

第六章 神经网络与深度学习

周世斌

中国矿业大学 计算机学院

May. 2022

1 Introduction

- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

2 前馈神经网络与 BP 算法

- 前馈神经网络
- 链式规则
- Forward computation
- Backward Computation
- BP 算法的矩阵与向量表示
- 交叉熵损失函数

In the linear regression models, the model prediction $f(\mathbf{x})$ was given by a linear function of the parameters \mathbf{w} . In the simplest case, the model is also linear in the input variables and therefore takes the form

$$f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$$

so that f is a real number. For classification problems, however, we wish to predict discrete class labels. To achieve this, we consider a generalization of this model in which we transform the linear function of \mathbf{w} using a nonlinear function $g(\cdot)$ so that

$$f(\mathbf{x}) = g(\mathbf{w}^T \mathbf{x} + w_0)$$

In the machine learning literature $g(\cdot)$ is known as an **activation function**, whereas its inverse is called a **link function** in the statistics literature. The decision surfaces correspond to $f(\mathbf{x}) = \text{constant}$, so that

$$\mathbf{w}^T \mathbf{x} + w_0 = \text{constant}$$

and hence the decision surfaces are linear functions of \mathbf{x} , even if the function $g(\cdot)$ is nonlinear.

1 Introduction

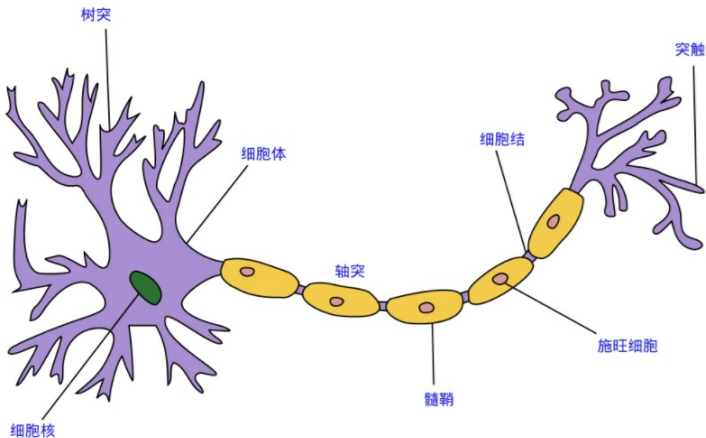
- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

2 前馈神经网络与 BP 算法

- 前馈神经网络
- 链式规则
- Forward computation
- Backward Computation
- BP 算法的矩阵与向量表示
- 交叉熵损失函数

人工神经网络

一个神经网络是一个由简单处理元构成规模宏大的并行分布式处理器，天然具有存储经验知识，并进行使用的特性。

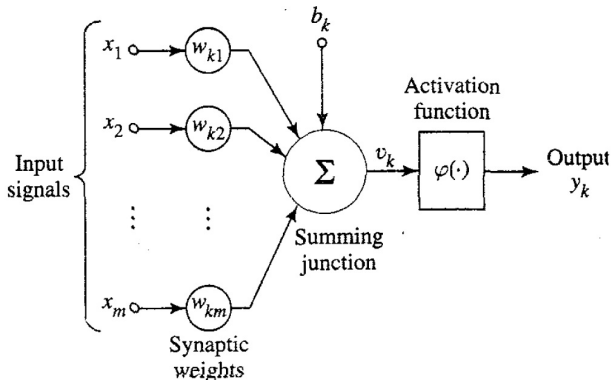


人工神经网络在两个方面与人脑相似

- ① 人工神经网络获取的知识，是从外界环境学习得来的
- ② 互联的神经元的连接强度，即突触权值，用于存储获取的知识，学习算法，以有序的方式，改变网络的突触权值以获得想要的目标。

人工神经元模型，MP 模型

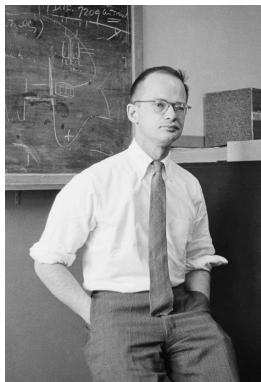
- ① 突触或连接链
- ② 加法器
- ③ 激活函数



v 是神经元的局部输出。 $v = \sum_{j=1}^p w_j x_j + b = \mathbf{w}^T \mathbf{x} + b$ ，而输出

$y = \begin{cases} 1 & v \geq 0 \\ -1 & v < 0 \end{cases}$ 这样一个神经元在文献中称为 McCulloch-Pitts 模型，以纪念 McCulloch 和 Pitts 的开拓性工作。

McCulloch and Pitts



The Doctor and the Hobo..

- Warren McCulloch: Neurophysician
- Walter Pitts: Homeless wannabe logician who arrived at his door

1 Introduction

- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

2 前馈神经网络与 BP 算法

- 前馈神经网络
- 链式规则
- Forward computation
- Backward Computation
- BP 算法的矩阵与向量表示
- 交叉熵损失函数

常用激活函数

激活函数的选择是构建神经网络过程中的重要环节，下面简要介绍常用的激活函数。

- ① 阈值函数
- ② Relu 函数
- ③ Sigmoid 函数 (Sigmoid Function)
- ④ tanh 函数 (tanh Function)
- ⑤ ...

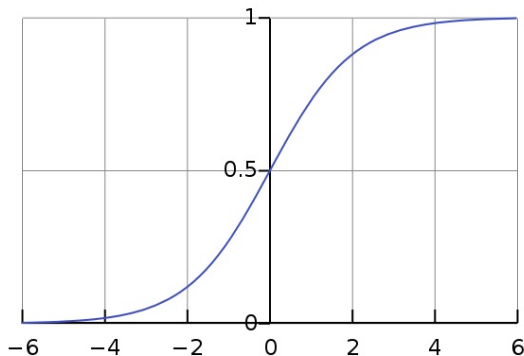
Sigmoid

Sigmoid 非线性函数将输入映射到

$(0, 1)$

之间。它的数学公式为：

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



历史上, sigmoid 函数曾非常常用, 然而现在它已经不太受欢迎, 实际很少使用了, 因为它主要有两个缺点:

1. 函数饱和使梯度消失

sigmoid 神经元在值为 0 或 1 的时候接近饱和, 这些区域, 梯度几乎为 0。因此在反向传播时, 这个局部梯度会与整个代价函数关于该单元输出的梯度相乘, 结果也会接近为 0。

这样, 几乎就没有信号通过神经元传到权重再到数据了, 因此这时梯度就对模型的更新没有任何贡献。

除此之外, 为了防止饱和, 必须对于权重矩阵的初始化特别留意。比如, 如果初始化权重过大, 那么大多数神经元将会饱和, 导致网络就几乎不学习。

2. sigmoid 函数不是关于原点中心对称的

这个特性会导致后面网络层的输入也不是零中心的，进而影响梯度下降的运作。

因为如果输入都是正数的话（如 $f = w^T x + b$ 中每个元素都 $x > 0$ ），那么关于 w 的梯度在反向传播过程中，要么全是正数，要么全是负数（具体依据整个表达式 f 而定），这将会导致梯度下降权重更新时出现 z 字型的下降。

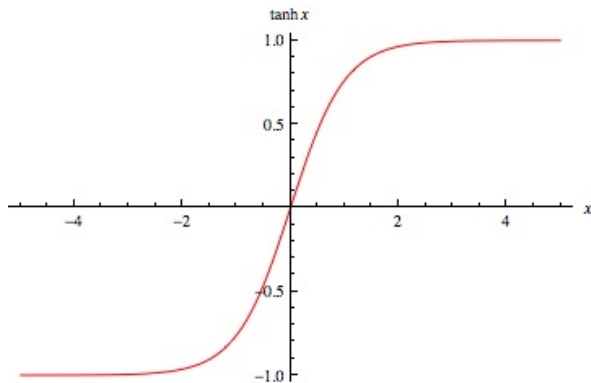
当然，如果是按 batch 去训练，那么每个 batch 可能得到不同的信号，整个批量的梯度加起来后可以缓解这个问题。因此，该问题相对于上面的神经元饱和问题来说只是个小麻烦，没有那么严重。

Tanh

tanh 函数同样存在饱和问题，但它的输出是零中心的，因此实际中 tanh 比 sigmoid 更受欢迎。

tanh 函数实际上是一个放大的 sigmoid 函数，数学关系为：

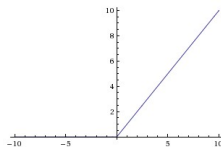
$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



