机器学习与深度学习
——神经元模型与激活函数

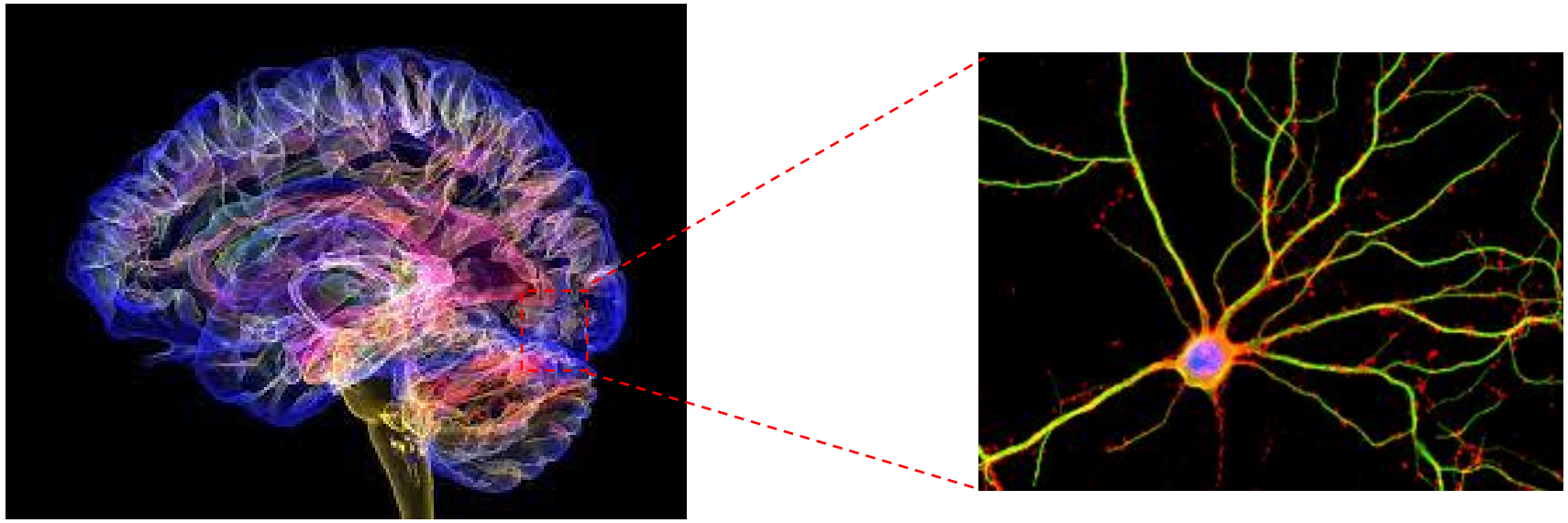
Personal Website: https://www.miaopeng.info/
Email: miaopeng@stu.scu.edu.cn
Github: https://github.com/MMeowhite
Youtube: https://www.youtube.com/@pengmiao-bmm

目录章节

CONTENTS

▶ 什么是神经网络（Neural Network）？

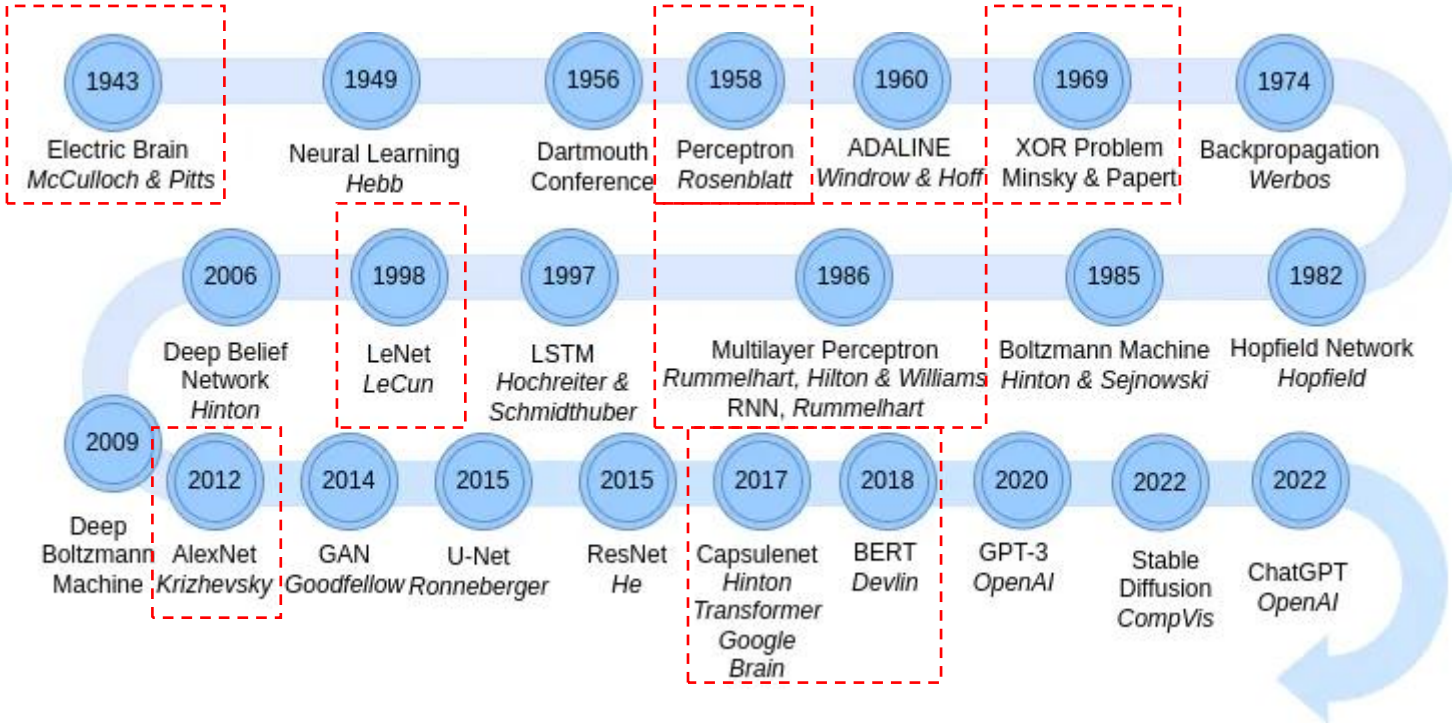
- 神经网络的起源可以追溯到对**人脑思维机制**的模拟，它最初的提出，源于一个核心思想：
- **“如果人类的大脑能通过神经元互相连接处理信息，那么我们是否也能构建一个类似的数学模型，让机器学习和思考？”**
- 神经网络是**模仿人脑神经元结构和学习机制**而设计的一类机器学习模型，它能通过“训练”从数据中**自动学习**模式，并用于预测、分类等任务。



核心：利用数学模型来模拟生物神经元的工作方式，让机器也能通过连接、学习和适应数据。

神经网络： 发展历史

- 神经网络起初源于对人脑神经元的模拟，历经感知机的提出、寒冬的质疑、反向传播的突破，到深度学习与Transformer的崛起，逐步发展为现代人工智能的核心驱动技术。

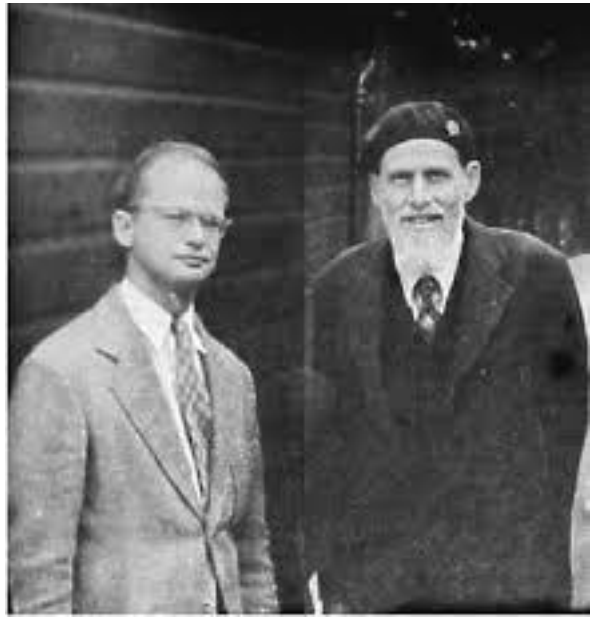


神经网络的发展几经沉浮，凭借算法和算力提升，终于成为当今人工智能的核心技术之一。

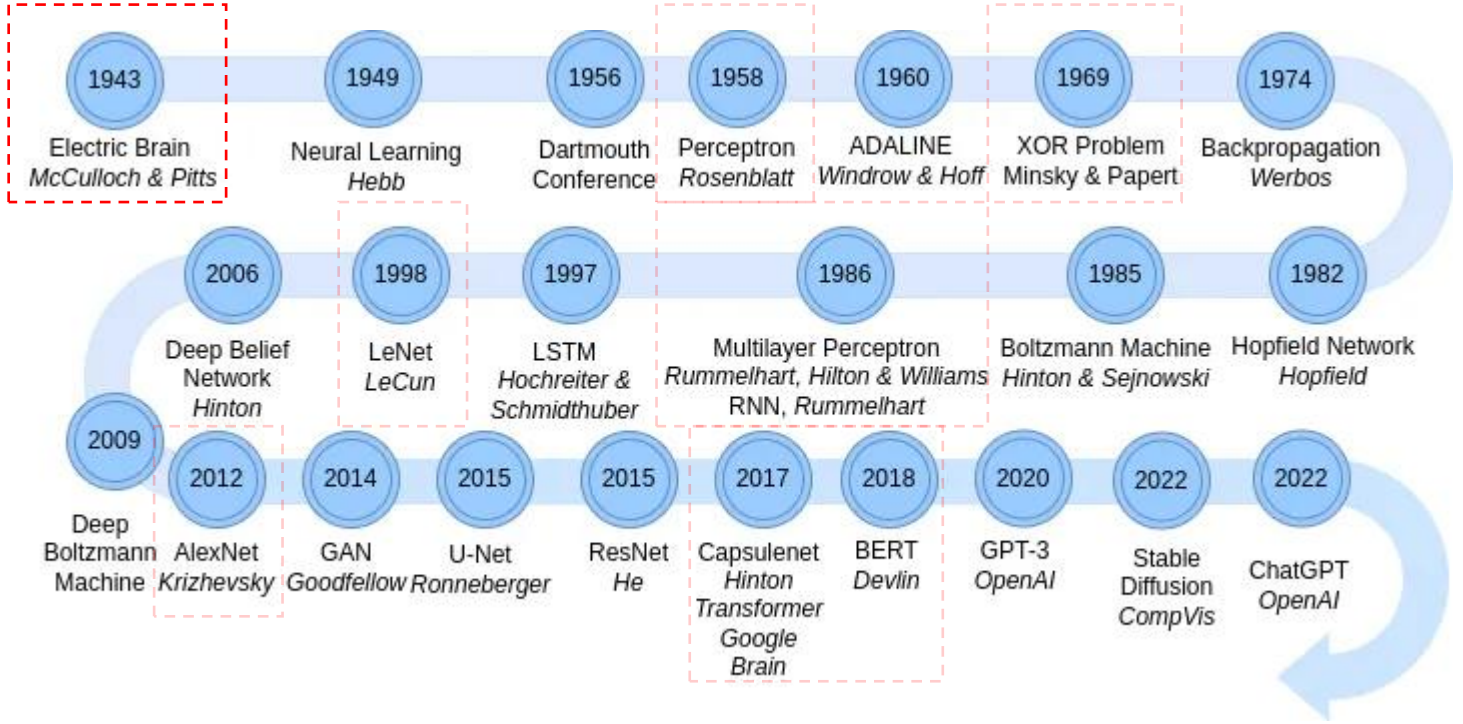
扩展阅读资料：https://pub.towardsai.net/a-brief-history-of-neural-nets-472107bc2c9c

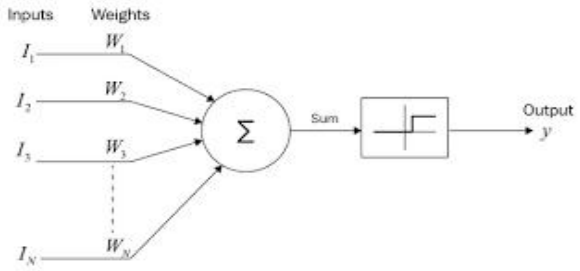
神经网络： 发展历史

- McCulloch & Pitts：提出第一个人工神经元模型，奠定神经网络理论基础。



McCulloch & Pitts, 1943

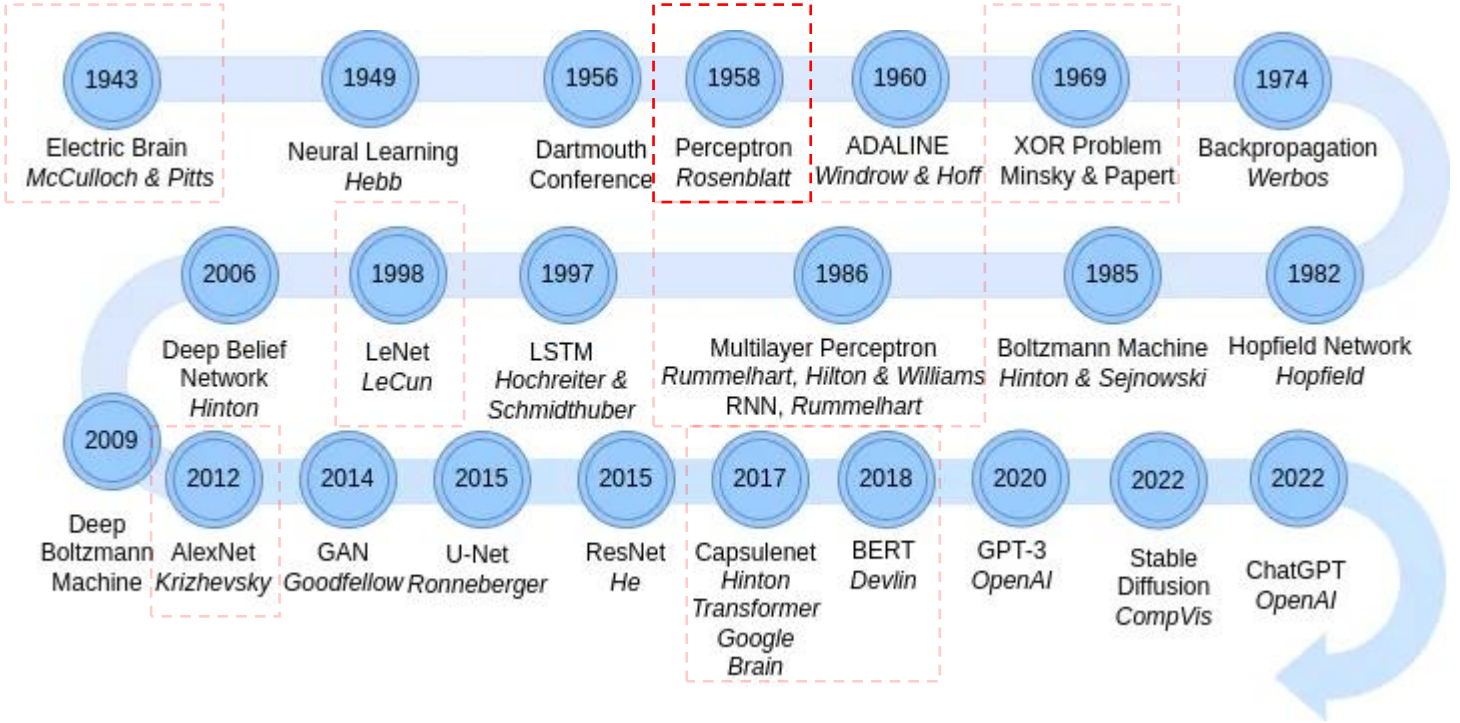


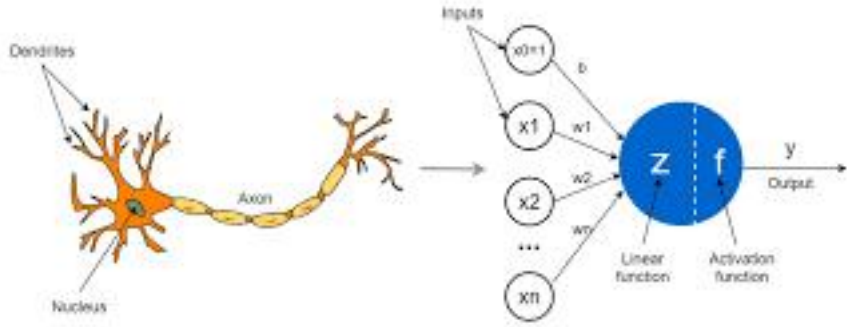


$$y = f\left(\sum_{i=1}^{n} x_i w_i - \theta\right) \qquad f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

神经网络： 发展历史

- Rosenblatt：提出感知机模型（Perceptron），能进行简单分类，是第一个“可学习”的神经网络。





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

▶ 扩展：两种模型的区别

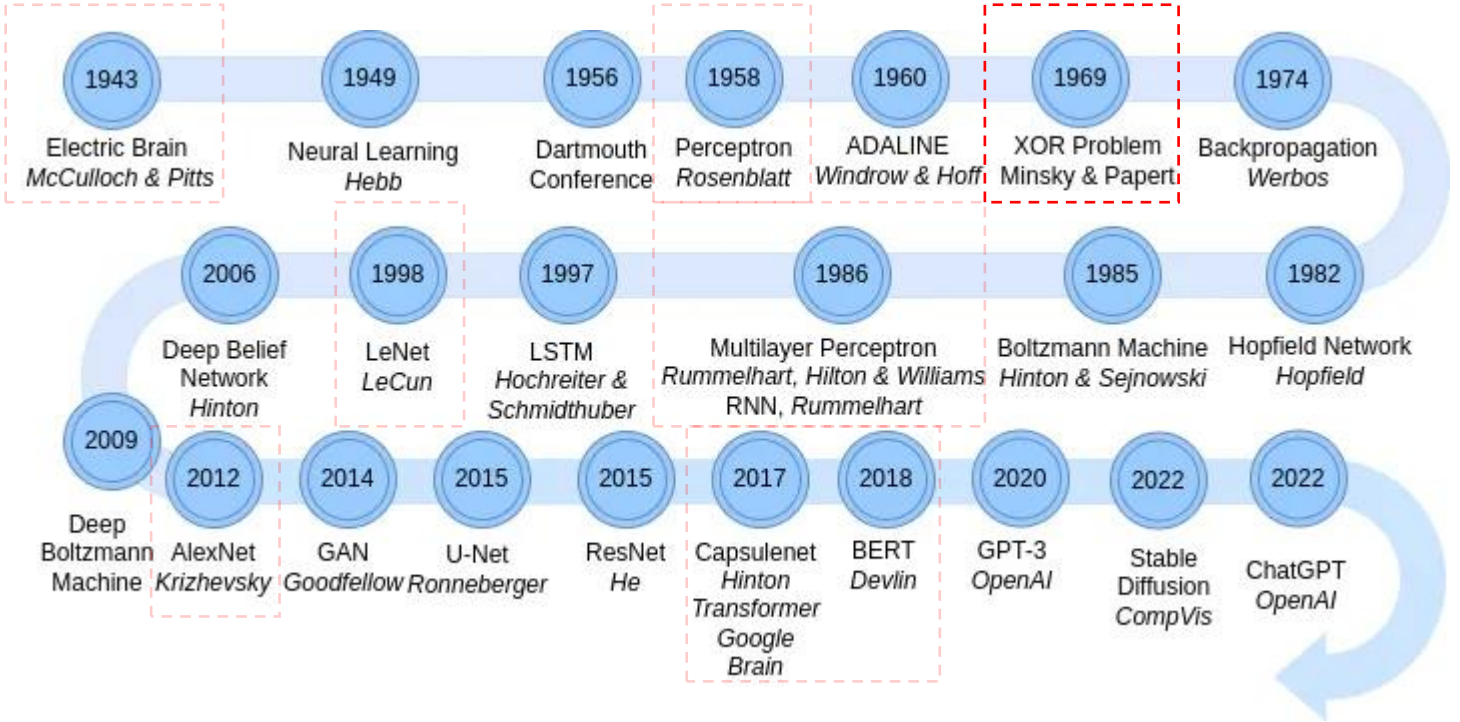
➢ Rosenblatt 感知机模型（1958）与McCulloch & Pitts 神经元模型（1943）具有类似的算法结构，但是仍有一定的区别

