



入门端智能

从 DIY 一个 NN 开始

莲叔

个人简介

- 花名：瞬咏
- SwiftGG 老炮，老司机周报编辑
- 从前的函数式编程死忠，近期主要在搞音视频与端智能
- 目前负责 UC 视频业务的客户端工作



端智能？

- 移动端运行人工智能相关任务

计算机视觉

NLP

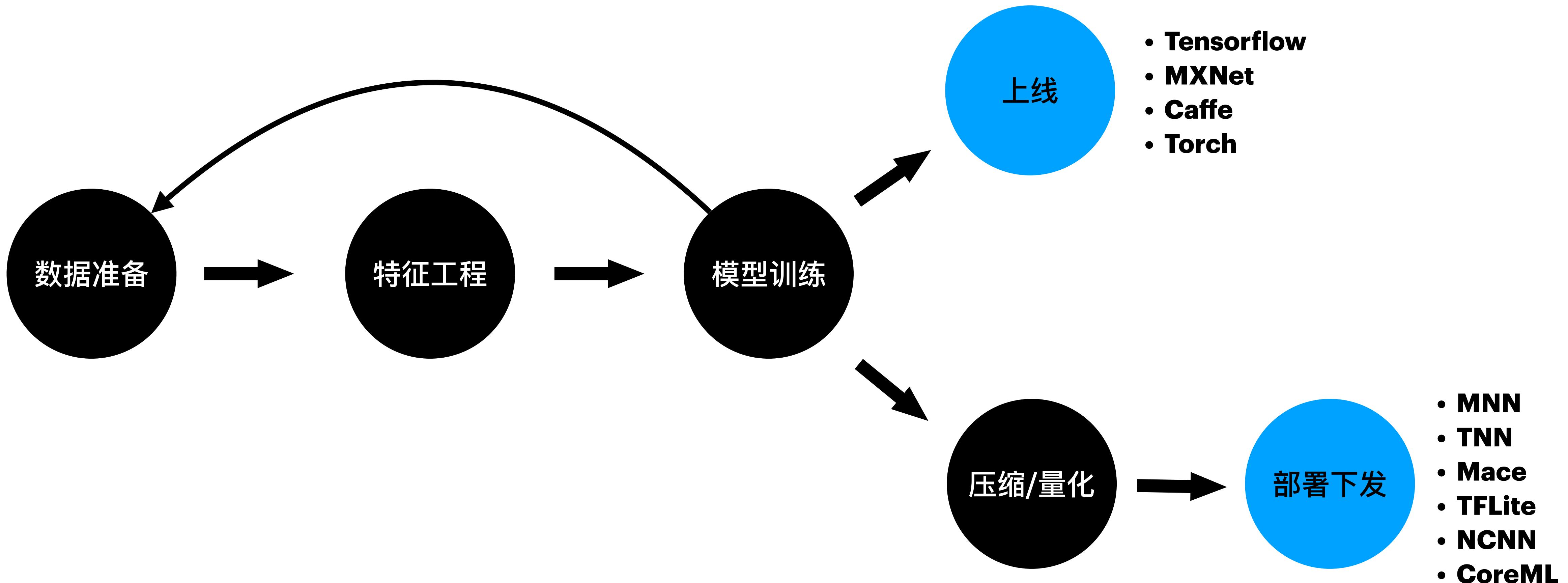
行为决策

- 识别
