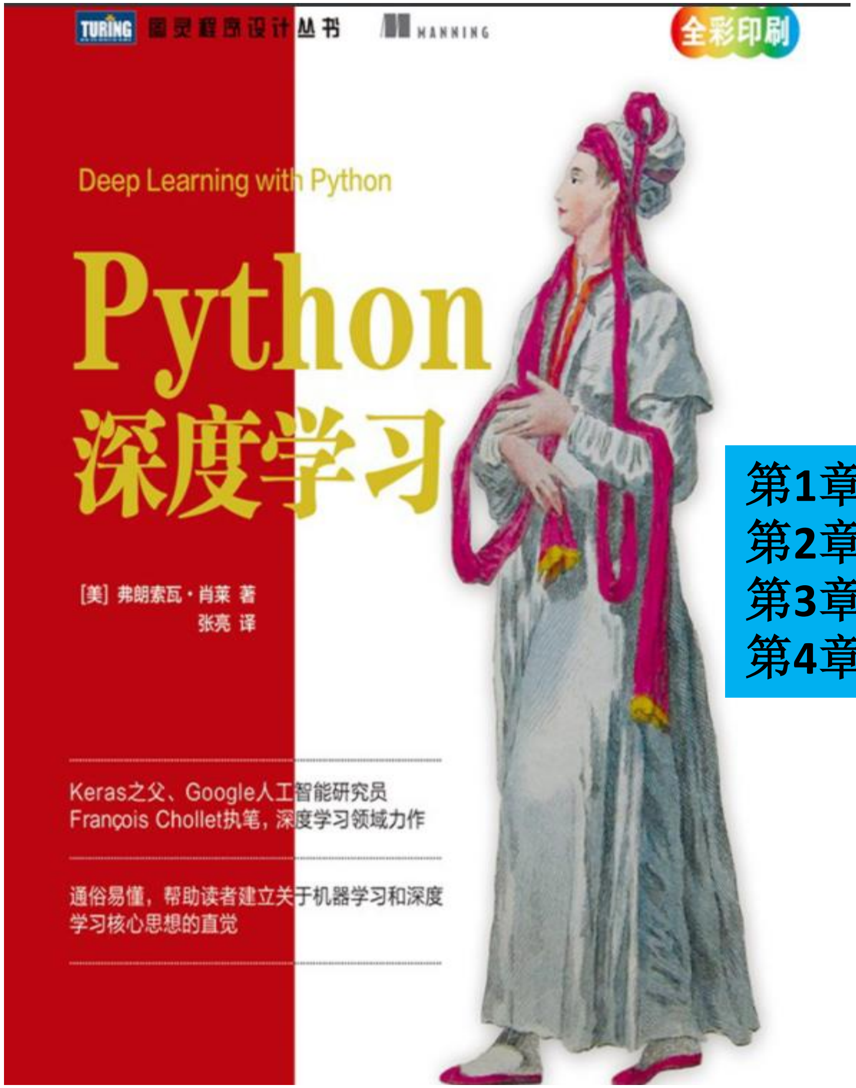
深度学习基础

参考资料



第1章 什么是深度学习
第2章 神经网络的数学基础
第3章 神经网络入门
第4章 机器学习基础

教材代码： https://github.com/fchollet/deep-learning-with-python-notebooks

本章内容

1 什么是深度学习

2 神经网络的数学基础

3 神经网络入门

4 神经网络的通用工作流程

1 什么是深度学习？

- 基本概念的定义
- 机器学习发展的时间线
- 深度学习日益流行的关键因素及其未来潜力

1 什么是深度学习？
  - 人工智能、机器学习与深度学习
  - 发展历程
  - 为什么是深度学习，为什么是现在
    - ABC

人工智能、机器学习与深度学习

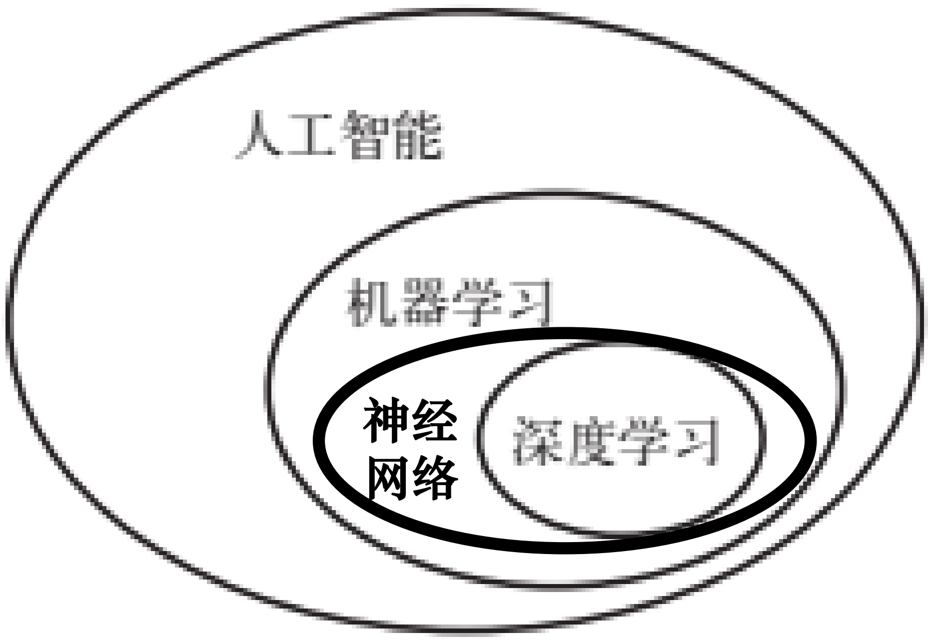


图 1-1 人工智能、机器学习与深度学习

人工智能是一个综合性的领域。

- 符号主义：用符号逻辑表达一些东西，目前并不是十分流行。
- 行为主义：行为能力不断增强，并没有很好的实现。
- 连接主义：最主要的智能技术，比如人工神经网络。

机器学习

1. 机器学习：通过观察数据自动学会数据处理规则。

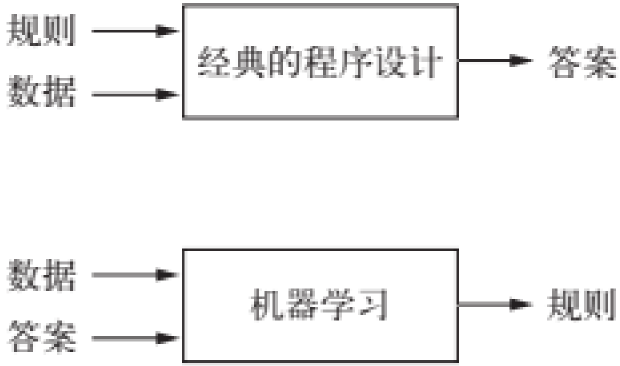


图 1-2　机器学习：一种新的编程范式

2. 机器学习中的**学习**指的是，寻找更好**数据表示**的**自动搜索**过程。
3. 计算机程序利用**经验E**学习**任务T**，性能是P，如果针对任务T的**性能P**随着经验E不断增长，则称为机器学习。——汤姆·米切尔，1997

神经网络

- 从连续的层中进行学习
- 深度：模型中的层数
- 结构特点：逐层堆叠

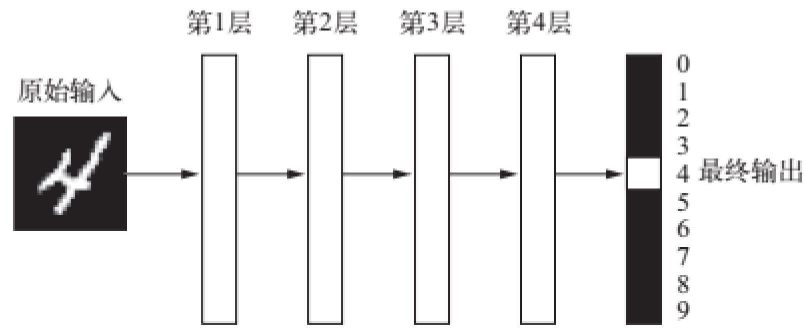
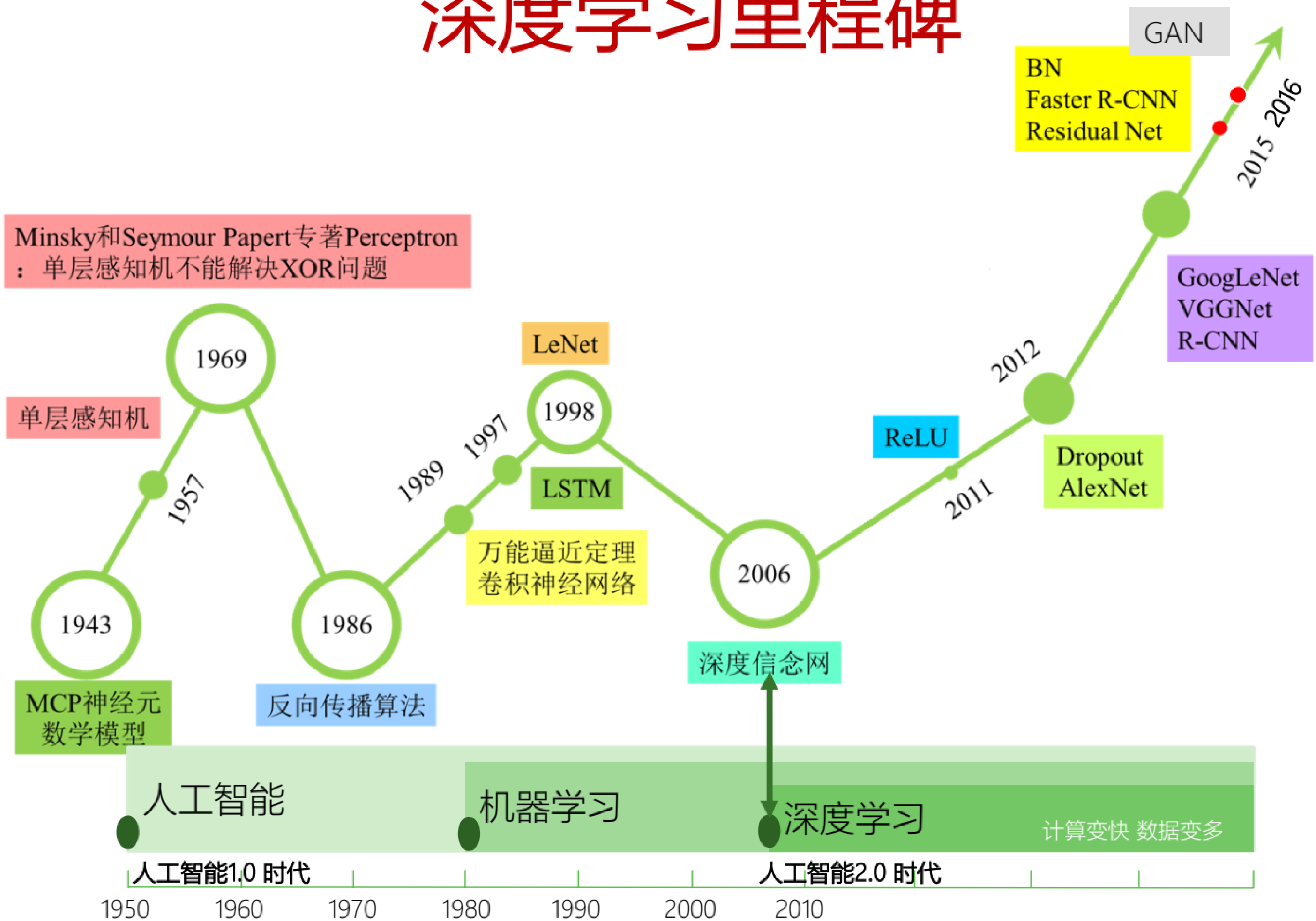


图 1-5　用于数字分类的深度神经网络

- 神经网络 VS. 机器学习
  - 神经网络：将特征工程完全自动化
    - 第一，通过渐进的、逐层的方式形成越来越复杂的表示；
    - 第二，对中间这些渐进的表示共同进行学习
  - 机器学习：
    - 也被称为浅层学习（shallow learning），往往是仅仅学习一两层的数据表示。
    - 需手动为数据设计好的表示层。