ReLU

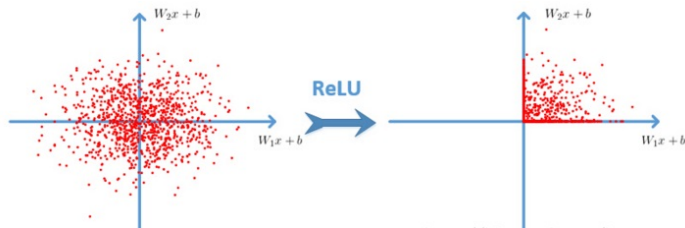
ReLU 近些年来非常流行。它的数学公式为：

$$f(x) = \max(0, x)$$



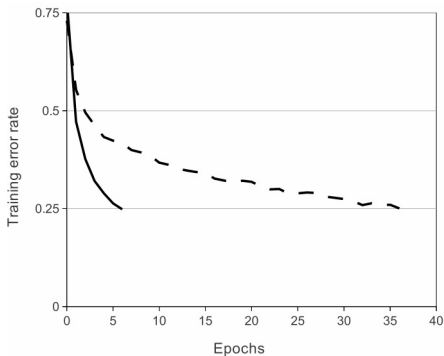
w

是二维时，ReLU 的效果如图：



ReLU 的优点:

1. 相较于 sigmoid 和 tanh 函数, ReLU 对于 SGD 的收敛有巨大的加速作用 (Alex Krizhevsky 指出有 6 倍之多)。有人认为这是由它的线性、非饱和的公式导致的。



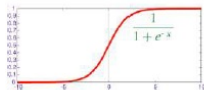
2. 相比于 sigmoid/tanh，ReLU 只需要一个阈值就可以得到激活值，而不用去算一大堆复杂的（指数）运算。ReLU 的缺点是，它在训练时比较脆弱并且可能“死掉”。

举例来说：一个非常大的梯度经过一个 ReLU 神经元，更新过参数之后，这个神经元再也不会对任何数据有激活现象了。如果这种情况发生，那么从此所有流过这个神经元的梯度将都变成 0。

也就是说，这个 ReLU 单元在训练中将不可逆转的死亡，导致了数据多样化的丢失。实际中，如果学习率设置得太高，可能会发现网络中 40% 的神经元都会死掉（在整个训练集中这些神经元都不会被激活）。

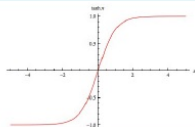
合理设置学习率，会降低这种情况的发生概率。

Activations and their derivatives



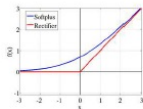
$$f(z) = \frac{1}{1 + \exp(-z)}$$

$$f'(z) = f(z)(1 - f(z))$$



$$f(z) = \tanh(z)$$

$$f'(z) = (1 - f^2(z))$$



$$f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

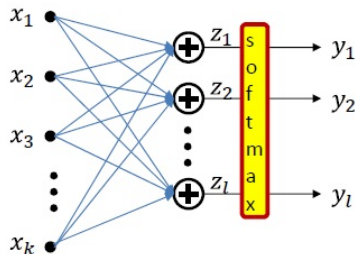
This space left intentionally (kind of) blank

$$f(z) = \log(1 + \exp(z))$$

$$f'(z) = \frac{1}{1 + \exp(-z)}$$

Some popular activation functions and their derivatives

Vector activation example: Softmax



Example: Softmax vector activation

$$z_i = \sum_j w_{ji} x_j + b_i$$

$$y = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Parameters are weights w_{ij} and bias b_i

1 Introduction

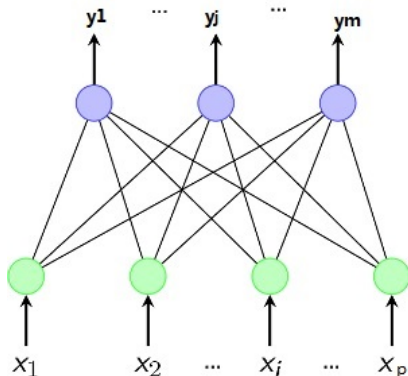
- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

2 前馈神经网络与 BP 算法

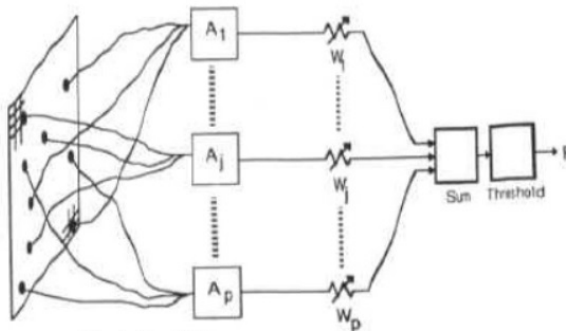
- 前馈神经网络
- 链式规则
- Forward computation
- Backward Computation
- BP 算法的矩阵与向量表示
- 交叉熵损失函数

最简单的神经网络结构，感知器

在 1958 年, 美国心理学家 Frank Rosenblatt 提出一种具有单层计算单元的神经网络, 称为感知器 (Perceptron)。它其实就是基于 M-P 模型的结构。我们可以看看它的拓扑结构图。



Perceptron(1957)

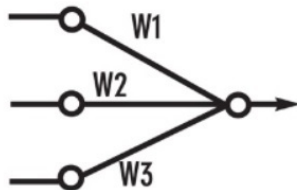


Frank Rosenblatt
(1928-1971)

Original Perceptron

(From *Perceptrons* by M. L. Minsky and S. Papert,
1969, Cambridge, MA: MIT Press. Copyright 1969
by MIT Press.)

Simplified model:



这个结构非常简单，其实就是输入输出两层神经元之间的简单连接。
输入层各节点的输入加权和

$$v_j = \sum_{i=1}^p w_{ij}x_i + b_j = \mathbf{w}_j^T \mathbf{x} + b_j$$

我们一般采用符号函数来当作单层感知器的传递函数，即输出

$$y_j = \text{sgn}(\mathbf{w}_j^T \mathbf{x} + b_j)$$

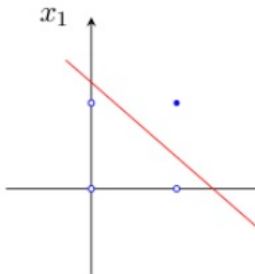
可以进一步表达为：

$$y_j = \begin{cases} 1 & v_j \geq 0 \\ -1 & v_j < 0 \end{cases}$$

单层感知器的局限性

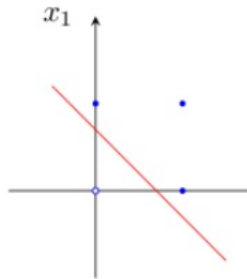
虽然单层感知器简单而优雅，但它显然不够聪明——它仅对线性问题具有分类能力。什么是线性问题呢？简单来讲，就是用一条直线可分的图形。比如，逻辑“与”和逻辑“或”就是线性问题，我们可以用一条直线来分隔 0 和 1。

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



逻辑“与”的真值表和二维样本图

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



逻辑“或”的真值表和二维样本图

为什么感知器就可以解决线性问题呢？这是由它的传递函数决定的。这里以两个输入分量 x_1 和 x_2 组成的二维空间为例，此时节点 j 的输出为

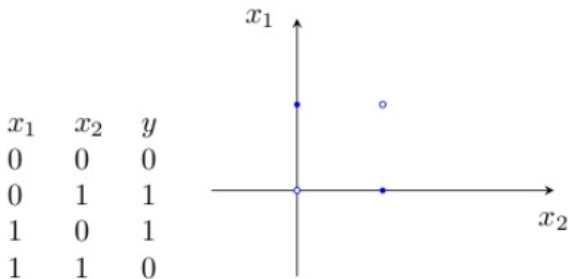
$$y_j = \begin{cases} 1 & w_{1j}x_1 + w_{2j}x_2 + b_j \geq 0 \\ -1 & w_{1j}x_1 + w_{2j}x_2 + b_j < 0 \end{cases}$$

所以，方程

$$w_{1j}x_1 + w_{2j}x_2 + b_j = 0$$

确定的直线就是二维输入样本空间上的一条分界线。

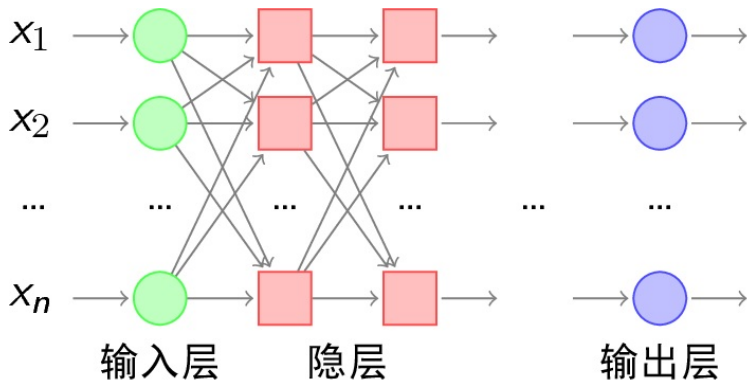
如果要用它来处理非线性的问题，单层感知器网就无能为力了。例如下面的“异或”，就无法用一条直线来分割开来，因此单层感知器网就没办法实现“异或”的功能。



