和目标 $\boldsymbol{y}$。该计算对于每一层 k 都产生了对激活 $\boldsymbol{a}^{(k)}$ 的梯度，从输出层开始向后计算一直到第一个隐藏层。这些梯度可以看作是对每层的输出应如何调整以减小误差的指导，根据这些梯度可以获得对每层参数的梯度。权重和偏置上的梯度可以立即用作随机梯度更新的一部分（梯度算出后即可执行更新），或者与其他基于梯度的优化方法一起使用。

在前向计算完成后，计算顶层的梯度：

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

将关于层输出的梯度转换为非线性激活输入前的梯度（如果 f 是逐元素的，则逐元素地相乘）：

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

计算关于权重和偏置的梯度（如果需要的话，还要包括正则项）：

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

关于下一更低层的隐藏层传播梯度：

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

end for

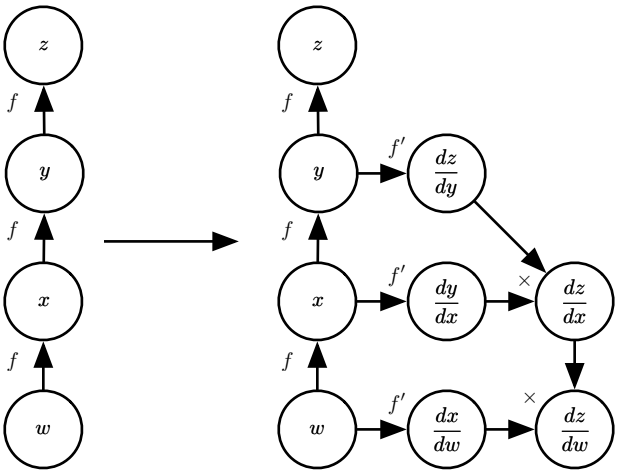

图 6.10: 使用符号到符号的方法计算导数的示例。在这种方法中，反向传播算法不需要访问任何实际的特定数值。相反，它将节点添加到计算图中来描述如何计算这些导数。通用图形求值引擎可以在随后计算任何特定数值的导数。*(左)* 在这个例子中，我们从表示 $z = f(f(f(w)))$ 的图开始。*(右)* 我们运行反向传播算法，指导它构造表达式 $\frac{dz}{dw}$ 对应的图。在这个例子中，我们不解释反向传播算法如何工作。我们的目的只是说明想要的结果是什么：符号描述的导数的计算图。

6.5.6 一般化的反向传播

反向传播算法非常简单。为了计算某个标量 z 关于图中它的一个祖先 $\boldsymbol{x}$ 的梯度，我们首先观察到它关于 z 的梯度由 $\frac{dz}{dz} = 1$ 给出。然后，我们可以计算对图中 z 的每个父节点的梯度，通过现有的梯度乘以产生 z 的操作的 Jacobian。我们继续乘以 Jacobian，以这种方式向后穿过图，直到我们到达 $\boldsymbol{x}$。对于从 z 出发可以经过两个或更多路径向后行进而到达的任意节点，我们简单地对该节点来自不同路径上的梯度进行求和。

更正式地，图 $\mathcal{G}$ 中的每个节点对应着一个变量。为了实现最大的一般化，我们将这个变量描述为一个张量 $\mathbf{V}$。张量通常可以具有任意维度，并且包含标量、向量和矩阵。

我们假设每个变量 $\mathbf{V}$ 与下列子程序相关联：

- `get_operation`($\mathbf{V}$)：它返回用于计算 $\mathbf{V}$ 的操作，代表了在计算图中流入 $\mathbf{V}$ 的边。例如，可能有一个 Python 或者 C++ 的类表示矩阵乘法操作，以及 `get_operation` 函数。假设我们的一个变量是由矩阵乘法产生的，$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$。

