Data Mining and Optimization

Lecture 6: Artificial Neural Network

Liu Yang

Nanjing University

Spring, 2025

Table of Contents

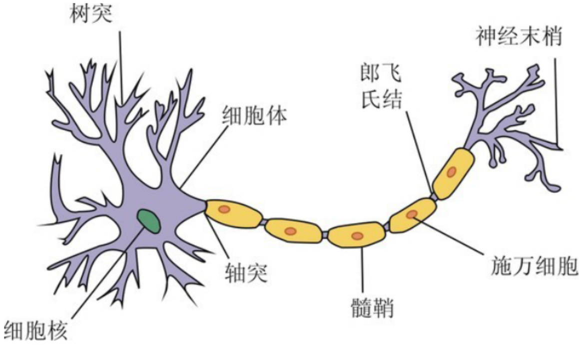
神经网络的基本概念

- 人工神经网络(Artificial Neural Network，ANN)，是一种模仿生物大脑的结构和功能的数学模型，用于对函数进行估计或近似。
- 和其他机器学习方法一样，神经网络可用于解决各种问题，例如机器视觉和语音识别，这些问题很难被传统基于规则的编程所解决。

Table of Contents

神经元

人类大脑是由大量“神经细胞”(neural cells)或神经元为基本单位而组成的神经网络。



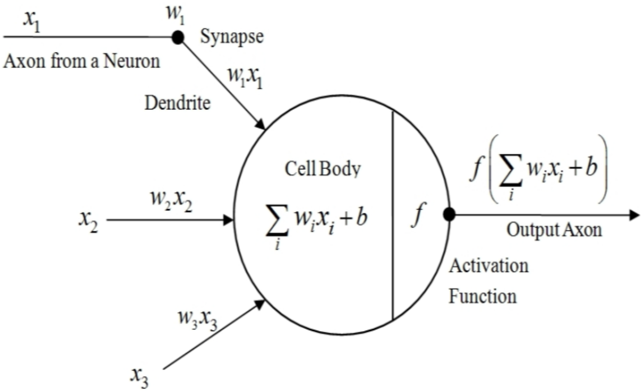
神经元

- 一个神经元通过左边的“树突”(dendrite)从其他神经元的“轴突”(axon)及“轴突末梢”(axon terminal)获取电子或化学信号；
- 两个神经元之间的连接部位(junction)，称为“神经突触”(synapse)(在图中标为“神经末梢”)。连接在一起的神经元，可以共同兴奋，即所谓“neurons wired together, fire together”；
- 从树突(dendrites)获得不同的信号后，神经元的“细胞体”(cell body)将这些信号进行加总处理；
- 如果这些信号的总量超过某个阈值(threshold)，则神经元会兴奋起来，并通过轴突向外传输信号，经过神经突触(synapses)，而为其他神经元的树突所接收。

Table of Contents

M-P神经元模型

1943年，美国神经生理学家Warren McCulloch与数学家Walter Pitts将生物神经元简化为一个数学模型(McCulloch and Pitts, 1943)，简称M-P神经元模型。



M-P神经元模型

- 将神经元视为一个计算单位，它首先从树突(dendrites)输入信号$(x_1, x_2, \cdots, x_k)'$，在细胞体(cell body)进行加权求和$\sum_{i=1}^k w_i x_i$，其中$(w_1, w_2, \cdots, w_k)'$为权重，表示不同信号重要程度的差异；
- 如果求和之后的总数，超过某个阈值(比如$-b$)，则神经元兴奋起来，通过轴突(axon)向外传递信号；反之，则神经元处于抑制状态；
- 使用如下函数表示神经元模型的输出：

$$\mathbb{I}\left(\sum_{i=1}^k w_i x_i + b > 0\right) = \begin{cases} 1 & \text{if } \sum_{i=1}^k w_i x_i + b > 0; \\ 0 & \text{if } \sum_{i=1}^k w_i x_i + b \leq 0 \end{cases}$$

  其中参数b表示阈值，示性函数$\mathbb{I}(\cdot)$称为激活函数(activation function)。
- M-P神经元模型本质上只是一个纯数学模型，其中的参数w_i与b需要人为指定，而无法通过训练样本进行学习。

感知机

- Rosenblatt(1958)提出感知机(Perceptron)，使得M-P神经元模型具备学习能力，成为神经网络模型的先驱。
- 对于二分类问题，考虑使用分离超平面$\sum_{i=1}^{k} w_i x_i + b = 0$进行分类，而响应变量$y \in \{-1, 1\}$。
- 如果$\sum_{i=1}^{k} w_i x_i + b > 0$，则预测$y = 1$；如果$\sum_{i=1}^{k} w_i x_i + b < 0$，则预测$y = -1$；如果$\sum_{i=1}^{k} w_i x_i + b = 0$，可随意预测。
- 正确分类要求$y(\sum_{i=1}^{k} w_i x_i + b) > 0$。若$y(\sum_{i=1}^{k} w_i x_i + b) < 0$，则分类错误。

感知机

- 从某个初始值$(w_1^0, w_2^0, \cdots, w_k^0, b^0)$出发，感知机希望通过调整参数$(w_1, w_2, \cdots, w_k, b)$，使得模型的错误分类最少。
- 感知机的目标函数为最小化所有分类错误观测值的“错误程度”之和：

$$\min_{w_1, w_2, \cdots, w_k, b} L(w_1, w_2, \cdots, w_k, b) = -\sum_{n \in \mathbb{M}} y_n \left(\sum_{i=1}^{k} w_i x_{in} + b\right),$$

其中，$\mathbb{M}$为所有错误分类(misclassified)的个体下标之集合。

- 使用梯度下降法不断迭代，损失函数$L(w_1, w_2, \cdots, w_k, b)$不断减小，直到变为0为止。

感知机

- 感知机算法的直观解释：当一个样本点被错误分类，即出现在分离超平面$\sum_{i=1}^{k} w_i x_i + b = 0$的错误一侧时，则调整参数$(w_1, w_2, \cdots, w_k, b)$，使得分离超平面向该误分类点的一侧移动，以减少此误分类点与超平面的距离，直至正确分类为止。
- 可以证明，对于线性可分的数据，感知机一定会收敛。
- 这表明，只要给予足够的数据，感知机具备学得参数$(w_1, w_2, \cdots, w_k, b)$的能力，仿佛拥有“感知”世界的能力，故名“感知机”。