- 分割
- 风格迁移
- 超分

- 语音识别
- 翻译
- 情绪理解

- 端云联合推荐
- 疲劳度控制

常规流程



端智能的过去、现在与未来

今天先不讲

```
let result = model.predict( inputImage )
```



背后到底发生了什么？

用 Swift DIY 一个神经网络

解决手写字符识别问题

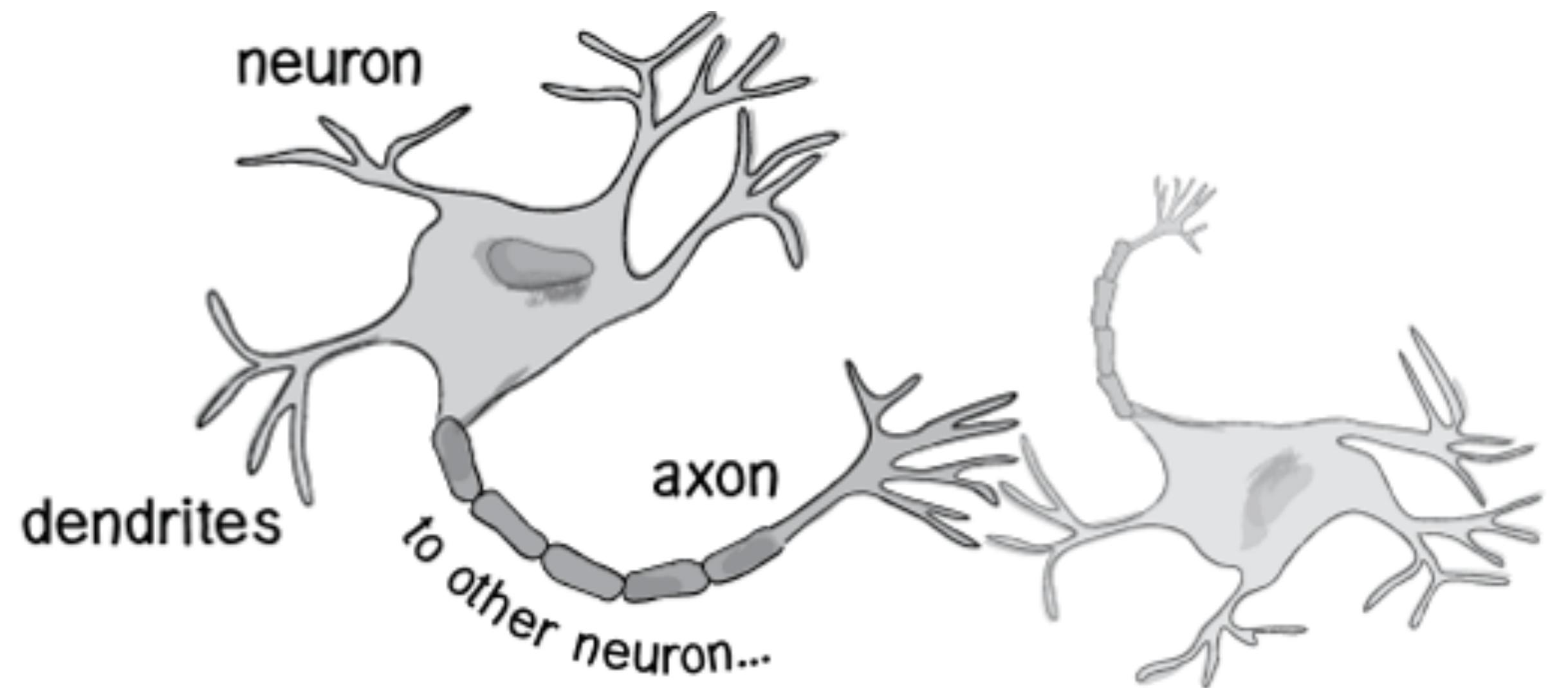