深度学习里程碑

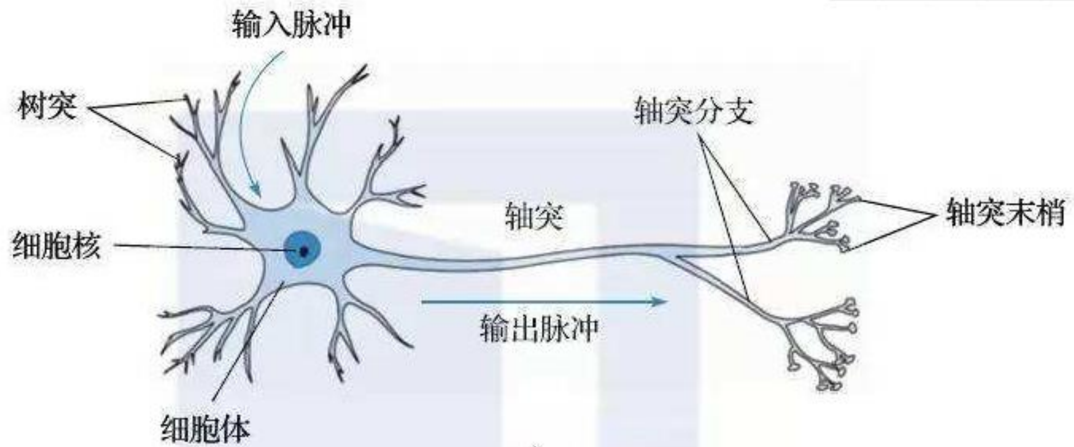


引自Prof. Meina Kan, PHD Student Xin Liu and Shuzhe Wu

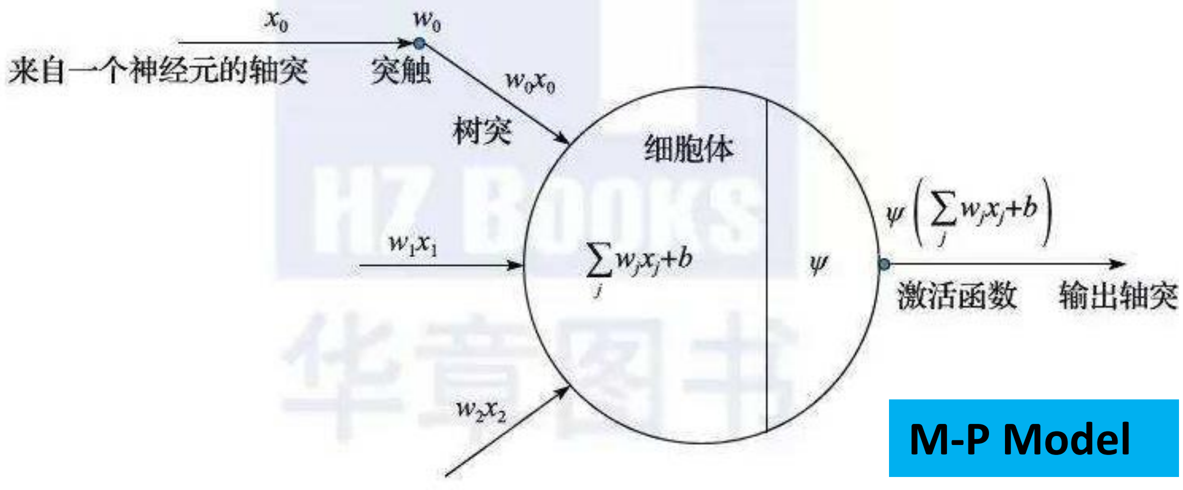
- 1943: M-P model (McCulloch-Pitts Model)
- 1958: Perceptron (linear model)
- 1969: Perceptron has limitation
- 1980s: Multi-layer perceptron
  – Do not have significant difference from DNN today
- 1986: Backpropagation
  – Usually more than 3 hidden layers is not helpful
- 1989: 1 hidden layer is “good enough”, why deep?
- 2006: “Deep Learning”; RBM(Restricted Boltzmann machine) initialization
- 2009: GPU
- 2011: Start to be popular in speech recognition
- 2012: win ILSVRC image competition
- 2015.2: Image recognition surpassing human-level performance
- 2016.3: Alpha GO beats Lee Sedol
- 2016.10: Speech recognition system as good as humans

浅层神经网络 -> 深层神经网络（深度学习）

人工神经元 VS. 生物神经元



a）



b）

M-P Model

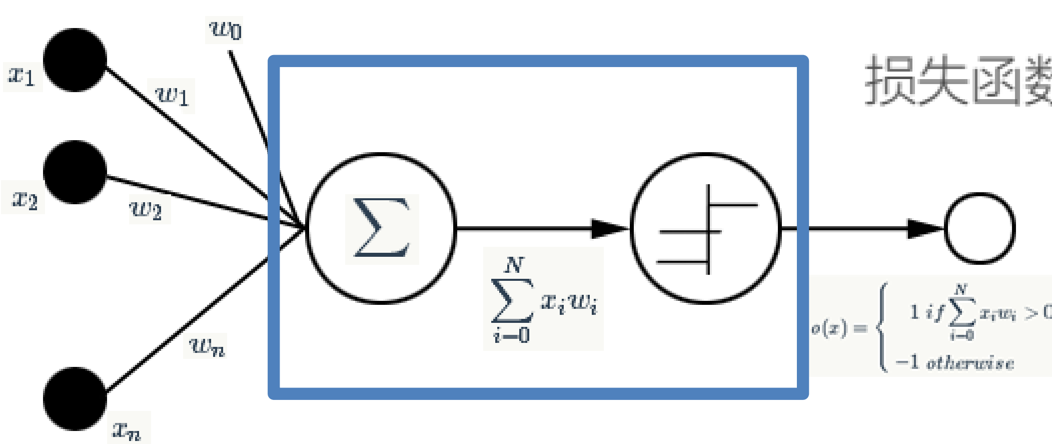
感知机（Perceptron）模型

- 目标：找到一个（$\boldsymbol{w}$, b），将线性可分的数据集中的所有样本点都正确地分为两类。
- 问题设定：寻找（$\boldsymbol{w}$, b），使得损失函数极小化的最优问题。

$$f(x) = sign(\boldsymbol{w}^T\boldsymbol{x} + b)$$

$$sign(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

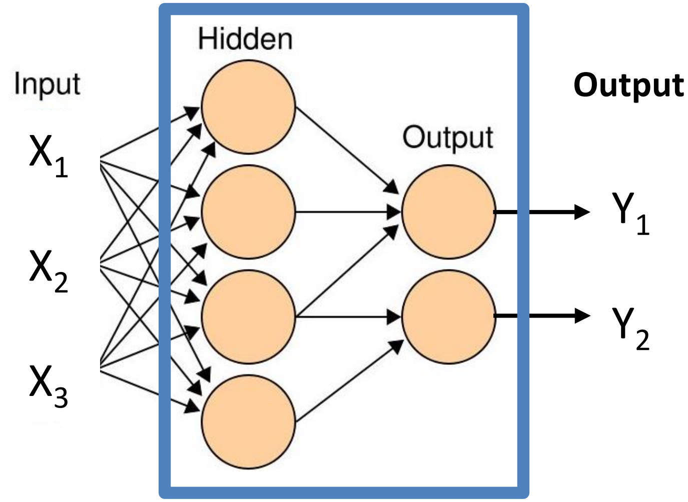
损失函数 $L(\boldsymbol{w}, b) = -\sum_{\boldsymbol{x}_j \in M} y_j(\boldsymbol{w}^T\boldsymbol{x}_j + b)$



不能解决异或问题！

多层感知机（ Multi-layer perceptron ）

- 是一种全连接的两层神经网络
- 相较于感知机，有三点改进：
  1. 加入了隐藏层，增强模型的表达能力，但同时也增加了模型的复杂度
  2. 输出层的神经元也可以不止一个输出；
  3. 对激活函数做扩展，增强模型的表达能力。
     - 感知机的激活函数是sign(z)，虽然简单但是处理能力有限。



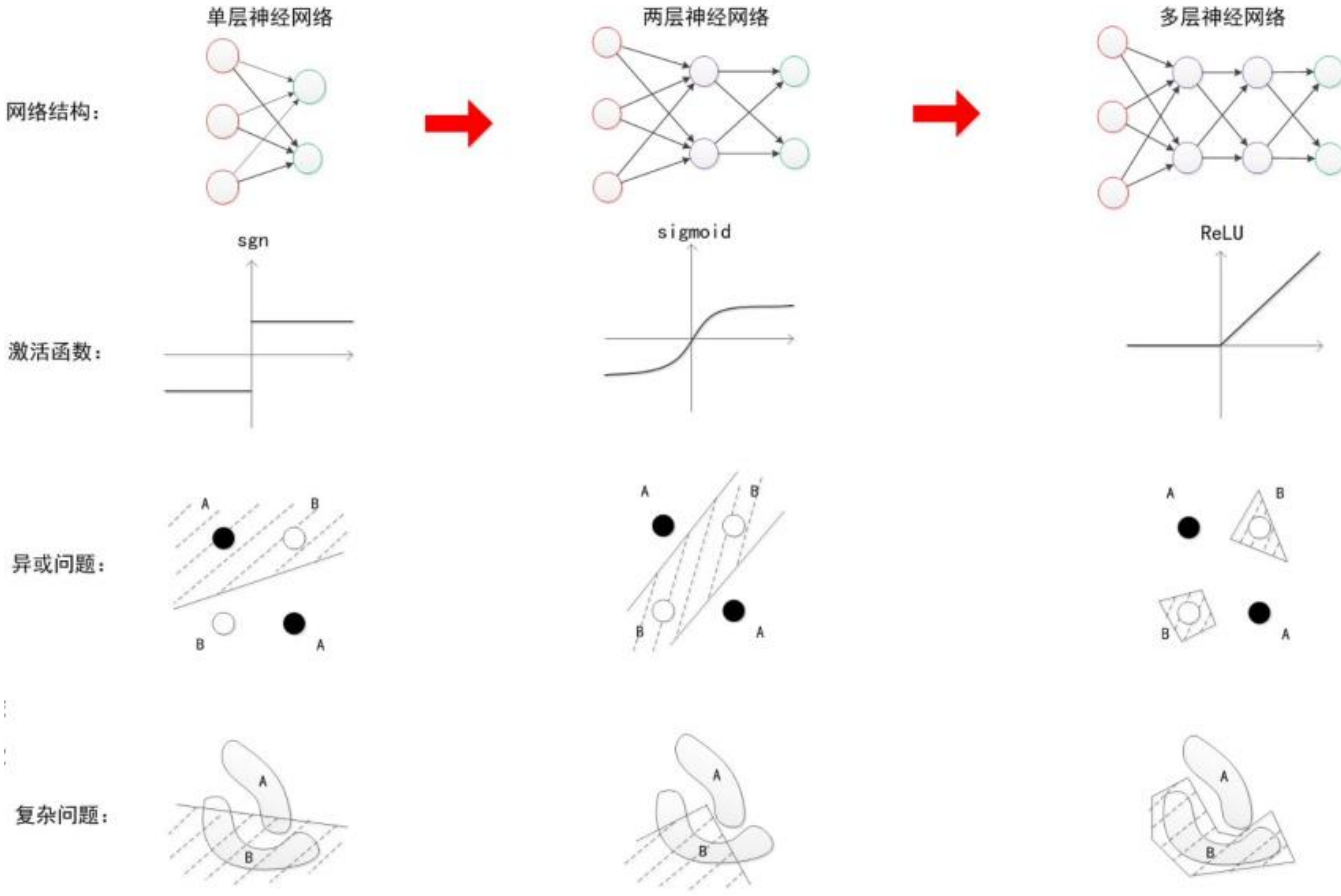
总结：浅层神经网络

- 优势：
  - 需要的数据量小
  - 训练快
- 局限性：对复杂问题的表示能力有限，泛化性弱。

- Why not go deeper?
  - Kurt Hornik证明：理论上两层神经网络足以拟合任何函数。
  - 数据量和计算能力的约束

多层神经网络（深度学习）

- 深度学习：最早是由多伦多大学的G. E.Hinton等于2006年在Science上发表的论文” Reducing the dimensionality of data with neural networks”中提出。
- 深度学习三位开创者：Hinton, LeCun, Bengio
  – Nature杂志为纪念AI 60周年推出的综述文献：《Deep Learning》



随着网络层数的增加，以及激活函数的调整，神经网络对非线性问题的拟合能力越来越强。

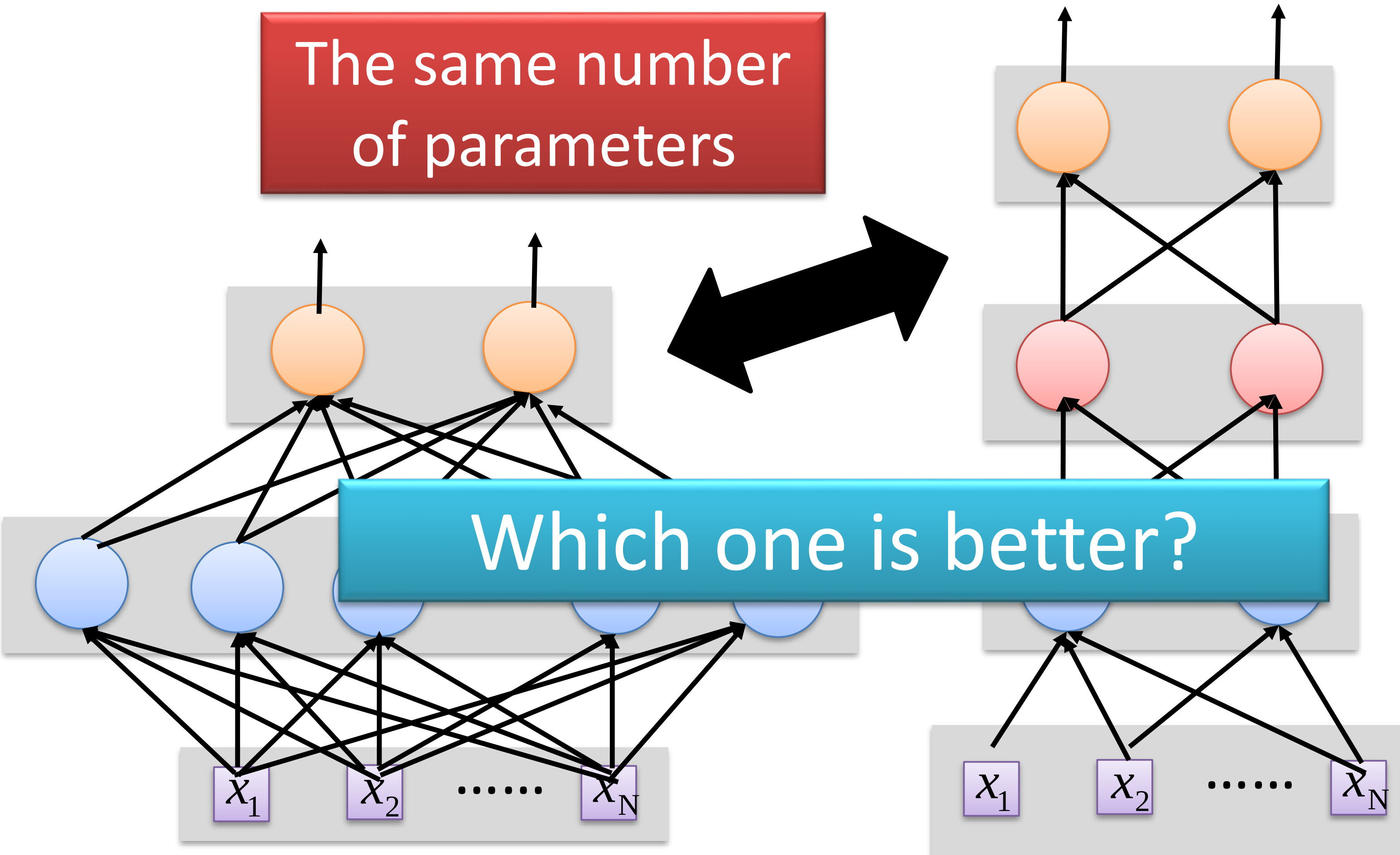
Deeper is Better?