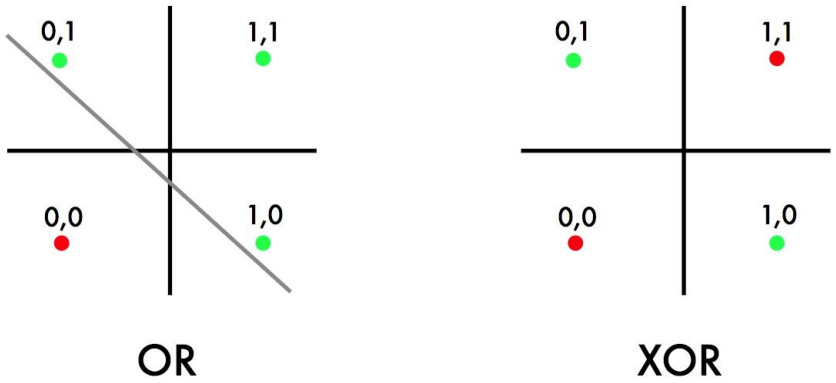
项目	McCulloch & Pitts 神经元模型（1943）	Rosenblatt 感知机模型（1958）
是否可学习	不可学习	可学习
输入类型	只能是0或1	实数输入、支持多个特征
激活函数	阈值函数（step function）	阶跃函数（step function）
功能	只能实现固定逻辑门（如AND，OR）	能自动调整权重，进行线性可分分类任务（如正负样本）
生物学动机	模拟神经元“是否激活”的逻辑机制	类似
数学公式	$y = f(\sum x_i w_i - \theta)$	$y = f(\mathbf{w}^T \mathbf{x} + b)$
学术意义	**人工神经元的原型**，提出神经计算思想的雏型	第一个**可训练**的神经网络模型，开创了机器学习的早期研究路线

神经网络： 发展历史

- Minsky：批评感知机，**指出感知机无法解决非线性问题**，研究陷入低谷（第一次寒冬）。

Marvin Lee Minsky, 1969



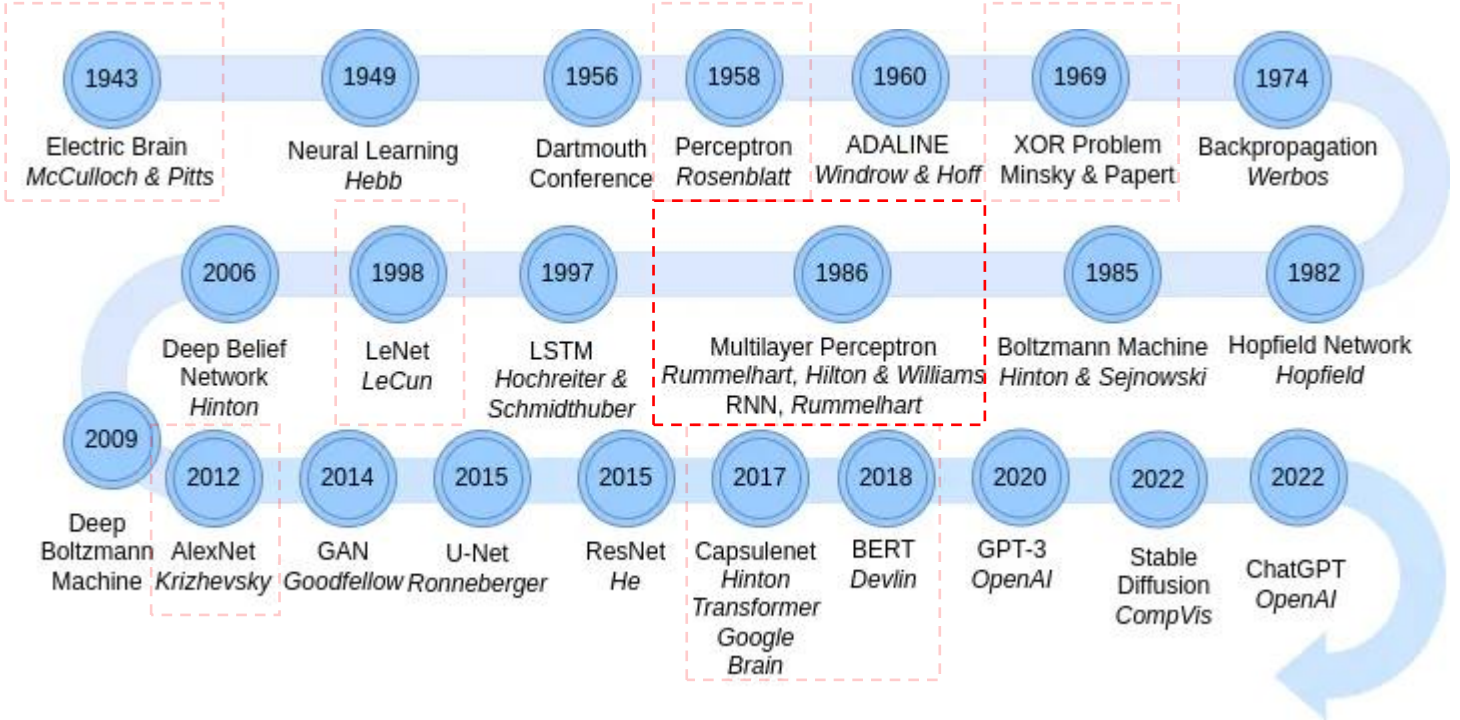


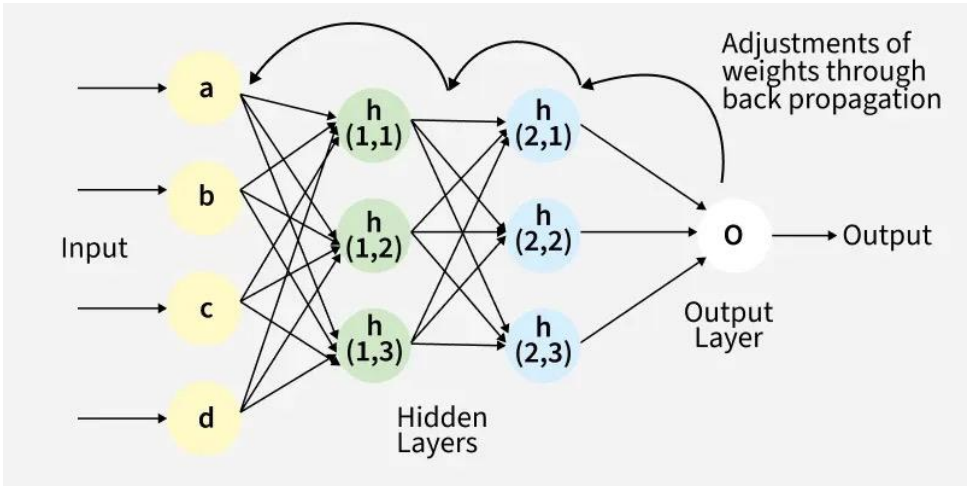
- XOR 问题：指的是 Marvin Minsky 在其 1969 年的经典著作《Perceptrons》中提出的一个挑战，揭示了 单层感知机 无法解决 XOR 问题（非线性），从而暴露了早期人工神经网络的局限性。

▶ 神经网络： 发展历史

➢ Hilton：**反向传播算法**兴起，多层神经网络可训练，掀起神经网络复兴浪潮。

Hinton, 1958



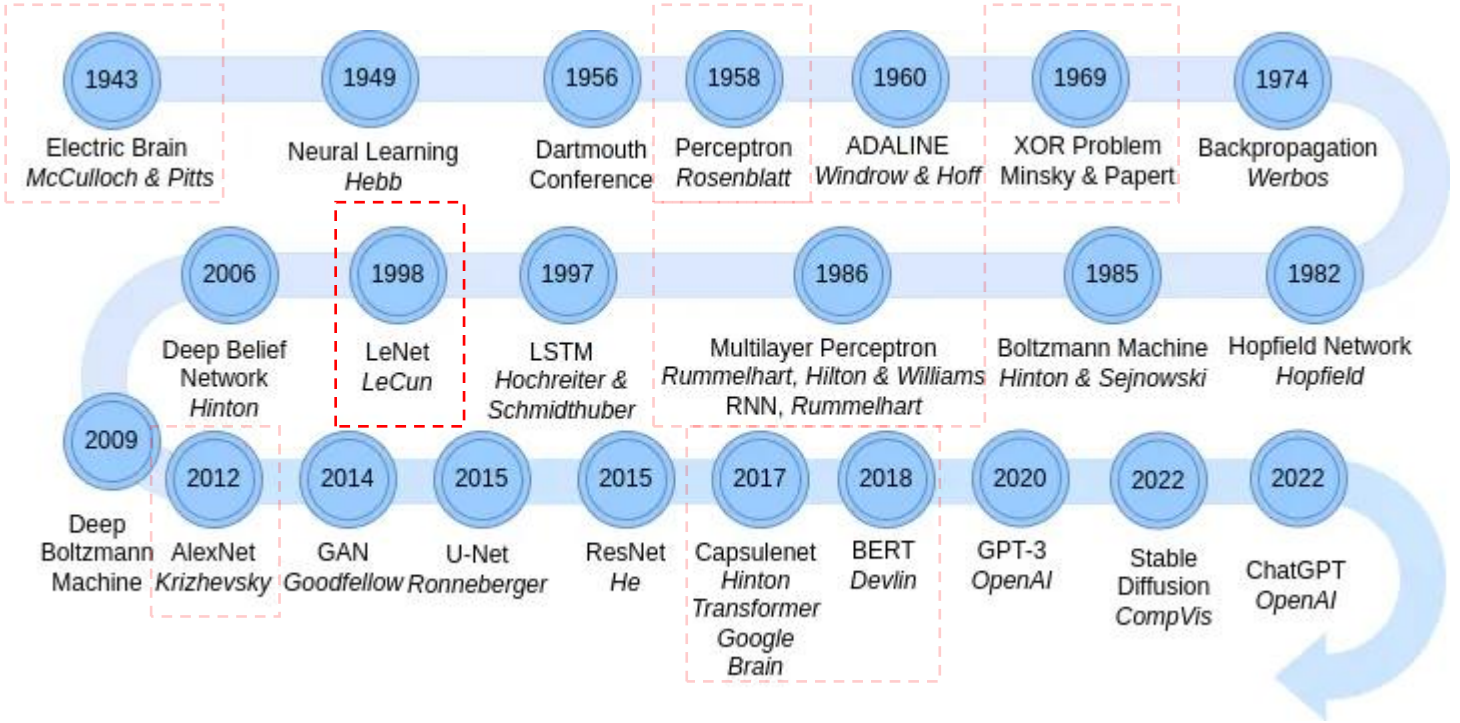


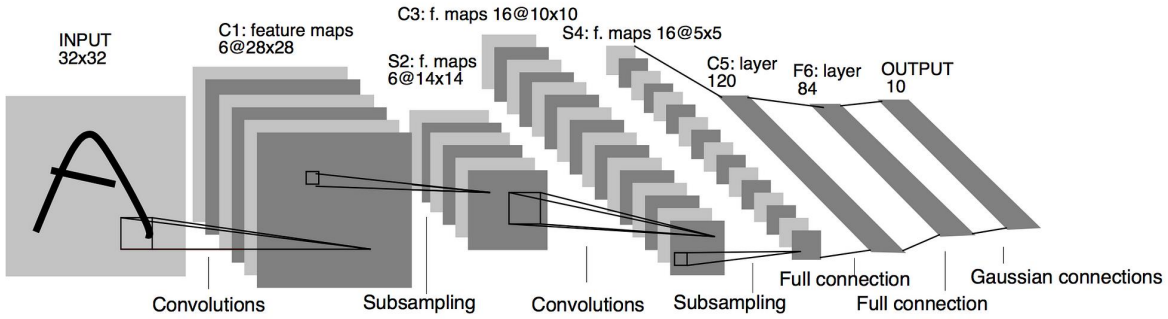
- 反向传播算法通过链式法则将误差从输出层向输入层逐层传播，以更新神经网络中的权重。

神经网络： 发展历史

- Yann LeCun：提出**LeNet-5**，用CNN成功识别手写数字，神经网络走入实用。

Yann LeCun, 1998



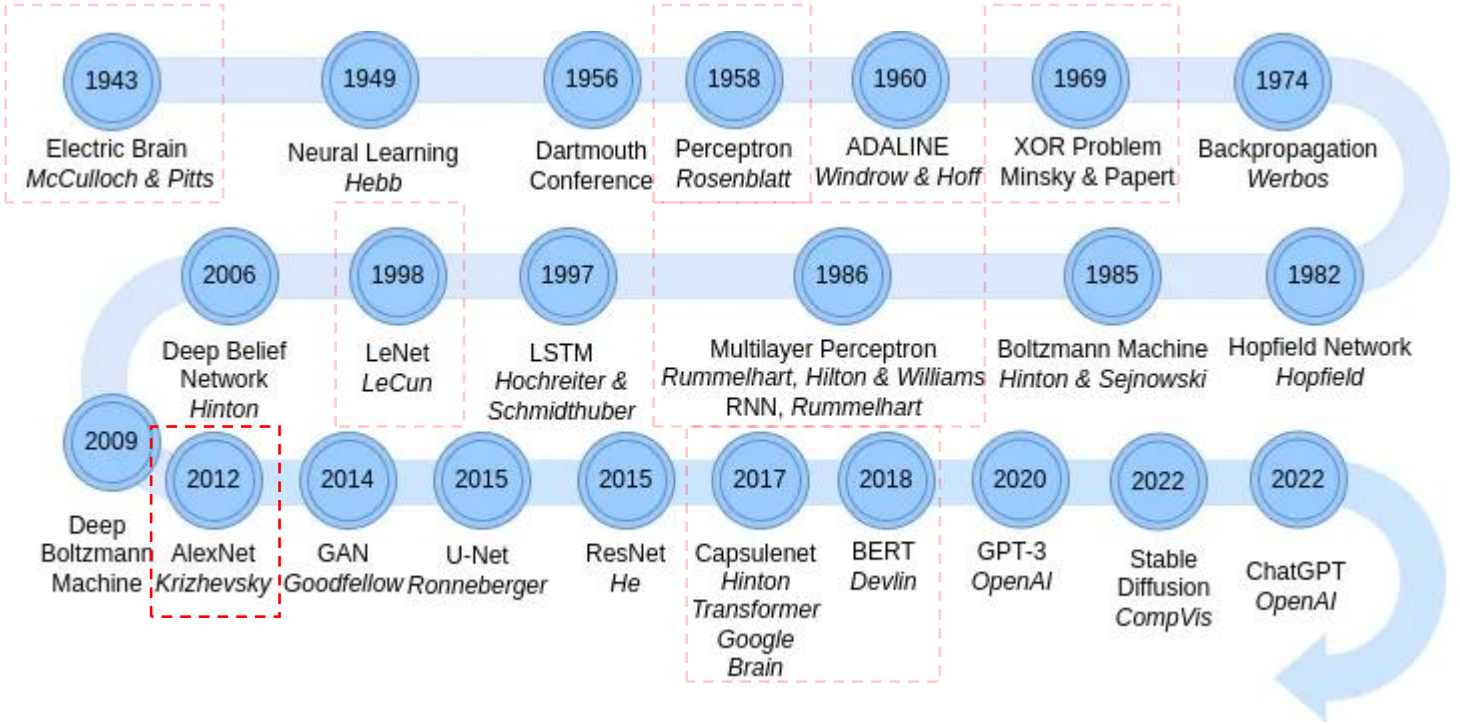


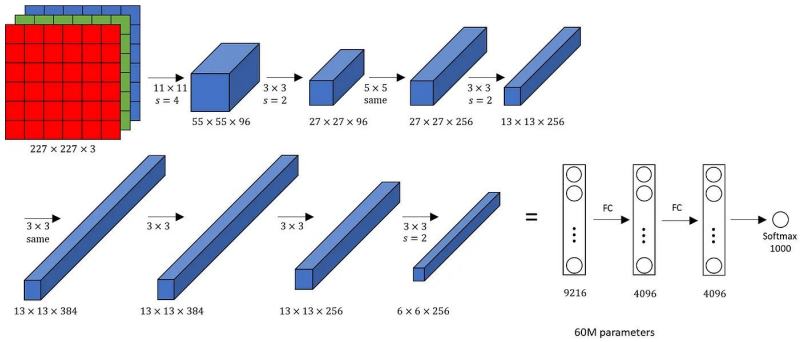
- LeNet-5是一种经典卷积神经网络，通过卷积层和池化层提取图像特征，最终通过全连接层进行分类，主要用于手写数字识别。

▶ 神经网络： 发展历史

➢ Krizhevsky：AlexNet 在 ImageNet 获胜，引爆深度学习热潮，**GPU 加速 + 大数据 = 视觉革命**。

Alex Krizhevsky, 2012



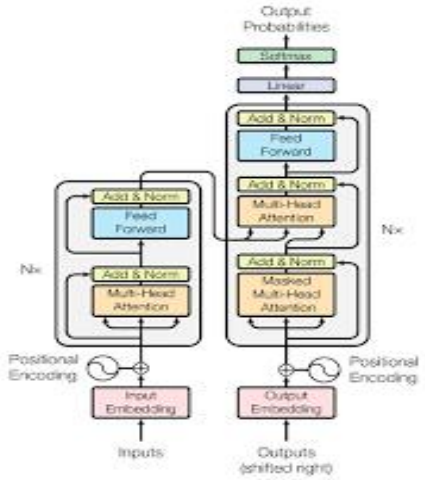


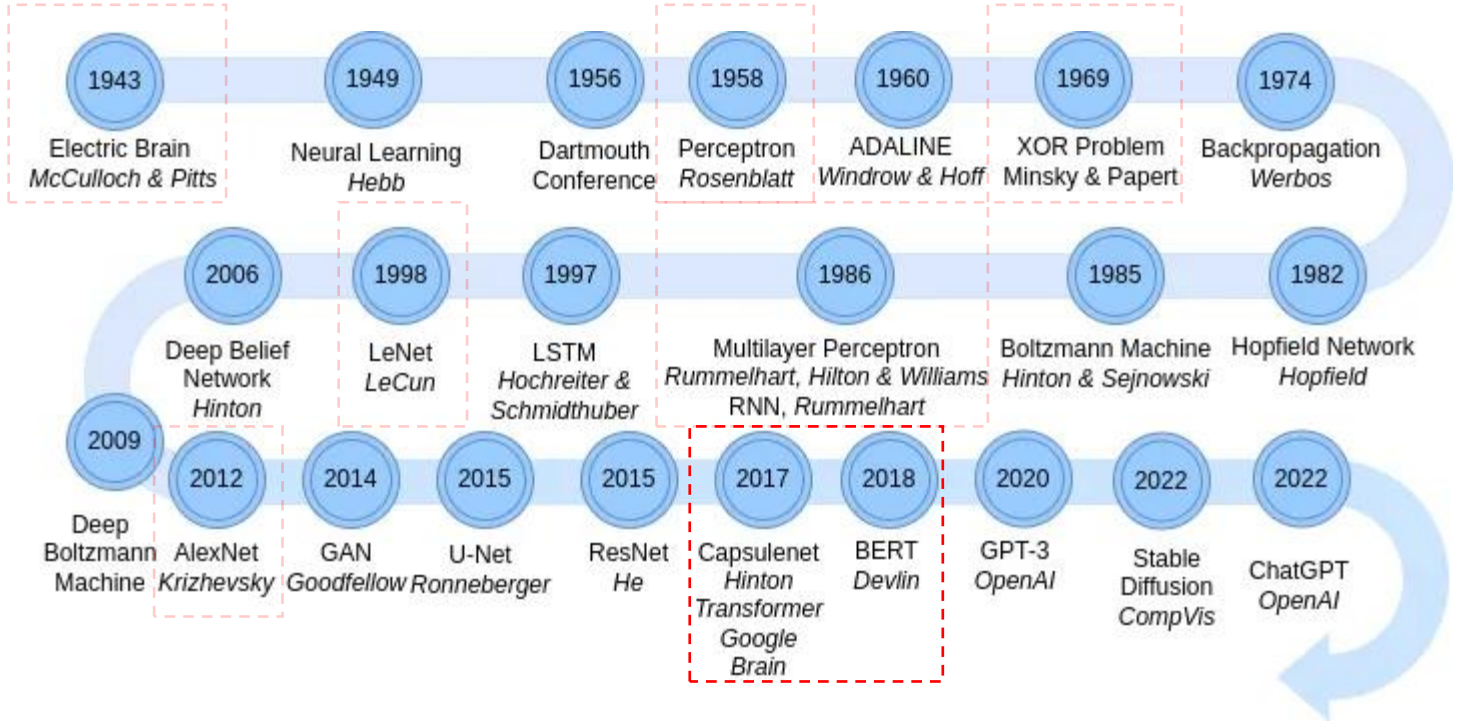
- AlexNet 是第一个在**大规模图像**识别中使用深度卷积神经网络并取得突破性成果的模型（5 个卷积层 + 3 个全连接层），标志着深度学习时代的开启。

