第一章　前馈神经网络

在人工智能领域，**人工神经网络**（Artificial Neural Network，ANN）是指一系列受生物学和神经学启发的数学模型。这些模型主要是通过对人脑的神经元网络进行抽象，构建人工神经元，并按照一定拓扑结构来建立人工神经元之间的连接，来模拟生物神经网络。在人工智能领域，人工神经网络也常常简称为神经网络（Neural Network，NN）或神经模型。

到目前为止，研究者已经提出了多种多样的神经网络模型，比如：感知器、前馈网络、卷积网络、循环网络、自组织映射、Hopfield 网络、Boltzmann 机等。这些神经网络模型的差异主要在于神经元的激活规则、网络连接的拓扑结构以及参数的学习规则等。

1.1　人工神经网络

人工神经网络主要由大量的神经元以及它们之间的有向连接构成。因此考虑三方面：

- 神经元的激活规则
- 网络的拓扑结构
- 学习算法。如果两个神经元间的存在连接，经过这条连接边的信息，称之为权重，

1.1.1　神经元

人工神经元（Artificial Neuron）是构成人工神经网络的基本单元。人工神经元和感知器非常类似，也是模拟生物神经元特性，接受一组输入信号并产出

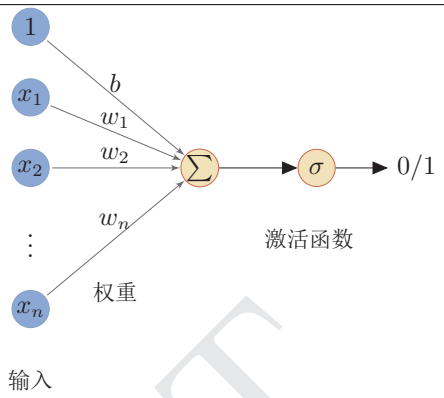


图 1.1: 人工神经元模型

输出。生物神经元有一个阀值，当神经元所获得的输入信号的积累效果超过阀值时，它就处于兴奋状态；否则，应该处于抑制状态。

净输入也叫**净活性值**（net activation）。

我们用 $\mathbf{x} = (x_1, x_2, \cdots, x_n)$ 来表示人工神经元的一组输入，用**净输入** z 表示一个神经元所获得的输入信号 x 的加权和，

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{1.1}$$

$$= \mathbf{w}^\top \mathbf{x} + b, \tag{1.2}$$

其中，$\mathbf{w} = (w_1, w_2, \cdots, w_n)$ 是 n 维的权重向量，b 是偏置。

净输入 z 在经过一个非线性函数后，人工神经元输出它的**活性值**（activation）a，

$$a = f(z), \tag{1.3}$$

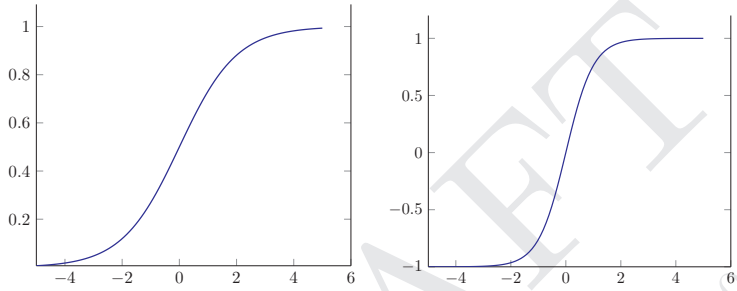
这里的非线性函数 f 也称为激活函数（activation function），有 sigmoid 型函数、非线性斜面函数等。

人工神经元的结构如图1.1所示。如果我们设激活函数 f 为 0 或 1 的阶跃函数，人工神经元就是感知器。

1.1.2 激活函数

数学小知识 | 饱和

对于函数 $f(x)$，若 $x \to -\infty$ 是，其导数 $f'(x) \to 0$，则称其为**左饱和**。若 $x \to +\infty$ 是，其导数 $f'(x) \to 0$，则称其为**右饱和**。当同时满足左、右饱和时，就称其为饱和。



(a) logistic 函数 (b) Tanh 函数

图 1.2: Sigmoid 型激活函数

为了增强网络的表达能力，我们需要使用连续非线性**激活函数**（Activation Function）。因为连续非线性激活函数可导，所以可以用最优化的方法来求解。

下面介绍几个在神经网络中常用的激活函数。

Logistic 函数 Logistic 函数是一种 sigmoid 型函数。其定义为 $\sigma(x)$

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{1.4}$$

图1.3a给出了 logistic 函数的形状。Logistic 函数可以看成是一个"挤压"函数，把一个实数域的输入"挤压"到 $(0,1)$。当输入值在 0 附近时，sigmoid 型函数近似为线性函数；当输入值靠近两端时，对输入进行抑制。输入越小，越接近于 0；输入越大，越接近于 1。这样的特点也和生物神经元类似，对一些输入会产生兴奋（输出为 1），对另一些输入产生抑制（输出为 0）。和感知器使用的阶跃激活函数相比，logistic 函数是连续可导的，其数学性质更好。

Sigmoid 型函数是指一类 S 型曲线函数，常用的 sigmoid 型函数有 logistic 函数和 tanh 函数。

Tanh 函数 Tanh 函数是也一种 sigmoid 型函数。其定义为

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.5}$$

Tanh 函数可以看作是放大并平移的 logistic 函数：

$$\tanh(x) = 2\sigma(2x) - 1. \tag{1.6}$$

图1.3b给出了 tanh 函数的形状，其值域是 $(-1, 1)$。

Hard-Logistic 和 Hard-Tanh 函数 Logistic 函数和 tanh 函数都是 sigmoid 型函数，具有饱和性，但是计算开销较大。因为这两个函数都是在中间（0 附近）近似线性，两端饱和。因此，这两个函数可以通过分段函数来近似。

以 logistic 函数 $\sigma(x)$ 为例，其导数为 $\sigma'(x) = \sigma(x)(1 - \sigma(x))$。logistic 函数在 0 附近的一阶泰勒展开（Taylor expansion）为

$$g_l(x) \approx \sigma(0) + \sigma'(0) \tag{1.7}$$

$$= 0.25x + 0.5. \tag{1.8}$$

$$\text{hard-logistic}(x) = \begin{cases} 1 & \text{for } g_l(x) \geq 1 \\ g(x) & 0 < g_l(x) < 1 \\ 0 & \text{for } g_l(x) \leq 0 \end{cases} \tag{1.9}$$