那么，get_operation($\mathbf{V}$) 返回一个指向相应 C++ 类的实例的指针。

- get_consumers($\mathbf{V}, \mathcal{G}$)：它返回一组变量，是计算图 $\mathcal{G}$ 中 $\mathbf{V}$ 的子节点。
- get_inputs($\mathbf{V}, \mathcal{G}$)：它返回一组变量，是计算图 $\mathcal{G}$ 中 $\mathbf{V}$ 的父节点。

每个操作 op 也与 bprop 操作相关联。该 bprop 操作可以计算如式 (6.47) 所描述的 Jacobian 向量积。这是反向传播算法能够实现很大通用性的原因。每个操作负责了解如何通过它参与的图中的边来反向传播。例如，我们可以使用矩阵乘法操作来产生变量 $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$。假设标量 z 关于 $\boldsymbol{C}$ 的梯度是 $\boldsymbol{G}$。矩阵乘法操作负责定义两个反向传播规则，每个规则对应于一个输入变量。如果我们调用 bprop 方法来请求关于 $\boldsymbol{A}$ 的梯度，那么在给定输出的梯度为 $\boldsymbol{G}$ 的情况下，矩阵乘法操作的 bprop 方法必须说明关于 $\boldsymbol{A}$ 的梯度是 $\boldsymbol{G}\boldsymbol{B}^\top$。类似的，如果我们调用 bprop 方法来请求关于 $\boldsymbol{B}$ 的梯度，那么矩阵操作负责实现 bprop 方法并指定希望的梯度是 $\boldsymbol{A}^\top\boldsymbol{G}$。反向传播算法本身并不需要知道任何微分法则。它只需要使用正确的参数调用每个操作的 bprop 方法即可。正式地， op.bprop(inputs, $\mathbf{X}, \mathbf{G}$) 必须返回

$$\sum_i (\nabla_{\mathbf{X}} \texttt{op.f(inputs)}_i)\mathsf{G}_i, \tag{6.54}$$

这只是如式 (6.47) 所表达的链式法则的实现。这里，inputs 是提供给操作的一组输入，op.f 是操作实现的数学函数，$\mathbf{X}$ 是输入，我们想要计算关于它的梯度，$\mathbf{G}$ 是操作对于输出的梯度。

op.bprop 方法应该总是假装它的所有输入彼此不同，即使它们不是。例如，如果 mul 操作传递两个 x 来计算 x^2，op.bprop 方法应该仍然返回 x 作为对于两个输入的导数。反向传播算法后面会将这些变量加起来获得 $2x$，这是 x 上总的正确的导数。

反向传播算法的软件实现通常提供操作和其 bprop 方法，所以深度学习软件库的用户能够对使用诸如矩阵乘法、指数运算、对数运算等等常用操作构建的图进行反向传播。构建反向传播新实现的软件工程师或者需要向现有库添加自己的操作的高级用户通常必须手动为新操作推导 op.bprop 方法。

反向传播算法的正式描述参考算法 6.5 。

在第 6.5.2 节中，我们使用反向传播作为一种策略来避免多次计算链式法则中的相同子表达式。由于这些重复子表达式的存在，简单的算法可能具有指数运行时间。现在我们已经详细说明了反向传播算法，我们可以去理解它的计算成本。如果我们

算法 6.5 反向传播算法最外围的骨架。这部分做简单的设置和清理工作。大多数重要的工作发生在算法 6.6 的子程序 `build_grad` 中。

Require: $\mathbb{T}$，需要计算梯度的目标变量集
Require: $\mathcal{G}$，计算图
Require: z，要微分的变量
  令 $\mathcal{G}'$ 为 $\mathcal{G}$ 剪枝后的计算图，其中仅包括 z 的祖先以及 $\mathbb{T}$ 中节点的后代。
  初始化 `grad_table`，它是关联张量和对应导数的数据结构。
  `grad_table`$[z] \leftarrow 1$
  for $\mathbf{V}$ in $\mathbb{T}$ **do**
    `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$
  end for
  Return `grad_table` restricted to $\mathbb{T}$