Layer X Size	Word Error Rate (%)
1 X 2k	24.2
2 X 2k	20.4
3 X 2k	18.4
4 X 2k	17.8
5 X 2k	17.2
7 X 2k	17.1

Not surprised, more parameters, better performance

Seide, Frank, Gang Li, and Dong Yu. "Conversational Speech Transcription Using Context-Dependent Deep Neural Networks." *Interspeech*. 2011.

Fat + Short v.s. Thin + Tall



Shallow

Deep

Fat + Short v.s. Thin + Tall

Layer X Size	Word Error Rate (%)	Layer X Size	Word Error Rate (%)
1 X 2k	24.2		
2 X 2k	20.4		
3 X 2k	18.4		
4 X 2k	17.8		
5 X 2k	17.2	1 X 3772	22.5
7 X 2k	17.1	1 X 4634	22.6
		1 X 16k	22.1

Why?

Seide, Frank, Gang Li, and Dong Yu. "Conversational Speech Transcription Using Context-Dependent Deep Neural Networks." *Interspeech*. 2011.

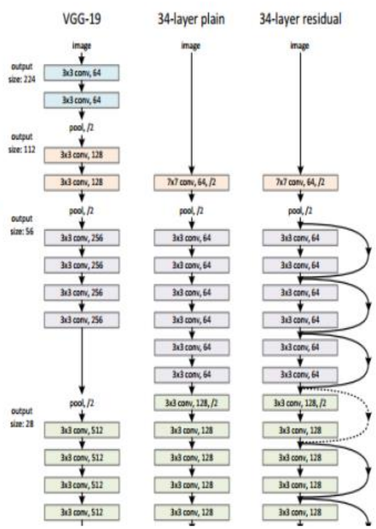
What Makes Deep Learning Succeed Now?

Big data:

- Massive labeled datasets

Algorithm:

Improved models

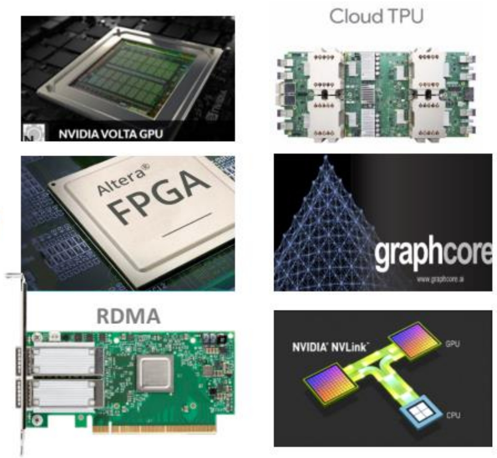


Software frameworks

TensorFlow
Microsoft CNTK
Caffe2
dmlc mxnet
PYTORCH

- Easy programming
- Fast model evolution
- Fast training and inferencing

Computing:



- Massive computing power
- Fast communication

来自：微软亚洲研究院 周礼栋《大数据系统的演化：理论、实践和展望》报告

Deep Learning Approach

训练过程：正向计算 & 反向传播（**backpropagation**）

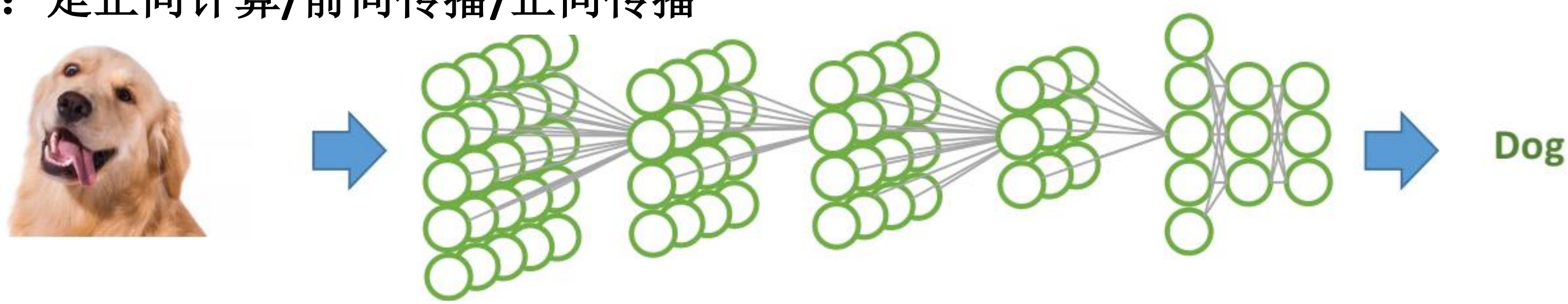
预测过程：是正向计算/前向传播/正向传播

来自：微软亚洲研究院 周礼栋《大数据系统的演化：理论、实践和展望》报告

Deep Learning Approach

训练过程：正向计算 & 反向传播（**backpropagation**）

预测过程：是正向计算/前向传播/正向传播

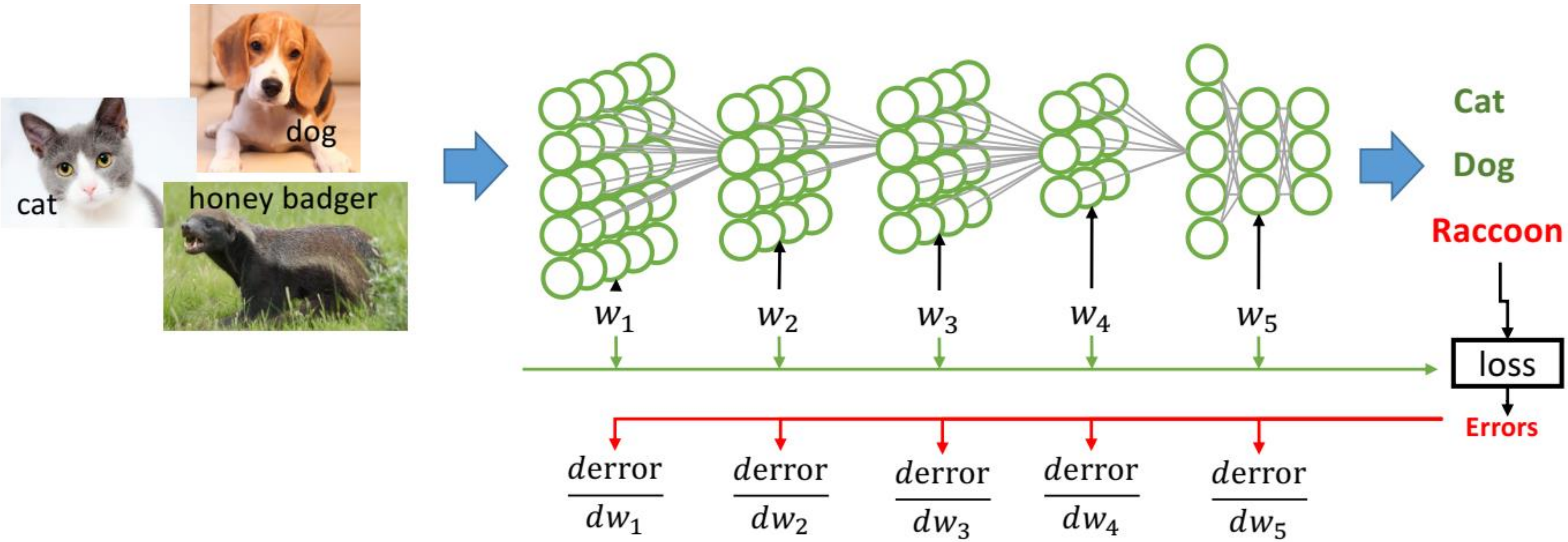


来自：微软亚洲研究院 周礼栋《大数据系统的演化：理论、实践和展望》报告

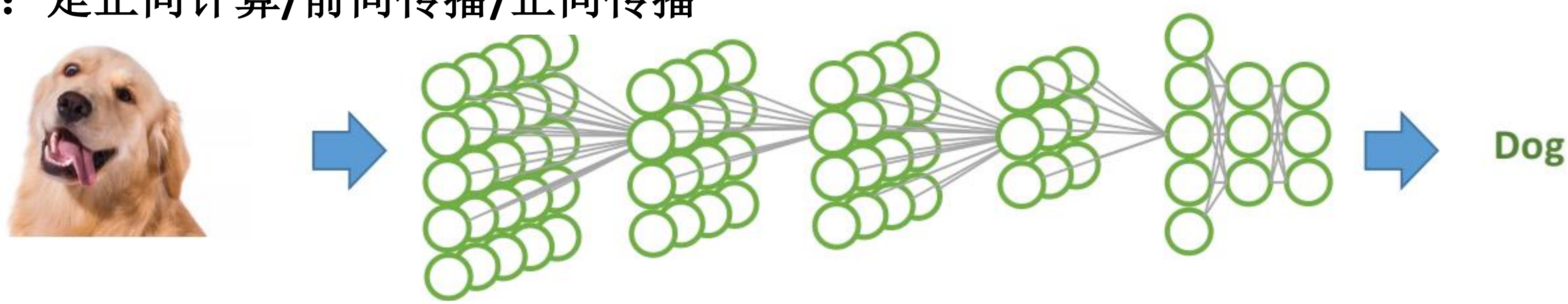
Deep Learning Approach

2个基本核心运算：
1. 正向计算
2. 反向传播

训练过程：正向计算 & 反向传播（**backpropagation**）



预测过程：是正向计算/前向传播/正向传播



来自：微软亚洲研究院 周礼栋《大数据系统的演化：理论、实践和展望》报告

本章内容

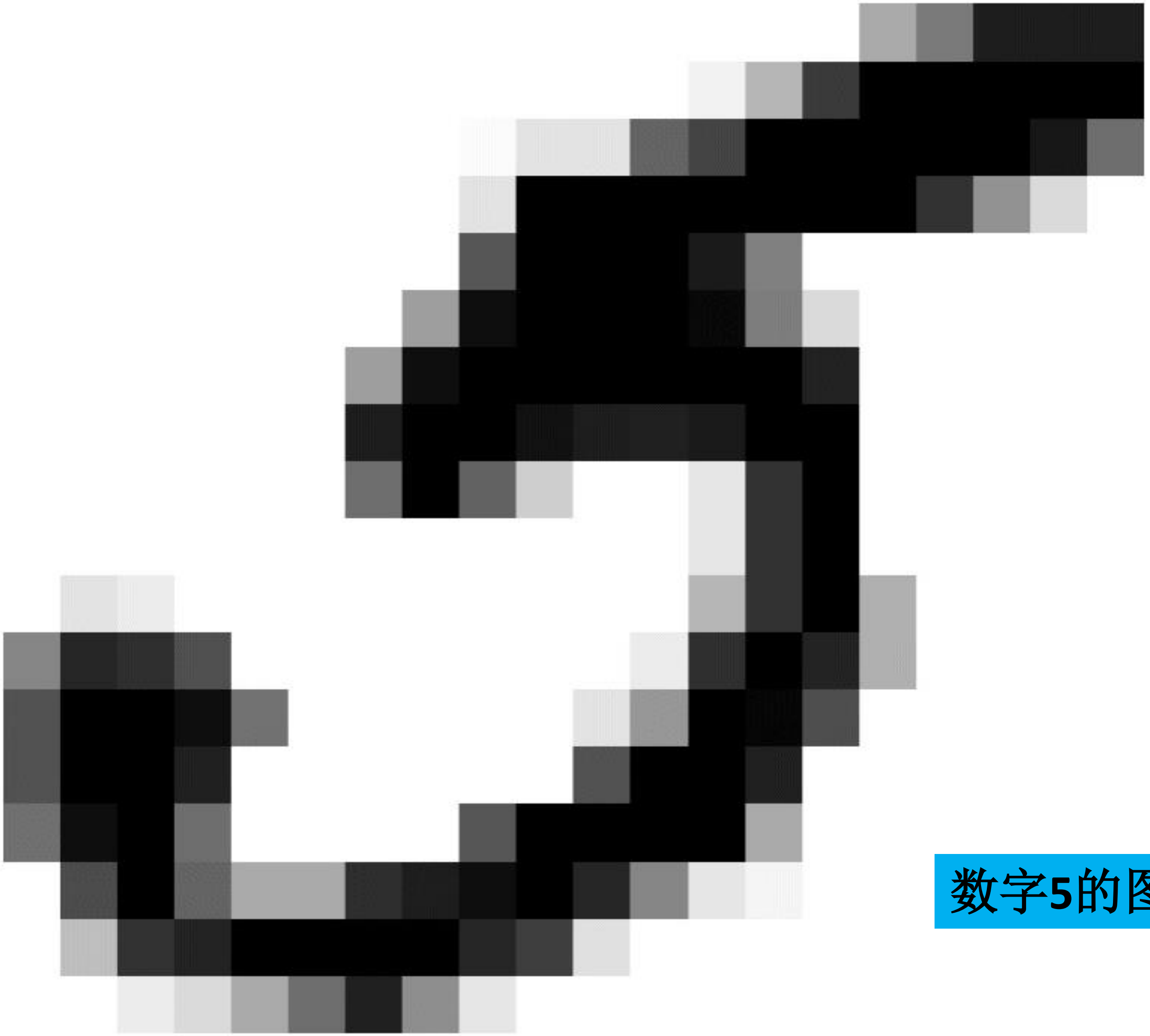
1 什么是深度学习

2 神经网络的数学基础

3 神经网络入门

4 神经网络的通用工作流程

- MNIST数据集
  - 包含70000张尺寸较小的手写数字图片，由美国的高中生和美国人口调查局的职员手写而成。
    - 图像被编码为Numpy数组，每个样本，即每图片有784个特征，因为每个图片都是28*28像素的。每个特征对应一个介于0~255之间的像素值
    - 标签是数字数组，取值范围为0~9。
    - 图像和标签一一对应。
- 任务：将手写数字的灰度图像（28像素×28像素）划分到10个类别中（0~9）。
- 问题设定：多分类问题
  - 机器学习中的“HelloWorld”



数字5的图片

- 安装Keras

```
C:\WINDOWS\system32>pip install keras
Collecting keras
  Downloading https://files.pythonhosted.org/packages/ad/fd/6bfe87920d7f4
479832bdc0fe9e589a60ceb/Keras-2.3.1-py2.py3-none-any.whl (377kB)
     |████████████████████████████████| 3
Requirement already satisfied: six>=1.9.0 in c:\program files (x86)\micro
d\anaconda3_64\lib\site-packages (from keras) (1.12.0)
Requirement already satisfied: pyyaml in c:\program files (x86)\microsoft
```