感知机

- 对于线性可分的数据，感知机虽然一定会收敛，但从不同的初始值出发，一般会得到不同的分离超平面，无法得到唯一解。
- 如果数据为线性不可分，则感知机的算法不会收敛。
- 感知机更严重的缺陷是，它的决策边界依然为线性函数。可将感知机的预测函数写为

$$f(x_1, x_2, \cdots, x_k) = sign(\sum_{i=1}^{k} w_i x_i + b),$$

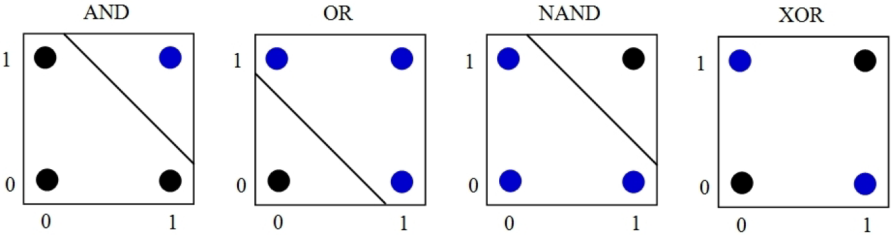
其中，$sign(\cdot)$为符号函数，满足

$$sign(z) = \begin{cases} 1 & \text{if } z \geq 0; \\ -1 & \text{if } z < 0 \end{cases}$$

感知机

- 虽然符号函数$sign(\cdot)$为非线性，但感知机的决策边界为$\sum_{i=1}^{k} w_i x_i + b = 0$依然为线性函数。
- 感知机无法适用于决策边界为非线性的数据。
- 例：感知机无法识别“异或函数”。在逻辑学中，有几个常见的逻辑运算，包括“与”(AND)、“或”(OR)、“与非”(NOT AND)、“异或”(Exclusive Or，简记XOR)。“异或”是一种排他性(exclusive)的“或”，即当二者取值不同时为“真”(TRUE)，而当二者取值相同时即为“假”(FALSE)。

感知机



逻辑判断TRUE记为1(以蓝点表示)，而FALSE记为0(以黑点表示)。

感知机

- 对AND运算而言，只有当输入值都是1(TRUE)时，经过AND运算后才是1(TRUE)，以右上角的蓝点表示；在这种情况下，存在线性的决策边界。
- 对于OR与NAND的运算，也存在线性的决策边界。
- 对于XOR的运算，由于TRUE与FALSE分别分布在两个对角上，故无法找到线性的决策边界，存在非线性的决策边界。
- 1969年，Marvin Minsky与Seymour Papert在专著Perceptrons指出，感知机连基本的异或函数都无法区分，功能十分有限。
- 当时学界普遍认为感知机无发展前途，使人工神经网络研究陷入低谷。

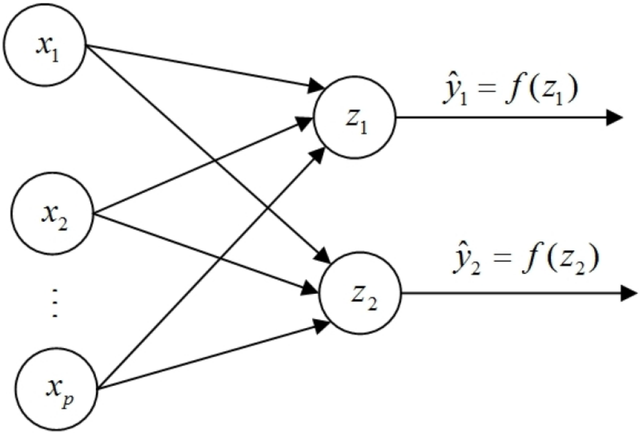
Table of Contents

神经网络模型

- 事实上，在感知机的基础上，并不难得到非线性的决策边界。
- 只要引入多层神经元，经过两个及以上的非线性激活函数迭代之后，即可得到非线性的决策边界。
- 非线性的激活函数是关键：如果使用线性的激活函数，则无论叠加或嵌套多少次(相当于复合函数)，所得结果还是线性函数。

多输出的感知机

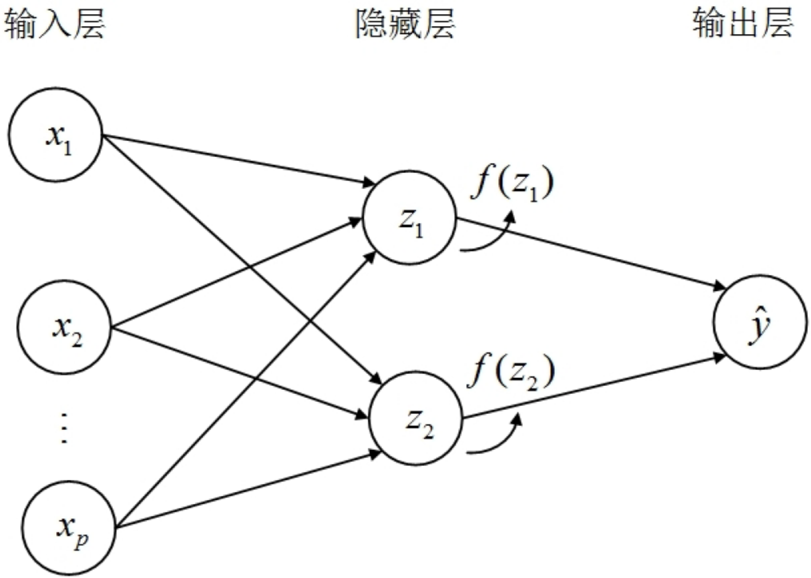
首先，考虑具有多个输出结果(multi output)的感知机，如图所示



多输出的感知机

- 在图中，共有两个输出(响应)变量，$\hat{y}_1$和$\hat{y}_2$。
- 其中，$z_1 = \sum_{i=1}^{p} w_{i1}x_i + b_1$与$z_2 = \sum_{i=1}^{p} w_{i2}x_i + b_2$，均为在施加激活函数之前的加总值；而$f(\cdot)$为激活函数。
- 其次，上图中的多个输出结果，可重新作为输入变量，经过加权求和后，再次施以激活函数，参见下图：