公式1.9也可以写为

$$\text{hard-logistic}(x) = \max(\min(g_l(x), 1), 0). \tag{1.10}$$

同理，tanh 函数在 0 附近的一阶泰勒展开为

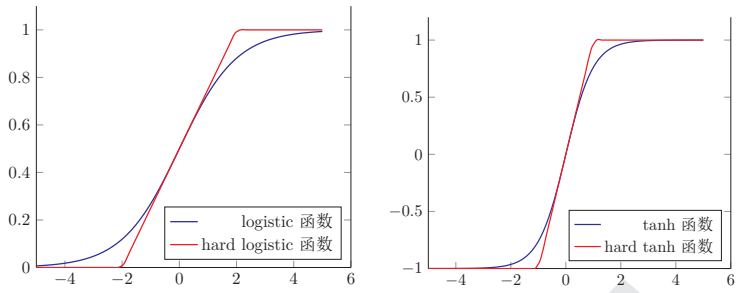
$$g_t(x) \approx \tanh(0) + \tanh'(0) \tag{1.11}$$

$$= x. \tag{1.12}$$

tanh 函数也可以用分段函数 $\text{hard-tanh}(x)$ 来近似。

$$\text{hard-tanh}(x) = \max(\min(g_t(x), 1), -1). \tag{1.13}$$

图1.3给出了 Hard-Logistic 和 Hard-Tanh 函数两种函数的形状。



(a) Hard logistic 函数 (b) Hard tanh 函数

图 1.3: Sigmoid 型激活函数近似

Rectifier 函数 Rectifier 函数 [Glorot et al., 2011] 也是一个分段函数，定义为

$$\text{rectifier}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{1.14}$$

$$= \max(0, x). \tag{1.15}$$

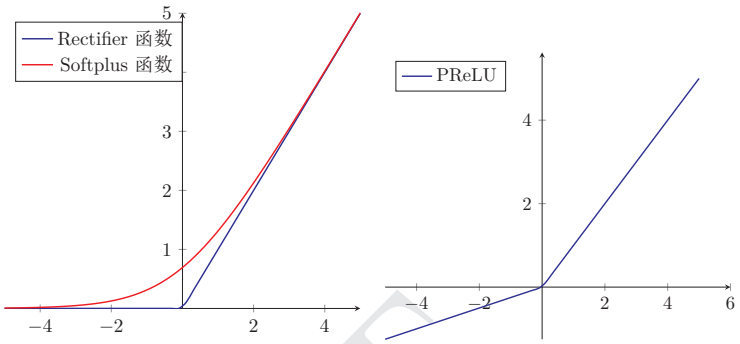
Rectifier 函数为左饱和函数，在 $x < 0$ 时饱和，在 $x > 0$ 时导数为 1。采用 Rectifier 函数，会使得神经网络只有比较、加、乘操作，计算上也更加高效。

此外，rectifier 函数被认为有生物上的解释性。神经科学家发现神经元具有**单侧抑制**、**宽兴奋边界**、**稀疏激活性**等特性。

稀疏激活性 生物神经元只对少数输入信号选择性响应，大量信号被屏蔽了。因为随机初始化的原因，sigmoid 系函数同时近乎有 50% 的神经元被激活，这不符合神经科学的发现。而 rectifier 函数却具备很好的稀疏性。

采用 rectifier 函数的单元也叫作**修正线性单元**（rectified linear unit，ReLU）[Nair and Hinton, 2010]。

带参数的 ReLU 在实际使用中，ReLU 存在很多变种。其中一种是带参数的 ReLU（Parametric ReLU，PReLU）。和之前的激活函数不同，PReLU 引入一个可学习的参数，不同神经元可以有不同的参数。对于第 i 个神经元，其 PReLU



(a) ReLU 和 softplus 函数 (b) PReLU

图 1.4: ReLU、PReLU 和 softplus 函数

的定义为

$$\text{PReLU}_i(x) = \begin{cases} x & x > 0 \\ a_i x & x \leq 0 \end{cases}, \tag{1.16}$$

其中，a_i 为 $x \leq 0$ 时函数的斜率。因此，PReLU 是非饱和函数。如果 $a_i = 0$，那么 PReLU 就退化为 ReLU。如果 a_i 为一个很小的常数，则 PReLU 可以看作是 Leaky ReLU(LReLU)。

这里，不同神经元对应的参数都可以是不同的，也可以一组神经元共享一个参数。

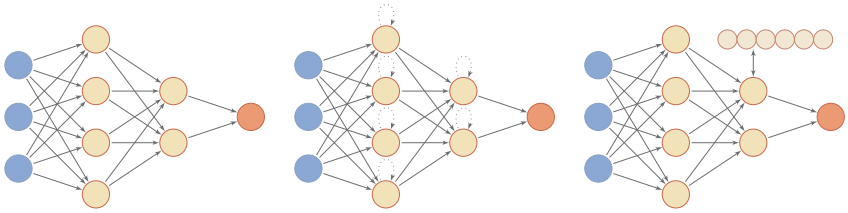
Softplus 函数 Softplus 函数 [Dugas et al., 2001] 可以看作是 rectifier 函数的平滑版本，其定义为

$$\textbf{softplus}(x) = \log(1 + e^x) \tag{1.17}$$

softplus 函数其导数刚好是 logistic 函数。softplus 虽然也有具有单侧抑制、宽兴奋边界的特性，却没有稀疏激活性。

图1.4给出了 ReLU、PReLU 以及 softplus 函数的示例。

Maxout 单元 Maxout 单元Goodfellow et al. [2013] 也是一种分段线性函数。和 Sigmoid 型函数、ReLU 等激活函数不同，Maxout 单元的输入就是神经元的



(a) 前馈网络 (b) 反馈网络 (c) 记忆网络

图 1.5: 三种不同的网络模型

全部输入，是一个向量 $\mathbf{x} = (x_1, x_2, \cdots, x_n)$，而其它激活函数的输入是神经元的净输入 z，是一个标量。

每个 Maxout 单元有 k 个权重向量 $\mathbf{w}_i \in \mathbb{R}^n$ 和偏置 b_i $(i \in [1, k])$。对于输入 $\mathbf{x}$，可以得到 k 个净输入 z

$$z_i = \mathbf{w}_i \mathbf{x} + b_i \tag{1.18}$$

$$= \sum_{j=1}^{n} w_{i,j} x_j + b_i, \tag{1.19}$$

其中，$\mathbf{w}_i = [w_{i,1}, \cdots, w_{i,n}]$ 为第 i 个权重向量。

Maxout 单元的非线性函数定义为

$$\text{maxout}(\mathbf{x}) = \max_{i \in [1,k]} (z_i). \tag{1.20}$$

Maxout 单元不单是净输入到输出之间的非线性映射，而是整体学习输入到输出之间的非线性映射关系。Maxout 激活函数可以看作任意凸函数的分段线性近似，并且在有限的点上是不可微的。

采用 maxout 单元的神经网络也就做 **maxout 网络**。

1.1.3 网络结构

一个生物神经细胞的功能比较简单，而人工神经元只是生物神经细胞的理想化和简单实现，功能更加简单。要想模拟人脑的能力，单一的神经元是远远不够的，需要通过很多神经元一起协作来完成复杂的功能。这样通过一定的连