代码清单 2-1　加载 Keras 中的 MNIST 数据集

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

我们来看一下训练数据：

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

下面是测试数据：

```
>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

神经网络的数学基础

- 神经网络的数据表示：张量（tensor）
- 经网络的“齿轮”：张量运算
- 神经网络的“引擎”：基于梯度的优化

神经网络的数据表示：张量（tensor）

- 是机器学习中的基本数据结构
- 是一个数据容器。它包含的数据几乎总是数值数据，因此它是数字的容器。
- 张量的维度（dimension）：通常叫作轴（axis）
  - 可以用ndim属性来查看一个Numpy张量的轴的个数。
  - 张量轴的个数也叫作阶（rank）
- 张量是矩阵向任意维度的推广。比如，矩阵 是二维张量。
- 张量是由以下三个关键属性来定义的。
  - 轴的个数（阶）。在Numpy等Python库中也叫张量的ndim。
  - 形状。这是一个整数元组，表示张量沿每个轴的维度大小（元素个数）。
  - 数据类型（在Python库中通常叫作dtype）。这是张量中所包含数据的类型。
  - 注意，Numpy（以及大多数其他库）中不存在字符串张量，因为张量存储在预先分配的连续内存段中，而字符串的长度是可变的，无法用这种方式存储。

- 观察MNIST数据集中的数据

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

接下来，我们给出张量 train_images 的轴的个数，即 ndim 属性。

```
>>> print(train_images.ndim)
3
```

下面是它的形状。

```
>>> print(train_images.shape)
(60000, 28, 28)
```

下面是它的数据类型，即 dtype 属性。

```
>>> print(train_images.dtype)
uint8
```

神经网络的数据表示：张量（tensor）

- 按照维度，张量分为：
  - 标量（0D张量）：仅包含一个数字的张量叫作标量（scalar，也叫标量张量、零维张量、0D张量）。
  - 向量（1D张量）
  - 矩阵（2D张量）
  - 3D张量与更高维张量

- **标量（0D张量）**：仅包含一个数字的张量叫作标量（scalar，也叫标量张量、零维张量、0D张量）。
- 比如，在Numpy中，一个float32或float64的数字就是一个标量张量（或标量数组）。
- 标量张量有0个轴（ndim == 0）。

```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

- # 向量（1D张量）

数字组成的数组叫作**向量**（vector）或一维张量（1D 张量）。一维张量只有一个轴。下面是一个 Numpy 向量。

```
>>> x = np.array([12, 3, 6, 14, 7])
>>> x
array([12, 3, 6, 14, 7])
>>> x.ndim
1
```

这个向量有 5 个元素，所以被称为 **5D 向量**。不要把 5D 向量和 5D 张量弄混！ 5D 向量只有一个轴，沿着轴有 5 个维度，而 5D 张量有 5 个轴（沿着每个轴可能有任意个维度）。**维度**（dimensionality）可以表示沿着某个轴上的元素个数（比如 5D 向量），也可以表示张量中轴的个数（比如 5D 张量），这有时会令人感到混乱。对于后一种情况，技术上更准确的说法是 **5 阶张量**（张量的阶数即轴的个数），但 **5D 张量**这种模糊的写法更常见。

- # 矩阵（2D张量）

向量组成的数组叫作**矩阵**（matrix）或二维张量（2D 张量）。矩阵有 2 个轴（通常叫作**行**和**列**）。你可以将矩阵直观地理解为数字组成的矩形网格。下面是一个 Numpy 矩阵。

```
>>> x = np.array([[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]])
>>> x.ndim
2
```

第一个轴上的元素叫作**行**（row），第二个轴上的元素叫作**列**（column）。在上面的例子中，`[5, 78, 2, 34, 0]` 是 x 的第一行，`[5, 6, 7]` 是第一列。

• 3D张量与更高维张量

将多个矩阵组合成一个新的数组，可以得到一个 3D 张量，你可以将其直观地理解为数字组成的立方体。下面是一个 Numpy 的 3D 张量。

```
>>> x = np.array([[[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]]])
>>> x.ndim
3
```

将多个 3D 张量组合成一个数组，可以创建一个 4D 张量，以此类推。深度学习处理的一般是 0D 到 4D 的张量，但处理视频数据时可能会遇到 5D 张量。

在Numpy中操作张量

- 使用语法train_images[i]来选择沿着第一个轴的特定数字。选择张量的特定元素叫作张量切片（tensor slicing）。

- Numpy数组上的张量切片运算示例：
  - 选择第10~100个数字（不包括第100个），并将其放在形状为(90, 28,28)的数组中。

```
>>> my_slice = train_images[10:100]
>>> print(my_slice.shape)
(90, 28, 28)
```

注意，:等同于选择整个轴。

```
>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

等同于前面的例子

也等同于前面的例子

数据批量

- 通常来说，深度学习中所有数据张量的第一个轴（0轴，因为索引从0开始）都是**样本轴**（samples axis，有时也叫**样本维度**）。
  - 比如，在MNIST的例子中，样本就是数字图像。
- 深度学习模型不会同时处理整个数据集，而是将数据拆分成小批量。
  - 比如，下面是MNIST数据集的一个批量，批量大小为128。

    ```
    batch = train_images[:128]
    ```
  - 然后是下一个批量。

    ```
    batch = train_images[128:256]
    ```
  - 然后是第n个批量。

    ```
    batch = train_images[128 * n:128 * (n + 1)]
    ```

- 对于这种批量张量，第一个轴（0轴）叫作**批量轴（batch axis）**或**批量维度（batch dimension）**。

现实世界中的数据张量

- 向量数据：2D张量，形状为(samples, features)。
- 时间序列数据或序列数据：3D张量，形状为(samples, timesteps, features)。
- 图像：4D张量，形状为(samples, height, width, channels)或(samples, channels, height, width)。
- 视频：5D张量，形状为(samples, frames, height, width, channels)或(samples, frames, channels, height, width)。

现实世界中的数据张量 – 向量数据

- 是最常见的数据。
- 对于MNIST数据集，每个数据点(即每个样本/每张图像）都被编码为一个向量，因此一个数据批量就被编码为2D张量（即向量组成的数组），其中第一个轴是样本轴，第二个轴是特征轴。
- 人口统计数据集，其中包括每个人的年龄、邮编和收入。每个人可以表示为包含3个值的向量，而整个数据集包含100 000个人，因此可以存储在形状为(100000, 3)的2D张量中。
- 文本文档数据集，我们将每个文档表示为每个单词在其中出现的次数（字典中包含20 000个常见单词）。每个文档可以被编码为包含20 000个值的向量（每个值对应于字典中每个单词的出现次数），整个数据集包含500个文档，因此可以存储在形状为(500, 20000)的张量中。

现实世界中的数据张量 —时间序列数据或序列数据

- 当时间（或序列顺序）对于数据很重要时，应该将数据存储在带有时间轴的3D张量中。每个样本可以被编码为一个向量序列（即2D张量），因此一个数据批量就被编码为一个3D张量（见图2-3）。
  - 根据惯例，时间轴始终是第2个轴（索引为1的轴）

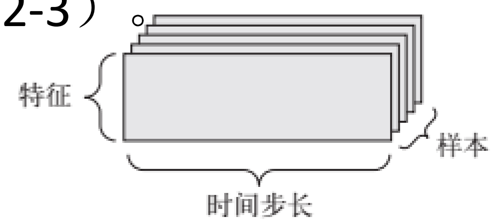


图 2-3　时间序列数据组成的 3D 张量

- 股票价格数据集。每一分钟，我们将股票的当前价格、前一分钟的最高价格和前一分钟的最低价格保存下来。因此每分钟被编码为一个3D向量，整个交易日被编码为一个形状为(390, 3)的2D张量（一个交易日有390分钟），而250天的数据则可以保存在一个形状为(250, 390, 3)的3D张量中。这里每个样本是一天的股票数据。
- 推文数据集。我们将每条推文编码为280个字符组成的序列，而每个字符又来自于128个字符组成的字母表。在这种情况下，每个字符可以被编码为大小为128的二进制向量（只有在该字符对应的索引位置取值为1，其他元素都为0）。那么每条推文可以被编码为一个形状为(280, 128)的2D张量，而包含100万条推文的数据集则可以存储在一个形状为(1000000, 280, 128)的张量中。

现实世界中的数据张量 – 图像数据

- 描述一张图像张量始终都是3D张量。
  - 图像通常具有三个维度：高度、宽度和颜色深度。
- 考虑到图像的通道数的不同，
  - 灰度图像的彩色通道只有一维。因此，如果图像大小为256×256，那么128张灰度图像组成的批量可以保存在一个形状为(128, 256, 256, 1)的张量中
    - 灰度图像（比如MNIST数字图像）只有一个颜色通道，因此可以保存在2D张量中
  - 128张彩色图像组成的批量则可以保存在一个形状为(128, 256, 256, 3)的张量中。

- 图像张量的形状有两种约定：
  - 通道在后（channels-last）的约定（在TensorFlow中使用）
    - (samples, height, width, color_depth)
  - 通道在前（channels-first）的约定（在Theano中使用）。
    - (samples, color_depth, height, width)
- Keras框架同时支持这两种格式。

现实世界中的数据张量 – 图像数据

- 描述一张图像张量始终都是3D张量。
  - 图像通常具有三个维度：高度、宽度和颜色深度。
- 考虑到图像的通道数的不同，图像数据是4D张量。
  - 灰度图像的彩色通道只有一维。因此，如果图像大小为256×256，那么128张灰度图像组成的批量可以保存在一个形状为(128, 256, 256, 1)的张量中
    - 灰度图像（比如MNIST数字图像）只有一个颜色通道，因此可以保存在2D张量中
  - 128张彩色图像组成的批量则可以保存在一个形状为(128, 256, 256, 3)的张量中。

- 图像张量的形状有两种约定：
  - 通道在后（channels-last）的约定（
    - (samples, height, width, color_depth)
  - 通道在前（channels-first）的约定（
    - (samples, color_depth, height, width)
- Keras框架同时支持这两种格式。

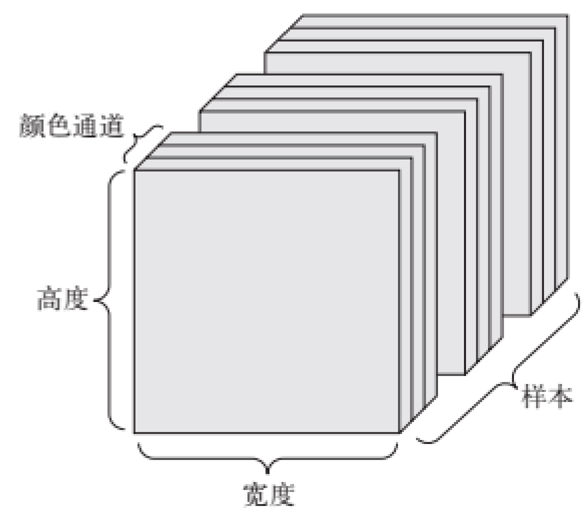


图 2-4　图像数据组成的 4D 张量（通道在前的约定）

现实世界中的数据张量 — 视频数据

- 视频数据是现实生活中需要用到5D张量的少数数据类型之一。
  - 视频可以看作一系列帧，每一帧都是一张彩色图像。每一帧都可以保存在一个形状为(height, width, color_depth)的3D张量中，因此一系列帧可以保存在一个形状为(frames, height, width,color_depth)的4D张量中，而不同视频组成的批量则可以保存在一个5D张量中，其形状为(samples, frames, height, width, color_depth)。
- 举个例子，一个以每秒4帧采样的60秒YouTube视频片段，视频尺寸为144×256，这个视频共有240帧。4个这样的视频片段组成的批量将保存在形状为(4, 240, 144, 256, 3)的张量中。总共有106 168 320个值！如果张量的数据类型（dtype）是float32，每个值都是32位，那么这个张量共有405MB。
  - 现实生活中遇到的视频要小得多，因为它们不以float32格式存储，而且通常被大大压缩，比如MPEG格式。

神经网络的“齿轮”：张量运算

- 加法运算（+）
- relu运算
- 点积运算（dot）
- 广播：两个形状不同的张量进行运算时，如果没有歧义的话，较小的张量会被广播（broadcast），以匹配较大张量的形状。广播包含以下两步。
  1. 向较小的张量添加轴（叫作广播轴），使其ndim与较大的张量相同。
  2. 将较小的张量沿着新轴重复，使其形状与较大的张量相同。

神经网络的“齿轮”：张量运算

- 张量变形

```
>>> x = np.array([[0., 1.],
                  [2., 3.],
                  [4., 5.]])
>>> print(x.shape)
(3, 2)
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
>>> x = x.reshape((2, 3))
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
```

- 转置（transposition）：特殊的张量变形
  - 对矩阵做转置是指将行和列互换，使x[i, :]变为x[:, i]。