Roadmap

- Theory
- Design
- Code
- Demo



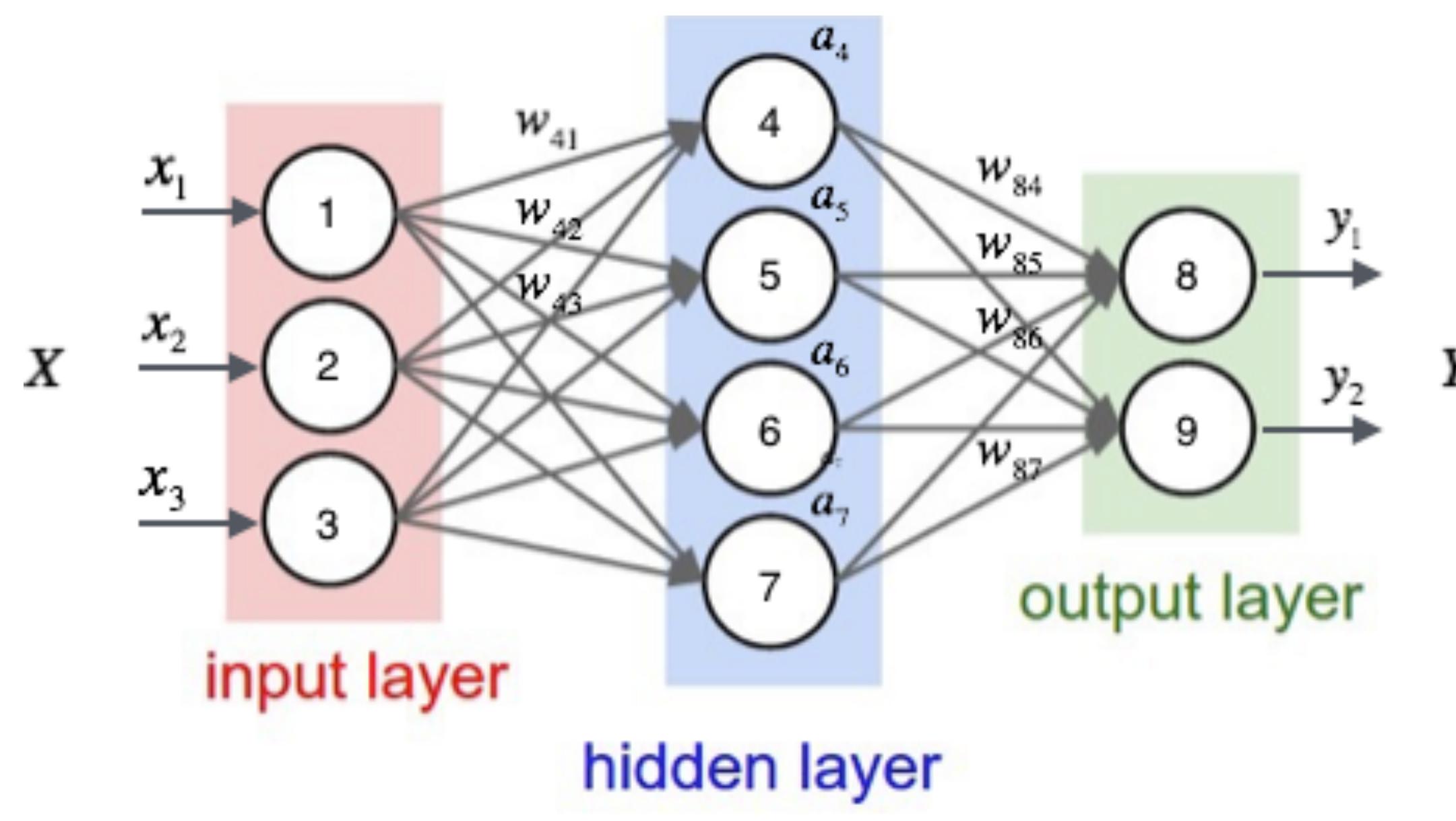
Theory

神经网络



- 神经元通过末梢接受输入信号
- 经过处理之后传递给邻接的神经元
- 不同神经元有不同的处理逻辑
- 无数的神经元构成神经网络
- 神经网络形成智能（记忆+推理）