神经网络： 发展历史

- Google Brain：提出**Transformer架构**，推动自然语言处理（如 ChatGPT）进入深度神经网络时代。

Google Brain, 2017





- Transformer 是一种**完全基于注意力机制**的深度学习模型，擅长处理序列数据，彻底改变了自然语言处理和人工智能的发展方向。

▶ 神经网络： 应用场景

➢ 神经网络广泛应用于图像处理、语音识别、自然语言处理、推荐系统等多个领域，助力解决复杂的模式识别和决策问题。

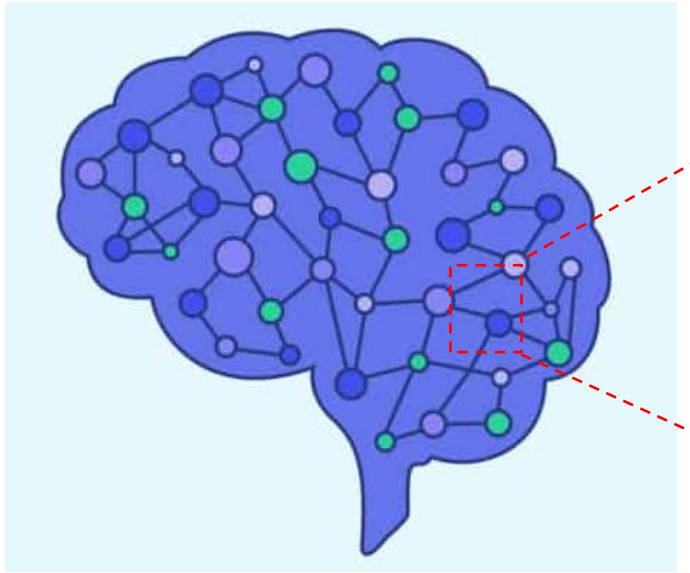
应用领域	具体应用场景	说明
图像处理	图像分类、目标检测、图像分割	使用卷积神经网络（CNN）进行图像的分类、识别、定位等任务。
语音处理	语音识别、语音合成、语音增强	基于循环神经网络（RNN）、长短期记忆（LSTM）进行语音的转换与处理。
自然语言处理	机器翻译、文本生成、情感分析	基于循环神经网络（RNN）、长短期记忆（LSTM）进行语音的转换与处理。
推荐系统	个性化推荐（商品推荐、影片推荐等）	利用神经网络建模用户与物品的关系，提供精准的推荐内容。
医疗诊断	图像诊断（如医学影像）、基因数据分析	使用深度学习分析医学图像（如 CT、MRI）和基因数据，辅助诊断。
自动驾驶	车辆识别、路径规划、环境感知	结合 CNN 和强化学习处理视觉信息，实现自动驾驶中的环境感知与决策。
金融领域	信用评估、算法交易、风险预测	神经网络用于金融预测、股市分析、信贷评分等复杂非线性问题。
游戏与娱乐	游戏AI、虚拟角色对话	使用深度强化学习和神经网络训练游戏中的AI玩家或虚拟角色。
艺术创作	图像生成、风格迁移、音乐创作	使用生成对抗网络（GAN）和神经网络进行艺术创作，如图像生成、风格迁移。

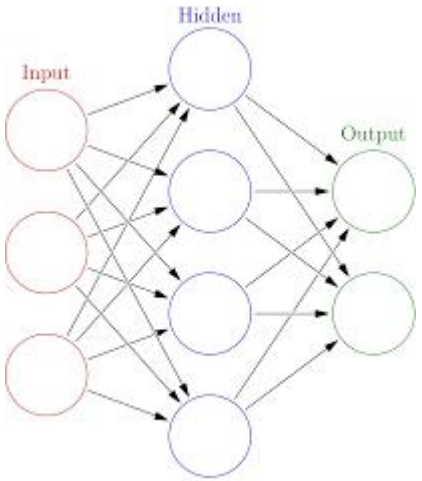
目录章节

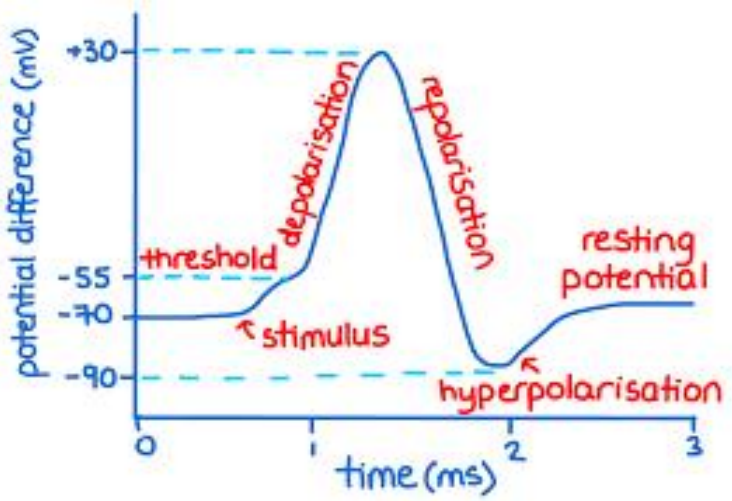
CONTENTS

内部结构：神经元模型概述

- 定义：神经网络是一种由**多个人工神经元（节点）连接**组成的计算模型，包含**输入层**、**隐藏层**和**输出层**。它通过对训练数据的前向传播与误差反向传播，不断优化权重参数，实现智能识别与预测。
- 人脑由数以亿计的神经元组成，彼此通过突触传递信息。神经网络用**权重模拟突触**，用**激活函数模拟神经元“是否被激活”（阈值）**。训练过程就像人在学习，通过不断试错来提升表现。

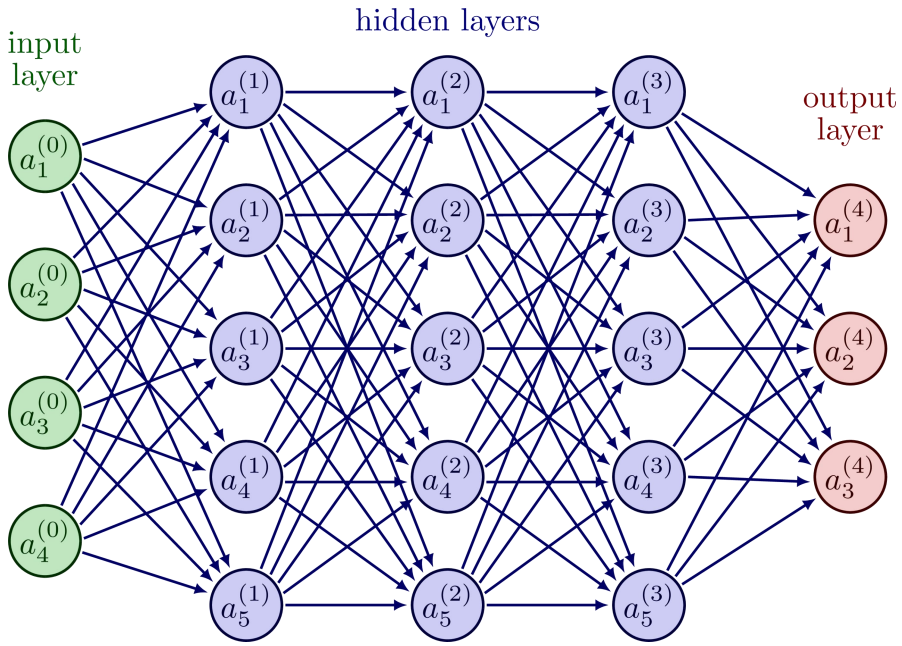






内部结构：神经元模型层次

- 神经网络的层次结构指的是神经网络由**若干“层（Layer）”组成**的有序结构，数据从输入开始，经过一层一层的处理，最终得到输出。



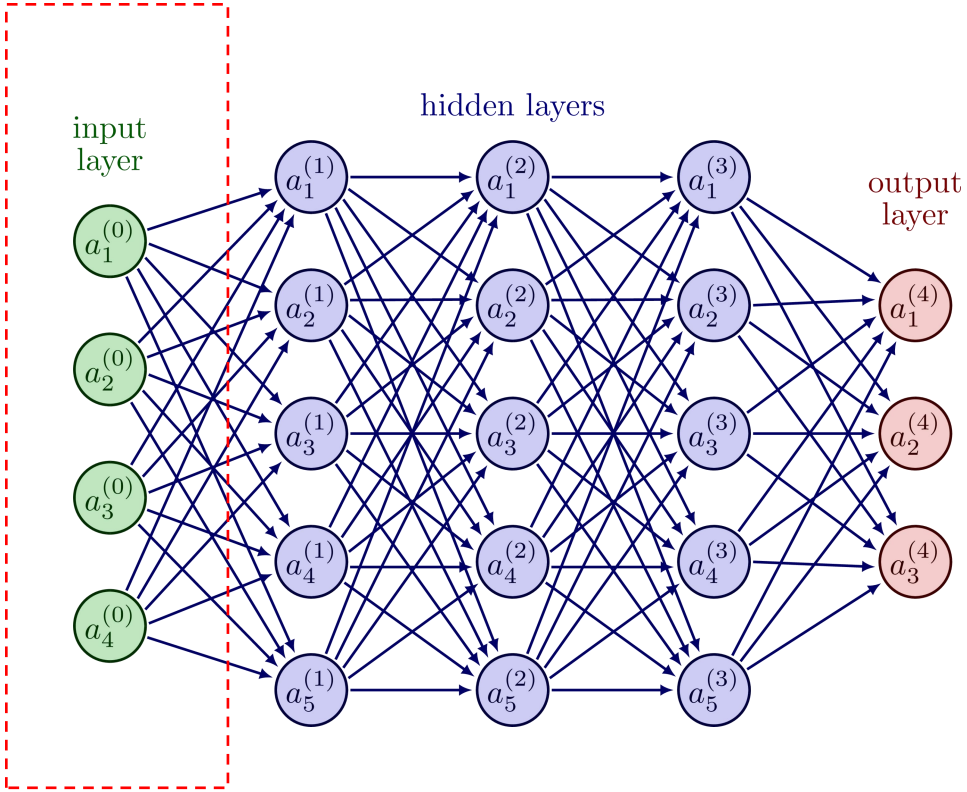
层名称	功能	常见示意
输入层 Input layer	接收原始数据，如图像、文本或数值	x1, x2, ... , xn
隐藏层 Hidden layer	执行特征提取与非线性映射，是神经网络的核心	z=f(Wx + b)
输出层 output layer	输出最终预测结果	分类或回归结果

神经网络：智能的“大脑”，层层神经元紧密相连，数据在其中流转，越深越能捕捉复杂模式。

扩展阅读资料：https://tikz.net/neural_networks/

▶ 内部结构：输入层

- 作用：接收外部输入数据，不进行任何计算。
- 特征：每个神经元代表一个输入变量（如像素值、数值特征等）。



28×28×3 的图像（宽 28 像素，高 28 像素，3 个颜色通道 RGB）

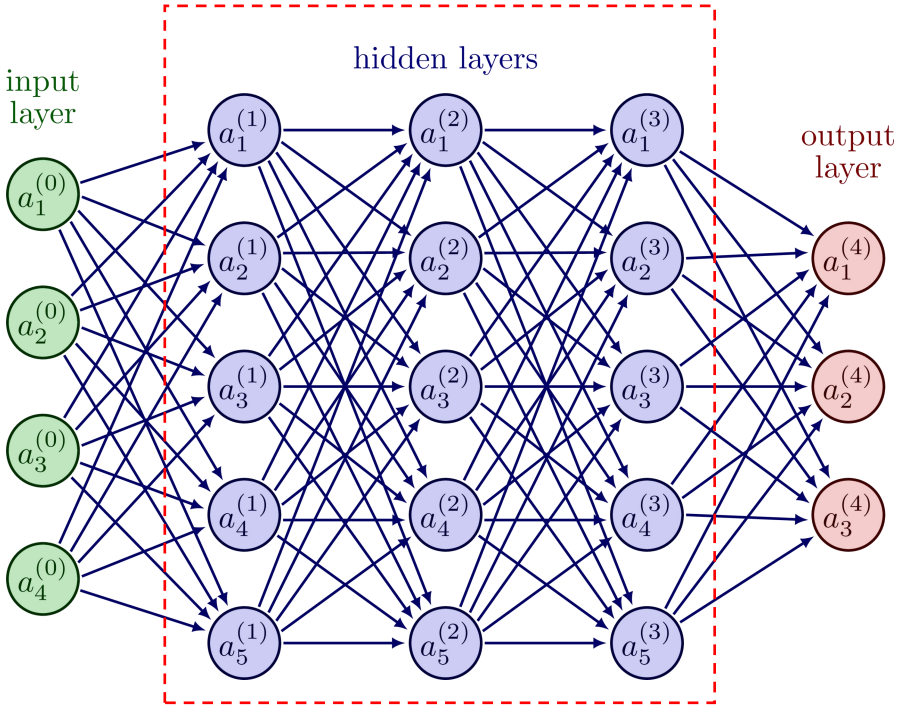
28×28×3 = 2352

输入层负责接收原始数据，每个神经元代表一个输入特征，是神经网络信息流动的起点。

扩展阅读资料：https://tikz.net/neural_networks/

内部结构：隐藏层

- 作用：负责数据的特征提取与映射。
- 特征：每个神经元代表一个输入变量（如像素值、数值特征等）。



- 内部结构：每个神经元执行以下的操作：

$$h = f(Wx + b)$$

其中：W -> 权重矩阵，x -> 上一层输出, b-> 偏置, f -> 激活函数（如ReLU、Sigmoid）

层数：一个隐藏层为浅层神经网络（Shallow NN），多个隐藏层为深度神经网络（Deep NN）

隐藏层越多，网络越深，模型表达能力越强，但也更容易过拟合。

扩展阅读资料：https://tikz.net/neural_networks/

▶ 内部结构：隐藏层

W -> 权重矩阵（Weights）

作用	表示当前层每个神经元与上一层所有神经元之间的连接强度。
形状	若上一层有n个神经元，当前层有m个神经元，则$W \in R^{m \times n}$
本质	决定了“输入特征”的线性组合方式。
特点	训练中会被不断更新，以最小化损失函数

x -> 上一层输出（input）

作用	表示传入当前层的输入数据，可能是原始输入，也可能是前一层的输出结果。
形状	一般为列向量 $x \in R^{n \times 1}$
注意	需要与权重矩阵的维度兼容，即W · x 可行。

$$h = f(Wx + b)$$

f -> 激活函数（Activation Function）

作用	引入非线性变换，使神经网络能表示复杂函数。
常见选择	ReLU、Sigmoid、Tanh、Leaky ReLU
应用位置	通常在每一层的线性变换后应用

b — 偏置向量（Bias）

作用	为每个神经元引入一个可调节的“基准”值，提升网络表达能力。
形状	若当前层有m个神经元，则$b \in R^{m \times 1}$
本质	允许神经元在没有输入信号时也能激活。

▶ 扩展：为什么选取一次函数形式？而不是其他任意函数？

- 涉及神经网络结构中**“线性变换 + 非线性激活”**的本质设计哲学。
- 原因如下：1）线性结构便于训练和优化

◆ 一次函数：

$$z_{i,j} = \sum_k w_k x_k + b$$

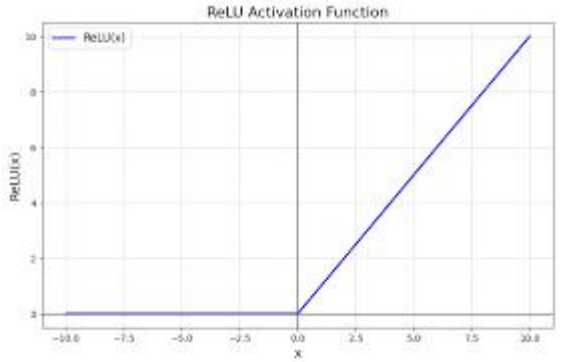
◆ 是一个线性变换（**本质上是矩阵乘法 + 偏置**），可以高效使用**梯度下降法**等方法优化。

◆ 复杂函数（如二次、三角、多项式）会导致：1）梯度爆炸或消失更严重；2）参数过多，训练不稳定；3）难以向深层扩展

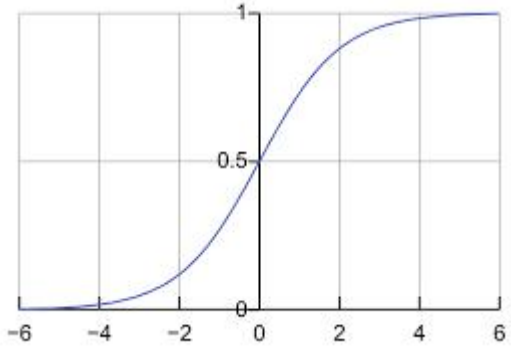
神经网络选用一次函数作为每层的核心计算，是为了结构简洁、可学习性强；而通过叠加层数和激活函数的非线性能力，实现对复杂关系的逼近。

▶ 扩展：为什么选取一次函数形式？而不是其他任意函数？

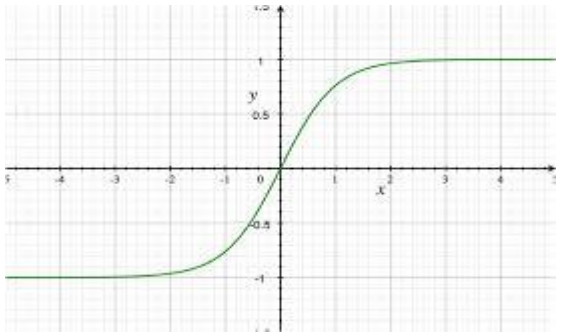
- 涉及神经网络结构中**“线性变换 + 非线性激活”**的本质设计哲学。
- 原因如下：2）线性变换+非线性激活 = 万能函数逼近器
- 单独的线性变换是不能逼近复杂函数的（多个线性变换合成还是线性）。