```
>>> x = np.zeros((300, 20))    <—— 创建一个形状为 (300, 20) 的零矩阵
>>> x = np.transpose(x)
>>> print(x.shape)
(20, 300)
```

神经网络的“引擎”：基于梯度的优化

- 一个训练循环（training loop）包含的具体过程如下：
  (1)抽取训练样本x和对应目标y组成的数据批量。
  (2)在x上运行网络，得到预测值y_pred。［这一步叫作前向传播（forward pass）］
  (3)计算网络在这批数据上的损失，用于衡量y_pred和y之间的距离。
  (4)更新网络的所有权重，使网络在这批数据上的损失略微下降。
- 经过多轮的训练循环后，最终得到的网络在训练数据上的损失非常小，即预测值y_pred和预期目标y之间的距离非常小。网络“学会”将输入映射到正确的目标。
- 存在问题：如何使得每次训练循环后，损失都下降？
  - 利用网络中所有运算都是可微（differentiable）的这一事实，计算 损失相对于网络系数的梯度（gradient），然后向梯度的反方向改变系数，从而使损失降低。

- 什么是导数？

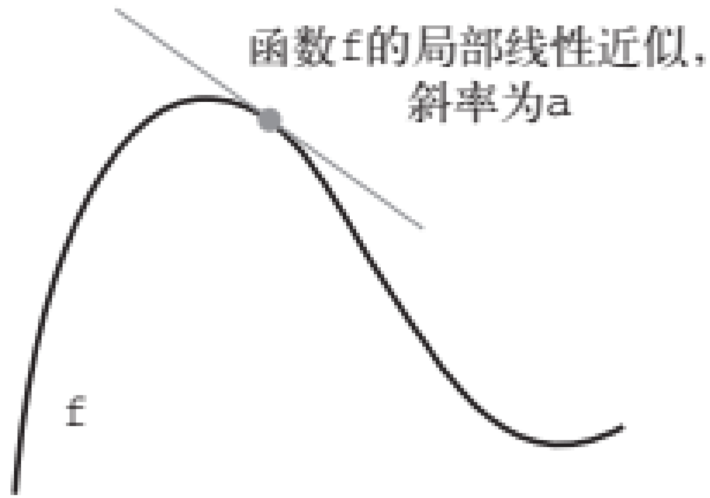


图 2-10 f 在 p 点的导数

- 某函数f(x)可微，是指“可以被求导”。
- 梯度：张量运算的导数
  – 梯度（gradient）是张量运算的导数。它是导数这一概念向多元函数导数的推广。多元函数是以张量作为输入的函数。

链式求导：反向传播算法(Backpropagation)

- Backpropagation: an efficient way to compute $\partial L / \partial w$ in neural network

TensorFlow

torch

theano

Caffe

Microsoft CNTK

Chainer

Deep Learning library produced by Amazon DSSTNE

mxnet

Gradient Descent

Network parameters $\theta = \{w_1, w_2, \cdots, b_1, b_2, \cdots\}$

Starting Parameters $\boldsymbol{\theta}^{<0>} \longrightarrow \boldsymbol{\theta}^{<1>} \longrightarrow \boldsymbol{\theta}^{<2>} \longrightarrow$ ......

$$\nabla \mathrm{L}(\theta) = \begin{bmatrix} \partial \mathrm{L}(\theta)/\partial w_1 \\ \partial \mathrm{L}(\theta)/\partial w_2 \\ \vdots \\ \partial \mathrm{L}(\theta)/\partial b_1 \\ \partial \mathrm{L}(\theta)/\partial b_2 \\ \vdots \end{bmatrix}$$

Compute $\nabla \mathbf{L}(\boldsymbol{\theta}^{<0>})$ $\quad \boldsymbol{\theta}^{<1>} = \boldsymbol{\theta}^{<0>} - \eta \nabla \mathbf{L}(\boldsymbol{\theta}^{<0>})$

Compute $\nabla \mathbf{L}(\boldsymbol{\theta}^{<1>})$ $\quad \boldsymbol{\theta}^{<2>} = \boldsymbol{\theta}^{<1>} - \eta \nabla \mathbf{L}(\boldsymbol{\theta}^{<1>})$

Millions of parameters ......

To compute the gradients efficiently, we use ***backpropagation***.

Chain Rule

Case 1

$$y = g(x) \qquad z = h(y)$$

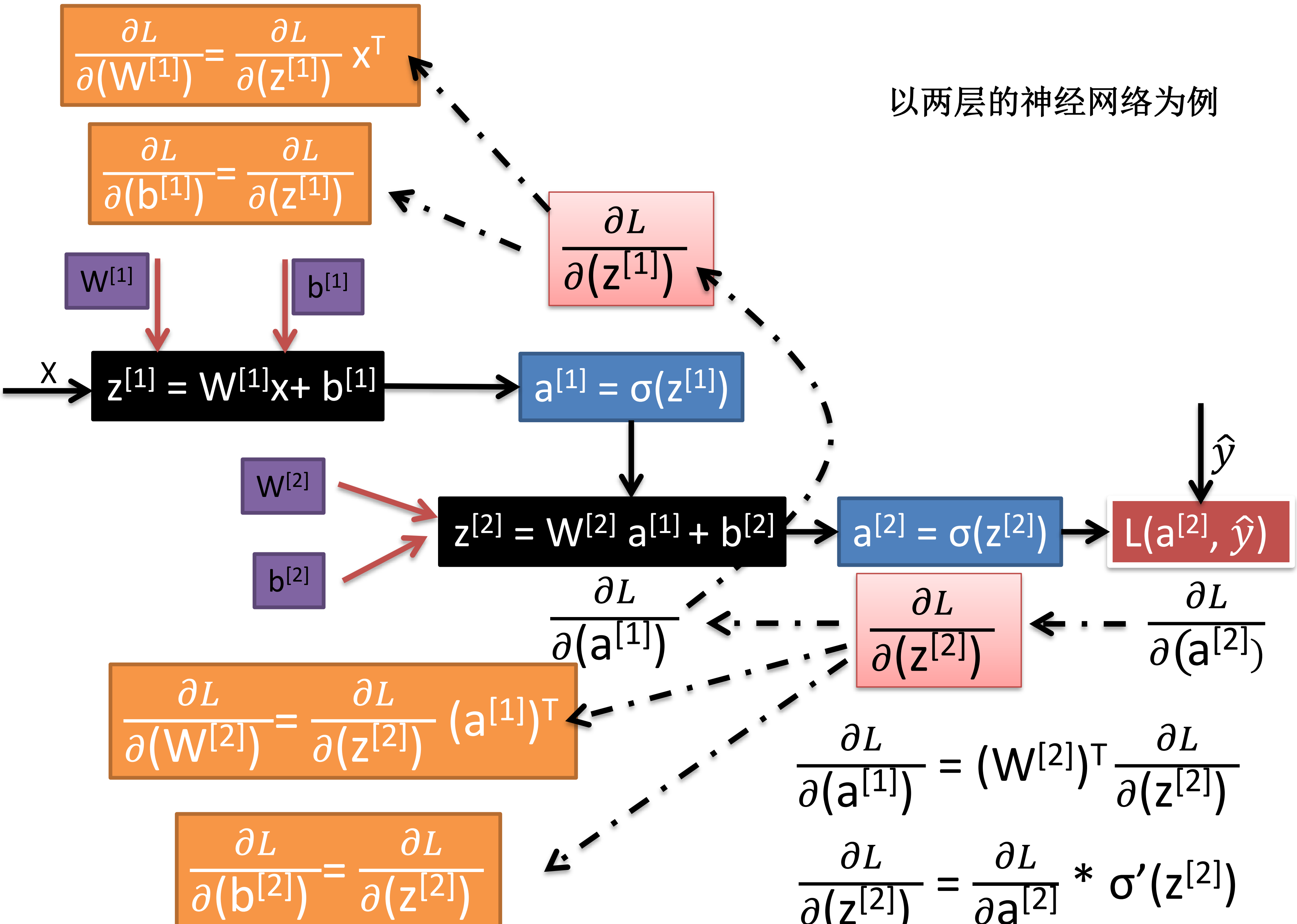
$$\Delta x \to \Delta y \to \Delta z \qquad\qquad \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Case 2