假设每个操作的执行都有大致相同的开销，那么我们可以依据执行操作的数量来分析计算成本。注意这里我们将一个操作记为计算图的基本单位，它实际可能包含许多算术运算（例如，我们可能将矩阵乘法视为单个操作）。在具有 n 个节点的图中计算梯度，将永远不会执行超过 $O(n^2)$ 个操作，或者存储超过 $O(n^2)$ 个操作的输出。这里我们是对计算图中的操作进行计数，而不是由底层硬件执行的单独操作，所以重要的是要记住每个操作的运行时间可能是高度可变的。例如，两个矩阵相乘可能对应着图中的一个单独的操作，但这两个矩阵可能每个都包含数百万个元素。我们可以看到，计算梯度至多需要 $O(n^2)$ 的操作，因为在最坏的情况下，前向传播的步骤将在原始图的全部 n 个节点上运行（取决于我们想要计算的值，我们可能不需要执行整个图）。反向传播算法在原始图的每条边添加一个 Jacobian 向量积，可以用 $O(1)$ 个节点来表达。因为计算图是有向无环图，它至多有 $O(n^2)$ 条边。对于实践中常用图的类型，情况会更好。大多数神经网络的代价函数大致是链式结构的，使得反向传播只有 $O(n)$ 的成本。这远远胜过简单的方法，简单方法可能需要在指数级的节点上运算。这种潜在的指数级代价可以通过非递归地扩展和重写递归链式法则（式 (6.49)）来看出：

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path}(u^{(\pi_1)}, u^{(\pi_2)}, \ldots, u^{(\pi_t)}), \\ \text{from } \pi_1 = j \text{ to } \pi_t = n}} \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}. \tag{6.55}$$

由于节点 j 到节点 n 的路径数目可以关于这些路径的长度上指数地增长，所以上述

算法 6.6 反向传播算法的内循环子程序 `build_grad`($\mathbf{V}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table}$)，由算法 6.5 中定义的反向传播算法调用。

Require: $\mathbf{V}$，应该被加到 $\mathcal{G}$ 和 `grad_table` 的变量。
Require: $\mathcal{G}$，要修改的图。
Require: $\mathcal{G}'$，根据参与梯度的节点 $\mathcal{G}$ 的受限图。
Require: `grad_table`，将节点映射到对应梯度的数据结构。
  if $\mathbb{V}$ is in `grad_table` **then**
    Return `grad_table`[$\mathbf{V}$]
  end if
  $i \leftarrow 1$
  for $\mathbf{C}$ in `get_consumers`($\mathbf{V}, \mathcal{G}'$) **do**
    `op` $\leftarrow$ `get_operation`($\mathbf{C}$)
    $\mathbf{D} \leftarrow$ `build_grad`($\mathbf{C}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table}$)
    $\mathbf{G}^{(i)} \leftarrow$ `op.bprop`(`get_inputs`($\mathbf{C}, \mathcal{G}'$), $\mathbf{V}, \mathbf{D}$)
    $i \leftarrow i + 1$
  end for
  $\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$
  `grad_table`[$\mathbf{V}$] $= \mathbf{G}$
  插入 $\mathbf{G}$ 和将其生成到 $\mathcal{G}$ 中的操作
  Return $\mathbf{G}$

求和符号中的项数（这些路径的数目），可能以前向传播图的深度的指数级增长。会产生如此大的成本是因为对于 $\frac{\partial u^{(i)}}{\partial u^{(j)}}$，相同的计算会重复进行很多次。为了避免这种重新计算，我们可以将反向传播看作一种表填充算法，利用存储的中间结果 $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ 来对表进行填充。图中的每个节点对应着表中的一个位置，这个位置存储对该节点的梯度。通过顺序填充这些表的条目，反向传播算法避免了重复计算许多公共子表达式。这种表填充策略有时被称为 **动态规划**（dynamic programming）。