接方式或信息传递方式进行协作的神经元可以看作是一个网络，叫做**人工神经网络**，简称**神经网络**。

到目前为止，研究者已经发明了各种各样的神经网络结构。目前常用的神经网络结构有以下三种：

前馈网络 网络中各个神经元按接受信息的先后分为不同的组。每一组可以看作一个神经层。每一层中的神经元接受前一层神经元的输出，并输出到下一层神经元。整个网络中的信息是朝一个方向传播，没有反向的信息传播。前馈网络可以用一个有向无环路图表示。前馈网络可以看作一个**函数**，通过简单非线性函数的多次复合，实现输入空间到输出空间的复杂映射。这种网络结构简单，易于实现。前馈网络包括全连接前馈网络和卷积神经网络等。

反馈网络 网络中神经元不但可以接收其它神经元的信号，也可以接收自己的反馈信号。和前馈网络相比，反馈网络在不同的时刻具有不同的状态，具有记忆功能，因此反馈网络可以看作一个**程序**，也具有更强的计算能力。反馈神经网络可用一个完备的无向图来表示。

记忆网络 记忆网络在前馈网络或反馈网络的基础上，引入一组记忆单元，用来保存中间状态。同时，根据一定的取址、读写机制，来增强网络能力。和反馈网络相比，反馈网络具有更强的记忆功能。

图1.5给出了前馈网络、反馈网络和记忆网络的网络结构示例。

1.2 前馈神经网络

给定一组神经元，我们可以以神经元为节点来构建一个网络。不同的神经网络模型有着不同网络连接的拓扑结构。一种比较直接的拓扑结构是前馈网络。**前馈神经网络**（Feedforward Neural Network）是最早发明的简单人工神经网络。

在前馈神经网络中，各神经元分别属于不同的层。每一层的神经元可以接收前一层神经元的信号，并产生信号输出到下一层。第一层叫**输入层**，最后一层叫**输出层**，其它中间层叫做**隐藏层**。整个网络中无反馈，信号从输入层向输出层单向传播，可用一个有向无环图表示。

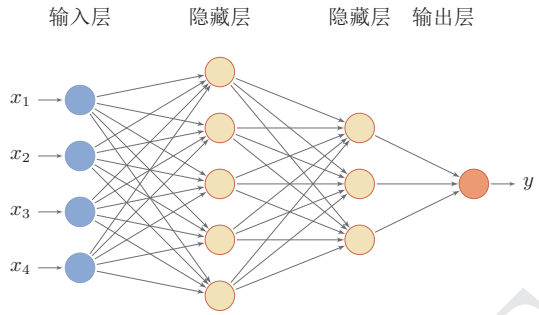


图 1.6: 多层神经网络

前馈神经网络也经常称为**多层感知器**（Multilayer Perceptron，MLP）。但**多层感知器**的叫法并不是否合理，因为前馈神经网络其实是由多层的logistic回归模型（连续的非线性函数）组成，而不是有多层的感知器（不连续的非线性函数）组成 [Bishop, 2006]。

图1.6给出了前馈神经网络的示例。

1.2.1 前馈计算

给定一个前馈神经网络，我们用下面的记号来描述这样网络。

- L：表示神经网络的层数；
- n^l：表示第l层神经元的个数；
- $f_l(\cdot)$：表示l层神经元的激活函数；
- $W^{(l)} \in \mathbb{R}^{n^l \times n^{l-1}}$：表示$l-1$层到第$l$层的权重矩阵；
- $\mathbf{b}^{(l)} \in \mathbb{R}^{n^l}$：表示$l-1$层到第$l$层的偏置；
- $\mathbf{z}^{(l)} \in \mathbb{R}^{n^l}$：表示$l$层神经元的净输入（净活性值）；
- $\mathbf{a}^{(l)} \in \mathbb{R}^{n^l}$：表示$l$层神经元的输出（活性值）。

前馈神经网络通过下面公式进行信息传播。

$$\mathbf{z}^{(l)} = W^{(l)} \cdot \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \tag{1.21}$$

$$\mathbf{a}^{(l)} = f_l(\mathbf{z}^{(l)}) \tag{1.22}$$

公式1.21和1.22也可以合并写为：

$$\mathbf{z}^{(l)} = W^{(l)} \cdot f_{l-1}(\mathbf{z}^{(l-1)}) + \mathbf{b}^{(l)} \tag{1.23}$$

或者

$$\mathbf{a}^{(l)} = f_l(W^{(l)} \cdot \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}). \tag{1.24}$$

这样，前馈神经网络可以通过逐层的信息传递，得到网络最后的输出 $\mathbf{a}^L$。整个网络可以看作一个复合函数 $\mathbf{y} = f(\mathbf{x}; W, \mathbf{b}))$，将输入 x 作为第 1 层的输入 $\mathbf{a}^{(0)}$，将第 L 层的输出 $\mathbf{a}^{(L)}$ 作为整个函数的输出 $\mathbf{y}$。

$$\mathbf{x} = \mathbf{a}^{(0)} \to \mathbf{z}^{(1)} \to \mathbf{a}^{(1)} \to \mathbf{z}^{(2)} \to \cdots \to \mathbf{a}^{(L-1)} \to \mathbf{z}^{(L)} \to \mathbf{a}^{(L)} = \mathbf{y} \tag{1.25}$$

1.2.2 应用到机器学习

在了解前馈网络的结构之后，我们可以将前馈网络应用于机器学习。

以监督学习为例，假设我们有一组训练样本 $(\mathbf{x}^{(i)}, y^{(i)}), 1 \leq i \leq N$，我们可以将前馈神经网络作为一个非线性映射函数 $\mathbf{y} = f(\mathbf{x}; W, \mathbf{b})$，并且来拟合 $\mathbf{x}^{(i)}$ 到 $y^{(i)}$ 之间的映射关系。

对于两类分类问题，我们可以在

注意，最后一层只用一个神经元，其输出可以

1.3 反向传播算法

给定一组样本 $(\mathbf{x}^{(i)}, y^{(i)}), 1 \leq i \leq N$，用前馈神经网络的输出为 $f(\mathbf{x}|\mathbf{w}, \mathbf{b})$，目标函数为：

$$\mathcal{R}(W, \mathbf{b}) = \sum_{i=1}^{N} \mathcal{L}(y^{(i)}, f(\mathbf{x}^{(i)}; W, \mathbf{b})) + \frac{1}{2}\lambda \|W\|_F^2, \tag{1.26}$$

$$= \sum_{i=1}^{N} \mathcal{L}(W, \mathbf{b}; \mathbf{x}^{(i)}, y^{(i)}) + \frac{1}{2}\lambda \|W\|_F^2, \tag{1.27}$$

这里，W 和 $\mathbf{b}$ 包含了每一层的权重矩阵和偏置向量；$\|W\|_F^2$ 是正则化项，用来防止过拟合；λ 是为正数的超参。λ 越大，W 越接近于 0。这里的 $\|W\|_F^2$ 一般使用 Frobenius 范数：

$$\|W\|_F^2 = \sum_{l=1}^{L} \sum_{j=1}^{n^{l+1}} \sum_{j=}^{n^l} W_{ij}^{(l)}. \tag{1.28}$$