$$x = g(s) \qquad y = h(s) \qquad z = k(x, y)$$

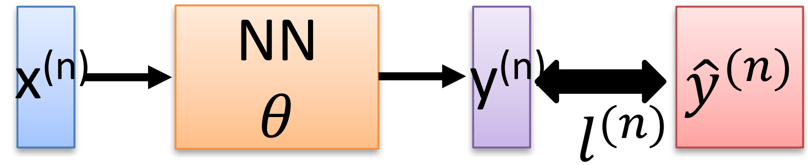
$$\begin{tikzcd}
 & \Delta x \arrow[dr] & \\
\Delta s \arrow[ur] \arrow[dr] & & \Delta z \\
 & \Delta y \arrow[ur] &
\end{tikzcd}$$

$$\frac{dz}{ds} = \frac{\partial z}{\partial x}\frac{dx}{ds} + \frac{\partial z}{\partial y}\frac{dy}{ds}$$

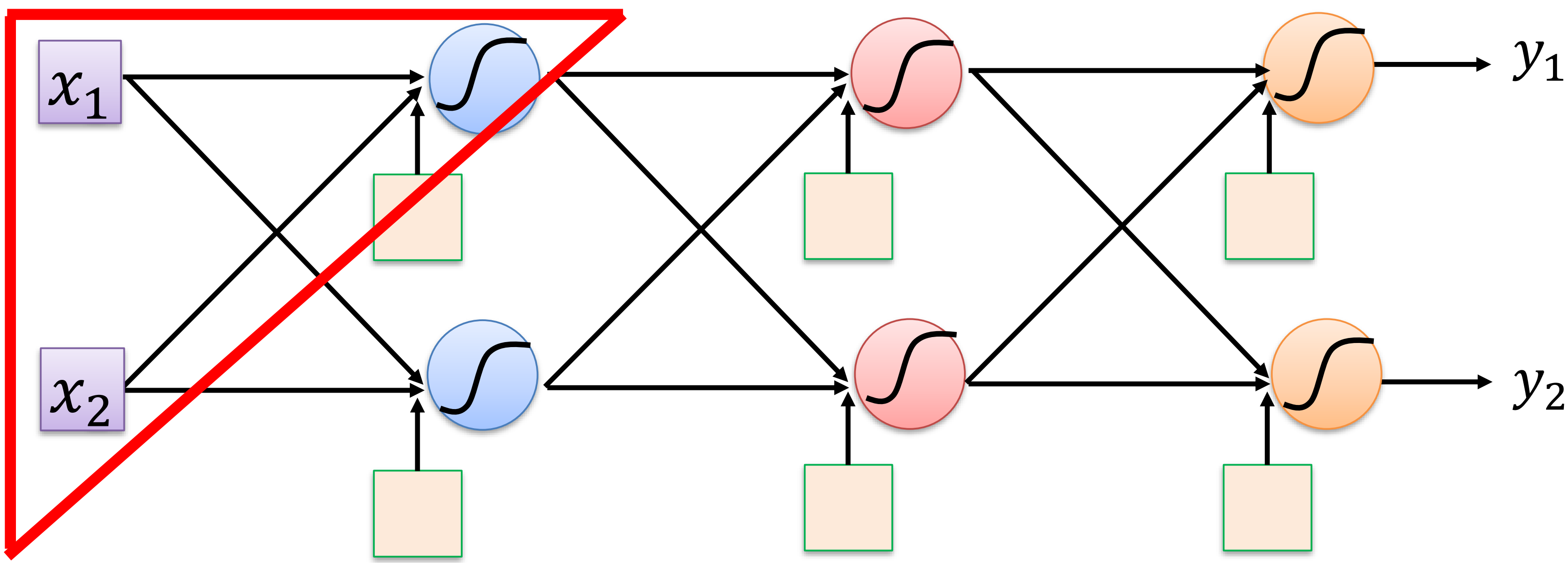


以两层的神经网络为例
$\frac{\partial L}{\partial (W^{[1]})} = \frac{\partial L}{\partial (z^{[1]})} x^T$
$\frac{\partial L}{\partial (b^{[1]})} = \frac{\partial L}{\partial (z^{[1]})}$
$\frac{\partial L}{\partial (z^{[1]})}$
$W^{[1]}$
$b^{[1]}$
X
$z^{[1]} = W^{[1]}x + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$\hat{y}$
$W^{[2]}$
$b^{[2]}$
$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$
$L(a^{[2]}, \hat{y})$
$\frac{\partial L}{\partial (a^{[1]})}$
$\frac{\partial L}{\partial (z^{[2]})}$
$\frac{\partial L}{\partial (a^{[2]})}$
$\frac{\partial L}{\partial (W^{[2]})} = \frac{\partial L}{\partial (z^{[2]})} (a^{[1]})^T$
$\frac{\partial L}{\partial (a^{[1]})} = (W^{[2]})^T \frac{\partial L}{\partial (z^{[2]})}$
$\frac{\partial L}{\partial (b^{[2]})} = \frac{\partial L}{\partial (z^{[2]})}$
$\frac{\partial L}{\partial (z^{[2]})} = \frac{\partial L}{\partial a^{[2]}} * \sigma'(z^{[2]})$

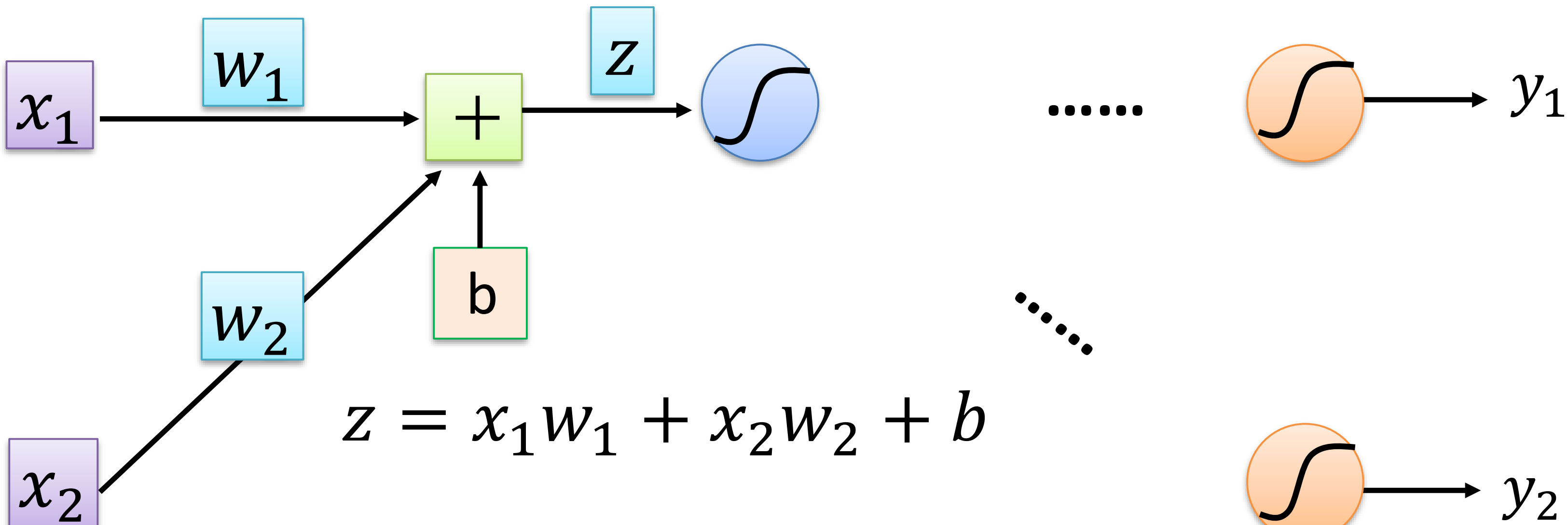
Backpropagation



$$L(\theta)=\sum_{n=1}^{N} l^{(n)}(\theta) \quad \Rightarrow \quad \frac{\partial L(\theta)}{\partial w}=\sum_{n=1}^{N}\frac{\partial l^{(n)}(\theta)}{\partial w}$$



Backpropagation



$$z = x_1 w_1 + x_2 w_2 + b$$

$$\frac{\partial l}{\partial w} = ? \quad \frac{\partial z}{\partial w}\frac{\partial l}{\partial z}$$

(Chain rule)

<u>Forward pass:</u>

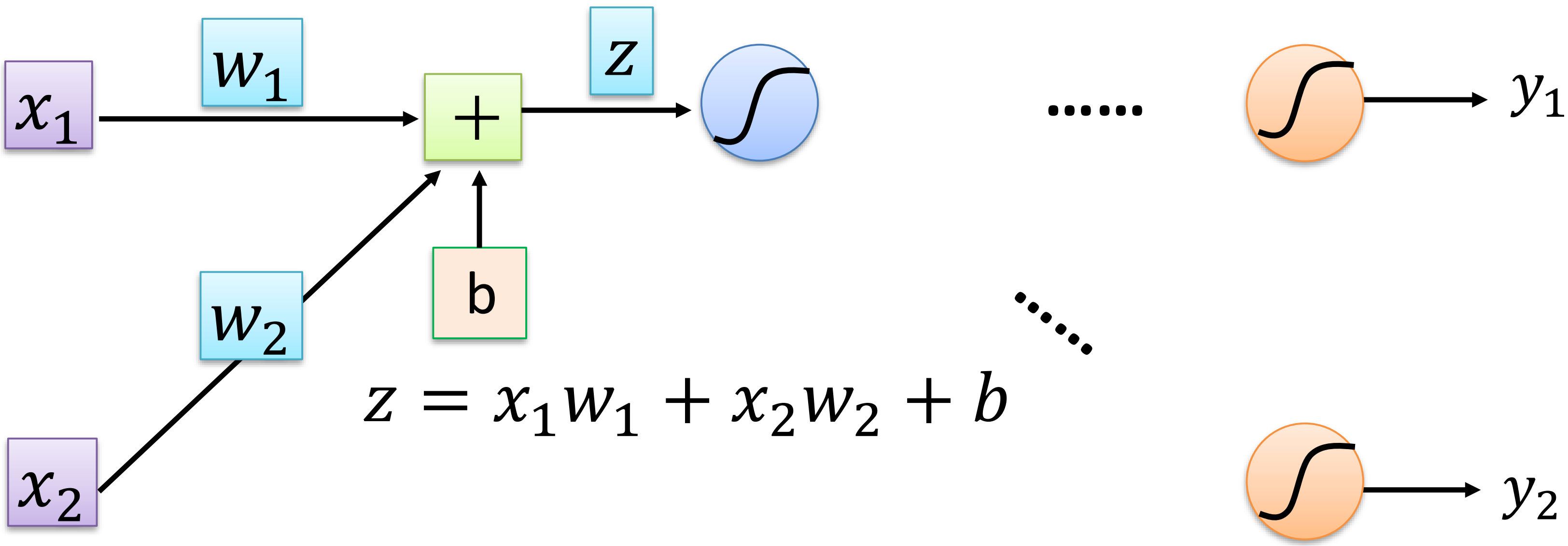
Compute $\partial z / \partial w$ for all parameters

<u>Backward pass:</u>

Compute $\partial l / \partial z$ for all activation function inputs z

Backpropagation – Forward pass

Compute $\partial z / \partial w$ for all parameters



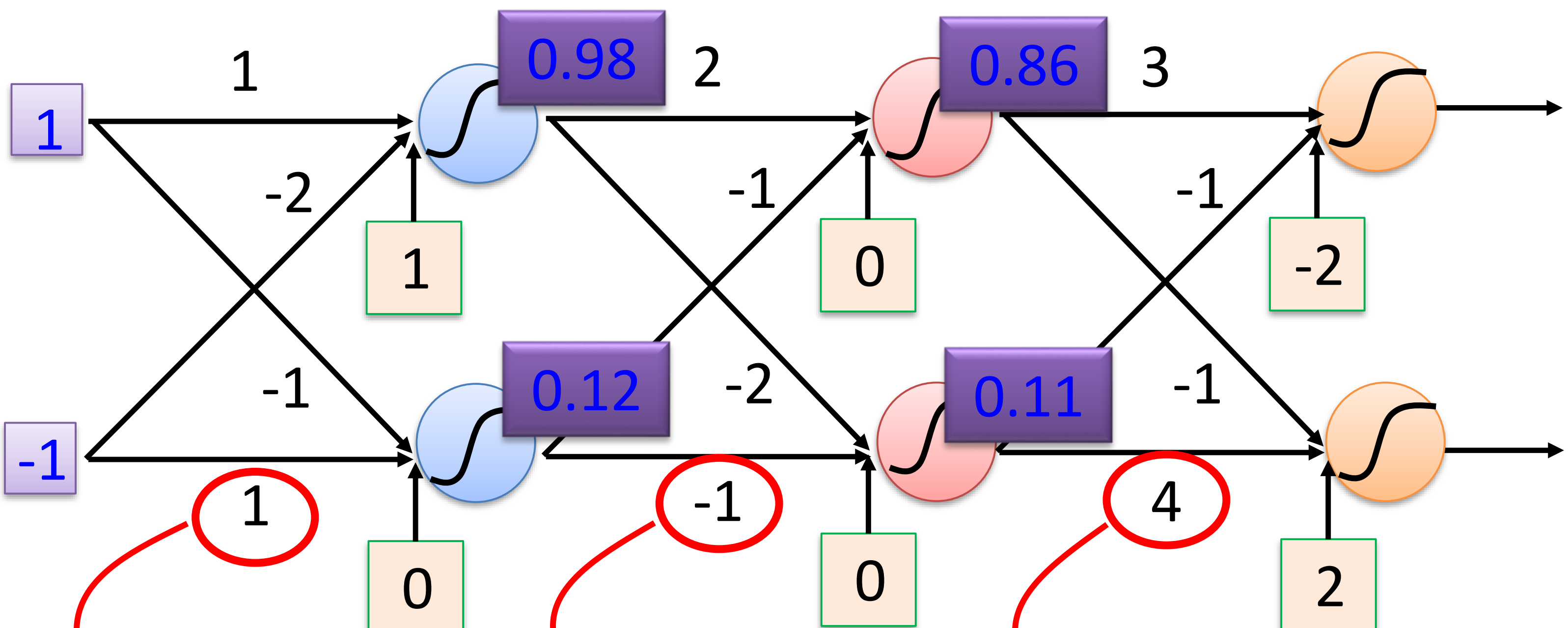