6.5.7 实例：用于 MLP 训练的反向传播

作为一个例子，我们利用反向传播算法来训练多层感知机。

这里，我们考虑一个具有单个隐藏层的非常简单的多层感知机。为了训练这个

模型，我们将使用小批量随机梯度下降算法。反向传播算法用于计算单个小批量上的代价的梯度。具体来说，我们使用训练集上的一小批量实例，将其规范化为一个设计矩阵 $\boldsymbol{X}$ 以及相关联的类标签向量 $\boldsymbol{y}$。网络计算隐藏特征层 $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W}^{(1)}\}$。为了简化表示，我们在这个模型中不使用偏置。假设我们的图语言包含 `relu` 操作，该操作可以对 $\max\{0, \boldsymbol{Z}\}$ 表达式的每个元素分别进行计算。类的非归一化对数概率的预测将随后由 $\boldsymbol{H}\boldsymbol{W}^{(2)}$ 给出。假设我们的图语言包含 `cross_entropy` 操作，用以计算目标 $\boldsymbol{y}$ 和由这些未归一化对数概率定义的概率分布间的交叉熵。所得到的交叉熵定义了代价函数 J_{MLE}。最小化这个交叉熵将执行对分类器的最大似然估计。然而，为了使得这个例子更加真实，我们也包含一个正则项。总的代价函数为

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right)^2 \right) \tag{6.56}$$

包含了交叉熵和系数为 λ 的权重衰减项。它的计算图在图 6.11 中给出。

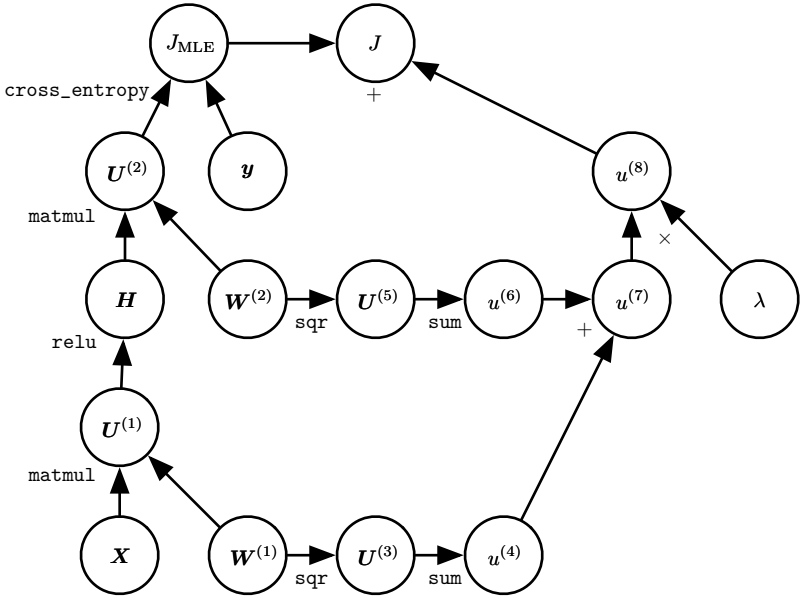

图 6.11: 用于计算代价函数的计算图，这个代价函数是使用交叉熵损失以及权重衰减训练我们的单层 MLP 示例所产生的。

这个示例的梯度计算图实在太大，以致绘制或者阅读都将是乏味的。这显示出了反向传播算法的优点之一，即它可以自动生成梯度，而这种计算对于软件工程师

来说需要进行直观但冗长的手动推导。

我们可以通过观察图6.11中的正向传播图来粗略地描述反向传播算法的行为。为了训练，我们希望计算 $\nabla_{\boldsymbol{W}^{(1)}} J$ 和 $\nabla_{\boldsymbol{W}^{(2)}} J$。有两种不同的路径从 J 后退到权重：一条通过交叉熵代价，另一条通过权重衰减代价。权重衰减代价相对简单，它总是对 $\boldsymbol{W}^{(i)}$ 上的梯度贡献 $2\lambda \boldsymbol{W}^{(i)}$。

