计算机科学188：人工智能导论
2024年春季 Note 22

作者（其他所有注释）：尼基尔·夏尔马
作者（贝叶斯网络注释）：乔希·胡格和杰基·梁，由王瑞吉娜编辑
作者（逻辑注释）：亨利·朱，由考佩林编辑
学分（机器学习和逻辑注释）：部分章节改编自教材《人工智能：一种现代方法》。

最后更新时间：2023年8月26日

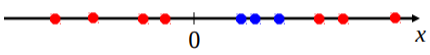
神经网络：动机

在接下来的内容中，我们将介绍神经网络的概念。在此过程中，我们将使用一些我们为二元逻辑回归和多类逻辑回归开发的建模技术。

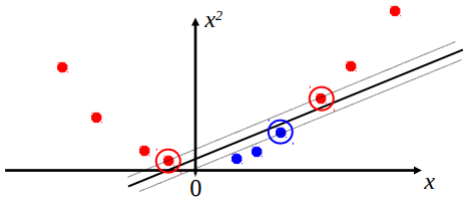
非线性分离器

我们知道如何构建一个学习二元分类任务线性边界的模型。这是一项强大的技术，当潜在的最优决策边界本身是线性时，它能很好地发挥作用。然而，许多实际问题需要非线性的决策边界，而我们的线性感知机模型表现力不足以捕捉这种关系。

考虑以下数据集：



我们想要分离这两种颜色，显然在一维空间中无法做到这一点（一维决策边界将是一个点，把轴分成两个区域）。为了解决这个问题，我们可以添加额外的（可能是非线性的）特征来构建决策边界。考虑添加 x^2 作为特征后的相同数据集：

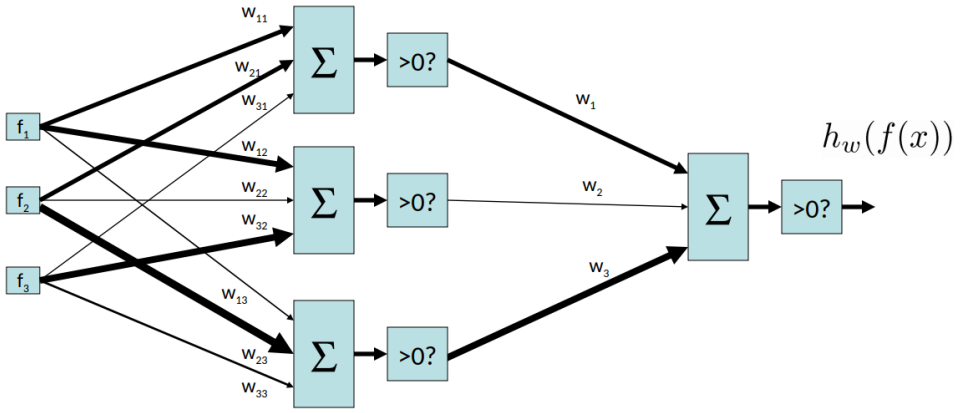


有了这条额外信息，我们现在能够在包含这些点的二维空间中构建一个线性分离器。

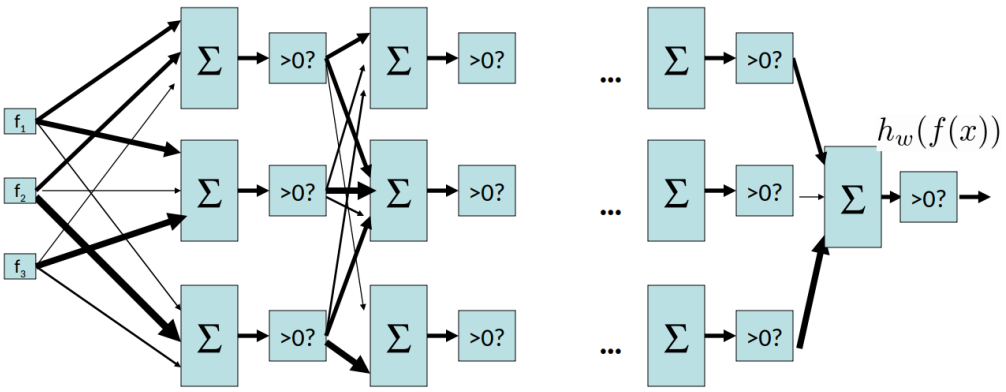
在这种情况下，我们通过手动向数据点添加有用特征，将数据映射到更高维空间，从而解决了问题。然而，在许多高维问题中，如图像分类，手动选择有用的特征是一项繁琐的任务。这需要特定领域的努力和专业知识，并且不利于跨任务泛化的目标。一个自然的想法是也学习这些特征化或变换函数，也许可以使用能够表示更广泛函数的非线性函数类。