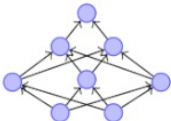
逻辑“异或”的真值表和二维样本图

既然一条直线无法解决分类问题，当然就会有人想到用弯曲的折线来进行样本分类。人们就想到了多层感知器。

所谓多层感知器，就是在输入层和输出层之间加入隐层，以形成能够将样本正确分类的凸域。多层感知器的拓扑结构如下图所示。



我们可以比较一下单层感知器和多层感知器的分类能力：

结构	决策区域类型	区域形状	异或问题
无隐层 	由一超平面分成两个		
单隐层 	开凸区域或闭凸区域		
双隐层 	任意形状（其复杂度由单元数目确定）		

由上图可以看出，随着隐层层数的增多，凸域将可以形成任意的形状，因此可以解决任何复杂的分类问题。实际上，Kolmogorov 理论指出：双隐层感知器就足以解决任何复杂的分类问题。

多层感知器确实是非常理想的分类器，但问题也随之而来：隐层的权值怎么训练？对于各隐层的节点来说，它们并不存在期望输出，所以也无法通过感知器的学习规则来训练多层感知器。因此，多层感知器心有余而力不足，虽然武功高强，但却无力可施。

1 Introduction

- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

2 前馈神经网络与 BP 算法

- 前馈神经网络
- 链式规则
- Forward computation
- Backward Computation
- BP 算法的矩阵与向量表示
- 交叉熵损失函数

1966 年, Minisky 和 Papert 在他们的《感知器》一书中提出了上述的感知器的研究瓶颈, 指出理论上还不能证明将感知器模型扩展到多层网络是有意义的。这在人工神经网络的历史上书写了极其灰暗的一章。对 ANN 的研究, 始于 1890 年开始于美国著名心理学家 W.James 对于人脑结构与功能的研究, 半个世纪后 W.S.McCulloch 和 W.A.Pitts 提出了 M-P 模型, 之后的 1958 年 Frank Rosenblatt 在这个基础上又提出了感知器, 此时对 ANN 的研究正处在升温阶段, 《感知器》这本书的出现就刚好为这刚刚燃起的人工神经网络之火泼了一大盆冷水。一时间人们仿佛感觉对以感知器为基础的 ANN 的研究突然间走到尽头, 看不到出路了。于是, 几乎所有为 ANN 提供的研究基金都枯竭了, 很多领域的专家纷纷放弃了这方面课题的研究。

ANN 研究的复苏和 BP 神经网络的诞生

所以说真理的果实总是垂青于能够忍受寂寞的科学家。尽管 ANN 的研究陷入了前所未有的低谷，但仍有为数不多的学者忍受住寂寞，坚持致力于 ANN 的研究。在长达 10 年的低潮时期之间，相继有一些开创性的研究成果被提出来，但还不足以激起人们对于 ANN 研究的热情。一直到上世纪 80 年代，两个璀璨的成果诞生了：1982 年美国加州理工学院的物理学家 John J.Hopfield 博士的 Hopfield 网络和 David E.Rumelhart 以及 James L.McCelland 研究小组发表的《并行分布式处理》。这两个成果重新激起了人们对 ANN 的研究兴趣，使人们对模仿脑信息处理的智能计算机的研究重新充满了希望。

前者暂不讨论，后者对具有非线性连续变换函数的多层感知器的误差反向传播 (Error Back Propagation) 算法进行了详尽的分析，实现了 Minsky 关于多层网络的设想

1 Introduction

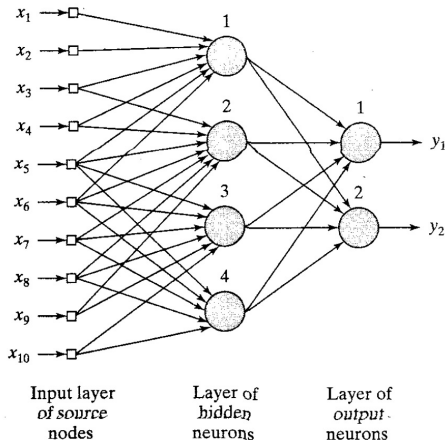
- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

2 前馈神经网络与 BP 算法

- 前馈神经网络
- 链式规则
- Forward computation
- Backward Computation
- BP 算法的矩阵与向量表示
- 交叉熵损失函数

怎样在神经网络设计中加入先验信息

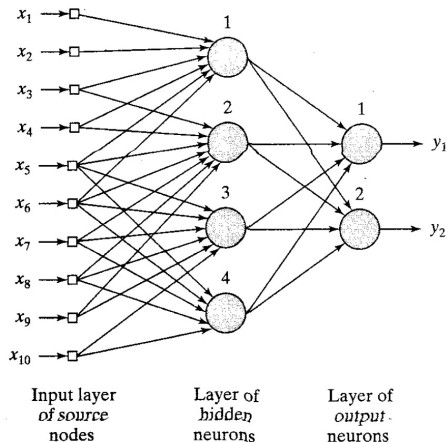
以此建立一种特定的网络结构，是必须考虑的重要问题。目前主要使用下面两种技术的结合



- 1 通过使用称为接收域的局部连接，限制网络结构
- 2 通过使用权值共享，限制突触权值的选择

联合利用接收域和权值共享的图例。所有四个隐神经元共享他们突触连接的不同权值集

怎样在神经网络设计中加入先验信息



每个隐藏神经元有 6 个局部连接，共有 4 个隐藏神经元，我们可以表示每个隐藏神经元的诱导局部域如下：

$$v_j = \sum_{i=1}^6 w_i x_{i+j-1}, j = 1, 2, 3, 4$$

其中， $\{w_i\}_{i=1}^6$ 构成所有四个隐藏神经元共享的同一权值集。

这里描述的前馈网络使用局部连接和权值共享的方式，这样的前馈网络称为卷积网络。

1 Introduction

- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

2 前馈神经网络与 BP 算法

- 前馈神经网络
- 链式规则
- Forward computation
- Backward Computation
- BP 算法的矩阵与向量表示
- 交叉熵损失函数

1 Introduction

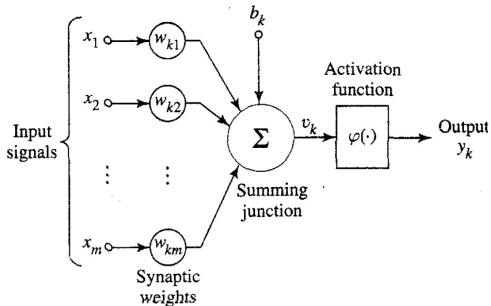
- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

2 前馈神经网络与 BP 算法

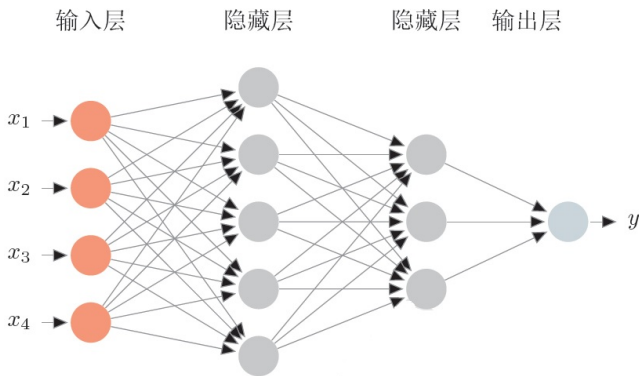
- 前馈神经网络
- 链式规则
- Forward computation
- Backward Computation
- BP 算法的矩阵与向量表示
- 交叉熵损失函数

人工神经元模型

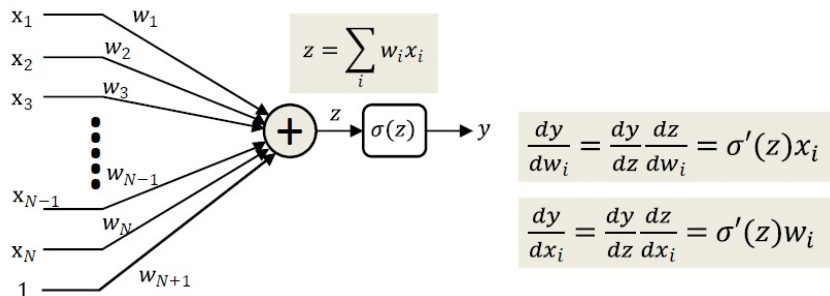
- ① 突触或连接链
- ② 加法器
- ③ 激活函数（在这里我们主要关注阈值函数）