ReLU



Sigmoid



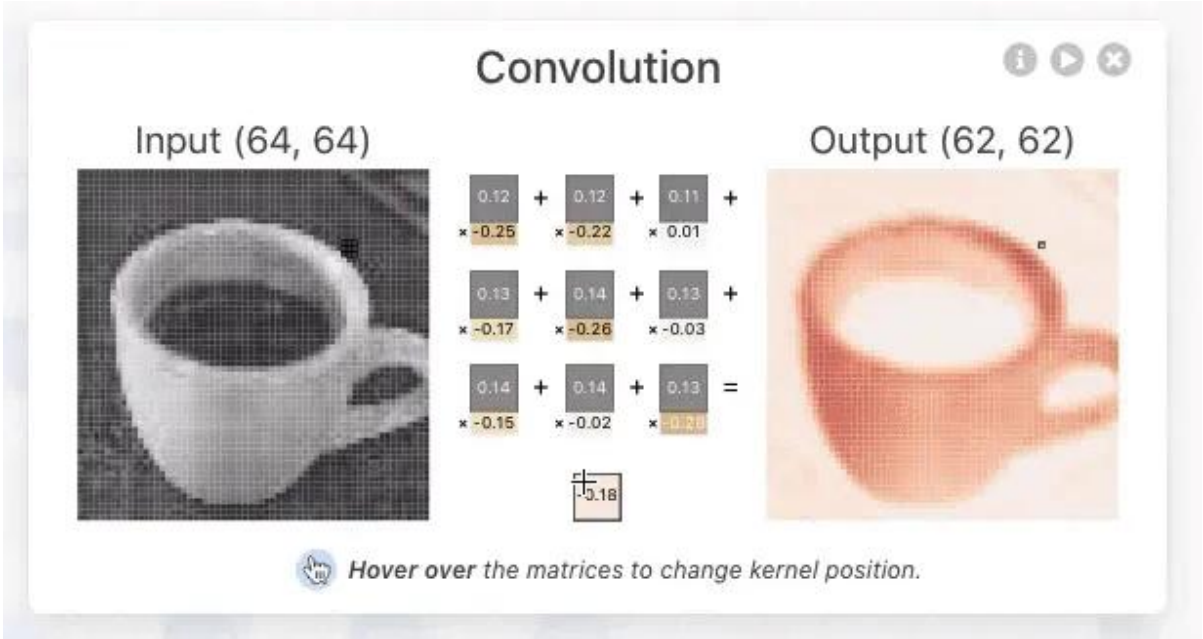
Tanh

- **加入非线性激活函数（如ReLU, Sigmoid, Tanh），只要网络层数足够，就可以逼近任何连续函数（通用逼近定理）。**

神经网络选用一次函数作为每层的核心计算，是为了结构简洁、可学习性强；而通过叠加层数和激活函数的非线性能力，实现对复杂关系的逼近。

▶ 扩展：为什么选取一次函数形式？而不是其他任意函数？

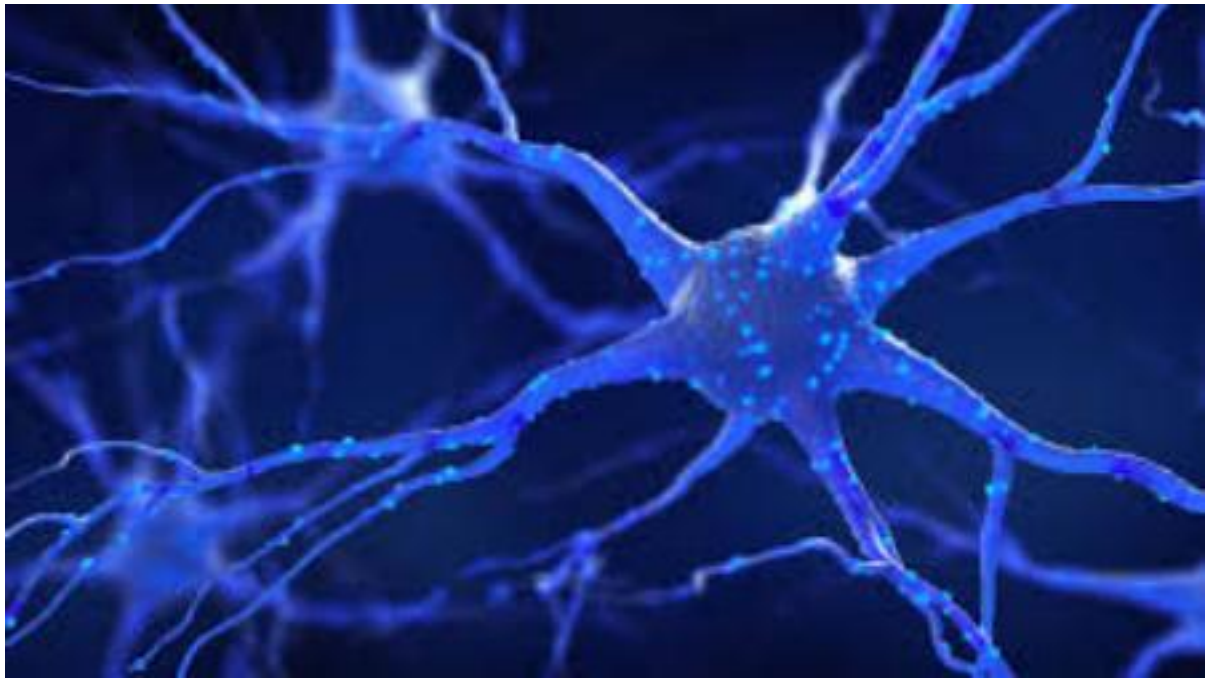
- 涉及神经网络结构中**“线性变换 + 非线性激活”**的本质设计哲学。
- 原因如下：3）易于解释、调试和实现



- ◆ 权重w_k的物理含义：每个输入对输出的“影响程度”。
- ◆ 偏置b：调整激活阈值，类似于“基线”。
- ◆ 形式简单清晰，便于硬件加速（如 GPU、TPU）。

▶ 扩展：为什么选取一次函数形式？而不是其他任意函数？

- 涉及神经网络结构中**“线性变换 + 非线性激活”**的本质设计哲学。
- 原因如下：4）从生物神经元的灵感而来



- ◆ 接收多个突触输入并进行叠加（EPSP/IPSP，兴奋性/抑制性突触后电位）。
- ◆ 若超过阈值则“激活”发射信号（形成动作电位）。

▶ 扩展：怎么进行权重初始化？

➢ **Xavier（Glorot）初始化** 和 **He 初始化**——是神经网络训练中的权重初始化策略，目的是避免梯度消失或爆炸，让网络更容易收敛。

初始化方法	推荐激活函数	保持什么不变	正态分布方差
Xaiver (Glorot)	Sigmoid / Tanh	输入输出的方差一致	$\frac{2}{n_{in}+n_{out}}$
He (Kaiming)	ReLU / Leaky ReLU	ReLU 激活后输出的方差保持稳定	$\frac{2}{n_{in}}$

总结：Xavier 初始化适用于 Sigmoid/Tanh，通过均衡输入输出方差防止梯度消失；He 初始化专为 ReLU 设计，放大初始权重以保持激活后方差稳定。

扩展：怎么进行权重初始化？

- **Xavier（Glorot）初始化**：适用激活函数：Sigmoid, Tanh（这些是对称、非零中心的函数）
- 核心思想：保持每层的输入和输出的方差一致，避免信号逐层放大或缩小。

Understanding the difficulty of training deep feedforward neural networks

Xavier Glorot **Yoshua Bengio**
DIRO, Université de Montréal, Montréal, Québec, Canada

Abstract

Whereas before 2006 it appears that deep multi-layer neural networks were not successfully trained, since then several algorithms have been shown to successfully train them, with experimental results showing the superiority of deeper vs less deep architectures. All these experimental results were obtained with new initialization or training mechanisms. Our objective here is to understand better why standard gradient descent from random initialization is doing so poorly with deep neural networks, to better understand these recent relative successes and help design better algorithms in the future. We first observe the influence of the non-linear activations functions. We find that the logistic sigmoid activation is unsuited for deep networks with random initialization because of its mean value, which can drive especially the top hidden layer into saturation. Surprisingly, we find that saturated units can move out of saturation by themselves, albeit slowly, and explaining the plateaus sometimes seen when training neural networks. We find that a new non-linearity that saturates less can often be beneficial. Finally, we study how activations and gradients vary across layers and during training, with the idea that training may be more difficult when the singular values of the Jacobian associated with each layer are far from 1. Based on these considerations, we propose a new initialization scheme that brings substantially faster convergence.

1 Deep Neural Networks

Deep learning methods aim at learning feature hierarchies with features from higher levels of the hierarchy formed by the composition of lower level features. They include learning methods for a wide array of *deep architectures*, including neural networks with many hidden layers (Vincent et al., 2008) and graphical models with many levels of hidden variables (Hinton et al., 2006), among others (Zhu et al., 2009; Weston et al., 2008). Much attention has recently been devoted to them (see (Bengio, 2009) for a review), because of their theoretical appeal, inspiration from biology and human cognition, and because of empirical success in vision (Ranzato et al., 2007; Larochelle et al., 2007; Vincent et al., 2008) and natural language processing (NLP) (Collobert & Weston, 2008; Mnih & Hinton, 2009). Theoretical results reviewed and discussed by Bengio (2009), suggest that in order to learn the kind of complicated functions that can represent high-level abstractions (e.g. in vision, language, and other AI-level tasks), one may need *deep architectures*.

Most of the recent experimental results with deep architecture are obtained with models that can be turned into deep supervised neural networks, but with initialization or training schemes different from the classical feedforward neural networks (Rumelhart et al., 1986). Why are these new algorithms working so much better than the standard random initialization and gradient-based optimization of a supervised training criterion? Part of the answer may be found in recent analyses of the effect of unsupervised pre-training (Erhan et al., 2009), showing that it acts as a regularizer that initializes the parameters in a "better" basin of attraction of the optimization procedure, corresponding to an apparent local minimum associated with better generalization. But earlier work (Bengio et al., 2007) had shown that even a purely supervised but greedy layer-wise procedure would give better results. So here instead of focusing on what unsupervised pre-training or semi-supervised criteria bring to deep architectures, we focus on analyzing what may be going wrong with good old (but deep) multi-layer neural networks.

Our analysis is driven by investigative experiments to monitor activations (watching for saturation of hidden units) and gradients, across layers and across training iterations. We also evaluate the effects on these of choices of activation function (with the idea that it might affect saturation) and initialization procedure (since unsupervised pre-training is a particular form of initialization and it has a drastic impact).

Appearing in Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS) 2010, Chia Laguna Resort, Sardinia, Italy. Volume 9 of JMLR: W&CP 9.

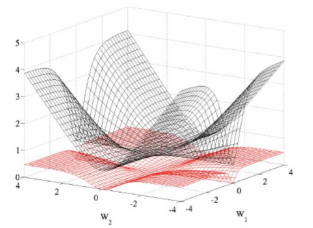


Figure 5: *Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, W_1 respectively on the first layer and W_2 on the second, output layer.*

For a dense artificial neural network using symmetric activation function f with unit derivative at 0 (i.e. $f'(0) = 1$), if we write $\mathbf{z}^i$ for the activation vector of layer i, and $\mathbf{s}^i$ the argument vector of the activation function at layer i, we have $\mathbf{s}^i = \mathbf{z}^i W^i + \mathbf{b}^i$ and $\mathbf{z}^{i+1} = f(\mathbf{s}^i)$. From these definitions we obtain the following:

$$\frac{\partial Cost}{\partial s_k^i} = f'(s_k^i) W_{k,\bullet}^{i+1} \frac{\partial Cost}{\partial \mathbf{s}^{i+1}} \quad (2)$$

$$\frac{\partial Cost}{\partial w_{l,k}^i} = z_l^i \frac{\partial Cost}{\partial s_k^i} \quad (3)$$

The variances will be expressed with respect to the input, output and weight initialization randomness. Consider the hypothesis that we are in a linear regime at the initialization, that the weights are initialized independently and that the inputs features variances are the same ($= Var[x]$). Then we can say that, with n_i the size of layer i and x the network input,

$$f'(\mathbf{s}_k^i) \approx 1, \quad (4)$$

$$Var[z^i] = Var[x] \prod_{i'=0}^{i-1} n_{i'} Var[W^{i'}], \quad (5)$$

We write $Var[W^{i'}]$ for the shared scalar variance of all weights at layer i'. Then for a network with d layers,

$$Var\left[\frac{\partial Cost}{\partial s^i}\right] = Var\left[\frac{\partial Cost}{\partial s^d}\right] \prod_{i'=i}^{d} n_{i'+1} Var[W^{i'}], \quad (6)$$

$$Var\left[\frac{\partial Cost}{\partial w^i}\right] = \prod_{i'=0}^{i-1} n_{i'} Var[W^{i'}] \prod_{i'=i}^{d-1} n_{i'+1} Var[W^{i'}] \times Var[x] Var\left[\frac{\partial Cost}{\partial s^d}\right]. \quad (7)$$

From a forward-propagation point of view, to keep information flowing we would like that

$$\forall (i, i'), Var[z^i] = Var[z^{i'}]. \quad (8)$$

From a back-propagation point of view we would similarly like to have

$$\forall (i, i'), Var\left[\frac{\partial Cost}{\partial s^i}\right] = Var\left[\frac{\partial Cost}{\partial s^{i'}}\right]. \quad (9)$$

These two conditions transform to:

$$\forall i, \quad n_i Var[W^i] = 1 \quad (10)$$

$$\forall i, \quad n_{i+1} Var[W^i] = 1 \quad (11)$$

As a compromise between these two constraints, we might want to have

$$\forall i, \quad Var[W^i] = \frac{2}{n_i + n_{i+1}} \quad (12)$$

Note how both constraints are satisfied when all layers have the same width. If we also have the same initialization for the weights we could get the following interesting properties:

$$\forall i, Var\left[\frac{\partial Cost}{\partial s^i}\right] = \left[n Var[W]\right]^{d-i} Var[x] \quad (13)$$

$$\forall i, Var\left[\frac{\partial Cost}{\partial w^i}\right] = \left[n Var[W]\right]^{d} Var[x] Var\left[\frac{\partial Cost}{\partial s^d}\right] \quad (14)$$

We can see that the variance of the gradient on the weights is the same for all layers, but the variance of the back-propagated gradient might still vanish or explode as we consider deeper networks. Note how this is reminiscent of issues raised when studying recurrent neural networks (Bengio et al., 1994), which can be seen as very deep networks when unfolded through time.

The standard initialization that we have used (eq.1) gives rise to variance with the following property:

$$n Var[W] = \frac{1}{3} \quad (15)$$

where n is the layer size (assuming all layers of the same size). This will cause the variance of the back-propagated gradient to be dependent on the layer (and decreasing).

The normalization factor may therefore be important when initializing deep networks because of the multiplicative effect through layers, and we suggest the following initialization procedure to approximately satisfy our objectives of maintaining activation variances and back-propagated gradients variance as one moves up or down the network. We call it the **normalized initialization**:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] \quad (16)$$

正态分布：

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

均匀分布：

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

n_{in}：上一层的神经元数量（输入维度）
n_{out}：当前层的神经元数量（输出维度）

source:
https://www.researchgate.net/publication/215616968_Understanding_the_difficulty_of_training_deep_feedforward_neural_networks

扩展：怎么进行权重初始化？

- **He 初始化**：适用激活函数：ReLU, Leaky ReLU（这些会将负数置零，造成非对称）。
- 核心思想：适当放大初始权重，使 ReLU 激活后的方差仍能保持稳定。

Delving Deep into Rectifiers:
Surpassing Human-Level Performance on ImageNet Classification

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun

Microsoft Research

{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

Abstract

Rectified activation units (rectifiers) are essential for state-of-the-art neural networks. In this work, we study rectifier neural networks for image classification from two aspects. First, we propose a Parametric Rectified Linear Unit (PReLU) that generalizes the traditional rectified unit. PReLU improves model fitting with nearly zero extra computational cost and little overfitting risk. Second, we derive a robust initialization method that particularly considers the rectifier nonlinearities. This method enables us to train extremely deep rectified models directly from scratch and to investigate deeper or wider network architectures. Based on our PReLU networks (PReLU-nets), we achieve ***4.94%*** *top-5 test error on the ImageNet 2012 classification dataset. This is a 26% relative improvement over the ILSVRC 2014 winner (GoogLeNet, 6.66% [29]). To our knowledge, our result is the first to surpass human-level performance (5.1%, [22]) on this visual recognition challenge.*

1. Introduction

Convolutional neural networks (CNNs) [17, 16] have demonstrated recognition accuracy better than or comparable to humans in several visual recognition tasks, including recognizing traffic signs [3], faces [30, 28], and handwritten digits [3, 31]. In this work, we present a result that surpasses human-level performance on a more generic and challenging recognition task - the classification task in the 1000-class ImageNet dataset [22].

In the last few years, we have witnessed tremendous improvements in recognition performance, mainly due to advances in two technical directions: building more powerful models, and designing effective strategies against overfitting. On one hand, neural networks are becoming more capable of fitting training data, because of increased complexity (*e.g.*, increased depth [25, 29], enlarged width [33, 24], and the use of smaller strides [33, 24, 2, 25]), new nonlinear activations [21, 20, 34, 19, 27, 9], and sophisticated layer designs [29, 11]. On the other hand, better generalization is achieved by effective regularization techniques [12, 26, 9, 31], aggressive data augmentation [16, 13, 25, 29], and large-scale data [4, 22].

Among these advances, the rectifier neuron [21, 8, 20, 34], *e.g.*, Rectified Linear Unit (ReLU), is one of several keys to the recent success of deep networks [16]. It expedites convergence of the training procedure [16] and leads to better solutions [21, 8, 20, 34] than conventional sigmoid-like units. Despite the prevalence of rectifier networks, recent improvements of models [33, 24, 11, 25, 29] and theoretical guidelines for training them [7, 23] have rarely focused on the properties of the rectifiers.

In this paper, we investigate neural networks from two aspects particularly driven by the rectifiers. First, we propose a new generalization of ReLU, which we call *Parametric Rectified Linear Unit* (PReLU). This activation function adaptively learns the parameters of the rectifiers, and improves accuracy at negligible extra computational cost. Second, we study the difficulty of training rectified models that are very deep. By explicitly modeling the nonlinearity of rectifiers (ReLU/PReLU), we derive a theoretically sound initialization method, which helps with convergence of very deep models (*e.g.*, with 30 weight layers) trained directly from scratch. This gives us more flexibility to explore more powerful network architectures.

On the 1000-class ImageNet 2012 dataset, our PReLU network (PReLU-net) leads to a single-model result of 5.71% top-5 error, which surpasses all existing multi-model results. Further, our multi-model result achieves **4.94%** top-5 error on the test set, which is a 26% relative improvement over the ILSVRC 2014 winner (GoogLeNet, 6.66% [29]). To the best of our knowledge, our result surpasses for the first time the reported human-level performance (5.1% in [22]) on this visual recognition challenge.

We let the initialized elements in W_l be mutually independent and share the same distribution. As in [7], we assume that the elements in $\mathbf{x}_l$ are also mutually independent and share the same distribution, and $\mathbf{x}_l$ and W_l are independent of each other. Then we have:

$$Var[y_l] = n_l Var[w_l x_l], \tag{6}$$

where now y_l, x_l, and w_l represent the random variables of each element in $\mathbf{y}_l$, W_l, and $\mathbf{x}_l$ respectively. We let w_l have zero mean. Then the variance of the product of independent variables gives us:

$$Var[y_l] = n_l Var[w_l] E[x_l^2]. \tag{7}$$

Here $E[x_l^2]$ is the expectation of the square of x_l. It is worth noticing that $E[x_l^2] \neq Var[x_l]$ unless x_l has zero mean. For the ReLU activation, $x_l = max(0, y_{l-1})$ and thus it does not have zero mean. This will lead to a conclusion different from [7].