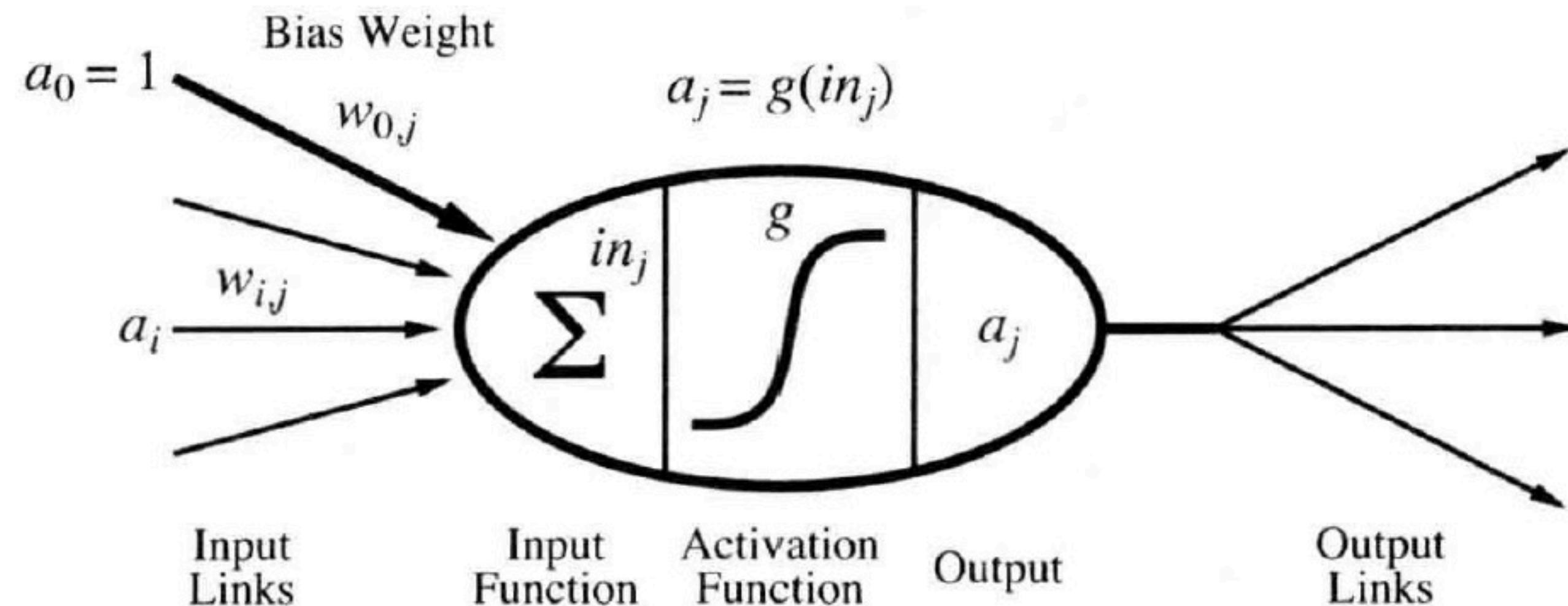
人工神经网络



- 按层组织神经元
- 输入层感知输入
- 输出层给出结果
- 一个或多个隐藏层
- 实现记忆+推理

以全连接网络为例

神经元



$$output = g\left(\sum_{i=0}^n a_i w_i\right)$$

- 数据存储
 - Weights: 来自上一层所有神经元的边
- 逻辑处理
 - Input function: hadamard 求和
 - Activation function: Sigmoid, ReLu
- 输入
 - $[a_0, a_1, \dots, a_n]$, n为上一层的神经元数
- 输出
 - 单一数值