多层感知器

让我们研究一下如何从原始感知器架构中导出更复杂的函数。考虑以下设置，一个两层感知器，它将另一个感知器的输出作为输入。



实际上，我们可以将其推广到一个 N 层感知器：



有了这个额外的结构和权重，我们可以表达更广泛的函数集。

通过增加模型的复杂度，我们相应地大幅提高了它的表达能力。多层感知器为我们提供了一种通用方式来表示更广泛的函数集。事实上，多层感知器是一种通用函数逼近器，能够表示任何实函数，这就只剩下为网络参数化选择最佳权重集的问题了。正式表述如下：

定理。（通用函数逼近器）具有足够数量神经元的两层神经网络可以以任意期望的精度逼近任何连续函数。

测量精度

进行 n 次预测后，二元感知器的精度可以表示为：

$$l^{acc}(w) = \frac{1}{n}\sum_{i=1}^{n}(\text{sgn}(w \cdot \mathbf{f}(\mathbf{x}_i)) == y_i)$$

其中 x_i 是数据点， $i, \mathbf{w}$ 是我们的权重向量， $\mathbf{f}$ 是我们从原始数据点导出特征向量的函数， y_i 是 $\mathbf{x}_i$ 的实际类别标签。在此上下文中， $\text{sgn}\left(x\right)$ 表示一个指示函数，当 x 为负时其值为 -1，当 x 为正时其值为 1。考虑到这种表示法，我们可以注意到上面的精度函数等同于将正确预测的总数除以预测的原始总数。

有时，我们希望得到一个比二元标签更具表现力的输出。那么，为我们想要分类的每个 N 类别生成一个概率就变得很有用，这个概率反映了我们对数据点属于每个可能类别的确定程度。为此，就像我们在多类逻辑回归的情况中所做的那样，我们从存储单个权重向量转变为为每个类别 j 存储一个权重向量，并使用softmax函数估计概率。softmax函数将分类为 $x^{(i)}$ 到类别 j 的概率定义为：

$$\sigma(\mathbf{x}_i)_j = \frac{e^{\mathbf{f}(\mathbf{x}_i)^T \mathbf{w}_j}}{\sum_{\ell=1}^{N} e^{\mathbf{f}(\mathbf{x}_i)^T \mathbf{w}_\ell}} = P(y_i = j|\mathbf{f}(\mathbf{x}_i); \mathbf{w}).$$

给定我们的函数 f 输出的一个向量，softmax进行归一化以输出一个概率分布。为了为我们的模型得出一个通用的损失函数，我们可以使用这个概率分布来生成一组权重的似然性表达式：

$$\ell(w) = \prod_{i=1}^{n} P(y_i|\mathbf{f}(\mathbf{x}_i); \mathbf{w}).$$

这个表达式表示特定权重集解释观测标签和数据点的可能性。我们希望找到能使这个量最大化的权重集。这等同于找到对数似然表达式的最大值（因为对数是单调函数，一个的最大化者将是另一个的最大化者）：

$$\log \ell(w) = \log \prod_{i=1}^{n} P(y_i|x_i; w) = \sum_{i=1}^{n} \log P(y_i|\mathbf{f}(\mathbf{x}_i); \mathbf{w}).$$

根据应用的不同，将其表述为对数概率之和可能更有用。在对数似然关于权重可微的情况下，我们可以使用梯度上升来对其进行优化。

多层前馈神经网络

我们现在引入（人工）神经网络的概念。这与多层感知器很相似，然而，我们在各个感知器节点之后选择应用不同的非线性函数。

请注意，正是这些额外的非线性因素使得整个网络呈现非线性且更具表现力（没有它们，多层感知器将仅仅是线性函数的组合，因此也是线性的）。对于多层感知器的情况，我们选择了一个阶跃函数：

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

让我们看看它的图像：

由于多种原因，这很难进行优化。首先，它不连续，其次，它在所有点的导数都为零。直观地说，这意味着我们不知道该朝着哪个方向寻找函数的局部最小值，这使得以平滑的方式最小化损失变得困难。
与其使用上述的阶跃函数，更好的解决方案是选择一个连续函数。对于这样的函数我们有很多选择，包括sigmoid函数（因其形状像希腊字母σ或's'而得名）以及修正线性单元（ReLU）。让我们在下面看看它们的定义和图像：Sigmoid：$\sigma\left(x\right) = \frac{1}{1+e^{-x}}$

$$ReLU\text{: } f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

多层感知器的输出计算方式与之前相同，不同之处在于，现在在每一层的输出处，我们应用新的非线性函数之一（作为神经网络架构的一部分进行选择），而不是初始指示函数。在实践中，非线性函数的选择是一种设计选择，通常需要进行一些实验，以便为每个具体用例选择一个合适的函数。