If we let w_{l-1} have a symmetric distribution around zero and $b_{l-1} = 0$, then y_{l-1} has zero mean and has a symmetric distribution around zero. This leads to $E[x_l^2] = \frac{1}{2} Var[y_{l-1}]$ when f is ReLU. Putting this into Eqn.(7), we obtain:

$$Var[y_l] = \frac{1}{2} n_l Var[w_l] Var[y_{l-1}]. \tag{8}$$

With L layers put together, we have:

$$Var[y_L] = Var[y_1] \left(\prod_{l=2}^{L} \frac{1}{2} n_l Var[w_l] \right). \tag{9}$$

This product is the key to the initialization design. A proper initialization method should avoid reducing or magnifying the magnitudes of input signals exponentially. So we expect the above product to take a proper scalar (*e.g.*, 1). A sufficient condition is:

$$\frac{1}{2} n_l Var[w_l] = 1, \quad \forall l. \tag{10}$$

This leads to a zero-mean Gaussian distribution whose standard deviation (std) is $\sqrt{2/n_l}$. This is our way of initialization. We also initialize $\mathbf{b} = 0$.

For the first layer ($l = 1$), we should have $n_1 Var[w_1] = 1$ because there is no ReLU applied on the input signal. But the factor $1/2$ does not matter if it just exists on one layer. So we also adopt Eqn.(10) in the first layer for simplicity.

Backward Propagation Case

For back-propagation, the gradient of a conv layer is computed by:

$$\Delta \mathbf{x}_l = \hat{\mathrm{W}}_l \Delta \mathbf{y}_l. \tag{11}$$

Here we use $\Delta \mathbf{x}$ and $\Delta \mathbf{y}$ to denote gradients ($\frac{\partial \varepsilon}{\partial \mathbf{x}}$ and $\frac{\partial \varepsilon}{\partial \mathbf{y}}$) for simplicity. $\Delta \mathbf{y}$ represents k-by-k pixels in d channels, and is reshaped into a $k^2 d$-by-1 vector. We denote $\hat{n} = k^2 d$. Note that $\hat{n} \neq n = k^2 c$. $\hat{\mathrm{W}}$ is a c-by-$\hat{n}$ matrix where the filters are rearranged in the way of back-propagation. Note that W and $\hat{\mathrm{W}}$ can be reshaped from each other. $\Delta \mathbf{x}$ is a c-by-1 vector representing the gradient at a pixel of this layer. As above, we assume that w_l and Δy_l are independent of each other, then Δx_l has zero mean for all l, when w_l is initialized by a symmetric distribution around zero.

In back-propagation we also have $\Delta y_l = f'(y_l) \Delta x_{l+1}$ where f' is the derivative of f. For the ReLU case, $f'(y_l)$ is zero or one, and their probabilities are equal. We assume that $f'(y_l)$ and Δx_{l+1} are independent of each other. Thus we have $E[\Delta y_l] = E[\Delta x_{l+1}]/2 = 0$, and also $E[(\Delta y_l)^2] = Var[\Delta y_l] = \frac{1}{2} Var[\Delta x_{l+1}]$. Then we compute the variance of the gradient in Eqn.(11):

$$\begin{aligned} Var[\Delta x_l] &= \hat{n}_l Var[w_l] Var[\Delta y_l] \\ &= \frac{1}{2} \hat{n}_l Var[w_l] Var[\Delta x_{l+1}]. \end{aligned} \tag{12}$$

The scalar $1/2$ in both Eqn.(12) and Eqn.(8) is the result of ReLU, though the derivations are different. With L layers put together, we have:

$$Var[\Delta x_2] = Var[\Delta x_{L+1}] \left(\prod_{l=2}^{L} \frac{1}{2} \hat{n}_l Var[w_l] \right). \tag{13}$$

We consider a sufficient condition that the gradient is not exponentially large/small:

$$\frac{1}{2} \hat{n}_l Var[w_l] = 1, \quad \forall l. \tag{14}$$

The only difference between this equation and Eqn.(10) is that $\hat{n}_l = k_l^2 d_l$ while $n_l = k_l^2 c_l = k_l^2 d_{l-1}$. Eqn.(14) results in a zero-mean Gaussian distribution whose std is $\sqrt{2/\hat{n}_l}$.

For the first layer ($l = 1$), we need not compute Δx_1 because it represents the image domain. But we can still adopt Eqn.(14) in the first layer, for the same reason as in the forward propagation case - the factor of a single layer does not make the overall product exponentially large/small.

We note that it is sufficient to use either Eqn.(14) or Eqn.(10) alone. For example, if we use Eqn.(14), then in Eqn.(13) the product $\prod_{l=2}^{L} \frac{1}{2} \hat{n}_l Var[w_l] = 1$, and in Eqn.(9) the product $\prod_{l=2}^{L} \frac{1}{2} n_l Var[w_l] = \prod_{l=2}^{L} n_l / \hat{n}_l = c_2 / d_L$, which is not a diminishing number in common network designs. This means that if the initialization properly scales the backward signal, then this is also the case for the forward signal; and vice versa. For all models in this paper, both forms can make them converge.

Discussions

If the forward/backward signal is inappropriately scaled by a factor β in each layer, then the final propagated signal

正态分布：

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$

均匀分布：

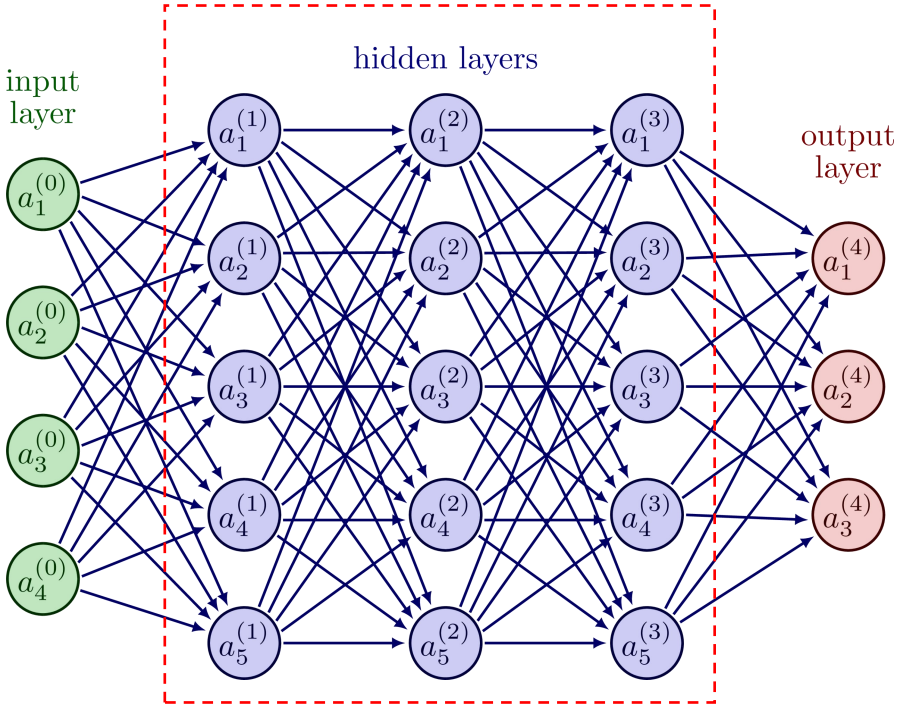
$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}}}}, \sqrt{\frac{6}{n_{\text{in}}}}\right)$$

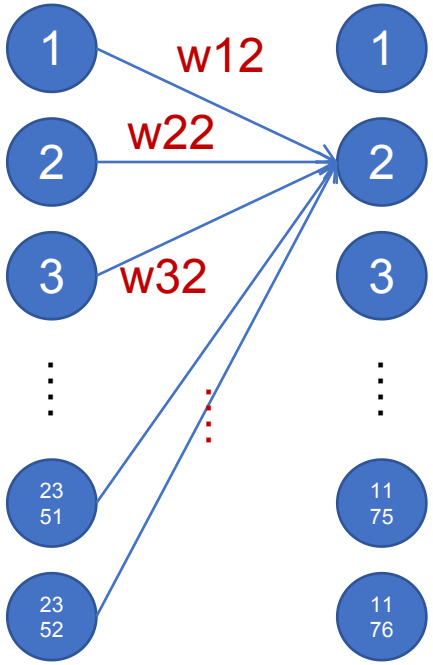
n_{in}：上一层的神经元数量（输入维度）
n_{out}：当前层的神经元数量（输出维度）

source: https://arxiv.org/abs/1502.01852

内部结构：隐藏层

- 作用：负责数据的特征提取与映射。
- 特征：每个神经元代表一个输入变量（如像素值、数值特征等）。





Step1: 根据上一层计算权重

$$x_{22} = w_{12} \times x_{11} + w_{22} \times x_{12} + w_{32} \times x_{13} + \ldots$$

Step2: 加入偏置值

$$z_{22} = x_{22} + b$$

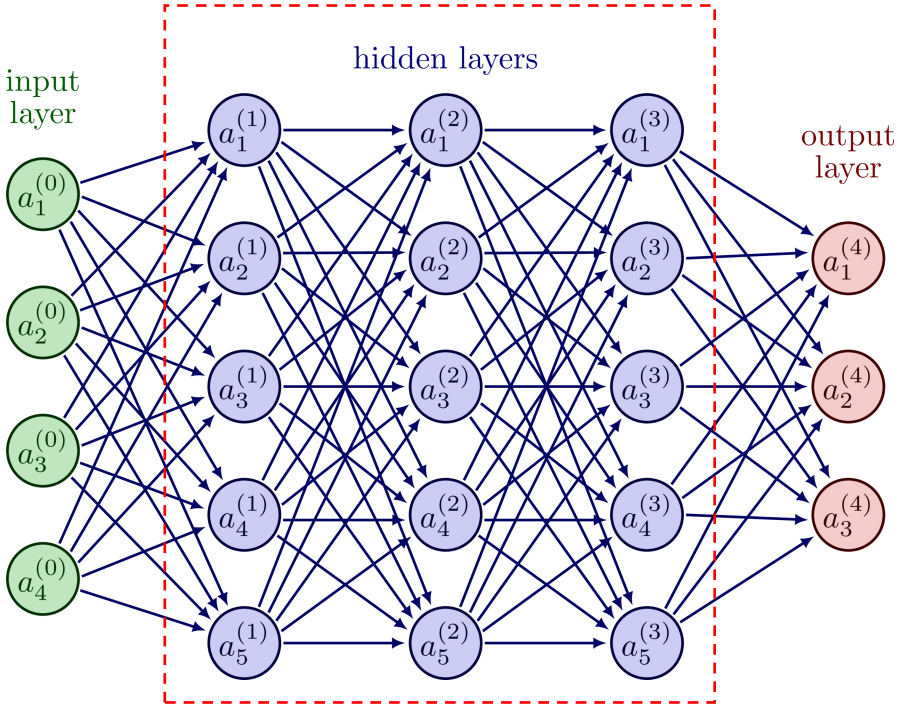
Step3: 送入激活函数

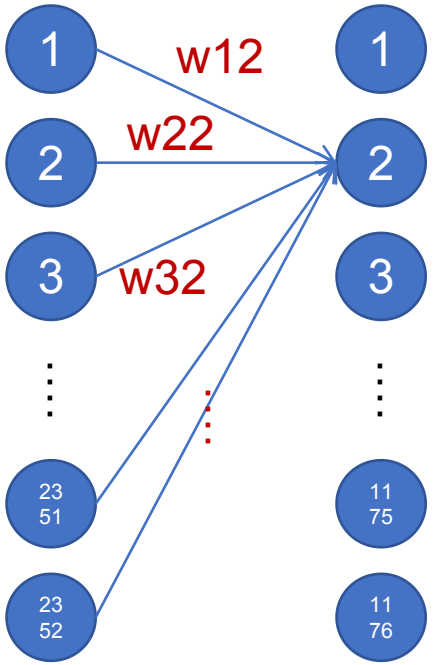
$$z_{22} = f(z_{22})$$

隐藏层通过线性变换与非线性激活逐层提取和组合特征，是神经网络学习复杂模式和表达能力的核心。

内部结构：隐藏层

- 作用：负责数据的特征提取与映射。
- 特征：每个神经元代表一个输入变量（如像素值、数值特征等）。





第 i 层第 j 个神经元的通用计算公式：

$$x_{i,j} = \sum_{k=1}^{n_{i-1}} w_{j,k}^{(i)} \cdot x_{i-1,k} + b_j^{(i)}, \quad z_{i,j} = f_i\left(x_{i,j}\right)$$

wijk：第i-1层第k个神经元到第i层第j个神经元的权重

bij：偏置

xi,j：第i层第j个神经元的线性加权和

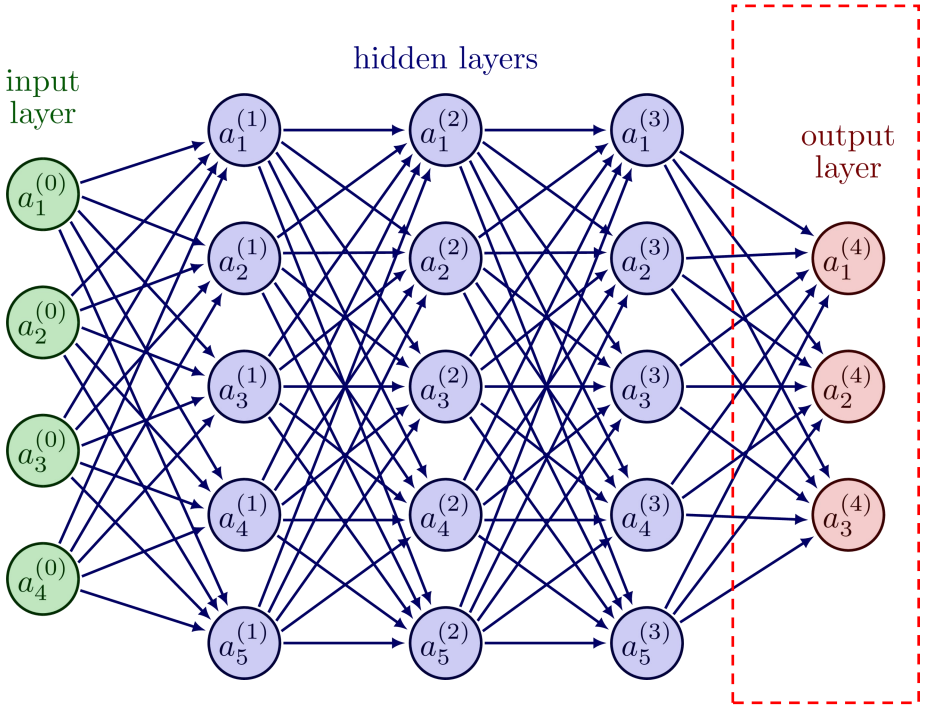
fi(·)：激活函数，例如ReLU, sigmoid, Tanh

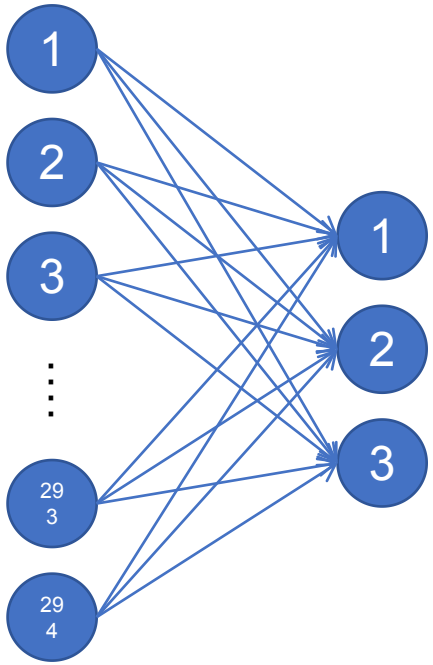
zi,j：激活输出

隐藏层通过线性变换与非线性激活逐层提取和组合特征，是神经网络学习复杂模式和表达能力的核心。

内部结构：输出层

- 作用：生成预测结果，最终输出分类、数值或概率。
- 输出数量：1）回归问题：1个连续值（无激活或ReLU）；2）二分类问题：1个输出节点（+Sigmoid）；3）多分类问题：n个输出节点（+Softmax）。





回归任务（Regression）：输出为连续数值

$$\hat{y} = f(Wx + b)$$

二分类任务（Binary Classification）：输出一个概率值，使用Sigmoid激活函数

$$\hat{y} = \sigma(Wx + b) = \frac{1}{1 + e^{-(Wx+b)}}$$

多分类任务（Multiclass Classification）：输出多个类别的概率分布，使用 Softmax 函数

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, 2, \ldots, K$$

输出层负责将神经网络的内部表示转换为最终预测结果，形式取决于具体任务（分类、回归或生成）。

▶ 内部结构：提升技巧与注意点

- **维度匹配**——每层输入输出的形状必须正确：线性变换：(输出节点数, 输入节点数) × (输入节点数, 1) = (输出节点数, 1)。
- **权重初始化**——不恰当的初始化可能导致：梯度消失或爆炸/网络训练困难。常用策略：Xavier（Glorot）初始化：适合 Sigmoid/Tanh；He 初始化：适合 ReLU/Leaky ReLU。
- **激活函数选择**——每一层激活函数应根据任务和位置选取：隐藏层：ReLU、Tanh、Leaky ReLU；输出层：二分类（Sigmoid）、多分类（Softmax）、多分类（Softmax）。
- **偏置项（bias）不可遗漏**——偏置相当于激活函数的“起点调整”，作用：避免所有输入为 0 时输出也为 0，提升模型表达能力。
- **注意数值稳定性**——指数函数中避免溢出；防止除以 0，例如 softmax 中加 +1e-9。
- **合理的输入归一化**——特征标准化（如 0 均值、单位方差）有助于加快训练收敛、避免梯度震荡。

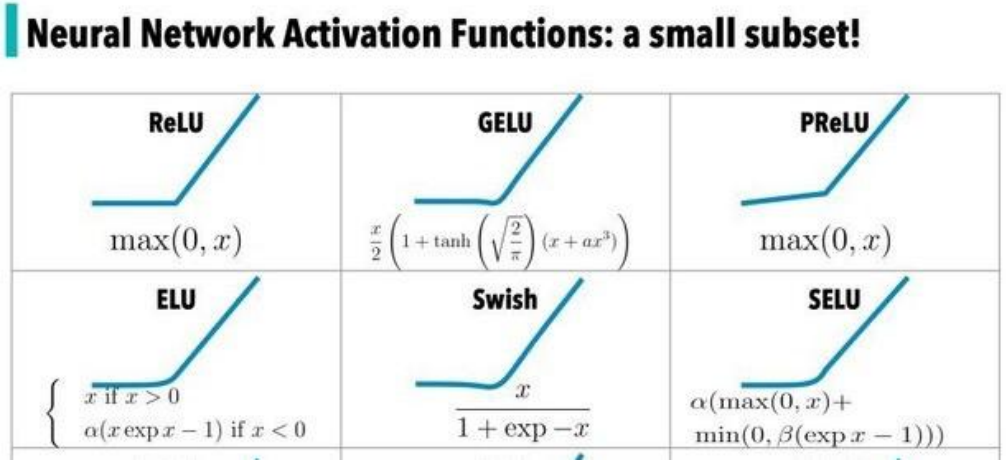
神经网络内部细节虽不直接可见，但任何小问题（如维度不对、初始化不当、激活不合理）都可能导致模型“无法学习”或性能不稳定。建议从构造到调试都保持结构清晰、数值可控。

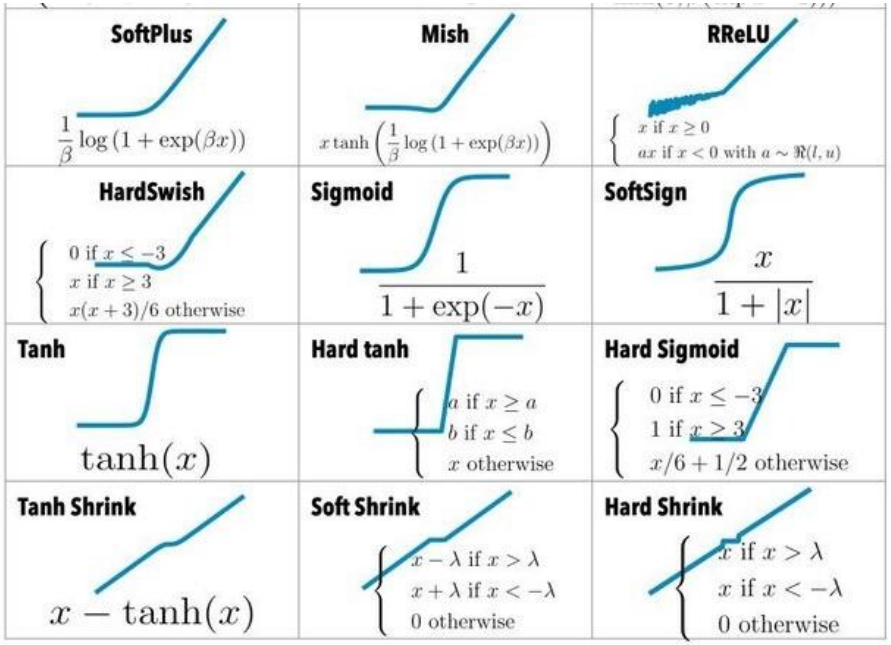
目录章节

CONTENTS

▶ 什么是激活函数？

- 激活函数是神经网络中对每个神经元输出进行非线性变换的函数，帮助模型捕捉复杂的模式和关系。
- 为什么需要激活函数？1）引入非线性，使网络能够拟合非线性数据和复杂函数；2）使网络具备更强的表达能力，而不仅仅是线性组合；3）使网络具备更强的表达能力，而不仅仅是线性组合。





激活函数是神经网络的“灵魂”，决定了模型能否捕获复杂的非线性关系。选择合适的激活函数，有助于提升模型的训练效率和效果。

激活函数：关键工作

Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control, Signals, and Systems

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j), \qquad (1)$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^T is the transpose of y so that $y^T x$ is the inner product of y and x.) Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ's:

$$\sigma(t) \to \begin{cases} 1 & \text{as} \quad t \to +\infty, \\ 0 & \text{as} \quad t \to -\infty. \end{cases}$$

Such functions arise naturally in neural network theory as the activation function of a neural node (or *unit* as is becoming the preferred term) [L1], [RHM]. The main result of this paper is a demonstration of the fact that sums of the form (1) are dense in the space of continuous functions on the unit cube if σ is any continuous sigmoidal

* Date received: October 21, 1988. Date revised: February 17, 1989. This research was supported in part by NSF Grant DCR-8619103, ONR Contract N000-86-G-0202 and DOE Grant DE-FG02-85ER25001.

† Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, U.S.A.

arbitrarily oriented rectangle. Notice that *no* scaling of the rectangle is allowed—only rigid-body motions in Euclidean space! We then have that summations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(U_j x + y_j)$$

are dense in $L^1(\mathbb{R}^n)$. This follows from direct application of the Wiener Tauberian theorem [R2] and the observation that the Fourier transform of σ vanishes on a mesh in $\mathbb{R}^n$ that does not include the origin. The intersection of all possible rotations of those meshes is empty and so σ together with its rotations and translations generates a space dense in $L^1(\mathbb{R}^n)$.

This last result is closely related to the classical *Pompeiu Problem* [BST] and using the results of [BST] we speculate that the rectangle in the above paragraph can be replaced by any convex set with a corner as defined in [BST].

5. Summary

We have demonstrated that finite superpositions of a fixed, univariate function that is *discriminatory* can uniformly approximate any continuous function of n real variables with support in the unit hypercube. Continuous sigmoidal functions of the type commonly used in real-valued neural network theory are discriminatory. This combination of results demonstrates that any continuous function can be uniformly approximated by a continuous neural network having only one internal, hidden layer and with an arbitrary continuous sigmoidal nonlinearity (Theorem 2). Theorem 3 and the subsequent discussion show in a precise way that arbitrary decision functions can be arbitrarily well approximated by a neural network with one internal layer and a continuous sigmoidal nonlinearity.

Table 1 summarizes the various contributions of which we are aware.

Table 1

Function type and transformations	Function space	References
$\sigma(y^T x + \theta)$, σ continuous sigmoidal, $y \in \mathbb{R}^n$, $\theta \in \mathbb{R}$	$C(I_n)$	This paper
$\sigma(y^T x + \theta)$, σ monotonic sigmoidal, $y \in \mathbb{R}^n$, $\theta \in \mathbb{R}$	$C(I_n)$	[F], [HSW]
$\sigma(y^T x + \theta)$, σ sigmoidal, $y \in \mathbb{R}^n$, $\theta \in \mathbb{R}$	$C(I_n)$	[J]
$\sigma(y^T x + \theta)$, $\sigma \in L^1(\mathbb{R})$ $\int \sigma(t)\,dt \neq 0$, $y \in \mathbb{R}^n$, $\theta \in \mathbb{R}$	$L^1(I_n)$	This paper
$\sigma(y^T x + \theta)$, σ continuous sigmoidal, $y \in \mathbb{R}^n$, $\theta \in \mathbb{R}$	$L^2(I_n)$	[CD]
$\sigma(Ux + y)$, $U \in \mathbb{R}^{n \times n}$, $y \in \mathbb{R}^n$, σ indicator of a rectangle	$L^1(I_n)$	This paper
$\sigma(tx + y)$, $t \in \mathbb{R}$, $\sigma \in L^1(\mathbb{R})$ $y \in \mathbb{R}^n$, $\int \sigma(x)\,dx \neq 0$	$L^1(\mathbb{R}^n)$	Wiener Tauberian theorem [R2]

While the approximating properties we have described are quite powerful, we have focused only on existence. The important questions that remain to be answered deal with feasibility, namely how many terms in the summation (or equivalently, how many neural nodes) are required to yield an approximation of a given quality? What properties of the function being approximated play a role in determining the number of terms? At this point, we can only say that we suspect quite strongly that the overwhelming majority of approximation problems will require astronomical numbers of terms. This feeling is based on the *curse of dimensionality* that plagues multidimensional approximation theory and statistics. Some recent progress concerned with the relationship between a function being approximated and the number of terms needed for a suitable approximation can be found in [MSJ] and [BH], [BEHW], and [V] for related problems. Given the conciseness of the results of this paper, we believe that these avenues of research deserve more attention.

Acknowledgments. The author thanks Brad Dickinson, Christopher Chase, Lee Jones, Todd Quinto, Lee Rubel, John Makhoul, Alex Samarov, Richard Lippmann, and the anonymous referees for comments, additional references, and improvements in the presentation of this material.

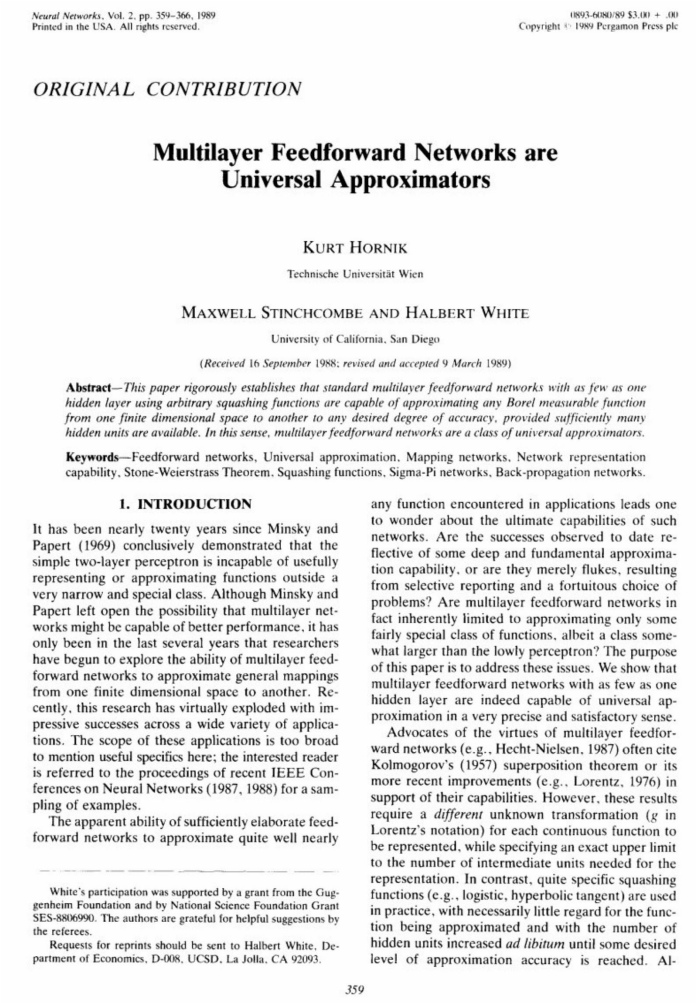
References

[A] R. B. Ash, *Real Analysis and Probability*, Academic Press, New York, 1972.
[BH] E. Baum and D. Haussler, What size net gives valid generalization?, *Neural Comput.* (to appear).
[B] B. Bavarian (ed.), Special section on neural networks for systems and control, *IEEE Control Systems Mag.*, **8** (April 1988), 3–31.
[BEHW] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, Classifying learnable geometric concepts with the Vapnik–Chervonenkis dimension, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, Berkeley, CA, 1986, pp. 273–282.
[BST] L. Brown, B. Schreiber, and B. A. Taylor, Spectral synthesis and the Pompeiu problem, *Ann. Inst. Fourier (Grenoble)*, **23** (1973), 125–154.
[CD] S. M. Carroll and B. W. Dickinson, Construction of neural nets using the Radon transform, preprint, 1989.
[C] G. Cybenko, Continuous Valued Neural Networks with Two Hidden Layers are Sufficient, Technical Report, Department of Computer Science, Tufts University, 1988.
[DS] P. Diaconis and M. Shahshahani, On nonlinear functions of linear combinations, *SIAM J. Sci. Statist. Comput.*, **5** (1984), 175–191.
[F] K. Funahashi, On the approximate realization of continuous mappings by neural networks, *Neural Networks* (to appear).
[G] L. J. Griffiths (ed.), Special section on neural networks, *IEEE Trans. Acoust. Speech Signal Process.*, **36** (1988), 1107–1190.
[HSW] K. Hornik, M. Stinchcombe, and H. White, Multi-layer feedforward networks are universal approximators, preprint, 1988.
[HL1] W. Y. Huang and R. P. Lippmann, Comparisons Between Neural Net and Conventional Classifiers, Technical Report, Lincoln Laboratory, MIT, 1987.
[HL2] W. Y. Huang and R.P. Lippmann, Neural Net and Traditional Classifiers, Technical Report, Lincoln Laboratory, MIT, 1987.
[H] P. J. Huber, Projection pursuit, *Ann. Statist.*, **13** (1985), 435–475.
[J] L. K. Jones, Constructive approximations for neural networks by sigmoidal functions, Technical Report Series, No. 7, Department of Mathematics, University of Lowell, 1988.

1. Cybenko, G. (1989).
Title: Approximation by superpositions of a sigmoidal function
Source: Mathematics of Control, Signals, and Systems
Contribution：这是第一个提出通用近似定理的经典论文，证明了含有单隐层的神经网络，使用sigmoid激活函数，能够以任意精度逼近任何连续函数。

source:
https://www.yumpu.com/en/document/view/29935370/approximation-by-superpositions-of-a-sigmoidal-function

激活函数：关键工作



Neural Networks, Vol. 2, pp. 359–366, 1989
Printed in the USA. All rights reserved.

ORIGINAL CONTRIBUTION

Multilayer Feedforward Networks are Universal Approximators

KURT HORNIK
Technische Universität Wien

MAXWELL STINCHCOMBE AND HALBERT WHITE
University of California, San Diego

(*Received* 16 *September* 1988; *revised and accepted* 9 *March* 1989)

Abstract—*This paper rigorously establishes that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.*

Keywords—Feedforward networks, Universal approximation, Mapping networks, Network representation capability, Stone-Weierstrass Theorem, Squashing functions, Sigma-Pi networks, Back-propagation networks.

1. INTRODUCTION

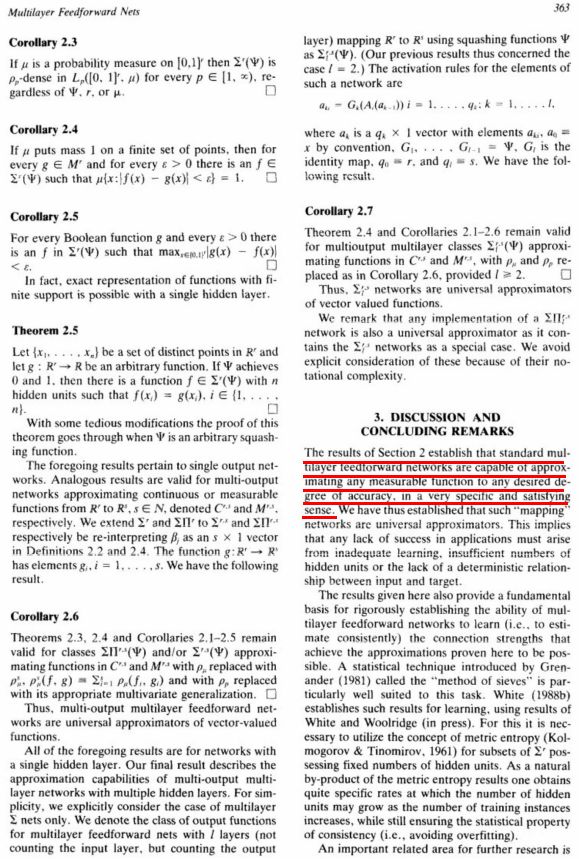
It has been nearly twenty years since Minsky and Papert (1969) conclusively demonstrated that the simple two-layer perceptron is incapable of usefully representing or approximating functions outside a very narrow and special class. Although Minsky and Papert left open the possibility that multilayer networks might be capable of better performance, it has only been in the last several years that researchers have begun to explore the ability of multilayer feedforward networks to approximate general mappings from one finite dimensional space to another. Recently, this research has virtually exploded with impressive successes across a wide variety of applications. The scope of these applications is too broad to mention useful specifics here; the interested reader is referred to the proceedings of recent IEEE Conferences on Neural Networks (1987, 1988) for a sampling of examples.

The apparent ability of sufficiently elaborate feedforward networks to approximate quite well nearly any function encountered in applications leads one to wonder about the ultimate capabilities of such networks. Are the successes observed to date reflective of some deep and fundamental approximation capability, or are they merely flukes, resulting from selective reporting and a fortuitous choice of problems? Are multilayer feedforward networks in fact inherently limited to approximating only some fairly special class of functions, albeit a class somewhat larger than the lowly perceptron? The purpose of this paper is to address these issues. We show that multilayer feedforward networks with as few as one hidden layer are indeed capable of universal approximation in a very precise and satisfactory sense.

Advocates of the virtues of multilayer feedforward networks (e.g., Hecht-Nielsen, 1987) often cite Kolmogorov's (1957) superposition theorem or its more recent improvements (e.g., Lorentz, 1976) in support of their capabilities. However, these results require a *different* unknown transformation (g in Lorentz's notation) for each continuous function to be represented, while specifying an exact upper limit to the number of intermediate units needed for the representation. In contrast, quite specific squashing functions (e.g., logistic, hyperbolic tangent) are used in practice, with necessarily little regard for the function being approximated and with the number of hidden units increased *ad libitum* until some desired level of approximation accuracy is reached. Al-

White's participation was supported by a grant from the Guggenheim Foundation and by National Science Foundation Grant SES-8806990. The authors are grateful for helpful suggestions by the referees.

Requests for reprints should be sent to Halbert White, Department of Economics, D-008, UCSD, La Jolla, CA 92093.



Corollary 2.3

If μ is a probability measure on $[0,1]^r$ then $\Sigma^r(\Psi)$ is ρ_p-dense in $L_p([0,1]^r, \mu)$ for every $p \in [1, \infty)$, regardless of Ψ, r, or μ. □

Corollary 2.4

If μ puts mass 1 on a finite set of points, then for every $g \in M^r$ and for every $\varepsilon > 0$ there is an $f \in \Sigma^r(\Psi)$ such that $\mu\{x : |f(x) - g(x)| < \varepsilon\} = 1$. □

Corollary 2.5

For every Boolean function g and every $\varepsilon > 0$ there is an f in $\Sigma^r(\Psi)$ such that $\max_{x \in \{0,1\}^r} |g(x) - f(x)| < \varepsilon$. □

In fact, exact representation of functions with finite support is possible with a single hidden layer.

Theorem 2.5

Let $\{x_1, \ldots, x_n\}$ be a set of distinct points in R^r and let $g : R^r \to R$ be an arbitrary function. If Ψ achieves 0 and 1, then there is a function $f \in \Sigma^r(\Psi)$ with n hidden units such that $f(x_i) = g(x_i)$, $i \in \{1, \ldots, n\}$. □

With some tedious modifications the proof of this theorem goes through when Ψ is an arbitrary squashing function.

The foregoing results pertain to single output networks. Analogous results are valid for multi-output networks approximating continuous or measurable functions from R^r to R^s, $s \in N$, denoted $C^{r,s}$ and $M^{r,s}$, respectively. We extend Σ^r and $\Sigma\Pi^r$ to $\Sigma^{r,s}$ and $\Sigma\Pi^{r,s}$ respectively be re-interpreting β_j as an $s \times 1$ vector in Definitions 2.2 and 2.4. The function $g : R^r \to R^s$ has elements g_i, $i = 1, \ldots, s$. We have the following result.

Corollary 2.6

Theorems 2.3, 2.4 and Corollaries 2.1–2.5 remain valid for classes $\Sigma\Pi^{r,s}(\Psi)$ and/or $\Sigma^{r,s}(\Psi)$ approximating functions in $C^{r,s}$ and $M^{r,s}$ with ρ_p replaced with ρ_p^s, $\rho_p^s(f, g) \equiv \sum_{i=1}^s \rho_p(f_i, g_i)$ and with ρ_p replaced with its appropriate multivariate generalization. □

Thus, multi-output multilayer feedforward networks are universal approximators of vector-valued functions.

All of the foregoing results are for networks with a single hidden layer. Our final result describes the approximation capabilities of multi-output multilayer networks with multiple hidden layers. For simplicity, we explicitly consider the case of multilayer Σ nets only. We denote the class of output functions for multilayer feedforward nets with l layers (not counting the input layer, but counting the output layer) mapping R^r to R^s using squashing functions Ψ as $\Sigma_l^{r,s}(\Psi)$. (Our previous results thus concerned the case $l = 2$.) The activation rules for the elements of such a network are

$$a_{ik} = G_k(A_i(a_{i-1})) \; i = 1, \ldots, q_k; \; k = 1, \ldots, l,$$

where a_k is a $q_k \times 1$ vector with elements a_{ik}, $a_0 \equiv x$ by convention, $G_1, \ldots, G_{l-1} = \Psi$, G_l is the identity map, $q_0 \equiv r$, and $q_l \equiv s$. We have the following result.

Corollary 2.7

Theorem 2.4 and Corollaries 2.1–2.6 remain valid for multioutput multilayer classes $\Sigma_l^{r,s}(\Psi)$ approximating functions in $C^{r,s}$ and $M^{r,s}$, with ρ_p and ρ_p replaced as in Corollary 2.6, provided $l \geq 2$. □

Thus, $\Sigma_l^{r,s}$ networks are universal approximators of vector valued functions.

We remark that any implementation of a $\Sigma\Pi_l^{r,s}$ network is also a universal approximator as it contains the $\Sigma_l^{r,s}$ networks as a special case. We avoid explicit consideration of these because of their notational complexity.

3. DISCUSSION AND CONCLUDING REMARKS

The results of Section 2 establish that standard multilayer feedforward networks are capable of approximating any measurable function to any desired degree of accuracy, in a very specific and satisfying sense. We have thus established that such "mapping" networks are universal approximators. This implies that any lack of success in applications must arise from inadequate learning, insufficient numbers of hidden units or the lack of a deterministic relationship between input and target.