另一条通过交叉熵代价的路径稍微复杂一些。令 $\boldsymbol{G}$ 是由 `cross_entropy` 操作提供的对未归一化对数概率 $\boldsymbol{U}^{(2)}$ 的梯度。反向传播算法现在需要探索两个不同的分支。在较短的分支上，它使用对矩阵乘法的第二个变量的反向传播规则，将 $\boldsymbol{H}^\top \boldsymbol{G}$ 加到 $\boldsymbol{W}^{(2)}$ 的梯度上。另一条更长些的路径沿着网络逐步下降。首先，反向传播算法使用对矩阵乘法的第一个变量的反向传播规则，计算 $\nabla_{\boldsymbol{H}} J = \boldsymbol{G}\boldsymbol{W}^{(2)\top}$。接下来，`relu` 操作使用其反向传播规则对先前梯度的部分位置清零，这些位置对应着 $\boldsymbol{U}^{(1)}$ 中所有小于 0 的元素。记上述结果为 $\boldsymbol{G}'$。反向传播算法的最后一步是使用对 `matmul` 操作的第二个变量的反向传播规则，将 $\boldsymbol{X}^\top \boldsymbol{G}'$ 加到 $\boldsymbol{W}^{(1)}$ 的梯度上。

在计算了这些梯度以后，梯度下降算法或者其他优化算法所要做的就是使用这些梯度来更新参数。

对于 MLP，计算成本主要来源于矩阵乘法。在前向传播阶段，我们乘以每个权重矩阵，得到了 $O(w)$ 数量的乘-加，其中 w 是权重的数量。在反向传播阶段，我们乘以每个权重矩阵的转置，这具有相同的计算成本。算法主要的存储成本是我们需要将输入存储到隐藏层的非线性中去。这些值从被计算时开始存储，直到反向过程回到了同一点。因此存储成本是 $O(mn_h)$，其中 m 是小批量中样本的数目，n_h 是隐藏单元的数量。

6.5.8 复杂化

我们这里描述的反向传播算法要比实践中实际使用的实现要简单。

正如前面提到的，我们将操作的定义限制为返回单个张量的函数。大多数软件实现需要支持可以返回多个张量的操作。例如，如果我们希望计算张量中的最大值和该值的索引，则最好在单次运算中计算两者，因此将该过程实现为具有两个输出的操作效率更高。

我们还没有描述如何控制反向传播的内存消耗。反向传播经常涉及将许多张量加在一起。在朴素方法中，将分别计算这些张量中的每一个，然后在第二步中对所

有这些张量求和。朴素方法具有过高的存储瓶颈，可以通过保持一个缓冲器，并且在计算时将每个值加到该缓冲器中来避免该瓶颈。

反向传播的现实实现还需要处理各种数据类型，例如 32 位浮点数、64 位浮点数和整型。处理这些类型的策略需要特别的设计考虑。

一些操作具有未定义的梯度，并且重要的是跟踪这些情况并且确定用户请求的梯度是否是未定义的。

各种其他技术的特性使现实世界的微分更加复杂。这些技术性并不是不可逾越的，本章已经描述了计算微分所需的关键知识工具，但重要的是要知道还有许多的精妙之处存在。

6.5.9 深度学习界以外的微分

深度学习界在某种程度上已经与更广泛的计算机科学界隔离开来，并且在很大程度上发展了自己关于如何进行微分的文化态度。更一般地，**自动微分**（automatic differentiation）领域关心如何以算法方式计算导数。这里描述的反向传播算法只是自动微分的一种方法。它是一种称为**反向模式累加**（reverse mode accumulation）的更广泛类型的技术的特殊情况。其他方法以不同的顺序来计算链式法则的子表达式。一般来说，确定一种计算的顺序使得计算开销最小，是困难的问题。找到计算梯度的最优操作序列是 NP 完全问题 (Naumann, 2008)，在这种意义上，它可能需要将代数表达式简化为它们最廉价的形式。