前馈神经网络 (Feed-forward Neural Network), 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。

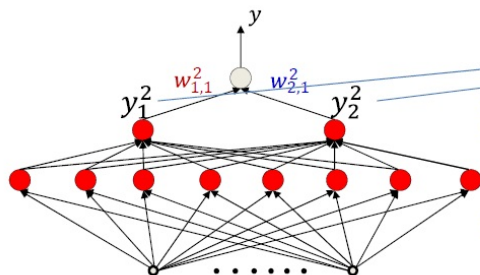


Perceptrons with differentiable activation functions



- ① $\sigma(z)$ is a differentiable function of z $\frac{d\sigma(z)}{dz}$ is well-defined and finite for all z
- ② Using the chain rule, y is a differentiable function of both inputs x_i and weights w_i
- ③ This means that we can compute the change in the output for small changes in either the input or the weights

Overall network is differentiable



$$y = \sigma(w_{1,1}^2 y_1^2 + w_{2,1}^2 y_2^2 + w_{3,1}^2)$$

y = output of overall network

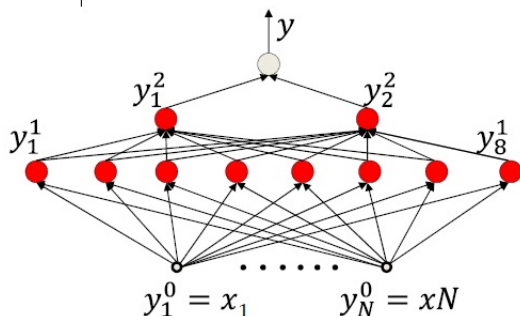
$w_{i,j}^k$ = weight connecting the i th unit of the k th layer to the j th unit of the $k+1$ -th layer

y_i^k = output of the i th unit of the k th layer

$\sigma()$ is differentiable w.r.t both w and y_i^k

- 1 Every individual perceptron is differentiable w.r.t its inputs and its weights (including “bias” weight)
- 2 By the chain rule, the overall function is differentiable w.r.t every parameter (weight or bias), Small changes in the parameters result in measurable changes in output

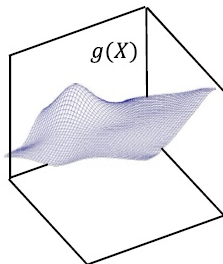
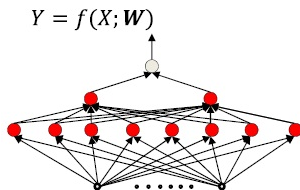
Overall function is differentiable



$$y_j^k = \sigma \left(\sum_i w_{i,j}^{k-1} y_i^{k-1} \right)$$

- The overall function is differentiable w.r.t every parameter
 - Small changes in the parameters result in measurable changes in the output
 - We will derive the actual derivatives using the chain rule later

Learning the function



- When $f(X; \mathbf{W})$ has the capacity to exactly represent $g(X)$

$$\widehat{\mathbf{W}} = \arg \min_{\mathbf{W}} \int_X \text{div}(f(X; \mathbf{W}), g(\mathbf{W})) dX$$

- $\text{div}()$ is a divergence function that goes to zero when $f(X; \mathbf{W}) = g(X)$

Training Neural Nets through Gradient Descent

Total training Error

$$\text{Err} = \frac{1}{T} \sum_t \text{Div}(Y_t, d_t)$$

- Gradient descent algorithm:
- Initialize all weights and biases $\{w_{ij}^{(k)}\}$
 - Using the extended notation: the bias is also a weight
- Do:
 - For every layer for all ij update:

$$w_{ij}^{(k)} = w_{ij}^{(k)} - \eta \frac{d\text{Err}}{dw_{ij}^{(k)}}$$

- Until has converged

The derivative

Total training Error

$$\text{Err} = \frac{1}{T} \sum_t \text{Div}(Y_t, d_t)$$

Compute the derivative

Total derivative

$$\frac{d\text{Err}}{dw_{ij}^{(k)}} = \frac{1}{T} \sum_t \frac{d \text{Div}(Y_t, d_t)}{dw_{ij}^{(k)}}$$

Training by Gradient Descent

- Initialize all weights $\{w_{ij}^{(k)}\}$
- Do:
 - For all i, j, k , initialize $\frac{dErr}{dw_{ij}^{(k)}} = 0$:
 - For all $t = 1 : T$
 - compute $\frac{d \text{Div}(Y_t, d_t)}{dw_{ij}^{(k)}}$
 - Compute $\frac{dErr}{dw_{ij}^{(k)}} + = \frac{d \text{Div}(Y_t, d_t)}{dw_{ij}^{(k)}}$
 - For every layer k for all i, j

$$w_{ij}^{(k)} = w_{ij}^{(k)} - \frac{\eta}{T} \frac{dErr}{dw_{ij}^{(k)}}$$

- Until has converged

The derivative

Total training Error

$$\text{Err} = \frac{1}{T} \sum_t \text{Div}(Y_t, d_t)$$

Total derivative

$$\frac{d\text{Err}}{dw_{ij}^{(k)}} = \frac{1}{T} \sum_t \frac{d \text{Div}(Y_t, d_t)}{dw_{ij}^{(k)}}$$

- So we must first figure out how to compute the derivative of divergences of individual training inputs

关键问题

- How to compute $\frac{d \text{Div}(Y_t, d_t)}{dw_{ij}^{(k)}}$
- Back Propagation

1 Introduction

- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

2 前馈神经网络与 BP 算法

- 前馈神经网络
- 链式规则
- Forward computation
- Backward Computation
- BP 算法的矩阵与向量表示
- 交叉熵损失函数


Basic rules of calculus

For any differentiable function

$$y = f(x)$$

with derivative

$$\frac{dy}{dx}$$

the following must hold for sufficiently small Δx  $\Delta y \approx \frac{dy}{dx} \Delta x$

For any differentiable function

$$y = f(x_1, x_2, \dots, x_M)$$

with partial derivatives

$$\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \dots, \frac{\partial y}{\partial x_M}$$

the following must hold for sufficiently small $\Delta x_1, \Delta x_2, \dots, \Delta x_M$

$$\Delta y \approx \frac{\partial y}{\partial x_1} \Delta x_1 + \frac{\partial y}{\partial x_2} \Delta x_2 + \dots + \frac{\partial y}{\partial x_M} \Delta x_M$$

Chain rule

For any nested function $y = f(g(x))$

$$\frac{dy}{dx} = \frac{\partial y}{\partial g(x)} \frac{dg(x)}{dx}$$

Check - we can confirm that : $\Delta y = \frac{dy}{dx} \Delta x$

$$z = g(x) \Rightarrow \Delta z = \frac{dg(x)}{dx} \Delta x$$

$$y = f(z) \Rightarrow \Delta y = \frac{dy}{dz} \Delta z = \frac{dy}{dz} \frac{dg(x)}{dx} \Delta x$$



Distributed Chain rule

$$y = f(g_1(x), g_1(x), \dots, g_M(x))$$

$$\frac{dy}{dx} = \frac{\partial y}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial y}{\partial g_2(x)} \frac{dg_2(x)}{dx} + \dots + \frac{\partial y}{\partial g_M(x)} \frac{dg_M(x)}{dx}$$

Check: $\Delta y = \frac{dy}{dx} \Delta x$

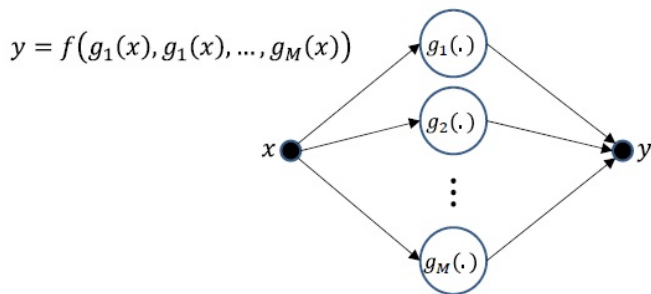
$$\Delta y = \frac{\partial y}{\partial g_1(x)} \Delta g_1(x) + \frac{\partial y}{\partial g_2(x)} \Delta g_2(x) + \dots + \frac{\partial y}{\partial g_M(x)} \Delta g_M(x)$$

$$\Delta y = \frac{\partial y}{\partial g_1(x)} \frac{dg_1(x)}{dx} \Delta x + \frac{\partial y}{\partial g_2(x)} \frac{dg_2(x)}{dx} \Delta x + \dots + \frac{\partial y}{\partial g_M(x)} \frac{dg_M(x)}{dx} \Delta x$$

$$\Delta y = \left(\frac{\partial y}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial y}{\partial g_2(x)} \frac{dg_2(x)}{dx} + \dots + \frac{\partial y}{\partial g_M(x)} \frac{dg_M(x)}{dx} \right) \Delta x$$