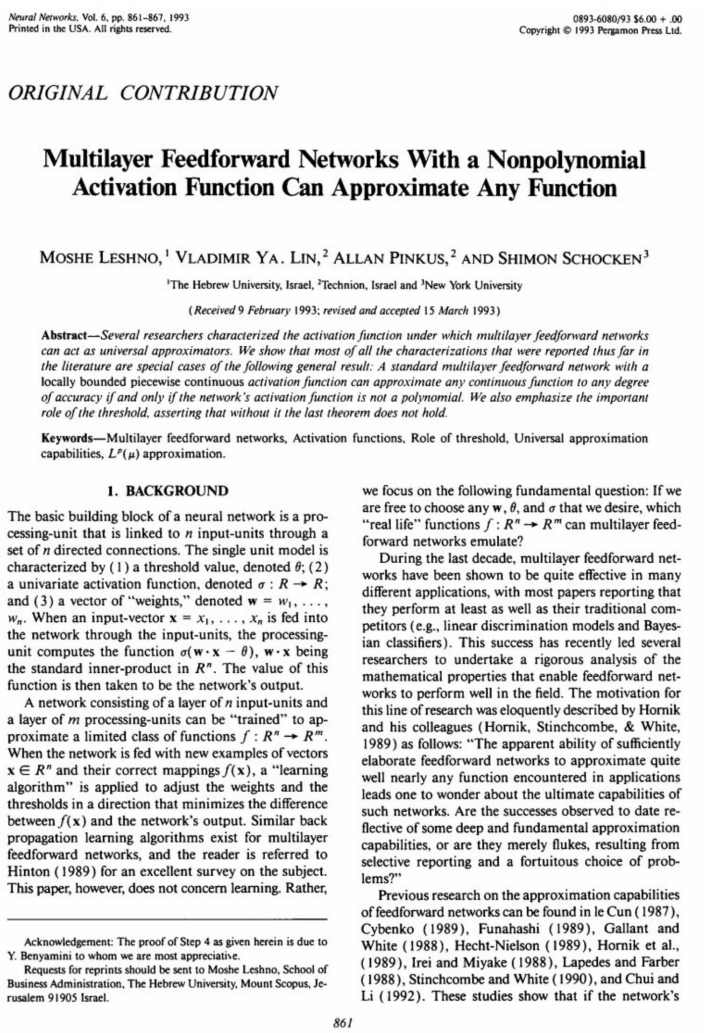
The results given here also provide a fundamental basis for rigorously establishing the ability of multilayer feedforward networks to learn (i.e., to estimate consistently) the connection strengths that achieve the approximations proven here to be possible. A statistical technique introduced by Grenander (1981) called the "method of sieves" is particularly well suited to this task. White (1988b) establishes such results for learning, using results of White and Woolridge (in press). For this it is necessary to utilize the concept of metric entropy (Kolmogorov & Tinomirov, 1961) for subsets of Σ^r possessing fixed numbers of hidden units. As a natural by-product of the metric entropy results one obtains quite specific rates at which the number of hidden units may grow as the number of training instances increases, while still ensuring the statistical property of consistency (i.e., avoiding overfitting).

An important related area for further research is

2. Hornik, K., Stinchcombe, M., & White, H. (1989).
Title: Multilayer feedforward networks are universal approximators
Source: Neural Networks
Contribution: 进一步推广了激活函数的类型，表明只要激活函数满足一定条件，神经可以作为通用函数逼近器。

source:
https://www.yumpu.com/en/document/view/29935370/approximation-by-superpositions-of-a-sigmoidal-function

激活函数：关键工作



Neural Networks, Vol. 6, pp. 861–867, 1993
Printed in the USA. All rights reserved.

0893-6080/93 $6.00 + .00

ORIGINAL CONTRIBUTION

Multilayer Feedforward Networks With a Nonpolynomial Activation Function Can Approximate Any Function

MOSHE LESHNO,[1] VLADIMIR YA. LIN,[2] ALLAN PINKUS,[2] AND SHIMON SCHOCKEN[3]

[1]The Hebrew University, Israel, [2]Technion, Israel and [3]New York University

(*Received* 9 *February* 1993; *revised and accepted* 15 *March* 1993)

Abstract—*Several researchers characterized the activation function under which multilayer feedforward networks can act as universal approximators. We show that most of all the characterizations that were reported thus far in the literature are special cases of the following general result: A standard multilayer feedforward network with a* locally bounded piecewise continuous *activation function can approximate any continuous function to any degree of accuracy if and only if the network's activation function is not a polynomial. We also emphasize the important role of the threshold, asserting that without it the last theorem does not hold.*

Keywords—Multilayer feedforward networks, Activation functions, Role of threshold, Universal approximation capabilities, $L^p(\mu)$ approximation.

1. BACKGROUND

The basic building block of a neural network is a processing-unit that is linked to n input-units through a set of n directed connections. The single unit model is characterized by (1) a threshold value, denoted θ; (2) a univariate activation function, denoted $\sigma : R \to R$; and (3) a vector of "weights," denoted $\mathbf{w} = w_1, \ldots, w_n$. When an input-vector $\mathbf{x} = x_1, \ldots, x_n$ is fed into the network through the input-units, the processing-unit computes the function $\sigma(\mathbf{w} \cdot \mathbf{x} - \theta)$, $\mathbf{w} \cdot \mathbf{x}$ being the standard inner-product in R^n. The value of this function is then taken to be the network's output.

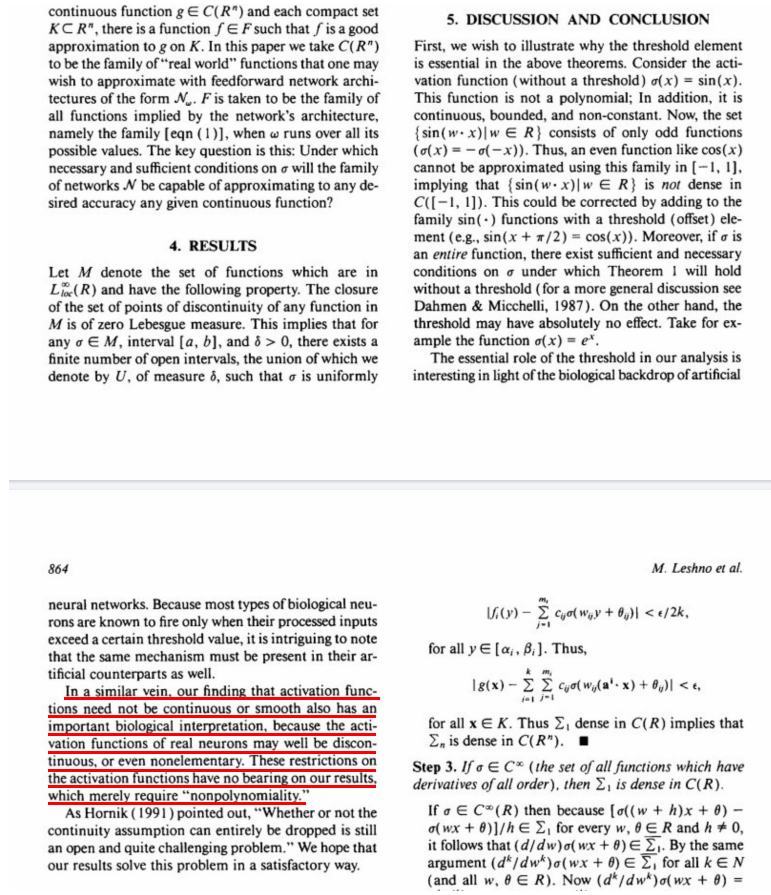
A network consisting of a layer of n input-units and a layer of m processing-units can be "trained" to approximate a limited class of functions $f : R^n \to R^m$. When the network is fed with new examples of vectors $\mathbf{x} \in R^n$ and their correct mappings $f(\mathbf{x})$, a "learning algorithm" is applied to adjust the weights and the thresholds in a direction that minimizes the difference between $f(\mathbf{x})$ and the network's output. Similar back propagation learning algorithms exist for multilayer feedforward networks, and the reader is referred to Hinton (1989) for an excellent survey on the subject. This paper, however, does not concern learning. Rather, we focus on the following fundamental question: If we are free to choose any $\mathbf{w}$, θ, and σ that we desire, which "real life" functions $f : R^n \to R^m$ can multilayer feedforward networks emulate?

During the last decade, multilayer feedforward networks have been shown to be quite effective in many different applications, with most papers reporting that they perform at least as well as their traditional competitors (e.g., linear discrimination models and Bayesian classifiers). This success has recently led several researchers to undertake a rigorous analysis of the mathematical properties that enable feedforward networks to perform well in the field. The motivation for this line of research was eloquently described by Hornik and his colleagues (Hornik, Stinchcombe, & White, 1989) as follows: "The apparent ability of sufficiently elaborate feedforward networks to approximate quite well nearly any function encountered in applications leads one to wonder about the ultimate capabilities of such networks. Are the successes observed to date reflective of some deep and fundamental approximation capabilities, or are they merely flukes, resulting from selective reporting and a fortuitous choice of problems?"

Previous research on the approximation capabilities of feedforward networks can be found in le Cun (1987), Cybenko (1989), Funahashi (1989), Gallant and White (1988), Hecht-Nielson (1989), Hornik et al., (1989), Irei and Miyake (1988), Lapedes and Farber (1988), Stinchcombe and White (1990), and Chui and Li (1992). These studies show that if the network's

Acknowledgement: The proof of Step 4 as given herein is due to Y. Benyamini to whom we are most appreciative.

Requests for reprints should be sent to Moshe Leshno, School of Business Administration, The Hebrew University, Mount Scopus, Jerusalem 91905 Israel.



continuous function $g \in C(R^n)$ and each compact set $K \subset R^n$, there is a function $f \in F$ such that f is a good approximation to g on K. In this paper we take $C(R^n)$ to be the family of "real world" functions that one may wish to approximate with feedforward network architectures of the form $\mathcal{N}_\sigma$. F is taken to be the family of all functions implied by the network's architecture, namely the family [eqn (1)], when ω runs over all its possible values. The key question is this: Under which necessary and sufficient conditions on σ will the family of networks $\mathcal{N}$ be capable of approximating to any desired accuracy any given continuous function?

4. RESULTS

Let M denote the set of functions which are in $L^\infty_{loc}(R)$ and have the following property. The closure of the set of points of discontinuity of any function in M is of zero Lebesgue measure. This implies that for any $\sigma \in M$, interval $[a, b]$, and $\delta > 0$, there exists a finite number of open intervals, the union of which we denote by U, of measure δ, such that σ is uniformly

5. DISCUSSION AND CONCLUSION

First, we wish to illustrate why the threshold element is essential in the above theorems. Consider the activation function (without a threshold) $\sigma(x) = \sin(x)$. This function is not a polynomial; In addition, it is continuous, bounded, and non-constant. Now, the set $\{\sin(w \cdot x) | w \in R\}$ consists of only odd functions ($\sigma(x) = -\sigma(-x)$). Thus, an even function like $\cos(x)$ cannot be approximated using this family in $[-1, 1]$, implying that $\{\sin(w \cdot x) | w \in R\}$ is *not* dense in $C([-1, 1])$. This could be corrected by adding to the family $\sin(\cdot)$ functions with a threshold (offset) element (e.g., $\sin(x + \pi/2) = \cos(x)$). Moreover, if σ is an *entire* function, there exist sufficient and necessary conditions on σ under which Theorem 1 will hold without a threshold (for a more general discussion see Dahmen & Micchelli, 1987). On the other hand, the threshold may have absolutely no effect. Take for example the function $\sigma(x) = e^x$.

The essential role of the threshold in our analysis is interesting in light of the biological backdrop of artificial

neural networks. Because most types of biological neurons are known to fire only when their processed inputs exceed a certain threshold value, it is intriguing to note that the same mechanism must be present in their artificial counterparts as well.

In a similar vein, our finding that activation functions need not be continuous or smooth also has an important biological interpretation, because the activation functions of real neurons may well be discontinuous, or even nonelementary. These restrictions on the activation functions have no bearing on our results, which merely require "nonpolynomiality."

As Hornik (1991) pointed out, "Whether or not the continuity assumption can entirely be dropped is still an open and quite challenging problem." We hope that our results solve this problem in a satisfactory way.

$$|f_i(y) - \sum_{j=1}^{m_i} c_{ij}\sigma(w_{ij}y + \theta_{ij})| < \epsilon/2k,$$

for all $y \in [\alpha_i, \beta_i]$. Thus,

$$|g(\mathbf{x}) - \sum_{i=1}^{k}\sum_{j=1}^{m_i} c_{ij}\sigma(w_{ij}(\mathbf{a}^i \cdot \mathbf{x}) + \theta_{ij})| < \epsilon,$$

for all $\mathbf{x} \in K$. Thus Σ_1 dense in $C(R)$ implies that Σ_n is dense in $C(R^n)$. ■

Step 3. *If $\sigma \in C^\infty$ (the set of all functions which have derivatives of all order), then Σ_1 is dense in $C(R)$.*

If $\sigma \in C^\infty(R)$ then because $[\sigma((w + h)x + \theta) - \sigma(wx + \theta)]/h \in \Sigma_1$ for every $w, \theta \in R$ and $h \neq 0$, it follows that $(d/dw)\sigma(wx + \theta) \in \overline{\Sigma_1}$. By the same argument $(d^k/dw^k)\sigma(wx + \theta) \in \overline{\Sigma_1}$ for all $k \in N$ (and all $w, \theta \in R$). Now $(d^k/dw^k)\sigma(wx + \theta) =$

3. Leshno, M., Lin, V. Y., Pinkus, A., & Schocken, S. (1993).
Title: Multilayer feedforward networks with a nonpolynomial activation function can approximate any function
Source: Neural Networks
Contribution: 论证了只要激活函数不是多项式函数，神经网络就能逼近任意连续函数，这说明了非线性激活函数的重要性。

source: https://www.academia.edu/19189632/Multilayer_feedforward_networks_with_a_nonpolynomial_activation_function_can_approximate_any_function

激活函数：常见种类

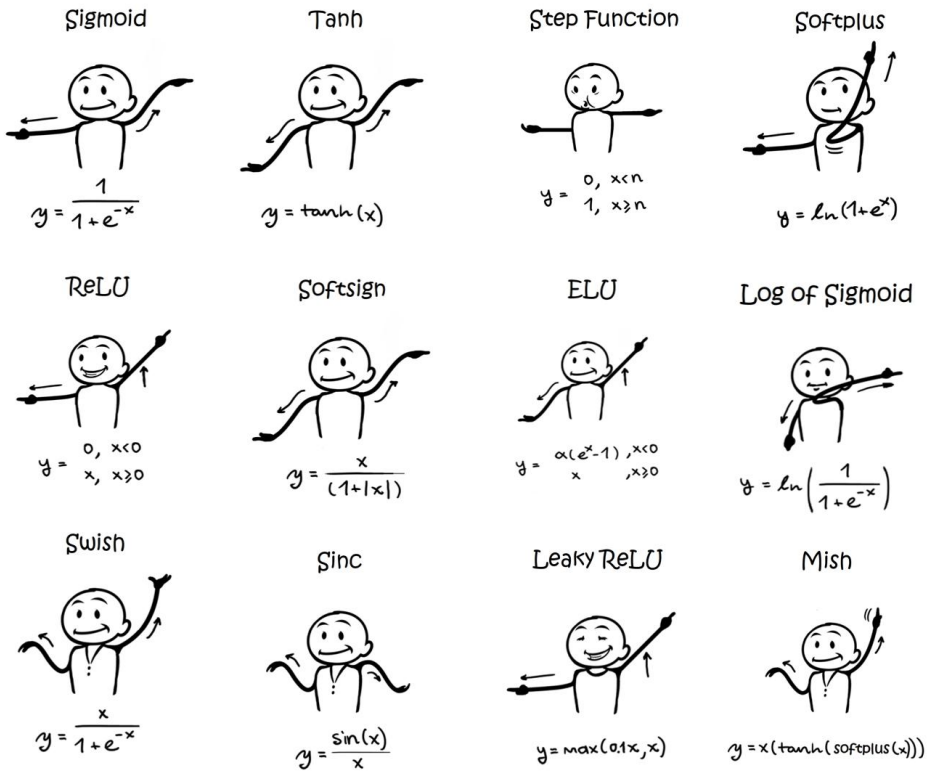
- 激活函数常见种类包括 Sigmoid、Tanh、ReLU、Leaky ReLU 等，分别适用于输出层和隐藏层不同需求。

激活函数	数学表达式	输出范围	优点	缺点	应用场景
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	(0, 1)	输出可解释为概率，适合二分类	梯度消失，输出非中心化	二分类任务的输出层
Tanh	$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	(-1, 1)	输出中心化，训练更稳定	梯度消失问题依然存在	隐藏层，优于 Sigmoid
ReLU	$\mathrm{ReLU}(x) = \max(0, x)$	[0, +∞)	计算高效，缓解梯度消失	神经元可能“死亡”	主流隐藏层激活函数
Leaky ReLU	$\mathrm{LeakyReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases}$	(-∞, +∞)	解决 ReLU 死亡问题	引入小负斜率，训练稍慢	ReLU 的改进版
Softmax	$\mathrm{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$	(0, 1) 总和为1	多分类输出概率化	仅适用于输出层	多分类任务输出层

它们主要作用是引入非线性、解决梯度问题、提升模型表达力。

▶ 激活函数：选择策略

➢ 选择合适的激活函数，是构建高效神经网络模型的关键一步，不同任务和层次有不同的最佳选择。



输出层怎么选择？

- Sigmoid：用于二分类输出概率。
- Softmax：用于多分类输出各类概率分布。
- 无激活/线性：用于回归任务。

隐藏层怎么选择？

- ReLU：首选！收敛快，效果好。
- Leaky ReLU：ReLU改进版，避免神经元“死亡”。
- Tanh：适合较浅的网络，对称、中心化。
- Sigmoid：已较少使用，易造成梯度消失。

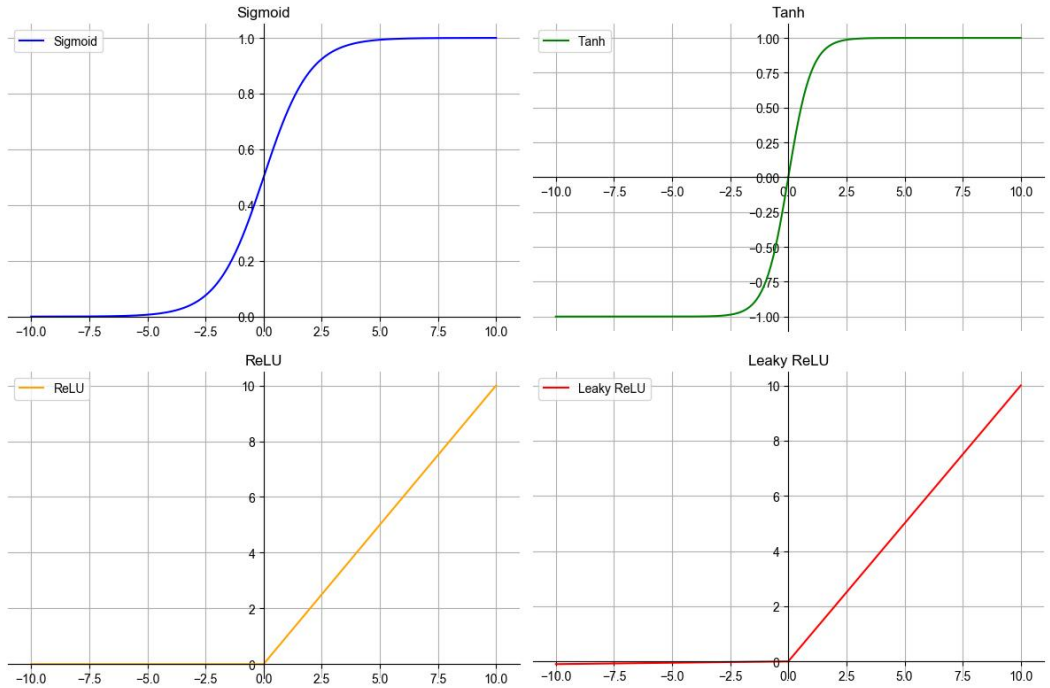
建议：深度网络优先用 ReLU 或其改进版本；试验多种函数，结合任务和数据表现评估；输出层根据任务目标明确匹配激活函数类型