例如，假设我们有变量 $p_1, p_2 \ldots, p_n$ 表示概率，以及变量 $z_1, z_2, \ldots, z_n$ 表示未归一化的对数概率。假设我们定义

$$q_i = \frac{\exp(z_i)}{\sum_i \exp(z_i)}, \tag{6.57}$$

其中我们通过指数化、求和与除法运算构建 softmax 函数，并构造交叉熵损失函数 $J = -\sum_i p_i \log q_i$。人类数学家可以观察到 J 对 z_i 的导数有一个非常简单的形式：$q_i - p_i$ 反向传播算法不能够以这种方式来简化梯度，而是会通过原始图中的所有对数和指数操作显式地传播梯度。一些软件库如 Theano (Bergstra *et al.*, 2010b; Bastien *et al.*, 2012b) 能够执行某些种类的代数替换来改进由纯反向传播算法提出的图。

当前向图 $\mathcal{G}$ 具有单个输出节点，并且每个偏导数 $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ 都可以用恒定的计算量来计算时，反向传播保证梯度计算的计算数目和前向计算的计算数目是同一个量级：

这可以在算法 6.2 中看出，因为每个局部偏导数 $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ 以及递归链式公式（式 (6.49)）中相关的乘和加都只需计算一次。因此，总的计算量是 $O(\#\text{edges})$。然而，可能通过对反向传播算法构建的计算图进行简化来减少这些计算量，并且这是 NP 完全问题。诸如 Theano 和 TensorFlow 的实现使用基于匹配已知简化模式的试探法，以便重复地尝试去简化图。我们定义反向传播仅用于计算标量输出的梯度，但是反向传播可以扩展到计算 Jacobian 矩阵（该 Jacobian 矩阵或者来源于图中的 k 个不同标量节点，或者来源于包含 k 个值的张量值节点）。朴素的实现可能需要 k 倍的计算：对于原始前向图中的每个内部标量节点，朴素的实现计算 k 个梯度而不是单个梯度。当图的输出数目大于输入的数目时，有时更偏向于使用另外一种形式的自动微分，称为 **前向模式累加**（forward mode accumulation）。前向模式计算已经被提出用于循环神经网络梯度的实时计算，例如 (Williams and Zipser, 1989)。这也避免了存储整个图的值和梯度的需要，是计算效率和内存使用的折中。前向模式和后向模式的关系类似于左乘和右乘一系列矩阵之间的关系，例如

$$\boldsymbol{ABCD}, \tag{6.58}$$

其中的矩阵可以认为是 Jacobian 矩阵。例如，如果 $\boldsymbol{D}$ 是列向量，而 $\boldsymbol{A}$ 有很多行，那么这对应于一幅具有单个输出和多个输入的图，并且从最后开始乘，反向进行，只需要矩阵-向量的乘积。这对应着反向模式。相反，从左边开始乘将涉及一系列的矩阵-矩阵乘积，这使得总的计算变得更加昂贵。然而，如果 $\boldsymbol{A}$ 的行数小于 D 的列数，则从左到右乘更为便宜，这对应着前向模式。

在机器学习以外的许多社区中，更常见的是使用传统的编程语言来直接实现微分软件，例如用 Python 或者 C 来编程，并且自动生成使用这些语言编写的不同函数的程序。在深度学习界中，计算图通常使用由专用库创建的明确的数据结构表示。专用方法的缺点是需要库开发人员为每个操作定义 `bprop` 方法，并且限制了库的用户仅使用定义好的那些操作。然而，专用方法也允许定制每个操作的反向传播规则，允许开发者以非显而易见的方式提高速度或稳定性，对于这种方式自动的过程可能不能复制。

因此，反向传播不是计算梯度的唯一方式或最佳方式，但它是一个非常实用的方法，继续为深度学习社区服务。在未来，深度网络的微分技术可能会提高，因为深度学习的从业者更加懂得了更广泛的自动微分领域的进步。

6.5.10 高阶微分

一些软件框架支持使用高阶导数。在深度学习软件框架中，这至少包括 Theano 和 TensorFlow。这些库使用一种数据结构来描述要被微分的原始函数，它们使用相同类型的数据结构来描述这个函数的导数表达式。这意味着符号微分机制可以应用于导数（从而产生高阶导数）。