多层感知机



多层感知机

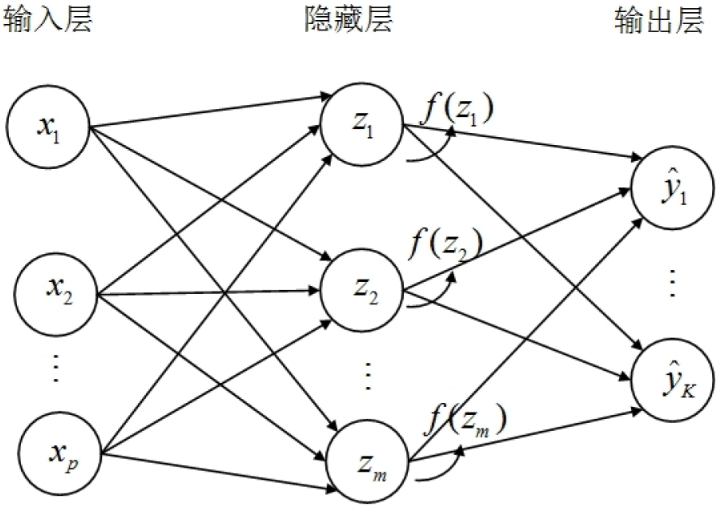
- 在上图中，最终输出结果为

$$\hat{y} = f\left(b^{(2)} + w_1^{(2)} f(z_1) + w_2^{(2)} f(z_2)\right), \tag{1}$$

  即对$f(z_1)$与$f(z_2)$再次加权求和，然后再施加激活函数$f(\cdot)$。
- 函数(1)所对应的决策边界为非线性的。
- 在上图中，最左边为输入层(input layer)，中间为隐藏层(hidden layer)，而最右边为输出层(output layer)。
- 之所以将中间层称为“隐藏层”，因为该层的计算在算法内部进行，从外面并不可见。

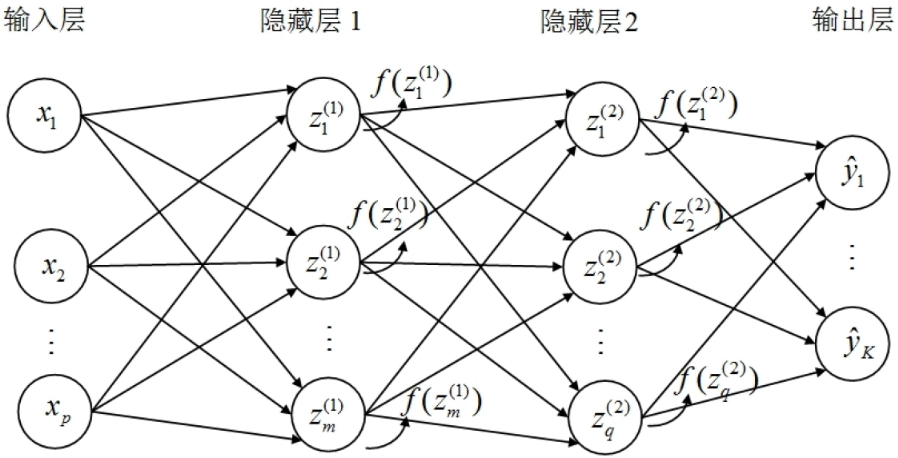
单隐藏层神经网络

隐藏层可有更多的神经元，而输出层也可有多个输出结果。



双隐藏层神经网络

更一般地，神经网络模型可有多个隐藏层。



神经网络的类型

- 这种标准的神经网络，称为前馈神经网络(feedforward neural network)，因为输入从左向右不断前馈，也称为全连接神经网络(fully-connected neural network)，因为相邻层的所有神经元都相互连接。
- 针对特殊的数据类型，可能还需要特别的网络结构，比如卷积神经网络(适用于图像识别)、循环神经网络(适用于自然语言等时间序列)等。
- 如果神经网络的隐藏层很多，则称为深度神经网络(deep neural networks)，简称深度学习(deep learning)。

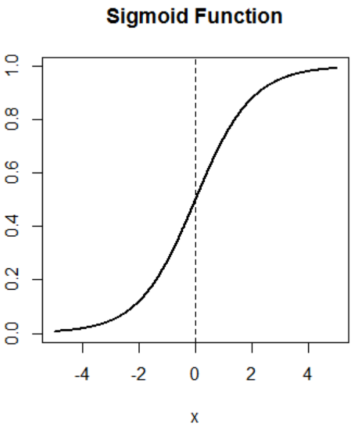
Table of Contents

神经网络的激活函数

- 感知机使用符号函数$sign(\cdot)$作为激活函数，是不连续的“阶梯函数”(step function)，不便于进行最优化。
- 激活函数必须为非线性函数。神经网络模型中常用的激活函数包括：
  - S型函数(Sigmoid Function)，参见下图。狭义的S型函数就是逻辑分布的累积分布函数，其表达式为

$$\Gamma(z) = \frac{e^z}{1+e^z}.$$

神经网络的激活函数



Sigmoid Function
1.0
0.8
0.6
0.4
0.2
0.0
-4
-2
0
2
4
x
Hyperbolic Tangent
1.0
0.5
0.0
-0.5
-1.0
-4
-2
0
2
4
x