损失函数与多元优化

现在我们已经了解了前馈神经网络是如何构建并进行预测的，我们希望开发一种方法来训练它，类似于我们在感知器的情况中所做的那样，迭代地提高其准确性。为了做到这一点，我们需要能够衡量它们的性能。回到我们想要最大化的对数似然函数，鉴于我们的函数是可微的，我们可以推导出一种直观的算法来优化我们的权重。

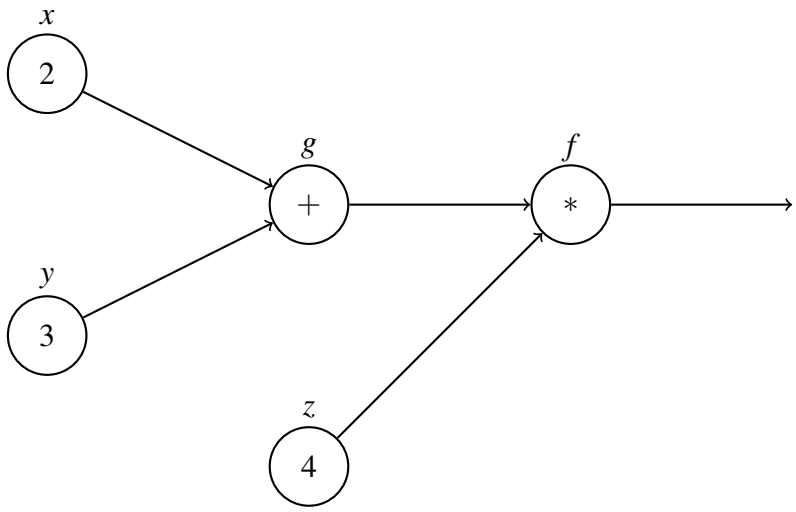
为了最大化我们的对数似然函数，我们对其求导以获得一个梯度向量，该向量由它对每个参数的偏导数组成：

$$\nabla_w \ell(\mathbf{w}) = \left[\frac{\partial \ell\ell(\mathbf{w})}{\partial \mathbf{w}_1}, \ldots, \frac{\partial \ell\ell(\mathbf{w})}{\partial \mathbf{w}_n}\right].$$

现在，我们可以使用前面描述的梯度上升法来找到参数的最优值。鉴于数据集通常很大，批量梯度上升是神经网络优化中梯度上升最流行的变体。

神经网络：反向传播

为了有效地计算神经网络中每个参数的梯度，我们将使用一种称为反向传播的算法。反向传播将神经网络表示为一个由运算符和操作数组成的依赖图，称为计算图，如下所示：



这种图结构使我们能够有效地计算网络在输入数据上的误差（损失），以及每个参数相对于损失的梯度。这些梯度可用于梯度下降，以调整网络参数并最小化训练数据上的损失。

链式法则

链式法则是微积分中的基本法则，它既推动了计算图的使用，又允许使用计算上可行的反向传播算法。从数学上讲，它表明对于一个变量 f，它是 n 个变量 $x_1, \ldots, x_n$ 的函数，并且每个 x_i 是 m 个变量 $t_1, \ldots, t_m$ 的函数，那么我们可以如下计算 f 相对于任何 t_i 的导数：

$$\frac{\partial f}{\partial t_i} = \frac{\partial f}{\partial x_1} \cdot \frac{\partial x_1}{\partial t_i} + \frac{\partial f}{\partial x_2} \cdot \frac{\partial x_2}{\partial t_i} + \ldots + \frac{\partial f}{\partial x_n} \cdot \frac{\partial x_n}{\partial t_i}.$$

在计算图的背景下，这意味着要计算给定节点 t_i 相对于输出 f 的梯度，我们需要对其子节点 (t_i) 项进行求和。

反向传播算法

图1展示了一个用于使用值 $x = 2, y = 3, z = 4$ 计算 $(x + y) * z$ 的示例计算图。我们将写成 $g = x + y$ 和 $f = g * z$。绿色的值是每个节点的输出，我们在正向传播中计算这些值，在正向传播中，我们将每个节点的操作应用于来自其父节点的输入值。