在深度学习的相关领域，很少会计算标量函数的单个二阶导数。相反，我们通常对 Hessian 矩阵的性质比较感兴趣。如果我们有函数 $f : \mathbb{R}^n \to \mathbb{R}$，那么 Hessian 矩阵的大小是 $n \times n$。在典型的深度学习应用中，n 将是模型的参数数量，可能很容易达到数十亿。因此，完整的 Hessian 矩阵甚至不能表示。

典型的深度学习方法是使用 **Krylov 方法**（Krylov method），而不是显式地计算 Hessian 矩阵。Krylov 方法是用于执行各种操作的一组迭代技术，这些操作包括像近似求解矩阵的逆、或者近似矩阵的特征值或特征向量等，而不使用矩阵-向量乘法以外的任何操作。

为了在 Hesssian 矩阵上使用 Krylov 方法，我们只需要能够计算 Hessian 矩阵 $\boldsymbol{H}$ 和一个任意向量 $\boldsymbol{v}$ 间的乘积即可。实现这一目标的一种直观方法 (Christianson, 1992) 是

$$\boldsymbol{H}\boldsymbol{v} = \nabla_{\boldsymbol{x}} \left[(\nabla_{\boldsymbol{x}} f(x))^\top \boldsymbol{v} \right]. \tag{6.59}$$

该表达式中两个梯度的计算都可以由适当的软件库自动完成。注意，外部梯度表达式是内部梯度表达式的函数的梯度。

如果 $\boldsymbol{v}$ 本身是由计算图产生的一个向量，那么重要的是指定自动微分软件不要对产生 $\boldsymbol{v}$ 的图进行微分。

虽然计算 Hessian 通常是不可取的，但是可以使用 Hessian 向量积。可以对所有的 $i = 1, \ldots, n$ 简单地计算 $\boldsymbol{H}\boldsymbol{e}^{(i)}$，其中 $\boldsymbol{e}^{(i)}$ 是 $e_i^{(i)} = 1$ 并且其他元素都为 0 的one-hot向量。

6.6 历史小记

前馈网络可以被视为一种高效的非线性函数近似器，它以使用梯度下降来最小化函数近似误差为基础。从这个角度来看，现代前馈网络是一般函数近似任务的几个世纪进步的结晶。

处于反向传播算法底层的链式法则是 17 世纪发明的 (Leibniz, 1676; L'Hôpital, 1696)。微积分和代数长期以来被用于求解优化问题的封闭形式，但梯度下降直到 19 世纪才作为优化问题的一种迭代近似的求解方法被引入 (Cauchy, 1847)。

从 20 世纪 40 年代开始，这些函数近似技术被用于导出诸如感知机的机器学习模型。然而，最早的模型都是基于线性模型。来自包括 Marvin Minsky 的批评指出了线性模型族的几个缺陷，例如它无法学习 XOR 函数，这导致了对整个神经网络方法的抵制。

学习非线性函数需要多层感知机的发展和计算该模型梯度的方法。基于动态规划的链式法则的高效应用开始出现在 20 世纪 60 年代和 70 年代，主要用于控制领域 (Kelley, 1960; Bryson and Denham, 1961; Dreyfus, 1962; Bryson and Ho, 1969; Dreyfus, 1973)，也用于灵敏度分析 (Linnainmaa, 1976)。Werbos (1981) 提出应用这些技术来训练人工神经网络。这个想法以不同的方式被独立地重新发现后 (LeCun, 1985; Parker, 1985; Rumelhart *et al.*, 1986a)，最终在实践中得以发展。**并行分布式处理**（Parallel Distributed Processing）一书在其中一章提供了第一次成功使用反向传播的一些实验的结果 (Rumelhart *et al.*, 1986b)，这对反向传播的普及做出了巨大的贡献，并且开启了一个研究多层神经网络非常活跃的时期。然而，该书作者提出的想法，特别是 Rumelhart 和 Hinton 提出的想法远远超过了反向传播。它们包括一些关键思想，关于可能通过计算实现认知和学习的几个核心方面，后来被冠以“联结主义”的名称，因为它强调了神经元之间的连接作为学习和记忆的轨迹的重要性。特别地，这些想法包括分布式表示的概念 (Hinton *et al.*, 1986)。