神经网络的激活函数

- 由于sigmoid函数的输出值介于0与1之间，故可将其解释为概率分布。
- 与感知机所用的阶梯激活函数相比，sigmoid函数为连续可导(continuously differentiable，即导函数存在且连续)，其数学性质更好。
- 但当输入靠近两端($|z|$很大)时，sigmoid函数的导数$\Gamma'(z)$趋向于0，故在训练神经网络时，可能导致“梯度消失”(vanishing gradient)的问题，使得梯度下降法失效。具体来说，sigmoid函数的导数为

$$\Gamma'(z) = \Gamma(z)(1 - \Gamma(z)),$$

当$z \to \infty$或$z \to -\infty$时，$\lim \Gamma'(z) = 0$。这种情形称为“两端饱和”。

神经网络的激活函数

- 双曲正切函数(Hyperbolic Tangent Function)，参见上图。双曲正切函数是一种广义的S型函数，因为它的形状也类似于拉长的英文大写字母S，其表达式为

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

- $tanh(\cdot)$函数可看作是将Logistic函数进一步拉伸到$(-1, 1)$区间。二者有如下关系：

$$tanh(z) = 2\Gamma(2z) - 1.$$

- $tanh(\cdot)$函数也是两端饱和的，依然可能发生梯度消失的问题。

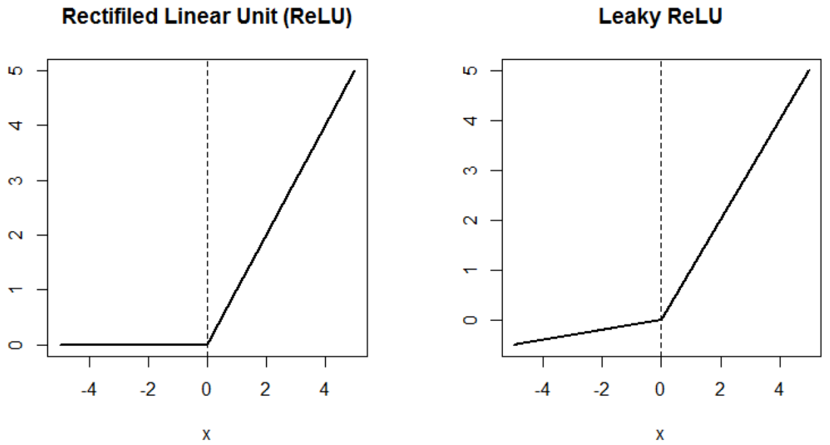
神经网络的激活函数

- 修正线性单元(Rectified Linear Unit，简记ReLU)，也称“线性整流函数”，参见下图。
- 为了解决Logistic函数与Tanh函数的两端饱和问题，Nair and Hinton(2010)提出如下ReLU函数，成为目前深度神经网络中经常使用的激活函数：

$$ReLU(z) = \max(0, z) = \begin{cases} z & \text{if } z \geq 0; \\ 0 & \text{if } z < 0 \end{cases}.$$

- ReLU实际上是一个斜坡(ramp)函数。当输入$z \geq 0$，其输出也是z，即所谓“线性单元”(linear unit)；而当输入$z < 0$时，则将输出“修正”(rectified)为0。

神经网络的激活函数



神经网络的激活函数

- 以ReLU函数作为激活函数，其计算非常方便。
- 相比于S型函数的两端饱和，ReLU函数为“左饱和函数”，即当$z \to -\infty$时，ReLU函数的导数趋向于0。
- 当$z > 0$时，ReLU函数的导数恒等于1，这可在一定程度上缓解神经网络训练中的梯度消失问题，加快梯度下降的收敛速度。
- ReLU函数被认为具有生物学上的解释，比如单侧抑制、宽兴奋边界(即兴奋度可以很高)。
- 在生物神经网络中，同时处于兴奋状态的神经元一般很稀疏。S型激活函数会导致非稀疏的神经网络，而ReLU激活函数可导致较好的稀疏性。

神经网络的激活函数

- 由于当$z < 0$时，ReLU函数的导数恒等于0，这导致神经元在训练时可能“死亡”，称为“死亡ReLU问题”(dying ReLU problem)。
- 所谓“神经元死亡”，就是无论该神经元的输入是什么，其输出永远是0，故无法更新其输入的权重。

- 泄露ReLU(Leaky ReLU，简记LReLU)，参见上图。解决“死亡ReLU问题”的一种方式是，当输入$z < 0$时，依然保持一个很小的梯度$\gamma > 0$。这使得当神经元处于非激活状态时，也有一个非零梯度可更新参数，避免永远不能被激活(Maas et al., 2013)。
- 泄露ReLU函数的定义为：

$$LReLU(z) = \begin{cases} z & \text{if } z \geq 0; \\ \gamma z & \text{if } z < 0 \end{cases}.$$

其中，$\gamma > 0$是一个很小的正数，比如0.01。当$\gamma < 1$时，泄露ReLU可写为

$$LReLU(z) = \max(z, \gamma z).$$

神经网络的激活函数

- 软加函数(Softplus Function)，参见下图。
- ReLU函数并不光滑，而且在$z < 0$时，导数一直为0。
- 软加函数可视为ReLU 函数的光滑版本，正好弥补ReLU的这些缺点。Softplus函数的定义为：

$$Softplus(z) = \ln(1 + e^{z}).$$

- Softplus函数也具有单侧抑制、宽兴奋边界的特性，但没有ReLU函数的稀疏激活性(因为Softplus函数的导数永远为正)。

神经网络的激活函数

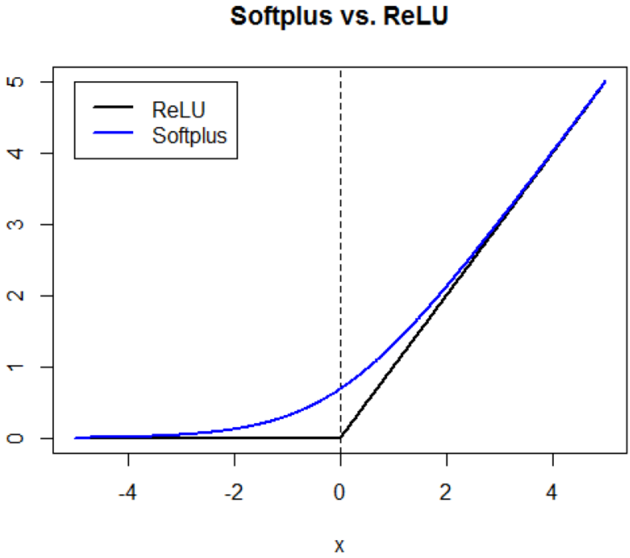


Table of Contents

通用函数近似器

- 前馈神经网络具有很强的函数拟合能力。
- 在一定意义上，神经网络可作为一种“通用近似器”(universal approximator)来使用。
- Cybenko(1988)与Hornik, Stinchcombe and White(1989)证明了神经网络的“通用近似定理”(Universal Approximation Theorem)。
- 主要结论为，包含单一隐藏层的前馈神经网络模型，只要其神经元数目足够多，则可以任意精度逼近任何一个在有界闭集上定义的连续函数。