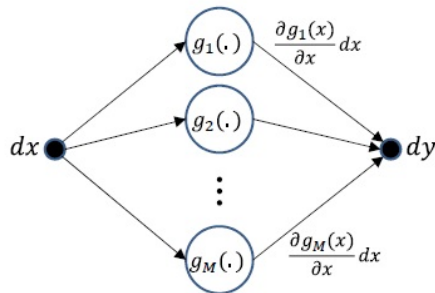


Distributed Chain Rule: Influence Diagram



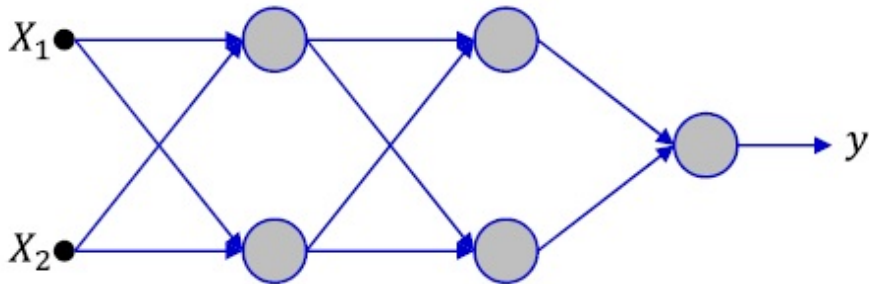
x affects y through each of g_1, \dots, g_m

Distributed Chain Rule: Influence Diagram



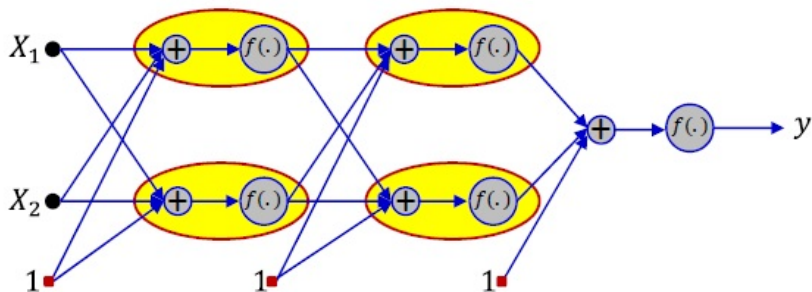
Small perturbations in x cause small perturbations in each of each of g_1, \dots, g_m which individually additively perturbs y

A first closer look at the network



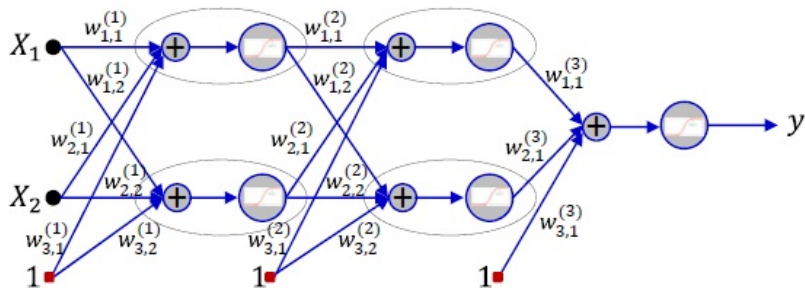
- Showing a tiny 2-input network for illustration
Actual network would have many more neurons and inputs

A first closer look at the network



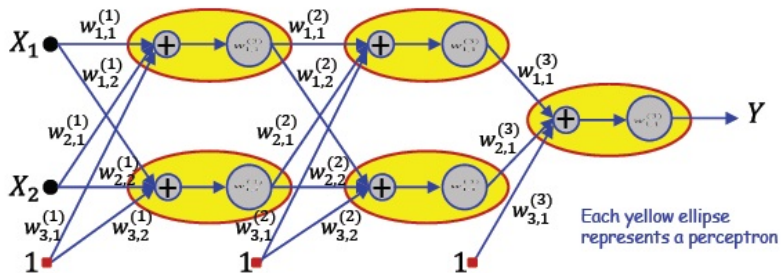
- Showing a tiny 2-input network for illustration
Actual network would have many more neurons and inputs
- Explicitly separating the weighted sum of inputs from the activation

A first closer look at the network



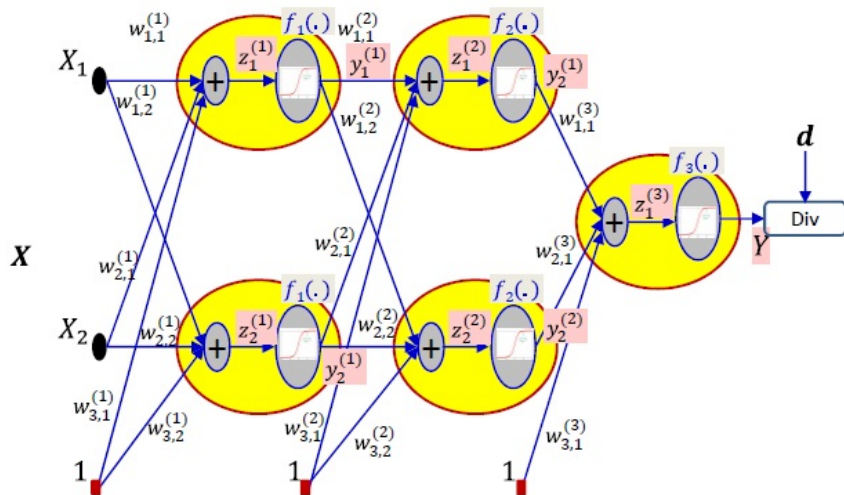
- Showing a tiny 2-input network for illustration
Actual network would have many more neurons and inputs
- Explicitly separating the weighted sum of inputs from the activation
- The overall function is differentiable w.r.t every weight, bias and input

A first closer look at the network



- Aim: compute derivative $\text{Div}(Y, d)$ of w.r.t. each of the weights
- But first, let's label all our variables and activation functions

A first closer look at the network



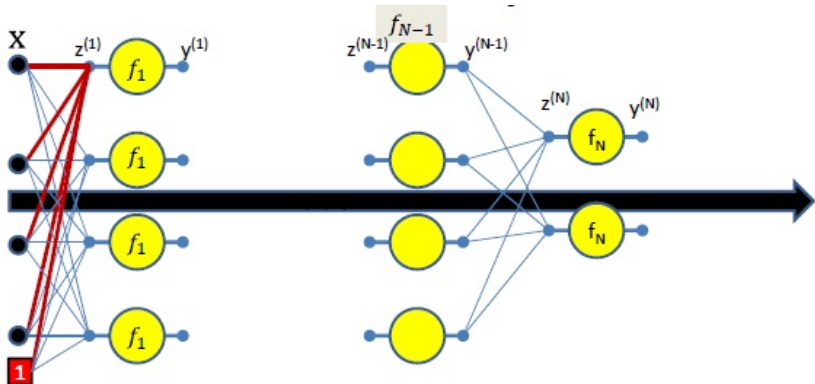
1 Introduction

- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

2 前馈神经网络与 BP 算法

- 前馈神经网络
- 链式规则
- **Forward computation**
- Backward Computation
- BP 算法的矩阵与向量表示
- 交叉熵损失函数