我们的目标是最小化 $\mathcal{L}(W, \mathbf{b}; \mathbf{x}, y)$。如果采用梯度下降方法，我们可以用如下方法更新参数：

$$W^{(l)} = W^{(l)} - \alpha \frac{\partial \mathcal{L}(W, \mathbf{b})}{\partial W^{(l)}}, \tag{1.29}$$

$$= W^{(l)} - \alpha \sum_{i=1}^{N} \left(\frac{\partial \mathcal{L}(W, \mathbf{b}; \mathbf{x}^{(i)}, y^{(i)})}{\partial W^{(l)}}\right) - \alpha\lambda W, \tag{1.30}$$

$$\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \alpha \frac{\partial \mathcal{L}(W, \mathbf{b}; \mathbf{x}^{(i)}, y^{(i)})}{\partial \mathbf{b}^{(l)}}, \tag{1.31}$$

$$= \mathbf{b}^{(l)} - \alpha \sum_{i=1}^{N} \left(\frac{\partial \mathcal{L}(W, \mathbf{b}; \mathbf{x}^{(i)}, y^{(i)})}{\partial \mathbf{b}^{(l)}}\right), \tag{1.32}$$

$$\tag{1.33}$$

这里 α 是参数的更新率。

我们首先来看下 $\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial W^{(l)}}$ 怎么计算。

根据链式法则，$\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial W_{ij}^{(l)}}$ 可以写为

$$\frac{\partial \mathcal{L}(W, \mathbf{b}; \mathbf{x}, y)}{\partial W_{ij}^{(l)}} = \left(\frac{\partial \mathcal{L}(W, \mathbf{b}; \mathbf{x}, y)}{\partial \mathbf{z}^{(l)}}\right)^{\top} \frac{\partial \mathbf{z}^{(l)}}{\partial W_{ij}^{(l)}}. \tag{1.34}$$

公式1.34的第一项是为目标函数关于第 l 层的神经元 $\mathbf{z}^{(l)}$ 的偏导数，我们称为**误差项** $\delta^{(l)}$：

$$\delta^{(l)} = \frac{\partial \mathcal{L}(W, \mathbf{b}; \mathbf{x}, y)}{\partial \mathbf{z}^{(l)}} \in \mathbb{R}^{n^{(l)}}. \tag{1.35}$$

误差项 $\delta^{(l)}$ 来表示第 l 层的神经元对最终误差的影响，也反映了最终的输出对第 l 层的神经元对最终误差的敏感程度。

公式1.34中的第二项是 l 层的神经元 $\mathbf{z}^{(l)}$ 关于参数 $W_{ij}^{(l)}$ 的偏导数。

我们分别来计算这两个偏导数。

意这里的正则化只包含权重参数而不包含偏置

链式法则参见第??页的公式??

计算误差项 $\delta^{(l)}$ 我们先来看下第 l 层的**误差项** $\delta^{(l)} = \frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial \mathbf{z}^{(l)}}$ 怎么计算。

$$\delta^{(l)} \triangleq \frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial \mathbf{z}^{(l)}} \tag{1.36}$$

$$= \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \cdot \frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial \mathbf{z}^{(l+1)}} \tag{1.37}$$

$$= \mathbf{diag}(f_l'(\mathbf{z}^{(l)})) \cdot (W^{(l+1)})^\top \cdot \delta^{(l+1)} \tag{1.38}$$

$$= f_l'(\mathbf{z}^{(l)}) \odot ((W^{(l+1)})^\top \delta^{(l+1)}), \tag{1.39}$$

其中 $\odot$ 是向量的点积运算符，表示每个元素相乘。

上述推导中，关键是公式1.37到公式1.38的推导。公式1.37中有三项，第三项根据定义为 $\delta^{(l+1)}$。

第二项因为 $\mathbf{z}^{(l+1)} = W^{(l+1)} \cdot \mathbf{a}^{(l)} + \mathbf{b}^{(l)}$，所以

$$\frac{\partial \mathbf{z}^{(l+1)}}{\partial a^{(l)}} = (W^{(l+1)})^\top. \tag{1.40}$$

第一项因为 $\mathbf{a}^{(l)} = f_l(\mathbf{z}^{(l)})$，而 $f_l(\cdot)$ 为按位计算的函数。因此

$$\frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} = \frac{\partial f_l(\mathbf{z}^{(l)})}{\partial \mathbf{z}^{(l)}} \tag{1.41}$$

$$= \mathbf{diag}(f_l'(\mathbf{z}^{(l)})). \tag{1.42}$$

从公式1.39可以看出，第 l 层的误差项可以通过第 $l+1$ 层的误差项计算得到。这就是误差的**反向传播**（Backpropagation，BP）。反向传播算法的含义是：第 l 层的一个神经元的误差项（或敏感性）是所有与该神经元相连的第 $l+1$ 层的神经元的误差项的权重和。然后，在乘上该神经元激活函数的梯度。

计算 $\frac{\partial \mathbf{z}^{(l)}}{\partial W_{ij}^{(l)}}$ 我们先来计算公式1.34中的第二项 $\frac{\partial \mathbf{z}^{(l)}}{\partial W_{ij}^{(l)}}$。

因为 $\mathbf{z}^{(l)} = W^{(l)} \cdot \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$，所以

$$\frac{\partial \mathbf{z}^{(l)}}{\partial W_{ij}^{(l)}} = \frac{\partial (W^{(l)} \cdot \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)})}{\partial W_{ij}^{(l)}} = \begin{bmatrix} 0 \\ \vdots \\ a_j^{(l-1)} \\ \vdots \\ 0 \end{bmatrix}. \tag{1.43}$$

在得到上面两个偏导数之后，公式1.34可以写为

$$\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial W_{ij}^{(l)}} = (\delta^{(l)})^{\top}\mathbf{a}^{(l-1)} \tag{1.44}$$

$$= \delta_i^{(l)} a_j^{(l-1)}. \tag{1.45}$$

进一步，$\mathcal{L}(W,\mathbf{b};\mathbf{x},y)$关于第$l$层权重$W^{(l)}$的梯度为

$$\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial W^{(l)}} = \delta^{(l)}(\mathbf{a}^{(l-1)})^{\top}. \tag{1.46}$$

同理可得，$\mathcal{L}(W,\mathbf{b};\mathbf{x},y)$关于第$l$层偏置$W^{(l)}$的梯度为

$$\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x},y)}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}. \tag{1.47}$$