$$z = x_1 w_1 + x_2 w_2 + b$$

$$\left.\begin{array}{l} \partial z / \partial w_1 = ? \;\; x_1 \\ \partial z / \partial w_2 = ? \;\; x_2 \end{array}\right\}$$

The value of the input connected by the weight

Backpropagation – Forward pass

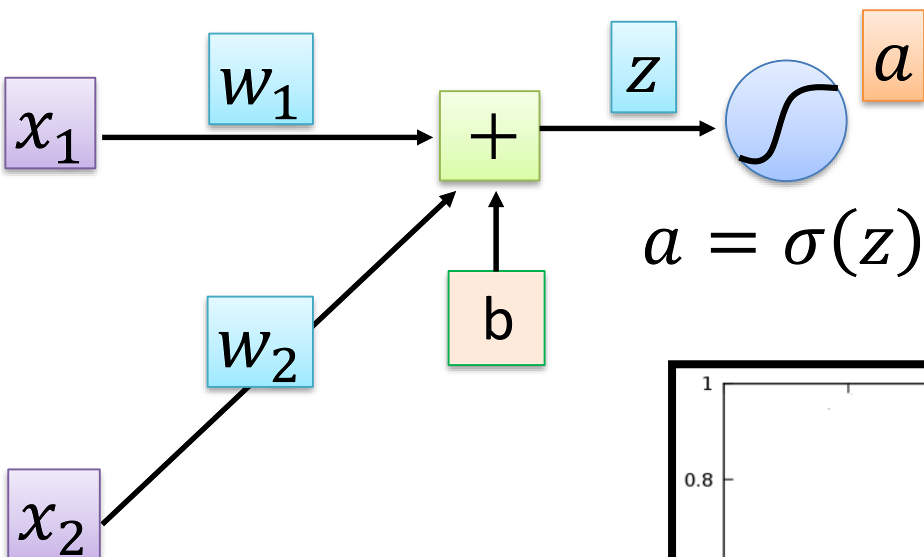
Compute $\partial z/\partial w$ for all parameters



$$\frac{\partial z}{\partial w} = -1 \qquad \frac{\partial z}{\partial w} = 0.12 \qquad \frac{\partial z}{\partial w} = 0.11$$

Backpropagation – Backward pass

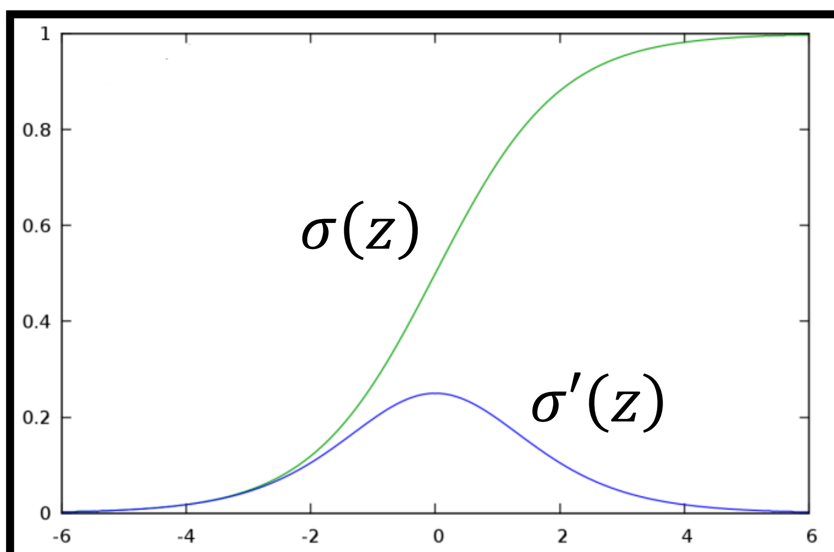
Compute $\partial l / \partial z$ for all activation function inputs z



$$a = \sigma(z)$$

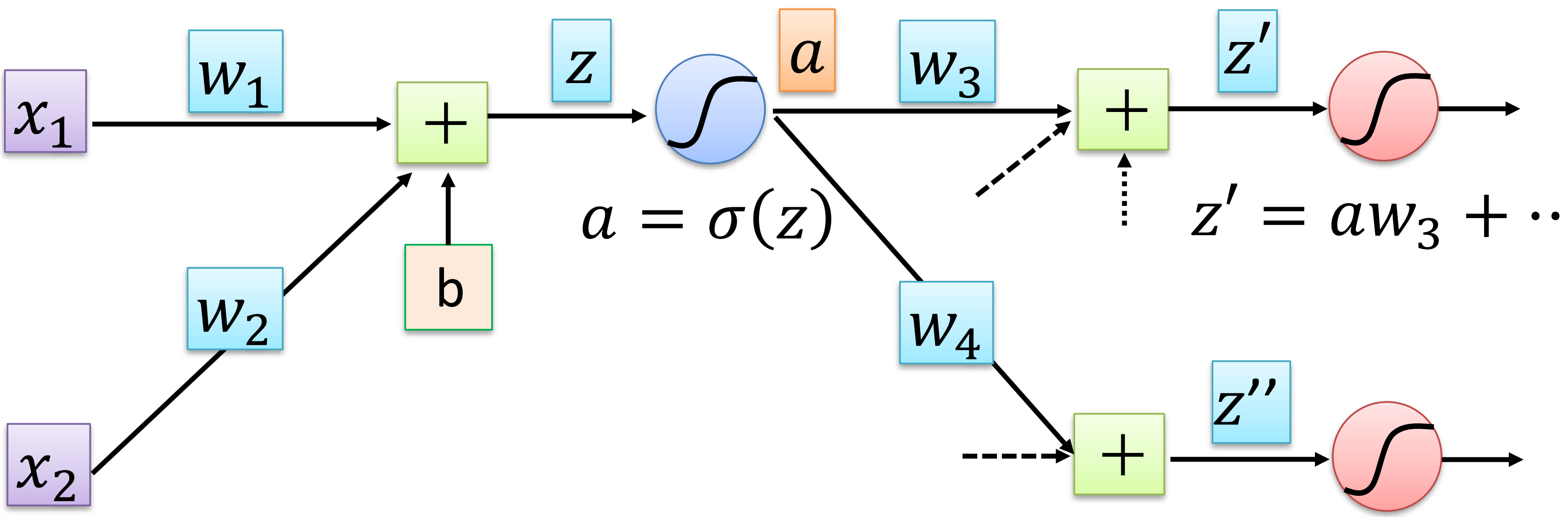
$$\frac{\partial l}{\partial z} = \underline{\frac{\partial a}{\partial z}} \frac{\partial l}{\partial a}$$

$\Rightarrow \sigma'(z)$



Backpropagation – Backward pass

Compute $\partial l / \partial z$ for all activation function inputs z



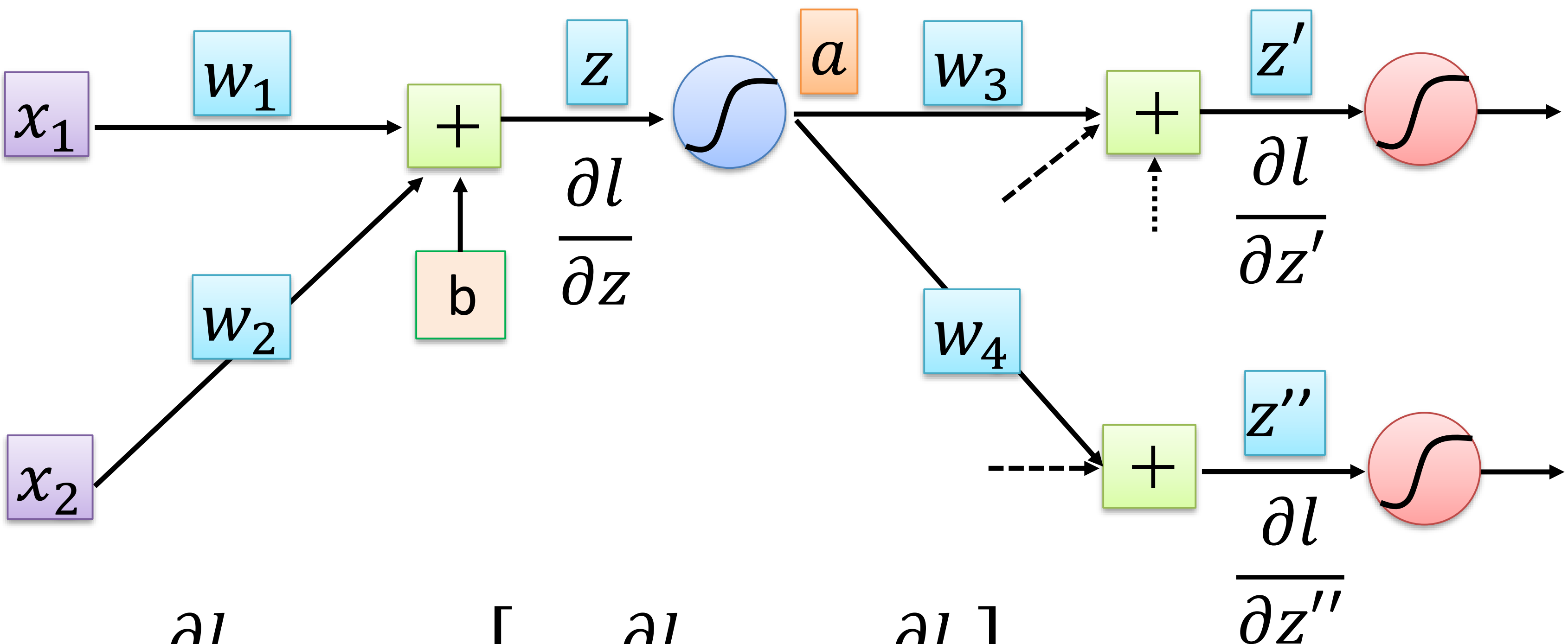
$$\frac{\partial l}{\partial z} = \frac{\partial a}{\partial z} \underline{\frac{\partial l}{\partial a}} \qquad \frac{\partial l}{\partial a} = \underbrace{\frac{\partial z'}{\partial a}}_{w_3} \underbrace{\frac{\partial l}{\partial z'}}_{?} + \underbrace{\frac{\partial z''}{\partial a}}_{w_4} \underbrace{\frac{\partial l}{\partial z''}}_{?}$$

(Chain rule)

Assumed it's known

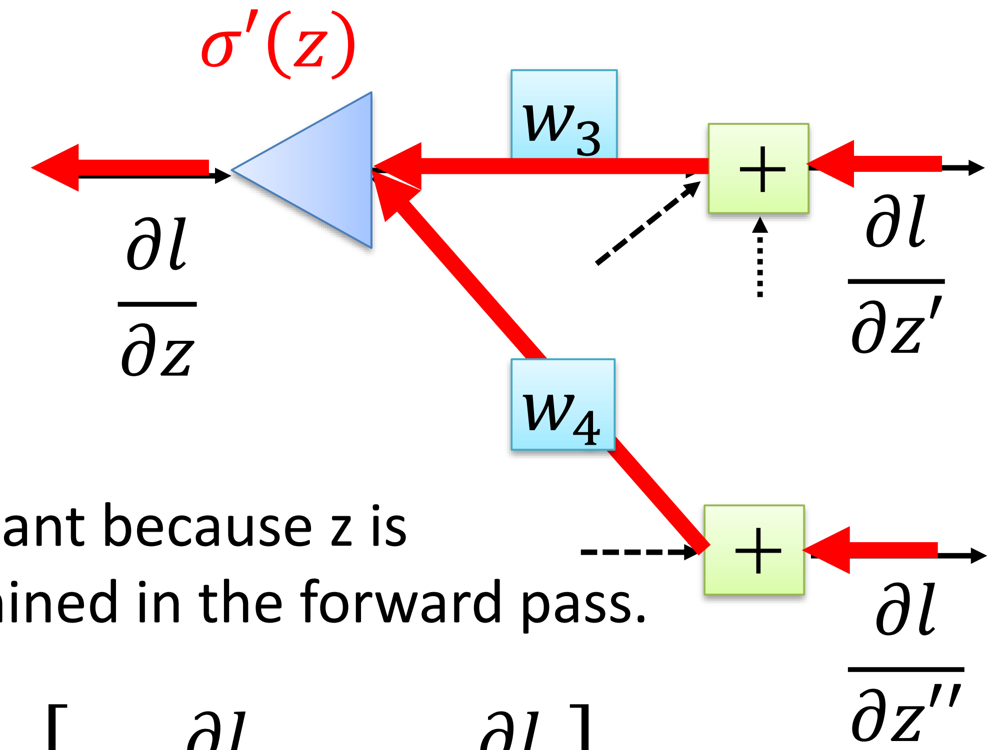
Backpropagation – Backward pass

Compute $\partial l / \partial z$ for all activation function inputs z



$$\frac{\partial l}{\partial z} = \sigma'(z)\left[w_3 \frac{\partial l}{\partial z'} + w_4 \frac{\partial l}{\partial z''}\right]$$

Backpropagation – Backward pass

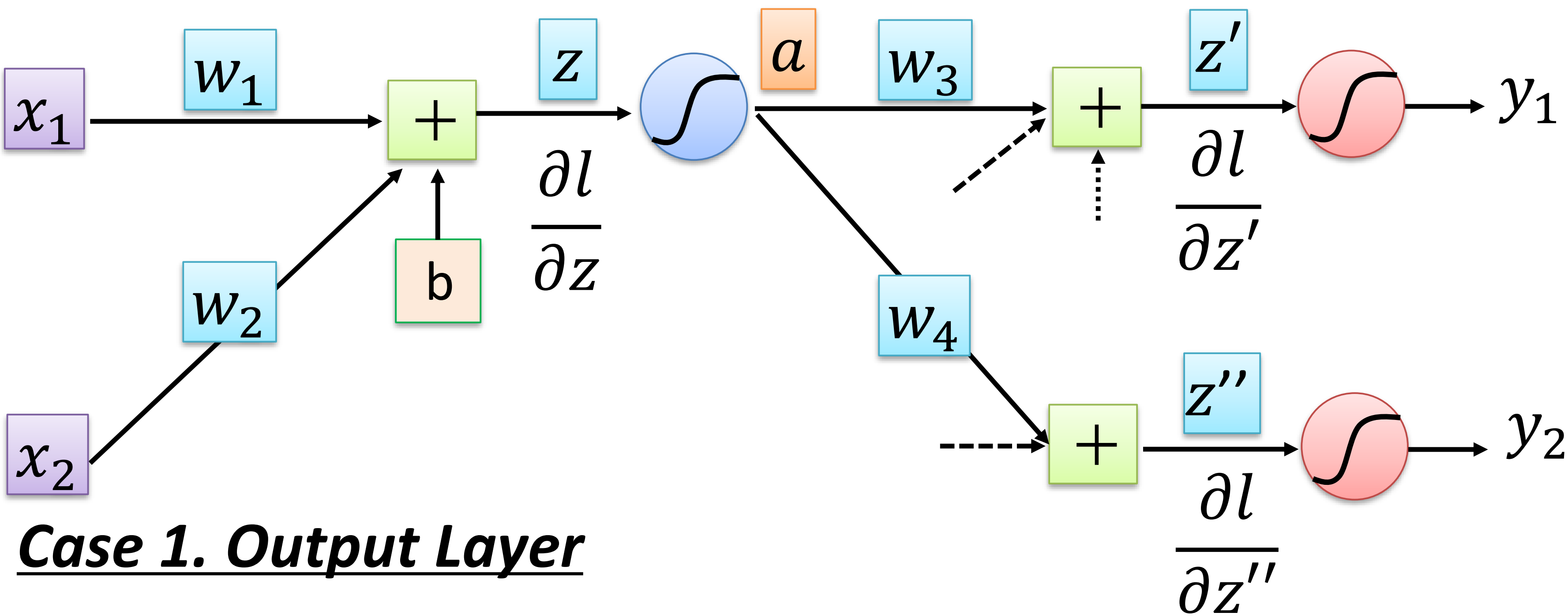


$\sigma'(z)$ is a constant because z is already determined in the forward pass.

$$\frac{\partial l}{\partial z} = \sigma'(z)\left[w_3 \frac{\partial l}{\partial z'} + w_4 \frac{\partial l}{\partial z''}\right]$$

Backpropagation – Backward pass

Compute $\partial l / \partial z$ for all activation function inputs z



Case 1. Output Layer

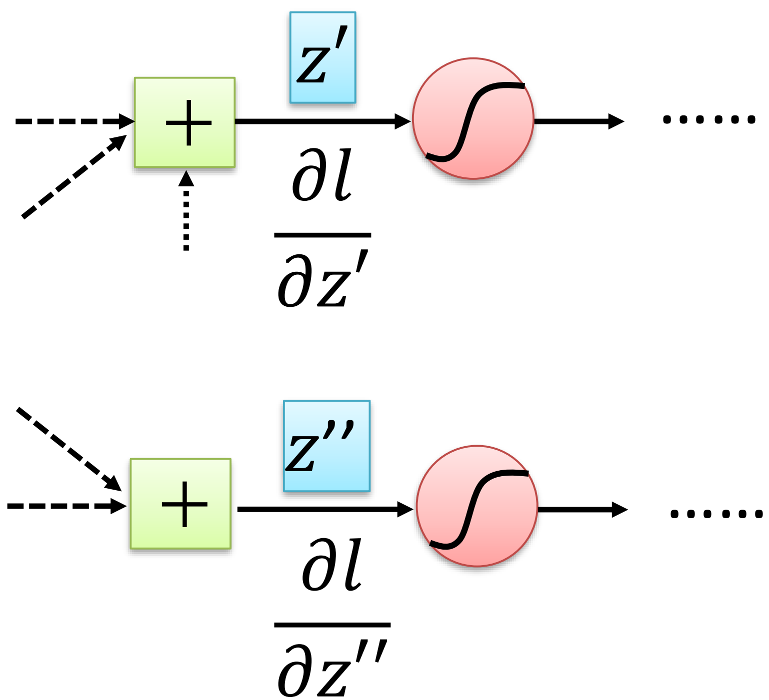