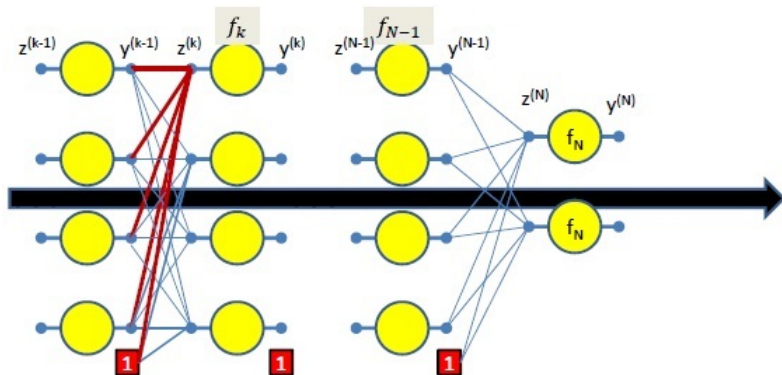
Forward computation



$$z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i$$

Assume $w_{0j}^{(1)} = b_j^{(1)}$ and $x_0 = 1$

Forward computation

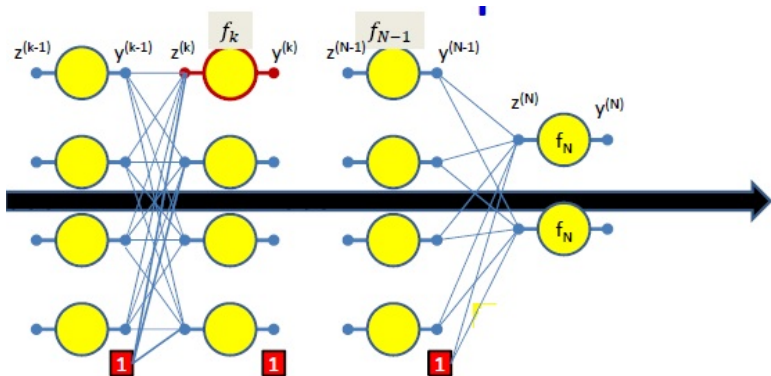


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i$$

$$z_j^{(k)} = \sum_i w_{ij}^{(k)} y_i^{(k-1)}$$

Assume $w_{0j}^{(k)} = b_j^{(k)}$ and $y_0^{(k-1)} = 1$

Forward computation

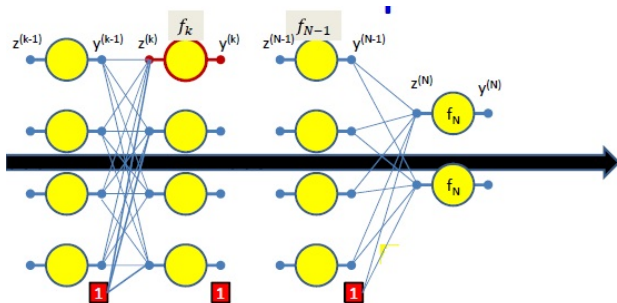


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i$$

$$z_j^{(k)} = \sum_i w_{ij}^{(k)} y_i^{(k-1)}$$

$$y_j^{(k)} = f(z_j^{(k)})$$

Forward computation



- Iterate for $k = 1 : N$
 - for $j = 1 : \text{layer-width}$

$$y_i^{(0)} = x_i$$

$$z_j^{(k)} = \sum_i w_{ij}^{(k)} y_i^{(k-1)}$$

$$y_j^{(k)} = f(z_j^{(k)})$$

Forward "Pass"

- Input: D dimensional vector $\mathbf{x} = \{x_j, j = 1 \dots D\}$
- Set
 - $D_0 = D$, is the width of the 0-th (input) layer
 - $y_j^{(0)} = x_j, j = 1 \dots D; y_0^{(k=1 \dots N)} = x_0 = 1$
- Iterate for $k = 1 : N$
 - for $j = 1 : \text{layer-width}$
$$z_j^{(k)} = \sum_i w_{ij}^{(k)} y_i^{(k-1)}$$
$$y_j^{(k)} = f(z_j^{(k)})$$
- Output: $Y = y_j^{(N)}, j = 1 \dots D_N$

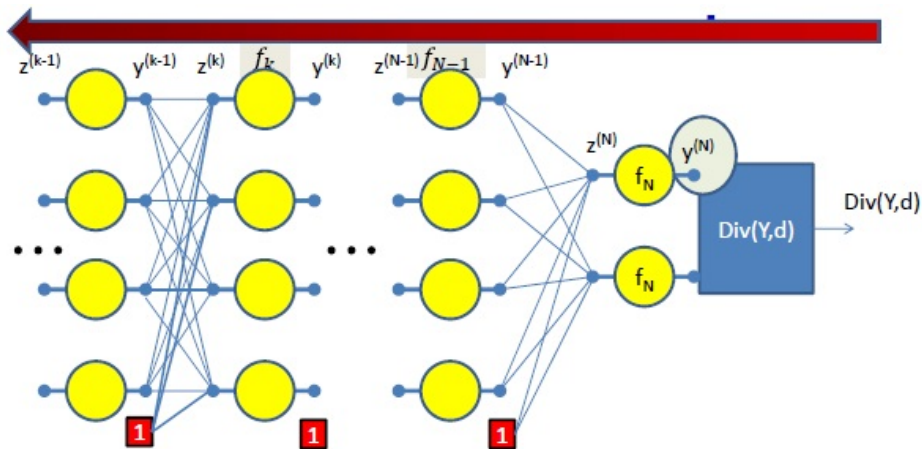
1 Introduction

- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

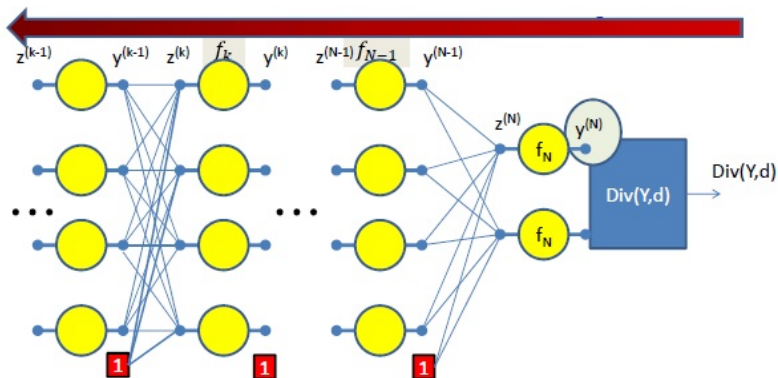
2 前馈神经网络与 BP 算法

- 前馈神经网络
- 链式规则
- Forward computation
- **Backward Computation**
- BP 算法的矩阵与向量表示
- 交叉熵损失函数

Gradients: Backward Computation

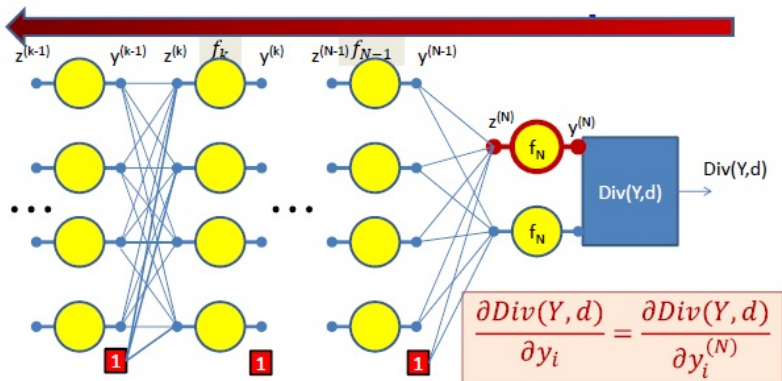


Gradients: Backward Computation

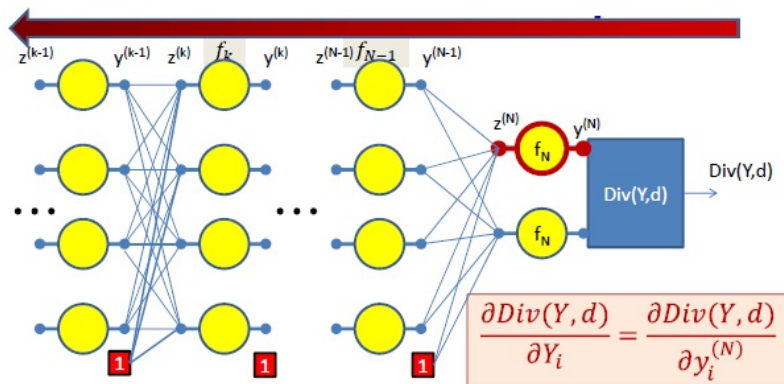


$$\frac{\partial \text{Div}(Y, d)}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}}$$

Gradients: Backward Computation



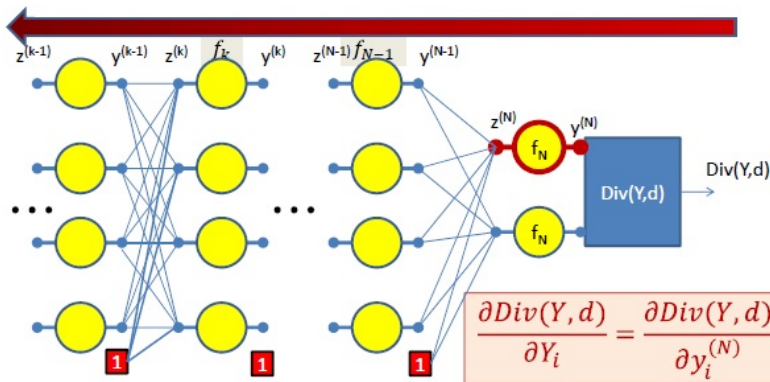
Gradients: Backward Computation



$z_i^{(N)}$ computed during the forward pass

$$\frac{\partial \text{Div}}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial \text{Div}}{\partial Y_i} = f'_N(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}}$$