在计算出每一层的误差项之后，我们就可以得到每一层参数的梯度。因此，前馈神经网络的训练过程可以分为以下三步：

1. 前馈计算每一层的状态和激活值，直到最后一层；

2. 反向传播计算每一层的误差项；

3. 计算每一层参数的偏导数，并更新参数。

← 第i行

具体的训练过程如算法1.1所示，也叫**反向传播算法**。

算法 1.1: 反向传播算法

输入: 训练集: $(\mathbf{x}^{(i)}, y^{(i)}), i = 1, \cdots, N$，最大迭代次数：$T$

输出: $W, \mathbf{b}$

初始化 $W, \mathbf{b}$;

for $t = 1 \cdots T$ **do**

  for $i = 1 \cdots N$ **do**

    (1) 前馈计算每一层的状态和激活值，直到最后一层;

    (2) 用公式1.39反向传播计算每一层的误差 $\delta^{(l)}$;

    (3) 用公式1.46和1.47每一层参数的导数;

$$\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x}^{(i)},y^{(i)})}{\partial W^{(l)}} = \delta^{(l)}(\mathbf{a}^{(l-1)})^{\top};$$

$$\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x}^{(i)},y^{(i)})}{\partial \mathbf{b}^{(l)}} = \delta^{(l)};$$

    (4) 更新参数;

$$W^{(l)} = W^{(l)} - \alpha \sum_{i=1}^{N} \left(\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x}^{(i)},y^{(i)})}{\partial W^{(l)}}\right) - \alpha\lambda W;$$

$$\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \alpha \sum_{i=1}^{N} \left(\frac{\partial \mathcal{L}(W,\mathbf{b};\mathbf{x}^{(i)},y^{(i)})}{\partial \mathbf{b}^{(l)}}\right);$$

  end

end

1.4 梯度消失问题

在神经网络中误差反向传播的迭代公式为

$$\delta^{(l)} = f_l'(\mathbf{z}^{(l)}) \odot ((W^{(l+1)})^{\top}\delta^{(l+1)}, \tag{1.48}$$

其中需要用到激活函数 f_l 的导数。

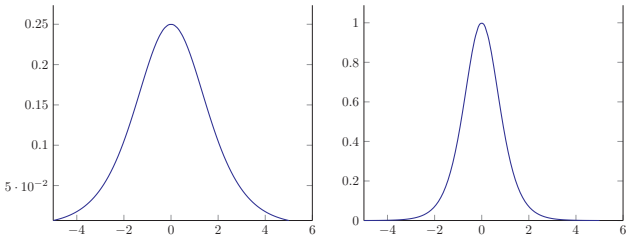
误差从输出层反向传播时，在每一层都要乘以该层的激活函数的导数。

当我们使用sigmoid型函数：logistic函数$\sigma(x)$或tanh函数时，其导数为

$$\sigma'(x) = \sigma(x)(1-\sigma(x)) \in [0, 0.25] \tag{1.49}$$

$$\tanh'(x) = 1 - (\tanh(x))^2 \in [0, 1]. \tag{1.50}$$

我们可以看到，sigmoid型函数导数的值域都小于1。这样误差经过每一层传递都会不断衰减。当网络层数很深时，梯度就会不停的衰减，甚至消失，使得整



(a) logistic 函数的导数 (b) tanh 函数的导数

图 1.7: 激活函数的导数

个网络很难训练。这就是所谓的**梯度消失问题**（Vanishing Gradient Problem），也叫**梯度弥散**。

减轻梯度消失问题的一个方法是使用线性激活函数（比如 rectifier 函数）或近似线性函数（比如 softplus 函数）。这样，激活函数的导数为 1，误差可以很好地传播，训练速度得到了很大的提高。

Sigmoid 的软饱和性，使得深度神经网络在二三十年里一直难以有效的训练，是阻碍神经网络发展的重要原因。具体来说，由于在后向传递过程中，sigmoid 向下传导的梯度包含了一个 f'(x) 因子（sigmoid 关于输入的导数），因此一旦输入落入饱和区，f'(x) 就会变得接近于 0，导致了向底层传递的梯度也变得非常小。此时，网络参数很难得到有效训练。这种现象被称为梯度消失。一般来说，sigmoid 网络在 5 层之内就会产生梯度消失现象 [2]。梯度消失问题至今仍然存在，但被新的优化方法有效缓解了，例如 DBN 中的分层预训练，Batch Normalization 的逐层归一化，Xavier 和 MSRA 权重初始化等代表性技术。Sigmoid 的饱和性虽然会导致梯度消失，但也有其有利的一面。例如它在物理意义上最为接近生物神经元。(0, 1) 的输出还可以被表示作概率，或用于输入的归一化，代表性的如 Sigmoid 交叉熵损失函数

1.5 训练方法

随机梯度下降 对于大规模数据，很难使用批量梯度下降。随机梯度下降（Stochastic Gradient Descent，SGD）

随机梯度下降：每次迭代（随机）（选择一个样本计算梯度，并更新参数。

$$\mathbf{a}_{t+1} = \mathbf{a}_t - \lambda \frac{\partial \mathcal{R}\big(\theta_t; x^{(t)}, y^{(t)}\big)}{\partial \theta}, \tag{1.51}$$

Mini-batch随机梯度下降 Mini-batch随机梯度下降：批量梯度下降和随机梯度下降的折中。

每次迭代选取$m(1 \leq m \leq N)$个样本进行参数更新。

$$\mathbf{a}_{t+1} = \mathbf{a}_t - \lambda \frac{1}{m} \sum_{i \in I_t} \frac{\partial \mathcal{R}\big(\theta_t; x^{(i)}, y^{(i)}\big)}{\partial \theta}, \tag{1.52}$$

1.6 经验

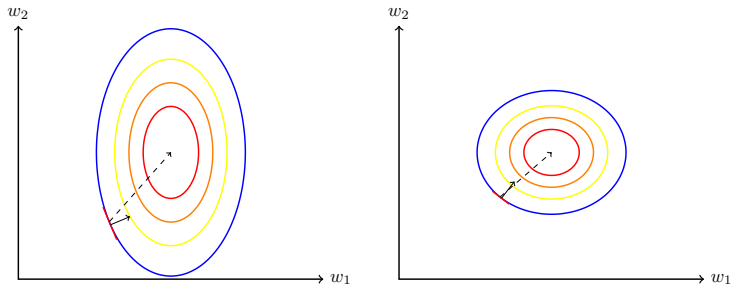
因为深度神经网络的表达能力很强，所以很容易产生过拟合。另外，大量的参数会导致训练比较慢。在训练深度神经网络时，同时也需要掌握一定的技巧。目前，人们在大量的实践中总结了一些经验技巧，可以从以下几个方面来提高学习效率并得到一个好的网络模型：1）数据增强；2）数据预处理；3）网络参数初始化；4）正则化；5）超参数优化等。