激活函数：如何实现

- 实现非常简单，只需要定义一个激活函数即可。

```python
# 无 numpy 实现激活函数

import math
import matplotlib.pyplot as plt

# 激活函数（单个数值输入）
def sigmoid(x):
    return 1 / (1 + math.exp(-x))

def tanh(x):
    ex = math.exp(x)
    enx = math.exp(-x)
    return (ex - enx) / (ex + enx)

def relu(x):
    return x if x > 0 else 0

def leaky_relu(x, alpha=0.01):
    return x if x > 0 else alpha * x

# 生成 x 值（-10 到 10 之间）
x_vals = [i * 0.05 for i in range(-200, 201)]  # 共 401 个点

# 分别计算各个激活函数对应的 y 值
y_funcs = {
    "Sigmoid": [sigmoid(x) for x in x_vals],
    "Tanh": [tanh(x) for x in x_vals],
    "ReLU": [relu(x) for x in x_vals],
    "Leaky ReLU": [leaky_relu(x) for x in x_vals]
}

# 绘图
plt.figure(figsize=(12, 8))
colors = ['blue', 'green', 'orange', 'red']

for i, (name, y_vals) in enumerate(y_funcs.items(), start=1):
    ax = plt.subplot(2, 2, i)
    ax.plot(x_vals, y_vals, color=colors[i-1], label=name)
    ax.set_title(name)
    ax.grid(True)

    # 设置中心坐标轴
    ax.spines['left'].set_position('zero')
    ax.spines['bottom'].set_position('zero')
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.yaxis.set_ticks_position('left')

    ax.legend(loc='upper left')

plt.tight_layout()
plt.show()
```

```python
# numpy 实现激活函数

import numpy as np
import matplotlib.pyplot as plt

# 激活函数定义
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def tanh(x):
    return (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))
    # return np.tanh(x)

def relu(x):
    return np.maximum(0, x)

def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)

# 生成xy数据
x = np.linspace(-10, 10, 400)
y_funcs = {
    "Sigmoid": sigmoid(x),
    "Tanh": tanh(x),
    "ReLU": relu(x),
    "Leaky ReLU": leaky_relu(x)
}

# 绘图
plt.figure(figsize=(12, 8))
colors = ['blue', 'green', 'orange', 'red']

for i, (name, y) in enumerate(y_funcs.items(), start=1):
    ax = plt.subplot(2, 2, i)
    ax.plot(x, y, color=colors[i-1], label=name)
    ax.set_title(name)
    ax.grid(True)

    # 中心坐标轴
    ax.spines['left'].set_position('zero')   # y轴移到x=0
    ax.spines['bottom'].set_position('zero') # x轴移到y=0
    ax.spines['right'].set_color('none')     # 隐藏右边框
    ax.spines['top'].set_color('none')       # 隐藏上边框
    ax.xaxis.set_ticks_position('bottom')
    ax.yaxis.set_ticks_position('left')

    ax.legend(loc='upper left')

plt.tight_layout()
plt.show()
```



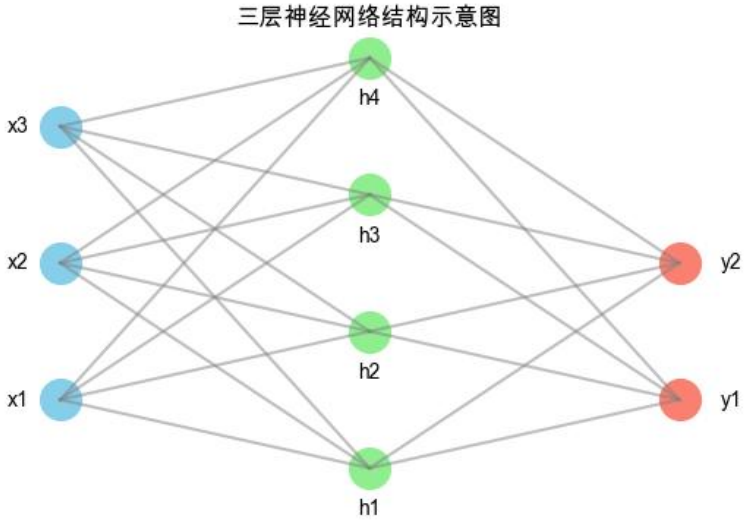
- 使用 NumPy 编写激活函数效率高、支持向量化计算，而不使用 NumPy 则更接近底层逻辑、适合逐步理解。

目录章节

CONTENTS

▶ 前向传播：实现

- 神经网络的前向传播是指数据从输入层依次经过每一层的计算直至输出层的过程。下面是前向传播的完整实现步骤，适用于两层（一个隐藏层 + 输出层）神经网络：
- 目标网络结构实现：一个输入层（例如：4维向量），一个隐藏层（ReLU 激活），一个输出层（Sigmoid 激活），所有层之间通过**线性变换 + 激活函数**连接。



网络结构说明：

该网络是一个只包含一个隐藏层的前馈神经网络（Feedforward Neural Network），结构简单但足以展示前向传播的基本流程，包括线性变换、激活函数处理和输出预测等关键步骤。

- 输入向量：$x \in \mathbb{R}^n$
- 第一层权重：$W_1 \in \mathbb{R}^{h \times n}$，偏置：$b_1 \in \mathbb{R}^h$
- 第二层权重：$W_2 \in \mathbb{R}^{m \times h}$，偏置：$b_2 \in \mathbb{R}^m$
- 激活函数：
  - 第一层使用 **ReLU**
  - 输出层使用 **Sigmoid**

注意：在实现前向传播时，必须确保前一层输出的维度与后一层权重矩阵的输入维度匹配，否则将导致矩阵乘法报错。建议在每一层操作前先确认维度一致性。

前向传播公式：

第 1 层（隐藏层）：

$$z_1 = W_1 x + b_1$$

$$a_1 = \text{ReLU}(z_1)$$

第 2 层（输出层）：

$$z_2 = W_2 a_1 + b_2$$

$$a_2 = \sigma(z_2)$$

激活函数定义：

$$\text{ReLU}(z) = \max(0, z)$$

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

关键：1）前一层的输出维度 == 后一层权重矩阵，确保维度能对上；2）每一层使用合适的激活函数。

▶ 前向传播：numpy实现

```python
import numpy as np
import matplotlib.pyplot as plt

# matplotlib 设置
plt.rcParams['font.family'] = 'Arial Unicode MS' # 设置字体，确保中文显示正常

# 设置 NumPy 随机种子，保证可重复
np.random.seed(42)

# 输入数据（3个特征）
X = np.array([1, 2, 3])

# 初始化权重和偏置
w1 = np.random.randn(4, 3)  # 第一层权重（4个神经元 * 3个输入）
b1 = np.random.randn(4)     # 第一层偏置（4个神经元）
w2 = np.random.randn(2, 4)  # 第二层权重（2个神经元 * 4个隐藏层输出）
b2 = np.random.randn(2)     # 第二层偏置（2个神经元）

# 激活函数：ReLU 和 Sigmoid
def relu(x):
    return np.maximum(0, x)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# === 前向传播 ===
# 第一层：加权求和 + 激活
z1 = np.dot(w1, X) + b1 # z1 = w1 * X + b1
a1 = relu(z1) # a1 = ReLU(z1)

# 第二层（输出层）：加权求和 + 激活
z2 = np.dot(w2, a1) + b2 # z2 = w2 * a1 + b2
a2 = sigmoid(z2) # a2 = Sigmoid(z2)

# === 输出结构化信息 ===
print("\n===== 第1层计算 =====")
print("加权和 z1 =", z1)
print("激活输出 a1 (ReLU) =", a1)

print("\n===== 第2层（输出层）计算 =====")
print("加权和 z2 =", z2)
print("激活输出 a2 (Sigmoid) =", a2)

# 结构化模拟图
print("\n=====\U0001f9e0 神经网络结构: =====")
print("输入层 (3个特征):", X)
print("↓")
print("隐藏层 (4个神经元):")
for i, (z, a) in enumerate(zip(z1, a1)):
    print(f"  神经元{i+1}: z={z:.4f}, a=ReLU(z)={a:.4f}")
print("↓")
print("输出层 (2个神经元):")
for i, (z, a) in enumerate(zip(z2, a2)):
    print(f"  输出{i+1}: z={z:.4f}, a=Sigmoid(z)={a:.4f}")
```

```
===== 第1层计算 =====
加权和 z1 = [ 2.40521344 -1.56096801 -0.01925872 -2.34375213]
激活输出 a1 (ReLU) = [2.40521344 0.         0.         0.        ]

===== 第2层（输出层）计算 =====
加权和 z2 = [-2.98045774  3.6361207 ]
激活输出 a2 (Sigmoid) = [0.04831658 0.97432234]

=====🧠 神经网络结构: =====
输入层 (3个特征): [1 2 3]
↓
隐藏层 (4个神经元):
  神经元1: z=2.4052, a=ReLU(z)=2.4052
  神经元2: z=-1.5610, a=ReLU(z)=0.0000
  神经元3: z=-0.0193, a=ReLU(z)=0.0000
  神经元4: z=-2.3438, a=ReLU(z)=0.0000
↓
输出层 (2个神经元):
  输出1: z=-2.9805, a=Sigmoid(z)=0.0483
  输出2: z=3.6361, a=Sigmoid(z)=0.9743
```

前向传播：pytorch实现

```python
import torch
import torch.nn.functional as F

# 设置随机种子，确保可重复性
torch.manual_seed(42)

# 输入数据（示例10维）
X = torch.arange(10, dtype=torch.float32)

# 打印输入数据
print("输入数据 X:\n", X)

# 第一层权重和偏置
W1 = torch.randn(4, 10)   # 4个隐藏单元，输入10维
b1 = torch.randn(4)

# 打印W1和b1
print("第一层权重 W1:\n", W1)
print("第一层偏置 b1:\n", b1)

# 第二层权重和偏置
W2 = torch.randn(2, 4)    # 输出2维
b2 = torch.randn(2)

# 打印W2和b2
print("第二层权重 W2:\n", W2)
print("第二层偏置 b2:\n", b2)

# 隐藏层计算：线性变换 + ReLU 激活
z1 = torch.matmul(W1, X) + b1
a1 = F.relu(z1)

# 输出层计算：线性变换 + Sigmoid 激活
z2 = torch.matmul(W2, a1) + b2
a2 = torch.sigmoid(z2)

# === 输出结构化信息 ===
print("\n===== 第1层计算 =====")
print("加权和 z1 =", z1)
print("激活输出 a1 (ReLU) =", a1)

print("\n===== 第2层（输出层）计算 =====")
print("加权和 z2 =", z2)
print("激活输出 a2 (Sigmoid) =", a2)

# 结构化模拟图
print("\n=====\U0001f9e0 神经网络结构：=====")
print("输入层 (3个特征):", X)
print("↓")
print("隐藏层 (4个神经元):")
for i, (z, a) in enumerate(zip(z1, a1)):
    print(f"  神经元{i+1}: z={z:.4f}, a=ReLU(z)={a:.4f}")
print("↓")
print("输出层 (2个神经元):")
for i, (z, a) in enumerate(zip(z2, a2)):
    print(f"  输出{i+1}: z={z:.4f}, a=Sigmoid(z)={a:.4f}")
```

```
输入数据 X:
 tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
第一层权重 W1:
 tensor([[ 1.9269,  1.4873,  0.9007, -2.1055,  0.6784, -1.2345, -0.0431, -1.6047,
          -0.7521,  1.6487],
         [-0.3925, -1.4036, -0.7279, -0.5594, -0.7688,  0.7624,  1.6423, -0.1596,
          -0.4974,  0.4396],
         [-0.7581,  1.0783,  0.8008,  1.6806,  0.0349,  0.3211,  1.5736, -0.8455,
           1.3123,  0.6872],
         [-1.0892, -0.3553, -1.4181,  0.8963,  0.0499,  2.2667,  1.1790, -0.4345,
          -1.3864, -1.2862]])
第一层偏置 b1:
 tensor([-0.0499,  0.5263, -0.0085,  0.7291])
第二层权重 W2:
 tensor([[ 0.1331,  0.8640, -1.0157, -0.8887],
         [ 0.1498, -0.2089, -0.3870,  0.9912]])
第二层偏置 b2:
 tensor([ 0.4679, -0.2049])

===== 第1层计算 =====
加权和 z1 = tensor([-9.2065,  5.4394, 29.6647, -6.8747])
激活输出 a1 (ReLU) = tensor([ 0.0000,  5.4394, 29.6647,  0.0000])

===== 第2层（输出层）计算 =====
加权和 z2 = tensor([-24.9622, -12.8220])
激活输出 a2 (Sigmoid) = tensor([1.4422e-11, 2.7006e-06])

=====\U0001f9e0 神经网络结构：=====
输入层 (3个特征): tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
↓
隐藏层 (4个神经元):
  神经元1: z=-9.2065, a=ReLU(z)=0.0000
  神经元2: z=5.4394, a=ReLU(z)=5.4394
  神经元3: z=29.6647, a=ReLU(z)=29.6647
  神经元4: z=-6.8747, a=ReLU(z)=0.0000
↓
输出层 (2个神经元):
  输出1: z=-24.9622, a=Sigmoid(z)=0.0000
  输出2: z=-12.8220, a=Sigmoid(z)=0.0000
```

▶ 前向传播：pytorch实现(nn.Sequential)

➢ nn.Sequential 是 PyTorch 中用于按顺序堆叠网络层的容器。你只需按顺序写出每一层，Sequential 会自动把前一层的输出作为后一层的输入。

```python
import torch.nn as nn
from collections import OrderedDict


model = nn.Sequential(
    nn.Linear(4, 8),     # 输入层到隐藏层
    nn.ReLU(), # ReLU 激活函数
    nn.Linear(8, 1),     # 隐藏层到输出层
    nn.Sigmoid() # Sigmoid 激活函数
)

model_complex = nn.Sequential(OrderedDict([
    ('fc1', nn.Linear(4, 8)),
    ('relu', nn.ReLU()),
    ('fc2', nn.Linear(8, 1)),
    ('sigmoid', nn.Sigmoid())
]))

# 模型输出
print("\n===== 模型输出 =====")
print("模型结构:", model)
print("复杂模型结构:", model_complex)

output = model(torch.randn(1, 4))  # 输入一个随机向量
output_complex = model_complex(torch.randn(1, 4))  # 输入一个随机向量


print("\n模型输出:", output)
print("\n复杂模型输出:", output_complex)
```

```
===== 模型输出 =====
模型结构: Sequential(
  (0): Linear(in_features=4, out_features=8, bias=True)
  (1): ReLU()
  (2): Linear(in_features=8, out_features=1, bias=True)
  (3): Sigmoid()
)
复杂模型结构: Sequential(
  (fc1): Linear(in_features=4, out_features=8, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=8, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)

模型输出: tensor([[0.3578]], grad_fn=<SigmoidBackward0>)

复杂模型输出: tensor([[0.6070]], grad_fn=<SigmoidBackward0>)
```

目录章节

CONTENTS

总结

- 神经网络是什么？模拟人脑神经元结构构建的函数逼近器，由**层**组成：输入层 → 隐藏层（多个）→ 输出层，每一层通过 **权重 + 偏置 + 激活函数** 进行连接与变换。

网络内部结构

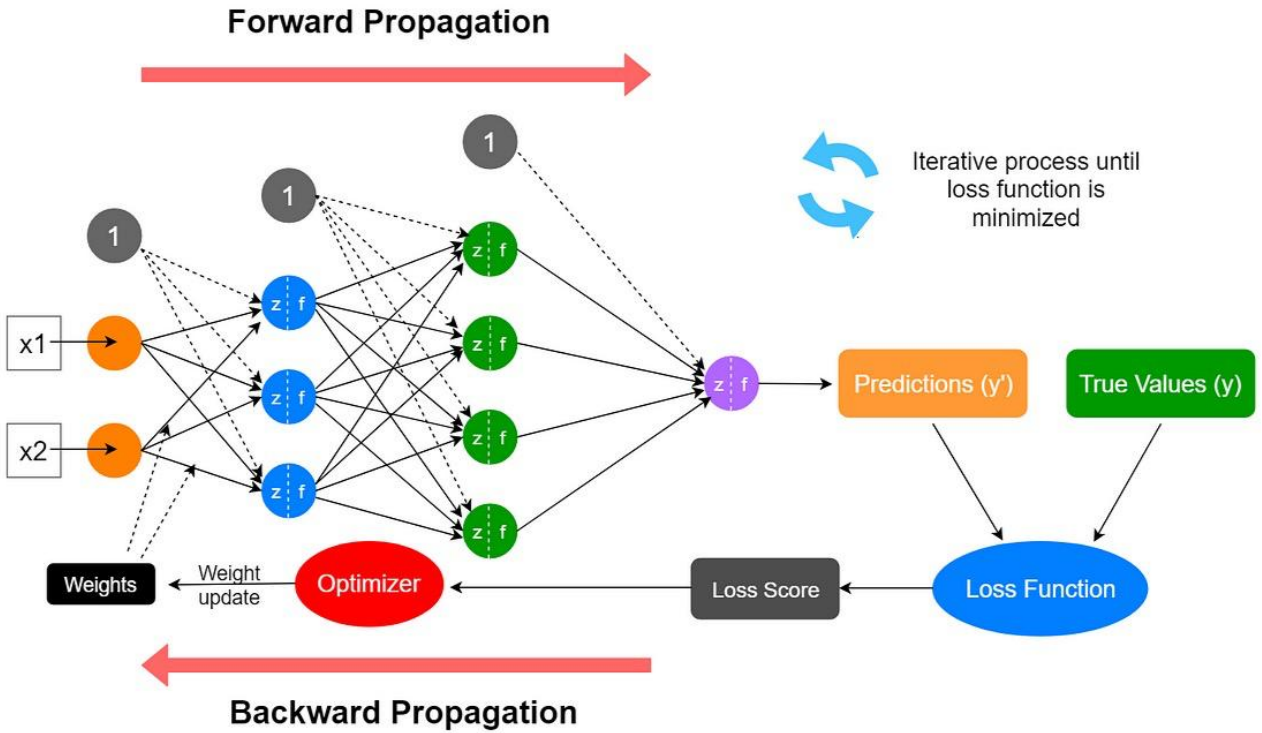
组成部分	作用说明
W（权重矩阵）	决定特征的重要性
b（偏置）	提供可调节的输出平移
x（输入/前一层输出）	层之间的数据传递
f（激活函数）	非线性变换，增强表达力

常见激活函数

激活函数	表达式	特点
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	输出范围 (0,1)，适合二分类
Tanh	$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	范围 (-1,1)，零中心化
ReLU	$\mathrm{ReLU}(x) = \max(0, x)$	快速计算，解决梯度消失
Leaky ReLU	$\mathrm{LeakyReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases}$	避免 ReLU 死神经元问题

总结

- 前向传播机制与实现：输入经过每层线性变换加上偏置，再通过激活函数逐层传递，最终输出预测结果。



前向传播就是从输入出发，经过每一层的加权、偏置和激活函数，最终输出预测结果。
问题：怎么更新权重呢？——反向传播：计算误差对参数的影响，从而更新权重。

感谢聆听

Personal Website: https://www.miaopeng.info/

Email: miaopeng@stu.scu.edu.cn

Github: https://github.com/MMeowhite

Youtube: https://www.youtube.com/@pengmiao-bmm