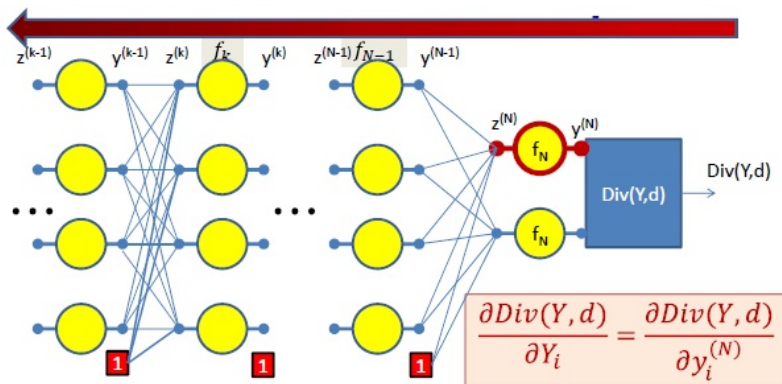
Gradients: Backward Computation



Derivative of the activation function of Nth layer

$$\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial Div}{\partial Y_i} = f'_N(z_i^{(N)}) \frac{\partial Div}{\partial y_i^{(N)}}$$

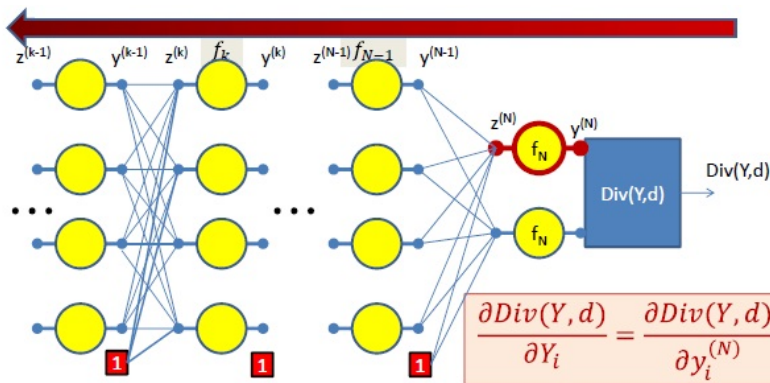
Gradients: Backward Computation



$z_i^{(N)}$ computed during the forward pass

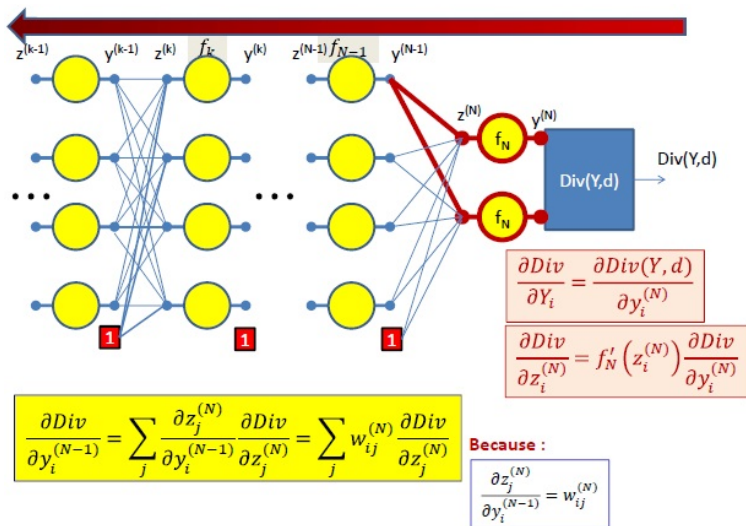
$$\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial Div}{\partial Y_i} = f'_N(z_i^{(N)}) \frac{\partial Div}{\partial y_i^{(N)}}$$

Gradients: Backward Computation

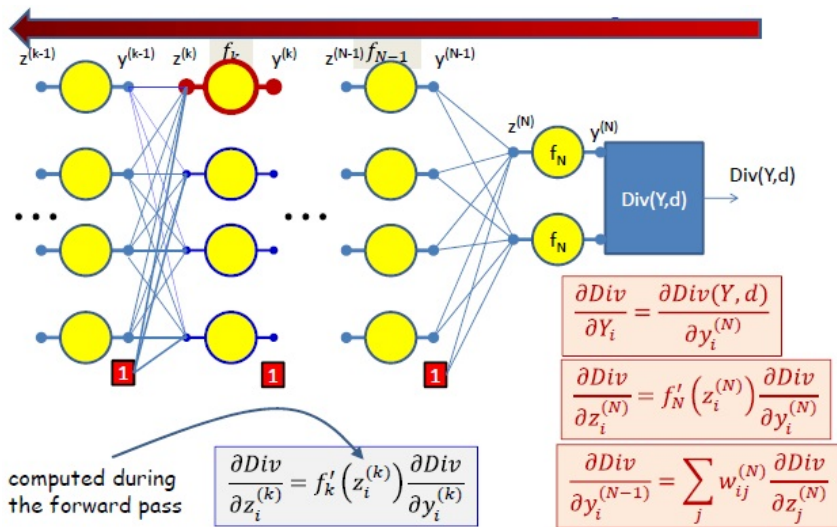


Derivative of the activation function of Nth layer

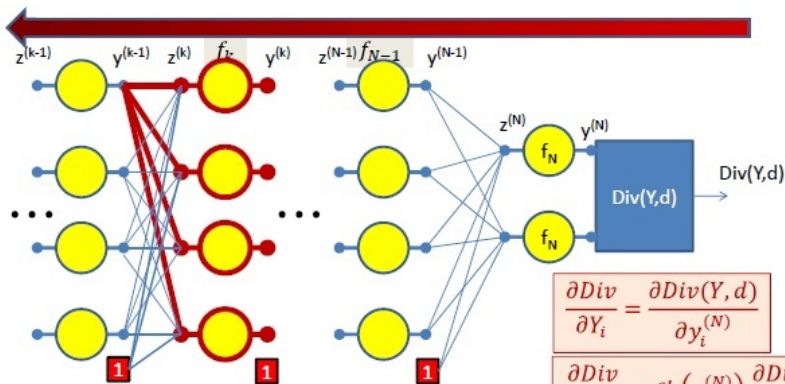
Gradients: Backward Computation



Gradients: Backward Computation



Gradients: Backward Computation

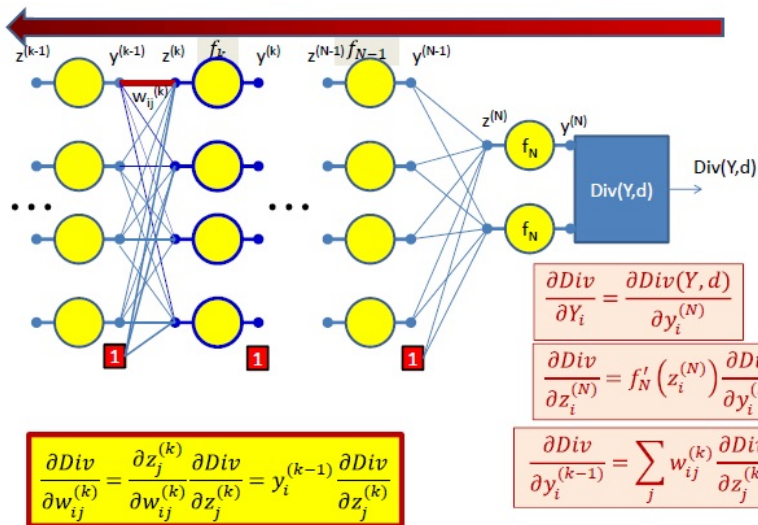


$$\frac{\partial Div}{\partial Y_i} = \frac{\partial Div(Y, d)}{\partial y_i^{(N)}}$$

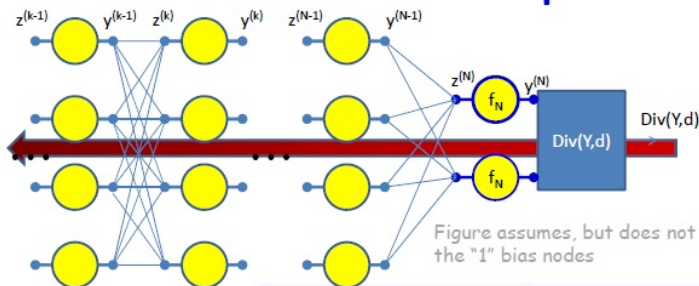
$$\frac{\partial Div}{\partial z_i^{(N)}} = f'_N(z_i^{(N)}) \frac{\partial Div}{\partial y_i^{(N)}}$$

$$\frac{\partial Div}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial Div}{\partial z_j^{(k)}} = \sum_j w_{ij}^{(k)} \frac{\partial Div}{\partial z_j^{(k)}}$$