通用函数近似器

- 首先，包含单隐藏层的前馈神经网络所代表的函数可写为

$$G(x) = \sum_{i=1}^{m} \alpha_i f(w_i' x + b_i) + \alpha_0. \tag{2}$$

其中，(w_i, b_i)为第i个神经元的权重与偏置参数，$f(\cdot)$为激活函数，α_0和α_i为连接隐藏层与输出层的参数，而m为神经元的数目。

- 通用近似定理表明，形如(2)的函数在定义于有界闭集上的连续函数之集合中是“稠密的”(dense)。
- 这意味着对于任意有界闭集上的连续函数，都可找到形如(2)的函数(即单隐层的前馈神经网络)，使二者的距离任意接近。

通用函数近似器

通用近似定理

令$f(\cdot)$为一个合适的激活函数(详见下文)，$\mathcal{I}_p$是一个p维的单位超立方体(unit hypercube)$[0,1]^p$ ，而$C(\mathcal{I}_p)$是定义在$\mathcal{I}_p$上的所有连续函数之集合。对于任意一个函数$g \in C(\mathcal{I}_p)$，给定任意小的正数$\epsilon > 0$，则存在一个正整数m(即神经元数目)，一组实数(a_i, b_i)，以及实数向量$w_i \in \mathbb{R}^p$, $i = 1, 2, \cdots, m$, 使得方程(2)所定义的函数$G(x)$，可以任意地接近$g(x)$，即

$$|G(x) - g(x)| < \epsilon, \forall x \in \mathcal{I}_p.$$

通用函数近似器

- 在上述定理中，假设定义域为p维单位超立方体$[0,1]^p$，只是为了叙述方便。通用近似定理在任意p维实数空间$\mathbb{R}^p$的有界闭集上依然成立。
- 在文献中，通用近似定理的激活函数可采取不同形式的非线性函数，既包括非常数(nonconstant)、有界(bounded)且单调递增的连续函数(例如S型函数、双曲正切函数)，也包括无界(unbounded)且单调递增的连续函数(例如ReLU)，甚至允许不连续函数(例如阶梯函数)。
- 通用近似定理表明，神经网络可作为“万能”函数来使用。

通用函数近似器

- 但通用近似定理只是说明，对于任意有界闭集上的连续函数，都存在与它非常接近的单隐层前馈神经网络。
- 但并未给出找到此神经网络的方法，也不知道究竟需要多少个神经元，才能达到既定的接近程度。
- 在实际应用中，一般并不知道真实函数$g(x)$，而我们更关心神经网络$G(x)$的泛化能力。
- 由于神经网络的强大拟合能力，反而容易在训练集上过拟合，故需要避免过拟合，以降低测试误差。

Table of Contents

神经网络的损失函数

- 未经训练的神经网络就像空白的大脑，并不具备预测与分类的能力。
- “训练”意味着估计神经网络模型的诸多参数。对于神经网络而言，知识就储存在这些参数中。
- 神经网络的通常训练方法为，在参数空间使用梯度下降法，使损失函数最小化。神经网络的损失函数之一般形式可写为

$$W^* = arg \min_{W} \frac{1}{N} \sum_{i=1}^{N} L(Y_i, G(X_i; W))$$

其中，W包含神经网络模型的所有参数(包括偏置)。

神经网络的损失函数

- W^*为W的最优值，$G(X_i; W)$为神经网络对观测值X_i所作的预测(即$\hat{Y}_i$)，而$L(Y_i, \hat{Y}_i)$为损失函数。
- 整个样本的损失函数为每个观测值之损$L(Y_i, \hat{Y}_i)$的平均值。
- 对于Y为连续的回归问题，一般使用平方损失函数(squared loss function)，最小化训练集的均方误差：

$$W^* = arg\min_W \frac{1}{N}\sum_{i=1}^{N}(Y_i - G(X_i; W))^2.$$

神经网络的损失函数

- 对于Y为离散的分类问题，则一般使用“交叉熵损失函数”(cross-entropy loss function)，即对数似然函数之负数。
- 对于二分类问题，一般使用“二值交叉熵损失函数”(binary cross-entropy loss function)：

$$W^* = arg\min_W -\frac{1}{N}\sum_{i=1}^{N}(Y_i \ln(G(X_i; W)) + (1 - Y_i)\ln(1 - G(X_i; W))).$$

Table of Contents

反向传播算法

- 使用梯度下降法训练神经网络需要计算$G(X_i; W)$的梯度向量。
- 对于神经网络，最常用的计算梯度向量方法为反向传播算法(Back Propagation，简记BP)。
- 对于多层的神经网络，越靠近网络右边(后端)的参数，其导数越容易计算，因为它们离输出层更近。
- 反向传播算法就是使用微积分的“链式法则”(chain rule)，将靠左边(前端)的参数之导数，递归地表示为靠右边(后端)的参数之导数之函数。