1.6.1 数据增强

深层神经网络一般都需要大量的训练数据才能获得比较理想的结果。

1.6.2 数据预处理

一般而言，原始的训练数据中，每一维特征的来源以及度量单位不同，会造成这些特征值的分布范围往往差异很大。当我们计算不同样本之间的欧式距离时，取值范围大的特征会起到主导作用。这样，对于基于相似度比较的机器学习方法（比如最近邻分类器），必须先对样本进行预处理，将各个维度的特征归一化到同一个取值区间，并且消除不同特征之间的相关性，才能获得比较理想的结果。虽然神经网络可以通过参数的调整来适应不同特征的取值范围，但是会导致训练效率比较低。



(a) 未归一化数据的梯度 (b) 归一化数据的梯度

图 1.8: 数据归一化对梯度的影响。

假设一个只有一层的网络 $y = \tanh(w_1x_1 + w_2x_2 + b)$，其中 $x_1 \in [0, 10]$，$x_2 \in [0, 1]$。之前我们提到 $\tanh()$ 的导数在区间 $[-2, 2]$ 上是敏感的，其余的导数接近于 0。因此，如果 $w_1x_1 + w_2x_2 + b$ 过大或过小，都会导致梯度过小，难以训练。为了提高训练效率，我们需要使 $w_1x_1 + w_2x_2 + b$ 在 $[-2, 2]$ 区间，我们需要将 w_1 设得小一点，比如在 $[-0.1, 0.1]$ 之间。可以想象，如果数据维数很多时，我们很难这样精心去选择每一个参数。因此，如果每一个特征的取值范围都在相似的区间，比如 $[0, 1]$ 或者 $[-1, 1]$，我们就不太需要区别对待每一个参数，减少人工干预。

除了参数初始化之外，不同特征取值范围差异比较大时还会梯度下降法的搜索效率。图1.8给出了数据归一化对梯度的影响。其中，图1.8a为未归一化数据的等高线图。取值范围不同会造成在大多数位置上的梯度方向并不是最优的搜索方向。当使用梯度下降法寻求最优解时，会导致需要很多次迭代才能收敛。如果我们把数据归一化为取值范围相同，如图1.8b所示，大部分位置的梯度方向近似于最优搜索方向。这样，在梯度下降求解时，每一步梯度的方向都基本指向最小值，训练效率会大大提高。

归一化的方法有很多种，比如之前我们介绍的 sigmoid 型函数等都可以将不同取值范围的特征挤压到一个比较受限的区间。这里，我们介绍几种在神经网络中经常使用的归一化方法。

标准归一化 **标准归一化**也叫 z-score 归一化，来源于统计上的标准分数。将每一个维特征都处理为符合标准正态分布（均值为 0，标准差为 1）。假设有 N 个

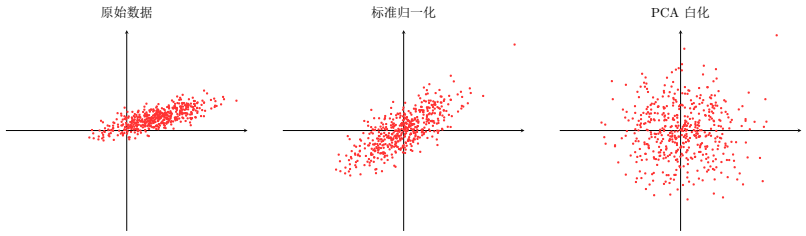


图 1.9: 数据归一化示例

样本 $\{\mathbf{x}^{(i)}\}, i = 1, \cdots, N$，对于每一维特征 x，我们先计算它的均值和标准差：

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x^{(i)}, \tag{1.53}$$

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - \mu)^2. \tag{1.54}$$

然后，将特征 $x^{(i)}$ 减去均值，并除以标准差，得到新的特征值 $\hat{x}^{(i)}$。

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu}{\sigma}, \tag{1.55}$$

这里，σ 不能为0。如果标准差为0，说明这一维特征没有任务区分性，可以直接删掉。

在标准归一化之后，每一维特征都服从标准正态分布。

缩放归一化 另外一种非常简单的归一化是通过缩放将特征取值范围归一到 $[0, 1]$ 或 $[-1, 1]$ 之间：

$$\hat{x}^{(i)} = \frac{x^{(i)} - \min(x)}{\max(x) - \min(x)}, \tag{1.56}$$

其中，$\min(x)$ 和 $\max(x)$ 分别为这一维特征在所有样本上的最小值和最大值。

PCA 使用 PCA (Principal Component Analysis) 方法可以去除掉各个成分之间的相关性。

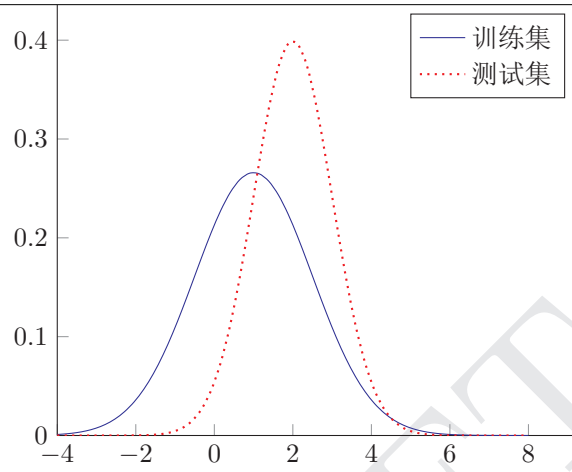


图 1.10: 协变量偏移。

1.6.3 批量归一化

在传统机器学习中，一个常见的问题的**协变量偏移**（Covariate Shift）。协变量是一个统计学概念，是可能影响预测结果的变量。在机器学习中，协变量可以看作是输入变量。一般的机器学习算法都要求输入变量在训练集和测试集上的分布是相似的。如果不满足这个要求，这些学习算法在测试集的表现会比较差。

在多层的神经网络中，中间某一层的输入是其前面网络的输出。因为前面网络的参数在每次迭代时也会被更新，而一旦这些参数变化会造成这一个中间层的输入也发生变化，其分布往往会和参数更新之前差异比较大。换句话说，从这一个中间层开始，之后的网络参数白学了，需要重新学习。这个中间层的深度很大时，这种现象就越明显。这种现象叫做**内部协变量偏移**（Internal Covariate Shift）。

为了解决这个问题，通过对每一层的输入进行归一化使其分布保存稳定。这种方法称为**批量归一化**方法（Batch Normalization）[Ioffe and Szegedy, 2015]。

归一化方法可以采用第1.6.2节中介绍的几种归一化方法。因为每一层都要进行归一化，所以要求归一化方法的速度要快。这样，PCA 白化的归一化方法就不太合适。为了提高归一化效率，这里使用标准归一化，对每一维特征都归一到标准正态分布。相当于每一层都进行一次数据预处理，从而加速收敛速度。