在反向传播的成功之后，神经网络研究获得了普及，并在 20 世纪 90 年代初达到高峰。随后，其他机器学习技术变得更受欢迎，直到 2006 年开始的现代深度学习复兴。

现代前馈网络的核心思想自 20 世纪 80 年代以来没有发生重大变化。仍然使用相同的反向传播算法和相同的梯度下降方法。1986 年至 2015 年神经网络性能的大部分改进可归因于两个因素。首先，较大的数据集减少了统计泛化对神经网络的挑战的程度。第二，神经网络由于更强大的计算机和更好的软件基础设施已经变得更大。然而，少量算法上的变化也显著改善了神经网络的性能。

其中一个算法上的变化是用交叉熵族损失函数替代均方误差损失函数。均方误差在 20 世纪 80 年代和 90 年代流行，但逐渐被交叉熵损失替代，并且最大似然原理的想法在统计学界和机器学习界之间广泛传播。使用交叉熵损失大大提高了具

有 sigmoid 和 softmax 输出的模型的性能，而当使用均方误差损失时会存在饱和和学习缓慢的问题。

另一个显著改善前馈网络性能的算法上的主要变化是使用分段线性隐藏单元来替代 sigmoid 隐藏单元，例如用整流线性单元。使用 $\max\{0, z\}$ 函数的整流在早期神经网络中已经被引入，并且至少可以追溯到认知机（Cognitron）和神经认知机 (Neocognitron)(Fukushima, 1975, 1980)。这些早期的模型没有使用整流线性单元，而是将整流用于非线性函数。尽管整流在早期很普及，在 20 世纪 80 年代，整流很大程度上被 sigmoid 所取代，也许是因为当神经网络非常小时，sigmoid 表现更好。到 21 世纪初，由于有些迷信的观念，认为必须避免具有不可导点的激活函数，所以避免了整流线性单元。这在 2009 年开始发生改变。Jarrett *et al.* (2009b) 观察到，在神经网络结构设计的几个不同因素中 ‘‘使用整流非线性是提高识别系统性能的最重要的唯一因素’’。

对于小的数据集，Jarrett *et al.* (2009b) 观察到，使用整流非线性甚至比学习隐藏层的权重值更加重要。随机的权重足以通过整流网络传播有用的信息，允许在顶部的分类器层学习如何将不同的特征向量映射到类标识。

当有更多数据可用时，学习开始提取足够的有用知识来超越随机选择参数的性能。Glorot *et al.* (2011a) 说明，在深度整流网络中的学习比在激活函数具有曲率或两侧饱和的深度网络中的学习更容易。

整流线性单元还具有历史意义，因为它们表明神经科学继续对深度学习算法的发展产生影响。Glorot *et al.* (2011a) 从生物学考虑整流线性单元的导出。半整流非线性旨在描述生物神经元的这些性质：(1) 对于某些输入，生物神经元是完全不活跃的。(2) 对于某些输入，生物神经元的输出和它的输入成比例。(3) 大多数时间，生物神经元是在它们不活跃的状态下进行操作（即它们应该具有 **稀疏激活**（sparse activation））。

当 2006 年深度学习开始现代复兴时，前馈网络仍然有不良的声誉。从 2006 年至 2012 年，人们普遍认为，前馈网络不会表现良好，除非它们得到其他模型的辅助，例如概率模型。现在已经知道，只要具备适当的资源和工程实践，前馈网络表现得非常好。今天，前馈网络中基于梯度的学习被用作发展概率模型的工具，例如第二十章中描述的变分自编码器和生成式对抗网络。前馈网络中基于梯度的学习自 2012 年以来一直被视为一种强大的技术，并应用于许多其他机器学习任务，而不是被视为必须由其他技术支持的不可靠技术。在 2006 年，业内使用无监督学习来支持监督