反向传播算法

- 具体来说，记第l个隐层的第i个神经元的输出值(激活值，activation)为$a_i^{(l)}$，则

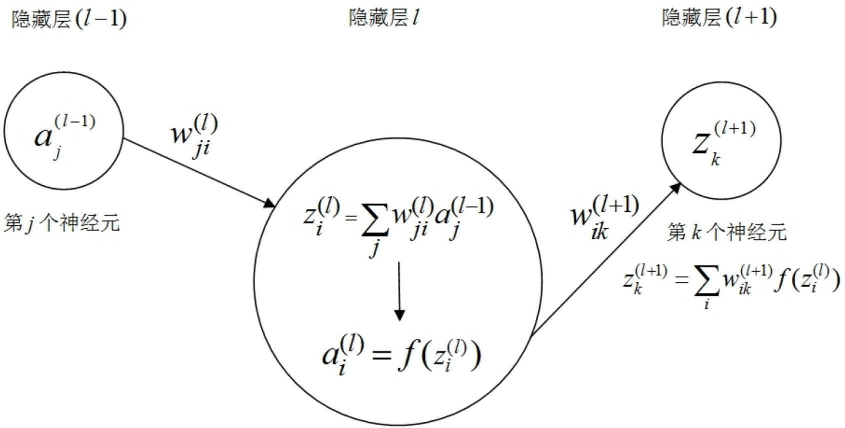
$$a_i^{(l)} = f\left(\sum_j w_{ji}^{(l)} a_j^{(l-1)}\right) \equiv f\left(z_i^{(l)}\right), \tag{3}$$

其中，$w_{ji}^{(l)}$为第$l-1$隐层第j个神经元的激活值$a_j^{(l-1)}$对$a_i^{(l)}$的作用权重，而$f(\cdot)$为激活函数。

- 在施加激活函数之前，记“净输入”(net input)为

$$z_i^{(l)} \equiv \sum_j w_{ji}^{(l)} a_j^{(l-1)}. \tag{4}$$

反向传播算法



隐藏层 $(l-1)$
隐藏层 l
隐藏层 $(l+1)$
$a_j^{(l-1)}$
$w_{ji}^{(l)}$
第 j 个神经元
$z_i^{(l)} = \sum_j w_{ji}^{(l)} a_j^{(l-1)}$
$a_i^{(l)} = f(z_i^{(l)})$
$w_{ik}^{(l+1)}$
$z_k^{(l+1)}$
第 k 个神经元
$z_k^{(l+1)} = \sum_i w_{ik}^{(l+1)} f(z_i^{(l)})$

反向传播算法

- 记神经网络的损失函数为L。考虑将损失函数L对参数$w_{ji}^{(l)}$求导。
- 由于$w_{ji}^{(l)}$仅通过影响净输入$z_i^{(l)}$而作用于L，故根据链式法则可得：

$$\frac{\partial L}{\partial w_{ji}^{(l)}} = \frac{\partial L}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{ji}^{(l)}} = \frac{\partial L}{\partial z_i^{(l)}} a_j^{(l-1)} \equiv \delta_i^{(l)} a_j^{(l-1)}, \tag{5}$$

其中，$\delta_i^{(l)} = \frac{\partial L}{\partial z_i^{(l)}}$称为“误差”(error)。

反向传播算法

- $z_i^{(l)}$影响损失函数L的途径为，通过第$l+1$层所有神经元的净输入$z_k^{(l+1)}$。
- 再次使用链式法则，可得到$\delta_i^{(l)}$的表达式：

$$\delta_i^{(l)} = \frac{\partial L}{\partial z_i^{(l)}} = \sum_k \frac{\partial L}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} = \sum_k \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}, \tag{6}$$

其中，根据定义$\delta_k^{(l+1)} = \frac{\partial L}{\partial z_k^{(l+1)}}$。

反向传播算法

- 进一步，由于$z_k^{(l+1)} = \sum_i w_{ik}^{(l+1)} f(z_i^{(l)})$，故

$$\frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} = w_{ik}^{(l+1)} f'(z_i^{(l)}). \tag{7}$$

- 将上式代入方程(6)可得：

$$\delta_i^{(l)} = \sum_k \delta_k^{(l+1)} w_{ik}^{(l+1)} f'(z_i^{(l)}) = f'(z_i^{(l)}) \sum_k \delta_k^{(l+1)} w_{ik}^{(l+1)}. \tag{8}$$

- (8)式将第l隐层的误差$\delta_i^{(l)}$表示为第$l+1$隐层的误差$\delta_k^{(l+1)}$之函数，这是一种反向的递推公式。

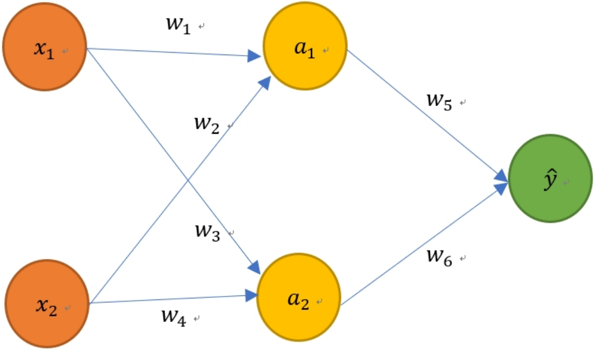
反向传播算法

- 可以用递归(recursive)的方法计算误差$\delta_i^{(l)}$，然后代入方程(5)，即可得到偏导数$\frac{\partial L}{\partial w_{ji}^{(l)}}$。
- 在计算误差$\delta_i^{(l)}$时，先算最后1个隐层的误差，再算倒数第2个隐层的误差，以此类推。
- 这种算法称为误差反向传播(error back propagation或backward pass)，简称BP算法(back-propagation algorithm)。

反向传播算法

- 在计算梯度向量时，依然需要知道每一层所有神经元的净输入$z_j^{(l)}$与激活值$a_j^{(l)}$。
- 故首先需要将每个观测值(X_i, Y_i)输入神经网络，从左到右进行正向传播(forward propagation或forward pass)，得到每一层所有神经元的$z_j^{(l)}$与$a_j^{(l)}$。
- 然后通过反向传播，计算每一层的误差$\delta_j^{(l)}$；再根据方程(5)计算每一层参数的偏导数，并通过梯度下降法更新参数。

A simple example



- $x_1 = 1, x_2 = 0.5, y = 4$
- Initial values: $w_1 = 0.5, w_2 = 1.5, w_3 = 2.3, w_4 = 3, w_5 = 1, w_6 = 1$
- S型激活函数：$f(z) = \Gamma(z) = \frac{e^z}{1+e^z}$