因为标准归一化会使得输入的取值集中的0附近，如果使用sigmoid型激活函数时，这个取值区间刚好是接近线性变换的区间，减弱的神经网络的非线性性质。因此，为了使得归一化不对网络的表示能力造成负面影响，我们可以通过一个附加的缩放和平移变换改变取值区间。从最保守的角度考虑，可以通过来标准归一化的逆变换来使得归一化后的变量可以被还原为原来的值。

$$y_k = \gamma_k \hat{x}_k + \beta_k, \tag{1.57}$$

这里γ_k和β_k分别代表缩放和平移的参数。当$\gamma_k = \sqrt{\sigma[x_k]}$，$\beta_k = \mu[x_k]$时，$y_k$即为原始的$x_k$。

算法 1.2: 批量归一化

输入: 一次mini-batch中的所有样本集合：

$\mathcal{B} = \{\mathbf{x}^{(i)}\}, i = 1, \cdots, m;$

参数：γ, β;

输出: $\{y^{(i)} = \mathbf{BN}_{\gamma,\beta}(\mathbf{x}^{(i)})\}$

for $k = 1 \cdots K$ **do**

$$\mu_{\mathcal{B},k} = \frac{1}{m}\sum_{i=1}^{m} x_k^{(i)}, \qquad \text{// mini-batch均值}$$

$$\sigma^2_{\mathcal{B},k} = \frac{1}{m}\sum_{i=1}^{m}(x_k^{(i)} - \mu_{\mathcal{B},k})^2. \qquad \text{//mini-batch方差}$$

$$\hat{x}_k^{(i)} = \frac{x_k^{(i)} - \mu_{\mathcal{B},k}}{\sqrt{\sigma^2_{\mathcal{B},k} + \epsilon}}, \forall i \qquad \text{//归一化}$$

$$y_k^{(i)} = \gamma \hat{x}_k^{(i)} + \beta \equiv \mathbf{BN}_{\gamma,\beta}(\mathbf{x}^{(i)}), \forall i \qquad \text{//缩放和平移}$$

end

当训练完成时，用整个数据集上的均值μ_k和方差σ_k^2来分别代替$\mu_{\mathcal{B},k}$和方差$\sigma^2_{\mathcal{B},k}$。

通过每一层的归一化，从而减少前面网络参数更新对后面网络输入带来的内部协变量偏移问题，提高训练效率。

1.6.4 参数初始化

神经网络的训练过程中的参数学习是基于梯度下降法进行优化的。梯度下降法需要在开始训练时给每一个参数赋一个初始值。这个初始值的选取十分关键。在感知器和logistic回归的训练中，我们一般将参数全部初始化为0。但是这在神经网络的训练中会存在一些问题。因为如果参数都为0，在第一遍前向计算时，所有的隐层神经元的激活值都相同。这样会导致深层神经元没有区分性。这种现象也称为**对称权重**现象。

为了打破这个平衡，比较好的方式是对每个参数都随机初始化，这样使得不同神经元之间的区分性更好。

但是一个问题是如何选取随机初始化的区间呢？如果参数太小，会导致神经元的输入过小。经过多层之后信号就慢慢消失了。参数过小还会使得sigmoid型激活函数丢失非线性的能力。以logistic函数为例，在0附近基本上是近似线性的。这样多层神经网络的优势也就不存在了。如果参数取得太大，会导致输入状态过大。对于sigmoid型激活函数来说，激活值变得饱和，从而导致梯度接近于0。

因此，如果要高质量地训练一个网络，给参数选取一个合适的初始化区间是非常重要的。经常使用的初始化方法有以下几种：

Gaussian初始化方法 Gaussian初始化方法是最简单的初始化方法，参数从一个固定均值（比如0）和固定方差（比如0.01）的Gaussian分布进行随机初始化。

Xavier初始化方法 Glorot and Bengio [2010]提出一个初始化方法，参数可以在区间$[-r, r]$内采用均匀分布进行初始化。

Xavier初始化方法中，Xavier是发明者Glorot的名字。

对于logistic函数，第$l-1$到l层的权重，

$$r = \sqrt{\frac{6}{n^{l-1} + n^1}}, \tag{1.58}$$

这里n^l是第l层神经元个数，n^{l-1}是第$l-1$层神经元个数。

对于tanh函数，

$$r = 4\sqrt{\frac{6}{n^{l-1} + n^l}}. \tag{1.59}$$

假设第 l 层的一个隐藏层神经元 z^l，其接受前一层的 n^{l-1} 个神经元的输出 $a_i^{(l-1)}$，$i \in [1, n^{(l-1)}]$，

$$z^l = \sum_{i=1}^{n} w_i^l a_i^{(l-1)} \tag{1.60}$$

为了避免初始化参数使得激活值变得饱和，我们需要尽量使得 z^l 处于激活函数的线性区间，也就是其绝对值比较小的值。这时，该神经元的激活值为 $a^l = f(z^l) \approx z^l$。

假设 w_i^l 和 $a_i^{(l-1)}$ 都是相互独立，并且均值都为0，则 a 的均值为

$$E(a^l) = E(\sum_{i=1}^{n} w_i^l a_i^{(l-1)}) = \sum_{i=1}^{d} E(\mathbf{w}_i) E(a_i^{(l-1)}) = 0. \tag{1.61}$$

a^l 的方差为

$$\begin{aligned} \mathrm{Var}(a^l) &= \mathrm{Var}(\sum_{i=1}^{n^{(l-1)}} w_i^l a_i^{(l-1)}) && (1.62)\\ &= \sum_{i=1}^{n^{(l-1)}} \mathrm{Var}(w_i^l)\,\mathrm{Var}(a_i^{(l-1)}) && (1.63)\\ &= n^{(l-1)}\,\mathrm{Var}(w_i^l)\,\mathrm{Var}(a_i^{(l-1)}). && (1.64) \end{aligned}$$

也就是说，输入信号的方差在经过该神经元后被放大或缩小了 $n^{(l-1)}\,\mathrm{Var}(w_i^l)$ 倍。为了使得在经过多层网络后，信号不被过分放大或过分减弱，我们尽可能保持每个神经元的输入和输出的方差一致。这样 $n^{(l-1)}\,\mathrm{Var}(w_i^l)$ 设为1比较合理，即