每个节点之后红色的值给出了由计算图计算出的函数梯度，这些梯度是在反向传播中计算得到的：每个节点之后的值是最后一个节点 f 的值关于该节点处变量的偏导数。例如， g 之后的红色值4是 $\frac{\partial f}{\partial g}$ ， x 之后的红色值4是 $\frac{\partial f}{\partial x}$ 。在我们的简单示例中， f 只是一个乘法节点，它输出其两个输入操作数的乘积，但在实际的神经网络中，最终节点通常会计算我们试图最小化的损失值。

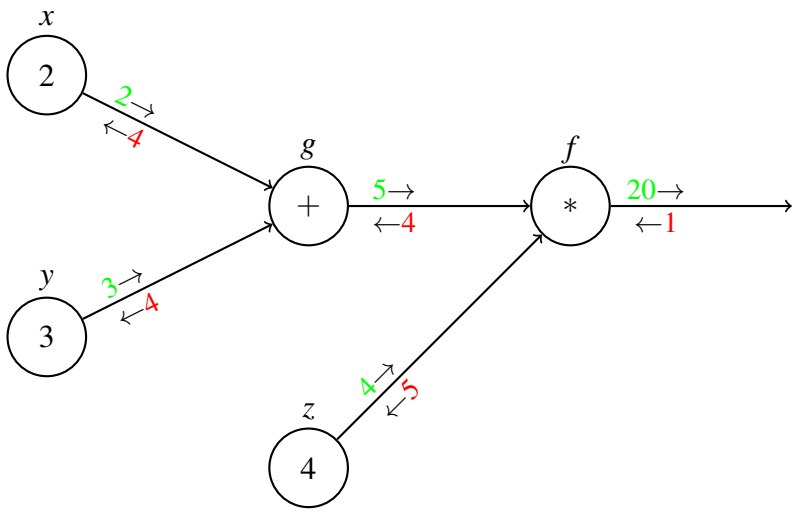


图1：一个用于使用值 $x = 2, y = 3, z = 4$ 计算 $(x + y) * z$ 的计算图。

反向传播通过从最终节点（由于 $\frac{\partial f}{\partial f} = 1$ 其梯度为1）开始，并在图中向后传递和更新梯度来计算梯度。直观地说，每个节点的梯度衡量该节点值的变化对最终节点值的变化有多大贡献。这将是该节点对其子节点变化的贡献量与子节点对最终节点变化的贡献量的乘积。每个节点接收并组合来自其子节点的梯度，并根据节点的输入和节点的操作更新此组合梯度，然后将更新后的梯度向后传递给其父节点。计算图是直观展示微积分中链式法则重复应用的好方法，因为这一过程是神经网络反向传播所必需的。

我们在反向传播过程中的目标是确定输出相对于每个输入的梯度。如图1所示，在这种情况下，我们要计算梯度 $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}$ 和 $\frac{\partial f}{\partial z}$：

1. 由于 f 是我们的最终节点，它的梯度为 $\frac{\partial f}{\partial f} = 1$。然后我们计算它的子节点 g 和 z 的梯度。我们有 $\frac{\partial f}{\partial g} = \frac{\partial}{\partial g}(g \cdot z) = z = 4$ 和 $\frac{\partial f}{\partial z} = \frac{\partial}{\partial z}(g \cdot z) = g = 5$。

2. 现在我们可以向上游推进，来计算 x 和 y 的梯度。为此，我们将使用链式法则，并复用我们刚刚为 $g, \frac{\partial f}{\partial g}$ 计算出的梯度。

3. 对于 x，根据链式法则我们有 $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$ ——来自 g 的梯度与该节点处 x 的偏导数的乘积。我们有 $\frac{\partial g}{\partial x} = \frac{\partial}{\partial x}(x + y) = \frac{\partial}{\partial x}x + \frac{\partial}{\partial x}y = 1 + 0$，所以 $\frac{\partial f}{\partial x} = 4 \cdot 1 = 4$。直观地说，$x$ 的变化对 f 的变化所做的贡献量，是 g 的变化对 f 的变化所做的贡献量，与 x 的变化对 g 的变化所做的贡献量的乘积。

4. 计算输出相对于 y 的梯度的过程几乎相同。对于 y，根据链式法则我们有 $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial y}$ ——来自 g 的梯度与该节点处 y 的偏导数的乘积。我们有 $\frac{\partial g}{\partial y} = \frac{\partial}{\partial y}(x + y) = \frac{\partial}{\partial y}x + \frac{\partial}{\partial y}y = 0 + 1$，所以 $\frac{\partial f}{\partial y} = 4 \cdot 1 = 4$。