A simple example

- 首先，在计算反向传播之前我们需要计算正向传播，也即是预测的$z_1, z_2, a_1, a_2, \hat{y}$和$L$，假设$L = (y - \hat{y})^2$。

$$
\begin{aligned}
z_1 &= w_1 x_1 + w_2 x_2 = 1.25, z_2 = w_3 x_1 + w_4 x_2 = 3.8 \\
a_1 &= \Gamma(z_1) = 0.777, a_2 = \Gamma(z_2) = 0.978 \\
\hat{y} &= w_5 a_1 + w_6 a_2 = 1.755 \\
L &= (y - \hat{y})^2 = 5.04.
\end{aligned}
$$

A simple example

- 然后，计算反向传播。$\hat{y}$是神经网络预测的值，真实的输出是y。那么，要更新w_5的值我们就要算$\frac{\partial L}{\partial w_5}$，根据链式法则有

$$\frac{\partial L}{\partial w_5} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_5} = -2(y - \hat{y})a_1 = -3.489.$$

- 运用梯度下降法的公式更新w_5，假设学习率$s = 0.1$

$$w_5 \leftarrow w_5 - s \cdot \frac{\partial L}{\partial w_5} = 1.349.$$

A simple example

- 类似地，计算$\frac{\partial L}{\partial w_6}$

$$\frac{\partial L}{\partial w_6} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_6} = -2(y - \hat{y})a_2 = -4.391.$$

- 运用梯度下降法的公式更新w_6

$$w_6 \leftarrow w_6 - s \cdot \frac{\partial L}{\partial w_6} = 1.439.$$

A simple example

- 下面我们再来看w_1, w_2, w_3, w_4，由于这四个参数在同一层，所以求梯度的方法是相同的，因此这里仅展示对w_1的推导。根据链式法则

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1} \frac{\partial a_1}{\partial z_1} \frac{\partial z_1}{\partial w_1}.$$

- 其中$\frac{\partial L}{\partial \hat{y}}$已经求过了。而根据$\hat{y} = w_5 a_1 + w_6 a_2$和$a_1 = \Gamma(z_1)$，我们可以得到

$$\frac{\partial \hat{y}}{\partial a_1} = w_5 = 1$$
$$\frac{\partial a_1}{\partial z_1} = \Gamma'(z_1) = 0.173.$$

- 又根据$z_1 = w_1 x_1 + w_2 x_2$，我们可以得到

$$\frac{\partial z_1}{\partial w_1} = x_1 = 1.$$

A simple example

- 因此我们有下面的公式

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1} \frac{\partial a_1}{\partial z_1} \frac{\partial z_1}{\partial w_1} = -4.49 \times 1 \times 0.173 \times 1 = -0.777.$$

- 最后使用梯度下降法更新

$$w_1 \leftarrow w_1 - s \cdot \frac{\partial L}{\partial w_1} = 0.578.$$

- 使用类似方法可更新剩余参数：w_2, w_3, w_4。

Table of Contents

神经网络的目标函数

- 对于神经网络的训练，考虑最小化如下损失函数：

$$\min_{W} \frac{1}{N} \sum_{i=1}^{N} L(Y_i, G(X_i; W)),$$

其中， $G(X_i; W)$为一个前馈神经网络模型。由于$G(X_i; W)$通常是一个高度非线性的函数，故上式中的求和式无法进一步简化。

- 如果样本容量为100万，则目标函数中共有100万项相加(对应于100万个观测值的损失之和)。

随机梯度下降

- 在求损失函数的梯度向量时，需要对每个观测值的损失$L(Y_i, G(X_i; W))$分别求梯度向量，然后再将这100万个梯度向量加总。
- 如果样本容量很大，则通常的梯度下降法过于费时，并不可行。
- 一种解决方法是，每次无放回地(without replacement)随机抽取一个观测值(X_i, Y_i)，计算该观测值的梯度向量$\frac{\partial L(Y_i, G(X_i; W))}{\partial W}$，然后沿着负梯度方向，使用合适的学习率$s$，进行参数更新：

$$W \leftarrow W - s\frac{\partial L(Y_i, G(X_i; W))}{\partial W}\ominus$$

随机梯度下降

- 这种方法称为随机梯度下降(Stochastic Gradient Descent，简记SGD)，最早由Robbins and Monro(1951)与Kiefer and Wolfowitz(1952)提出。
- SGD的计算速度大大加快，因为每次仅需计算一个观测值的梯度向量。
- 但单个观测值的负梯度方向并不一定与整个样本的负梯度方向一致或类似，这导致随机梯度下降的过程充满噪音(noisy)，有时反而会使损失函数上升。
- 当然，经过不断迭代后，SGD 的长期趋势依然指向损失函数的最小值。

小批量梯度下降

- 为克服随机梯度下降的不稳定与噪音，一种折衷方法应运而生。
- 每次无放回地(without replacement)随机抽取部分观测值，比如B个观测值(例如$B=32$)，计算这B个观测值的梯度向量，再作平均，然后进行参数更新：

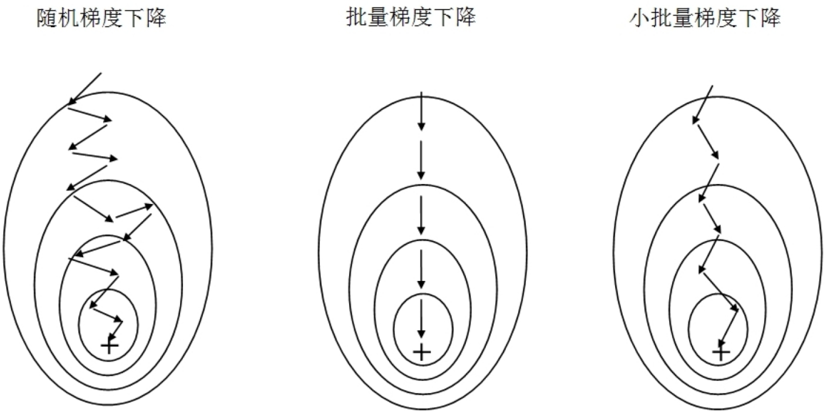
$$W \leftarrow W - s\frac{1}{B}\sum_{i=1}^{B}\frac{\partial L(Y_i, G(X_i; W))}{\partial W}.$$