$$\mathrm{Var}(w_i^l) = \frac{1}{n^{(l-1)}}. \tag{1.65}$$

同理，为了使得在反向传播中，误差信号也不被放大或缩小，需要将 w_i^l 的方差保持为

$$\mathrm{Var}(w_i^l) = \frac{1}{n^{(l)}}. \tag{1.66}$$

作为折中，同时考虑信号在前向和反向传播中都不被放大或缩小，可以设置

$$\mathrm{Var}(w_i^l) = \frac{2}{n^{(l-1)} + n^{(l)}}. \tag{1.67}$$

假设随机变量 x 在区间 $[a, b]$ 内均匀分布，则其方差为：

$$\text{Var}(x) = \frac{(b-a)^2}{12}. \tag{1.68}$$

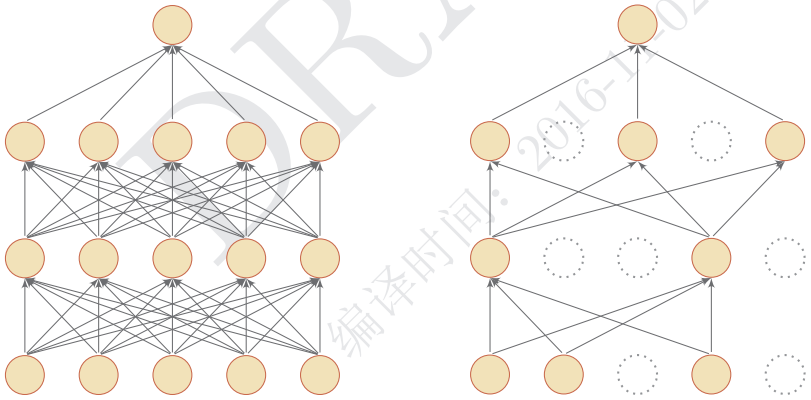
因此，若让 $w_i^l \in [-r, r]$，则 r 的取值为

$$r = \sqrt{\frac{6}{n^{l-1} + n^1}}. \tag{1.69}$$

1.6.5 正则化

深度神经网络很容易产生过拟合现象，因为增加的抽象层使得模型能够对训练数据中较为罕见的依赖关系进行建模。对此，权重递减（ℓ_2 正规化）或者稀疏（ℓ_1 -正规化）等方法可以利用在训练过程中以减小过拟合现象 [**?**]。

1.6.6 Dropout



(a) 标准网络　　(b) Dropout 神经网络

图 1.11: Dropout 神经网络模型

训练一个大规模的神经网络经常容易过拟合。过拟合在很多机器学习中都会出现。一般解决过拟合的方法有正则化、早期终止、集成学习以及使用验证集等。Srivastava et al. [2014] 提出了适用于神经网络的避免过拟合的方法，叫**dropout**方法（丢弃法），即在训练中随机丢弃一部分神经元（同时丢弃其对应的连接边）来避免过拟合。图1.11给出了 dropout 网络的示例。

每做一次dropout，相当于从原始的网络中采样得到一个更瘦的网络。如果一个神经网络有n个神经元，那么总共可以采样出2^n个子网络。每次迭代都相当于训练一个不同的子网络，这些子网络都共享原始网络的参数。那么，最终的网络可以近似看作是集成了指数级个不同网络的组合模型。每次选择丢弃的神经元是随机的。最简单的方法是设置一个固定的概率p。对每一个神经元都一个概率p来判定要不要保留。p可以通过验证集来选取一个最优的值。或者，p也可以设为0.5，这对大部分的网络和任务有比较有效。

随机森林？

一般情况下dropout是针对神经元进行随机丢弃，但是也可以扩展到对每条边进行随机丢弃，或每一层进行随机丢弃。

当$p = 0.5$时，在训练时有一半的神经元被丢弃，只剩余一半的神经元是可以激活的。而在测试时，所有的神经元都是可以激活的。因此每个神经元训练时的净输入值平均比测试时小一半左右。这会造成训练和测试时网络的输出不一致。为了缓解这个问题，在测试时需要将每一个神经元的输出都折半，也相当于把不同的神经网络做了平均。

一般来讲，对于隐藏层的神经元，其dropout率等于0.5时效果最好，因为此时通过dropout方法，随机生成的网络结构最具多样性。对于输入层的神经元，其dropout率通常设为更接近1的数，使得输入变化不会太大。当对输入层神经元进行dropout时，其可以被是给数据增加噪声，以此来提高网络的鲁棒性。

1.6.7 超参数优化

在构建前馈神经网络时，有下面的超参数需要设置。

- 网络层数
- 每层的神经元数量
- 激活函数的类型
- 学习率（以及动态调整算法）
- 正则化系数
- 每次小批量梯度下降的

对于超参数的设置，一般用**网格搜索**（Grid Search）或者人工搜索的方法来进行。假设总共有 K 个超参数，第 k 个超参数的可以取 m_k 个值。如果参数是连续的，可以将参数离散化，选择几个"经验"值。比如学习率 α，我们可以设置

$$\alpha \in \{0.01, 0.1, 0.5, 1.0\}. \tag{1.70}$$

这样，这些超参数可以有 $m_1 \times m_2 \times \cdots \times m_K$ 个取值组合。所谓网格搜索就是根据这些超参数的不同组合分别训练一个模型，然后评价这些模型在检验数据集上的性能，选取一组性能最好的组合。

随机搜索优于网格搜索。Bergstra 和 Bengio 在文章 Random Search for Hyper-Parameter Optimization 中说"随机选择比网格化的选择更加有效"，而且在实践中也更容易实现。

1.7 总结和深入阅读

Anderson and Rosenfeld [2000]

参考文献

James A Anderson and Edward Rosenfeld. *Talking nets: An oral history of neural networks*. MiT Press, 2000.

Yoshua Bengio. Learning deep architectures for AI. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

Yoshua Bengio, Jean-Sébastien Senécal, et al. Quick training of probabilistic neural nets by importance sampling. In *AISTATS Conference*, 2003.

C.M. Bishop. *Pattern recognition and machine learning*. Springer New York., 2006.

P.F. Brown, P.V. Desouza, R.L. Mercer, V.J.D. Pietra, and J.C. Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4): 467–479, 1992.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Michael Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, 2002.

Hal Daumé III. A course in machine learning. http://ciml.info/. [Online].

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley, New York, 2nd edition, 2001. ISBN 0471056693.

Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. Incorporating second-order functional knowledge for better option pricing. *Advances in Neural Information Processing Systems*, pages 472–478, 2001.

Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

Yoav Freund and Robert E Schapire. Large margin classification using the perceptron algorithm. *Machine learning*, 37(3):277–296, 1999.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

Ian Goodfellow, Aaron Courville, and Yoshua Bengio. Deep learning. Book in preparation for MIT Press, 2015. URL http://goodfeli.github.io/dlbook/.

Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C Courville, and Yoshua Bengio. Maxout networks. In *ICML*, volume 28, pages 1319–1327, 2013.

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, 2001.

Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

M.I. Jordan. *Learning in Graphical Models*. Kluwer Academic Publishers, 1998.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Ryan McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 456–464. Association for Computational Linguistics, 2010.

Marvin Minsky and Papert Seymour. Perceptrons. 1969.

Marvin L Minsky and Seymour A Papert. *Perceptrons - Expanded Edition: An Introduction to Computational Geometry*. MIT press Boston, MA:, 1987.

T.M. Mitchell. *Machine learning*. Burr Ridge, IL: McGraw Hill, 1997.

Andriy Mnih and Yee Whye Teh. A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*, 2012.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

Albert BJ Novikoff. On convergence proofs for perceptrons. Technical report, DTIC Document, 1963.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386–408, 1958.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5:3, 1988.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. 1974.

Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

DE Rumelhart GE Hinton RJ Williams and GE Hinton. Learning representations by back-propagating errors. *Nature*, pages 323–533, 1986.

Matthew D Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.