Gradients: Backward Computation



Gradients: Backward Computation



Initialize: Gradient
w.r.t network output

$$\frac{\partial \text{Div}}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}}$$

For $k = N..1$

For $i = 1:\text{layer} - \text{width}$

$$\frac{\partial \text{Div}}{\partial z_i^{(k)}} = f'_k(z_i^{(k)}) \frac{\partial \text{Div}}{\partial y_i^{(k)}}$$

$$\frac{\partial \text{Div}}{\partial y_i^{(k-1)}} = \sum_j w_{ij}^{(k)} \frac{\partial \text{Div}}{\partial z_j^{(k)}}$$

$$\frac{\partial \text{Div}}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial \text{Div}}{\partial z_j^{(k)}}$$

- Output layer (N):

- for $i = 1 \dots D_N$
$$\frac{\partial \text{Div}}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}}$$

$$\frac{\partial \text{Div}}{\partial z_i^{(k)}} = \frac{\partial \text{Div}}{\partial y_i^{(k)}} \frac{\partial y_i^{(k)}}{\partial z_i^{(k)}}$$

- for layer $k=N-1$ downto 0:

- for $i = 1 \dots D_k$
$$\frac{\partial \text{Div}}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$$

$$\frac{\partial \text{Div}}{\partial z_i^{(k)}} = \frac{\partial \text{Div}}{\partial y_i^{(k)}} \frac{\partial y_i^{(k)}}{\partial z_i^{(k)}}$$

$$\frac{\partial \text{Div}}{\partial w_{ij}^{k+1}} = y_j^{(k)} \frac{\partial \text{Div}}{\partial z_i^{(k+1)}}, \text{ for } j = 1 \dots D_{k+1}$$

1 Introduction

- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构

2 前馈神经网络与 BP 算法

- 前馈神经网络
- 链式规则
- Forward computation
- Backward Computation
- **BP 算法的矩阵与向量表示**
- 交叉熵损失函数

- n^k 表示第 k 层神经元的个数
- $f(\cdot)$ 表示神经元的激活函数，一般假设每个神经元的激活函数是一样的。这里我们为了更简洁的表示，将激活函数扩展为用向量（分量的形式）来表示，即 $f[z_1, z_2, z_3] = [f(z_1), f(z_2), f(z_3)]$,
- $\mathbf{W}^{(k)}$ 表示第 $k-1$ 层到第 k 层神经元的权重矩阵；
- $\mathbf{z}^{(k)}$ 表示第 k 层神经元的局部输出；
- $\mathbf{y}^{(k)}$ 表示第 k 层神经元的激活函数输出值。

前馈神经网络通过下面公式进行信息传播。

$$\begin{aligned}\mathbf{z}^{(k)} &= \mathbf{W}^{(k)} \cdot \mathbf{y}^{(k-1)} \\ \mathbf{y}^{(k-1)} &= f(\mathbf{z}^{(k-1)}), & \mathbf{y}^{(k)} &= f(\mathbf{z}^{(k)}) \\ \implies \mathbf{z}^{(k)} &= \mathbf{W}^{(k)} \cdot f(\mathbf{z}^{(k-1)})\end{aligned}$$

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。

$$\mathbf{x} = \mathbf{y}^{(0)} \rightarrow \mathbf{z}^{(1)} \rightarrow \mathbf{y}^{(1)} \rightarrow \mathbf{z}^{(2)} \rightarrow \dots \rightarrow \mathbf{y}^{(k)} \rightarrow \text{Div} \rightarrow \text{Err}$$

给定一组样本 $\{\mathbf{x}_i, y_i\}_{i=1}^n$ ，我们的目标是最小化前馈网络的损失函数 Err。如果采用梯度下降方法，我们可以用如下方法更新参数：

$$\mathbf{W}^{(k)} = \mathbf{W}^{(k)} - \alpha \frac{\partial \text{Err}}{\partial \mathbf{W}^{(k)}}$$

我们首先来看下 $\frac{\partial \text{Err}}{\partial \mathbf{W}^{(k)}}$ 怎么计算。根据链式法则：

$$\frac{\partial \text{Err}}{\partial \mathbf{W}_{ij}^{(k)}} = \left(\frac{\partial \text{Err}}{\partial \mathbf{z}^{(k)}} \right)^T \frac{\partial \mathbf{z}^{(k)}}{\partial \mathbf{W}_{ij}^{(k)}}$$

定义误差项 $\delta^k = \frac{\partial \text{Err}}{\partial \mathbf{z}^{(k)}}$ （局域梯度），表示第 k 层的神经元对最终误差的影响。也反映了最终的输出对第 k 层的神经元对最终误差的敏感程度。

$$\begin{aligned} \delta^k &= \frac{\partial \text{Err}}{\partial \mathbf{z}^{(k)}} = \frac{\partial \mathbf{y}^{(k)}}{\partial \mathbf{z}^{(k)}} \frac{\partial \mathbf{z}^{(k+1)}}{\partial \mathbf{y}^{(k)}} \frac{\partial \text{Err}}{\partial \mathbf{z}^{(k+1)}} \\ &= \text{diag}(f'(\mathbf{z}^k)) \cdot (\mathbf{W}^{k+1})^T \cdot \delta^{k+1} \\ &= f'(\mathbf{z}^k) \odot \left((\mathbf{W}^{k+1})^T \cdot \delta^{k+1} \right) \end{aligned} \quad (1)$$

其中 \odot 为 Hadamard 乘积。

$$\frac{\partial \mathbf{z}^{(k)}}{\partial \mathbf{W}_{ij}^{(k)}} = \frac{\partial (\mathbf{W}^{(k)} \cdot \mathbf{y}^{(k-1)})}{\partial \mathbf{W}_{ij}^{(k)}} = \begin{pmatrix} 0 \\ \vdots \\ y_j^{(k-1)} \\ \vdots \\ 0 \end{pmatrix} \cdot \leftarrow i\text{-th row}$$

因此,

$$\frac{\partial \text{Err}}{\partial \mathbf{W}_{ij}^{(k)}} = \delta_i^{(k)} y_j^{(k-1)}$$

最后得到:

$$\frac{\partial \text{Err}}{\partial \mathbf{W}^{(k)}} = \boldsymbol{\delta}^{(k)} (\mathbf{y}^{(k-1)})^T \quad (2)$$

第 k 层的误差项 δ^k 可以通过第 $k+1$ 层的误差项 δ^{k+1} 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 k 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 $k+1$ 层的神经元的误差项的权重和。然后，再乘上该神经元激活函数的梯度。

梯度弥散

在神经网络中误差反向传播的迭代公式为：

$$\delta^k = f'(z^k) \odot \left((\mathbf{W}^{k+1})^T \cdot \delta^{k+1} \right)$$

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，其导数为

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

我们可以看到，sigmoid 型函数导数的值域小于 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，误差可以很好的传播，训练速度得到了很大的提高。


1 Introduction

- 基本概念
- 激活函数
- 网络结构
- 神经网络的发展
- 其他网络结构


2 前馈神经网络与 BP 算法

- 前馈神经网络
- 链式规则
- Forward computation
- Backward Computation
- BP 算法的矩阵与向量表示
- 交叉熵损失函数

交叉熵损失函数

 相对熵。信息论中一个重要的概念是“相对熵” Kullback-Leibler Divergence，是以它的两个提出者库尔贝克和莱伯勒的名字命名的。相对熵用来衡量两个正函数是否相似，对于两个完全相同的函数，它们的相对熵等于零。在自然语言处理中可以用相对熵来衡量两个常用词（在语法上和语义上）是否同义，或者两篇文章的内容是否相近等等。

$$D_{\text{KL}}(p\|q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

 交叉熵

交叉熵（Cross Entropy）是 Shannon 信息论中一个重要概念，主要用于度量两个概率分布间的差异性信息。

假设现在有一个样本集中两个概率分布 p, q ，其中 p 为真实分布， q 为非真实（估计）分布。

$$H(p, q) = - \sum_x p(x) \log q(x)$$

```
import torch.nn as nn
```

```
# Example of target with class indices
```

```
loss = nn.CrossEntropyLoss()
```

```
input = torch.randn(3, 5, requires_grad=True)
```

```
target = torch.empty(3, dtype=torch.long).random_(5)
```

```
output = loss(input, target)
```

```
# Example of target with class probabilities
```

```
input = torch.randn(3, 5, requires_grad=True)
```

```
target = torch.randn(3, 5).softmax(dim=1)
```

```
output = loss(input, target)
```

```
~~~~~
```