- 这种方法称为小批量梯度下降(Mini-batch Gradient Descent)。
- 由于B通常不大，故小批量梯度下降依然计算较快。

批量梯度下降

- 经过对B个观测值平均之后，可得到对于全样本的真实梯度向量更为准确的估计，故小批量梯度下降的过程更为稳定。
- 传统的梯度下降法，在计算梯度向量时，同时考虑所有观测值，故称为批量梯度下降(Batch Gradient Descent)。
- 以上三种梯度下降的方法，主要区别在于其“批量规模”(batch size)。对于随机梯度下降，批量规模$B = 1$。
- 对于批量梯度下降，批量规模就是样本容量，即$B = N$。
- 对于小批量梯度下降，则$1 < B < N$；常见的B选择包括32，64，128或256(设为2的指数次方，以适应二进制的CPU或显卡GPU的内存)。

三种梯度下降算法



轮

- 另一相关概念为轮(epoch)。在训练模型时，将所有样本数据都用了一遍，即称为“一轮”(one epoch)。经过一轮之后，所有观测值都有机会影响参数更新。
- 对于批量梯度下降，每次迭代(iteration)都用全部样本计算梯度向量，故一次迭代就是一轮。
- 对于随机梯度下降，每次仅用一个观测值计算梯度向量，故N次迭代才算一轮(N为样本容量)。
- 对于小批量梯度下降，由于每次无放回地使用B个观测值计算梯度向量，故N/B(假设可整除)次迭代后，才算一轮。

Table of Contents

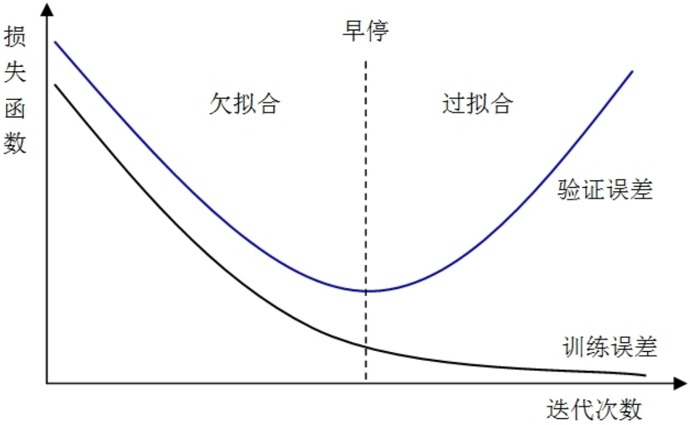
正则化

- 包含多个隐藏层的深度神经网络是表达能力很强的模型(very expressive models)，可学习输入与输出之间非常复杂的函数关系。
- 如果进行很多轮(epoch)的训练，则容易导致过拟合。
- 需要对神经网络模型进行“正则化”(regularization)处理。
- 常见的正则化方法包括早停，丢包和惩罚。

早停(Early Stopping)

- 早停：提前停止训练，而不必等到神经网络达到损失函数或训练误差的最小值。
- 一般建议将全样本随机地一分为三，即训练集(training set)、验证集(validation set)与测试集(test set)。
- 首先，在训练集中进行训练，并同时将学得的神经网络模型同步地在验证集中作预测，并计算“验证误差”(validation error)。
- 其次，当验证误差开始上升时，即停止训练。
- 最后，将所得的最终模型在测试集中进行预测，并计算“测试误差”(test error)。

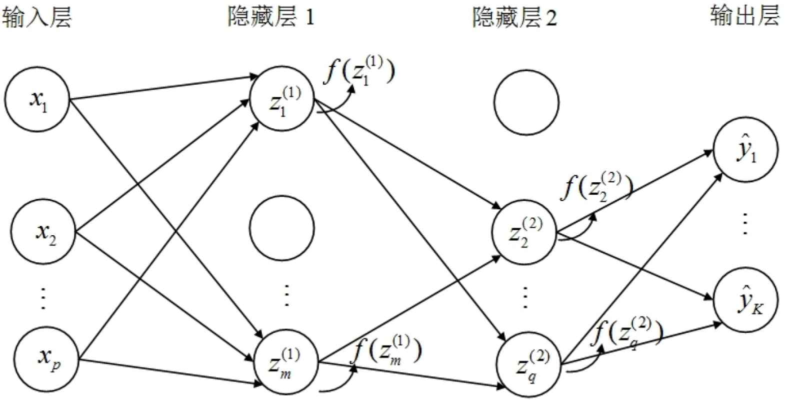
早停(Early Stopping)



丢包(Dropout)

- 为避免过拟合，Geoffrey Hinton的团队(Srivastava et al., 2014)提出，在训练样本时，随机地让某些神经元的激活值取值为0，即让某些神经元“死亡”，而不再影响神经网络。
- 通常随机地丢弃50%的神经元(以及它们在网络中的连接)，这样可以迫使神经网络不过分依赖于某些神经元而导致过拟合。

丢包(Dropout)



输入层
隐藏层 1
隐藏层 2
输出层
x_1
x_2
x_p
$z_1^{(1)}$
$z_m^{(1)}$
$f(z_1^{(1)})$
$f(z_m^{(1)})$
$z_2^{(2)}$
$z_q^{(2)}$
$f(z_2^{(2)})$
$f(z_q^{(2)})$
$\hat{y}_1$
$\hat{y}_K$

惩罚(Penalization)

- 在神经网络模型的目标函数中，可引入L_2惩罚项，以进行正则化：

$$\min_{W} \frac{1}{N} \sum_{i=1}^{N} L(Y_i, G(X_i; W)) + \lambda \|W\|_F^2,$$

  其中，$\|W\|_F$为矩阵W的“弗罗贝尼乌斯范数”(Frobenius norm)，即矩阵W所有元素的平方和之开根号；而λ为调节参数，可通过验证集法确定。

- 这是一个“收缩估计量”(shrinkage estimator)，但在神经网络的文献中，则称为权重衰减(weight decay)。