$$\frac{\partial l}{\partial z'} = \frac{\partial y_1}{\partial z'} \frac{\partial l}{\partial y_1} \qquad \frac{\partial l}{\partial z''} = \frac{\partial y_2}{\partial z''} \frac{\partial l}{\partial y_2}$$

Done!

Backpropagation – Backward pass

Compute $\partial l/\partial z$ for all activation function inputs z

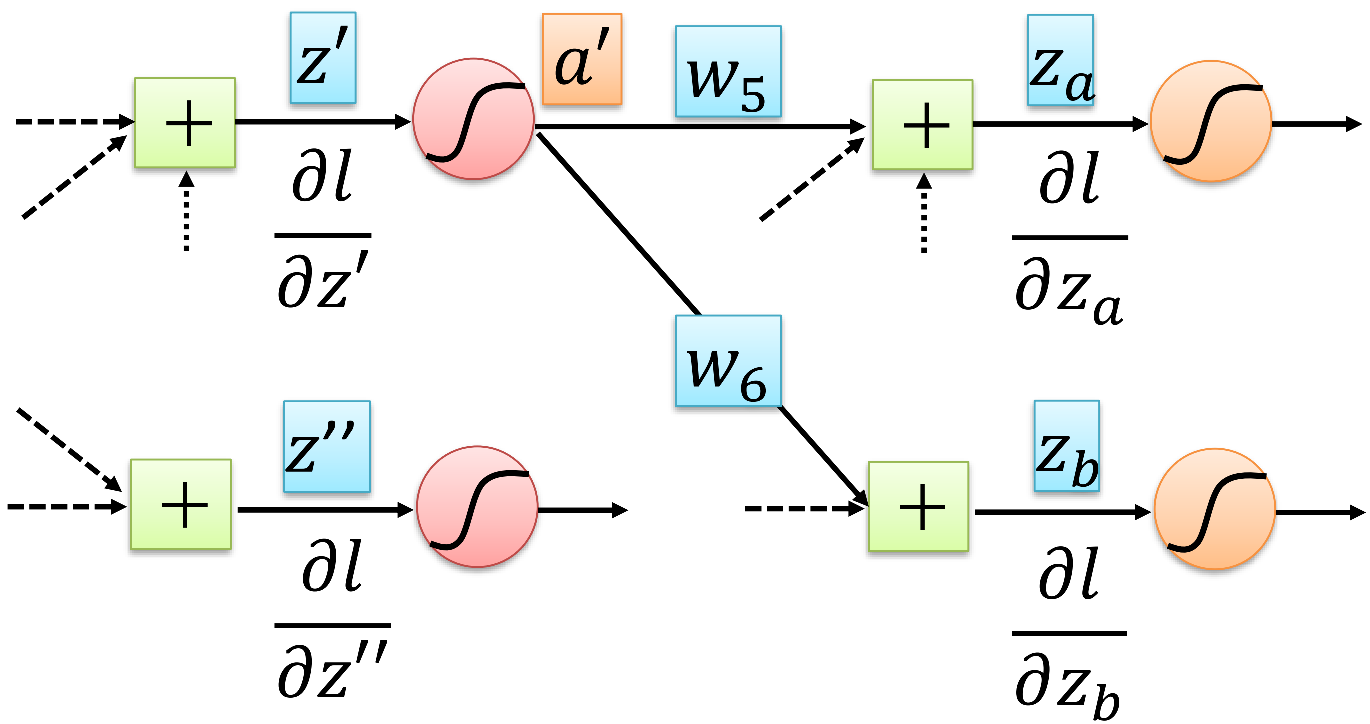
Case 2. Not Output Layer



Backpropagation – Backward pass

Compute $\partial l/\partial z$ for all activation function inputs z

<u>Case 2. Not Output Layer</u>

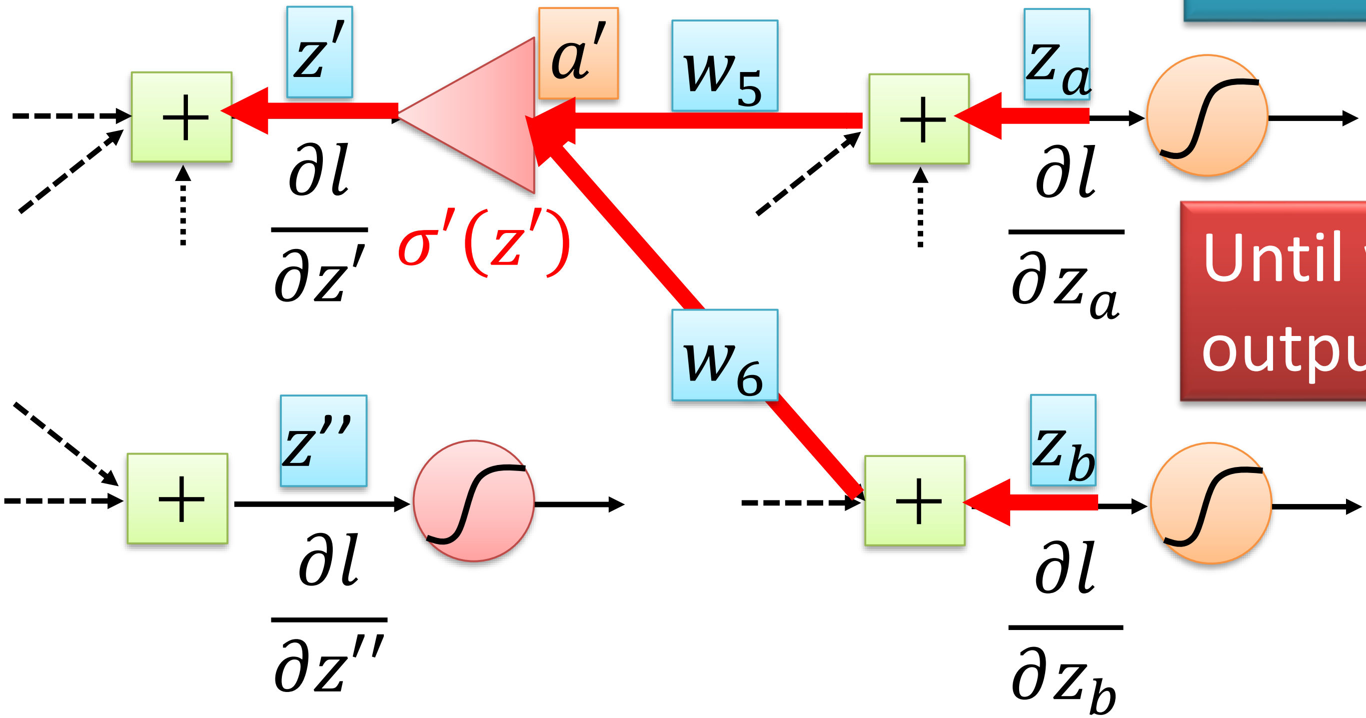


Backpropagation – Backward pass

Compute $\partial l / \partial z$ for all activation function inputs z

Case 2. Not Output Layer

Compute $\partial l / \partial z$ recursively

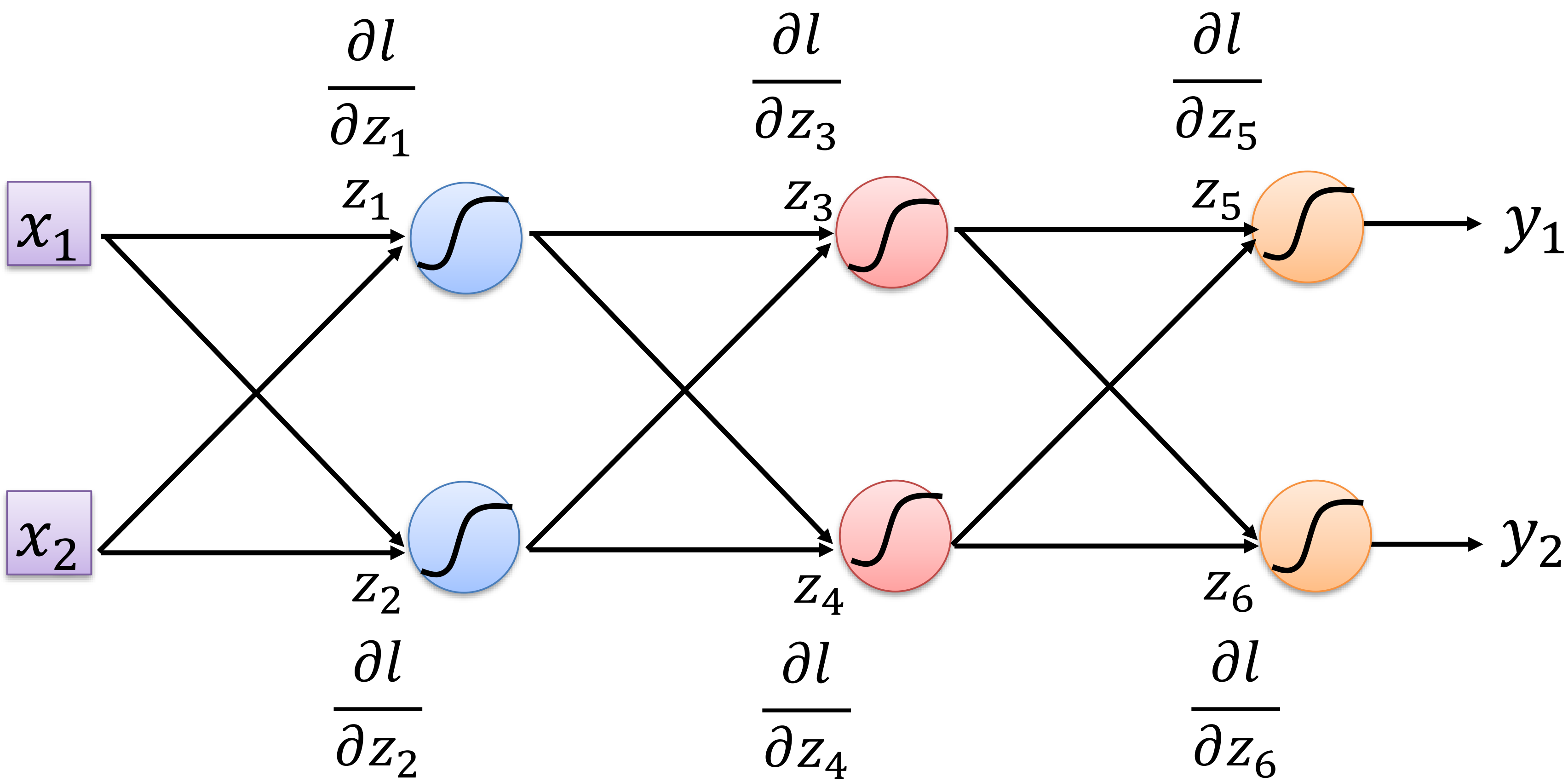


Until we reach the output layer ……

Backpropagation – Backward Pass

Compute $\partial l / \partial z$ for all activation function inputs z

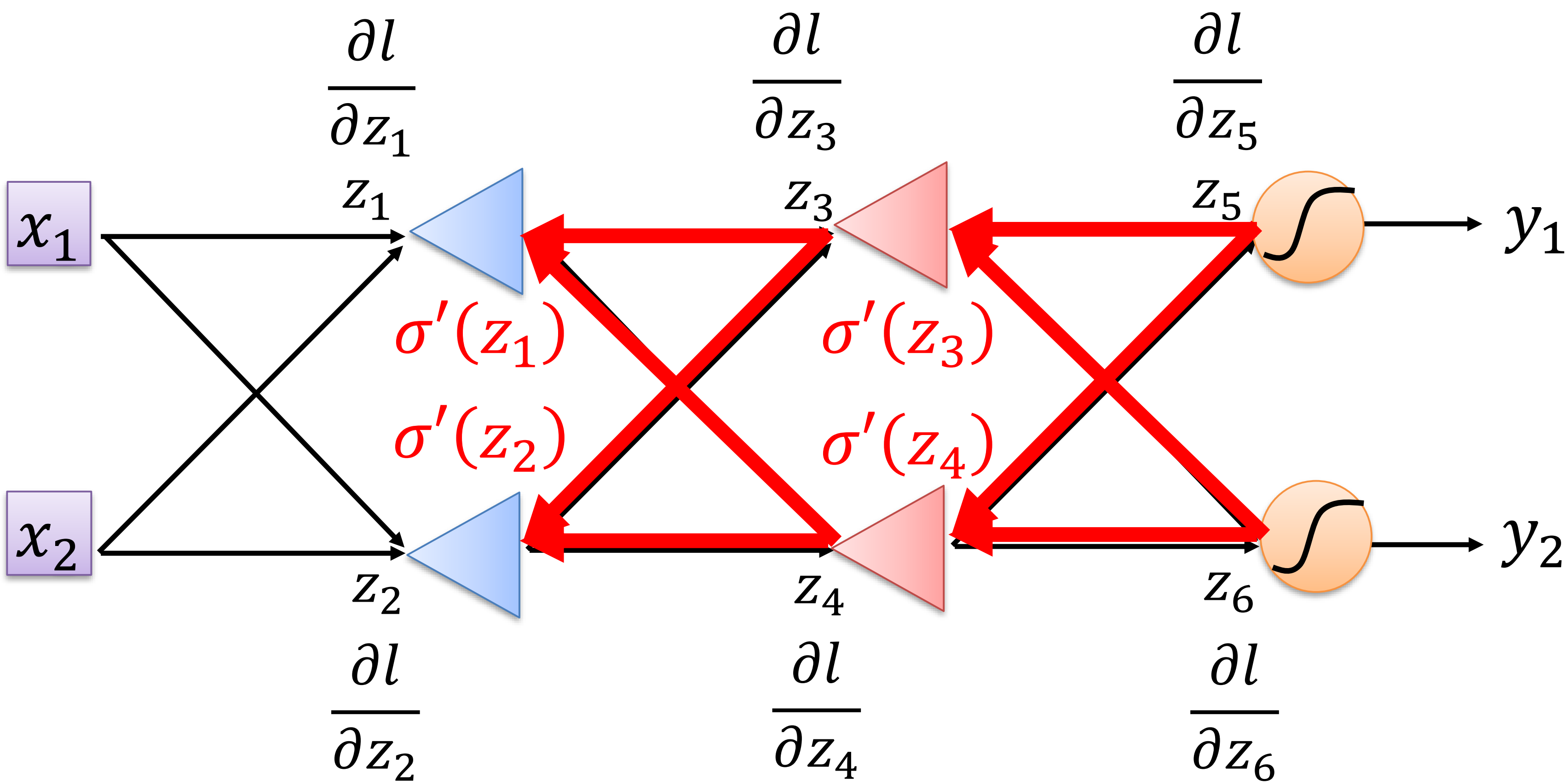
Compute $\partial l / \partial z$ from the output layer



Backpropagation – Backward Pass

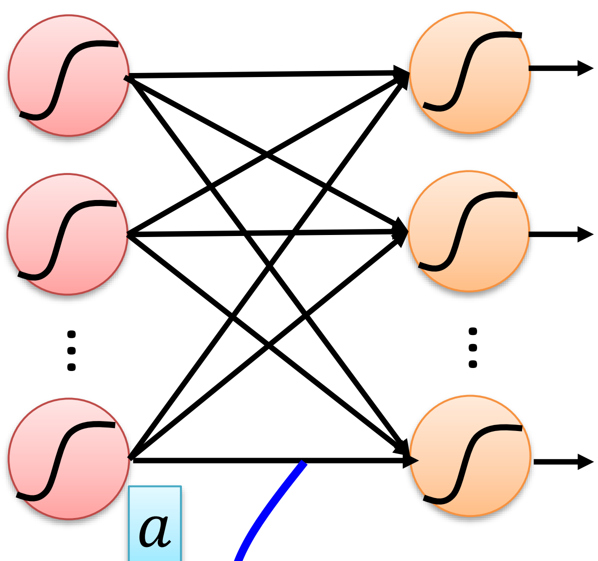
Compute $\partial l / \partial z$ for all activation function inputs z

Compute $\partial l / \partial z$ from the output layer

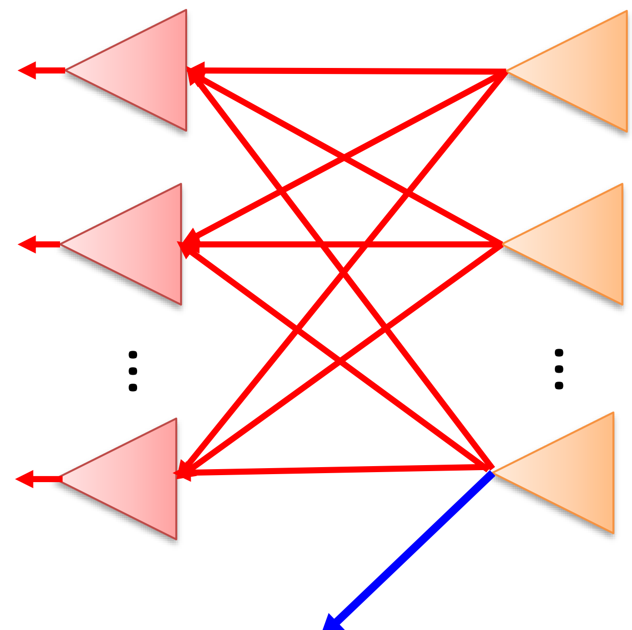


Backpropagation – Summary

Forward Pass



Backward Pass



$$\frac{\partial z}{\partial w} = a \quad \times \quad \frac{\partial l}{\partial z} \quad = \frac{\partial l}{\partial w}$$

for all w

链式求导：反向传播算法(Backpropagation)

- 输入数据。

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

输入图像保存在float32格式的Numpy张量中，形状分别为(60000,784)（训练数据）和(10000, 784)（测试数据）

- 构建网络。

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

- 网络的编译。

```
network.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

- 训练循环。

```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

  - 调用fit时发生了什么？
  - 网络开始在训练数据上进行迭代（每个小批量包含128个样本），共迭代5次［在所有训练数据上迭代一次叫作一个轮次（epoch）］。在每次迭代过程中，网络会计算批量损失相对于权重的梯度，并相应地更新权重。5轮之后，网络进行了2345次梯度更新（每轮469次），网络损失值将变得足够小，使得网络能够以很高的精度对手写数字进行分类。