由于一般来说，节点的反向传播步骤取决于该节点的输入（这些输入是在正向传播中计算得到的），以及由该节点的子节点在当前节点"下游"计算得到的梯度（在反向传播中更早计算得到），为了提高效率，我们在计算图中缓存所有这些值。总的来说，计算图上的正向传播和反向传播共同构成了反向传播算法。

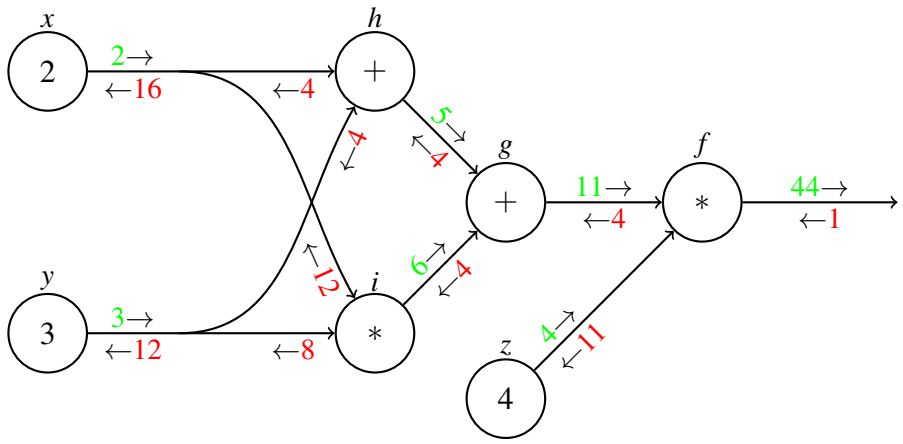


图2：用于计算 $((x+y)+(x\cdot y))\cdot z$ 的计算图，其中包含 $x=2, y=3, z=4$ 。

对于一个对具有多个子节点的节点应用链式法则的示例，考虑图2中的计算图，它表示 $((x+y)+(x\cdot y))\cdot z$ ，其中 $x=2, y=3, z=4$.x 和 y 分别在2个操作中使用，因此每个都有两个子节点。根据链式法则，它们的梯度值是由其子节点为它们计算的梯度之和（即梯度值在路径交汇处相加）。例如，要计算 x 的梯度，我们有

$$\frac{\partial f}{\partial x}=\frac{\partial f}{\partial h}\frac{\partial h}{\partial x}+\frac{\partial f}{\partial i}\frac{\partial i}{\partial x}=4\cdot 1+4\cdot 3=4+12=16.$$

既然我们已经有了一种为网络的所有参数计算梯度的方法，那么我们就可以使用梯度下降方法来优化参数，以便在训练数据上获得高精度。例如，假设我们设计了某个分类网络，用于为数据点 y 输出类别 x 的概率，并且有 m 个不同的训练数据点（有关更多信息，请参阅"测量准确率"部分）。设 $\mathbf{w}$ 为我们网络的所有参数。我们想要找到参数 $\mathbf{w}$ 的值，以最大化我们数据的真实类别概率的似然性，因此我们有以下要对其运行梯度上升的函数：

$$\ell\ell(\mathbf{w})=\log\prod_{i=1}^{m}P(y^{(i)}\mid x^{(i)};\mathbf{w})=\sum_{i=1}^{m}\log P(y^{(i)}\mid x^{(i)};\mathbf{w})$$

其中 $x^{(1)},\ldots,x^{(m)}$ 是我们训练集中的 m 个数据点。

为了使对数似然的负值最小化，在梯度下降的每次迭代中，我们使用数据点 $x^{(1)},\ldots,x^{(m)}$ 来计算参数 $\mathbf{w}$ 的梯度，更新参数，然后重复此过程，直到参数收敛（此时我们达到了函数的局部最小值）。
神经网络是强大的（且通用的！）函数逼近器，但设计和训练起来可能很困难。深度学习领域有许多正在进行的研究，专注于神经网络设计的各个方面，例如：

1. 网络架构 - 设计一个适合特定问题的网络（选择激活函数、层数等）

2. 学习算法 - 如何找到能使损失函数值较低的参数，这是个难题，因为梯度下降是一种贪心算法，而神经网络可能有许多局部最优解
3. 泛化与迁移学习 - 由于神经网络有许多参数，很容易过拟合训练数据 - 如何确保它们在未见过的测试数据上也有低损失？

Summary

在本笔记中，我们介绍了逻辑回归及其紧密变体——多类逻辑回归。

我们还研究了如何使用梯度方法解决优化问题。

结合这些概念，我们引出了神经网络的概念，神经网络本质上是具有中间非线性激活函数的多层感知器。这些网络在逼近函数方面极其强大，并使用梯度下降进行训练。