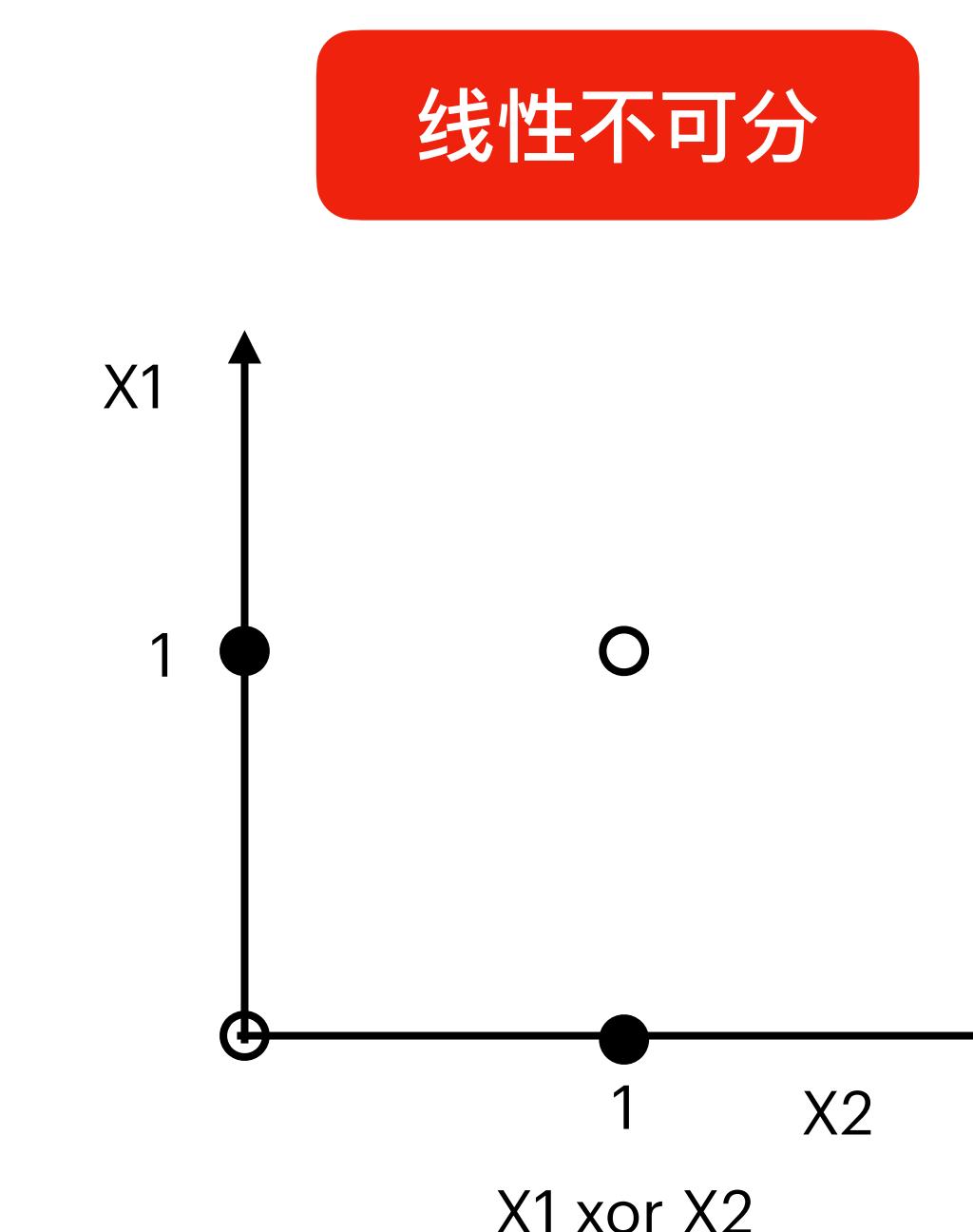
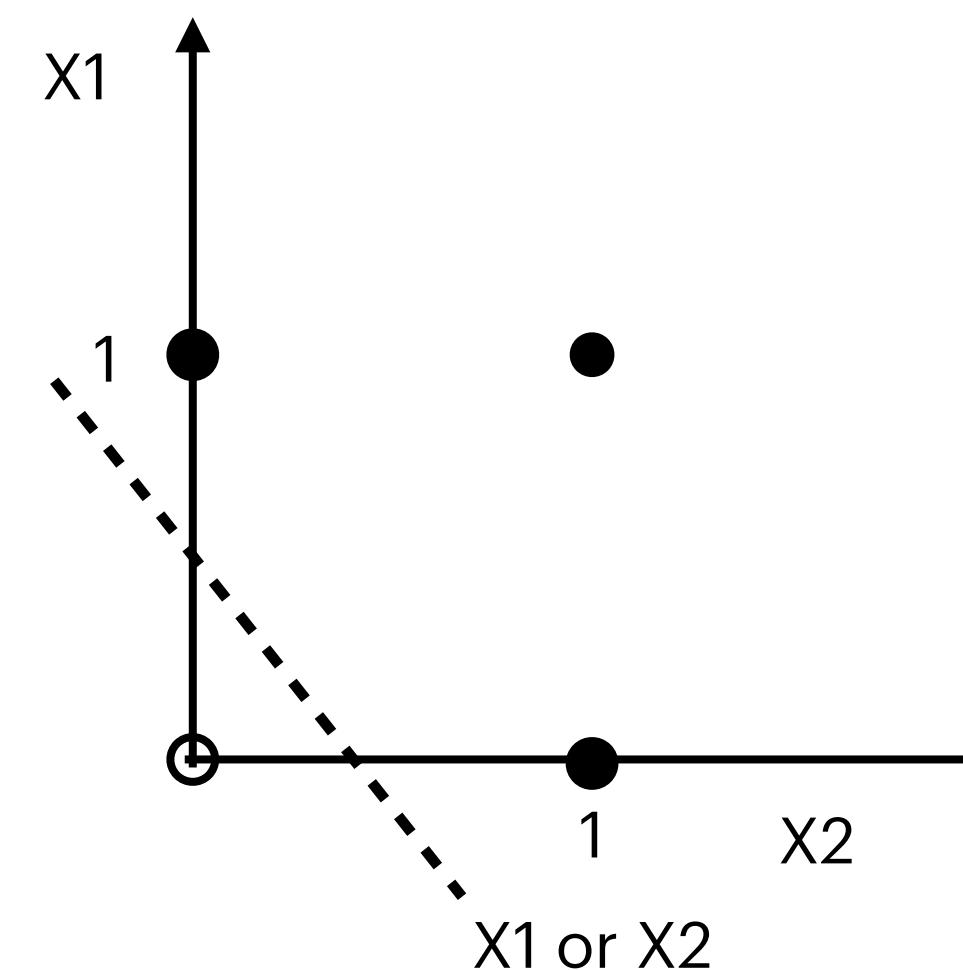
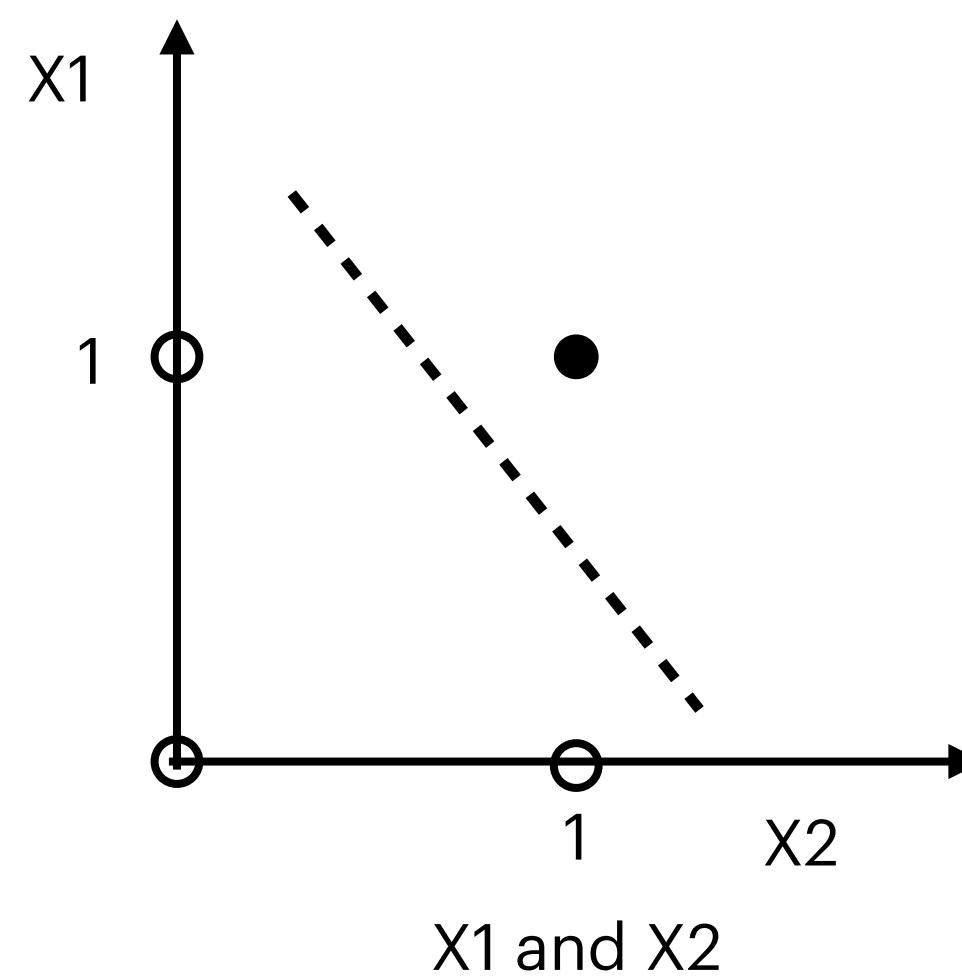
神经元

为什么需要激活函数

$$output = \sum_{i=0}^n a_i w_i \rightarrow y = W^T X$$

- 每一层的输出都是前一层输入的线性组合
- 最终整个网络的输出也是初始输入的线性组合

线性分类的局限：异或问题

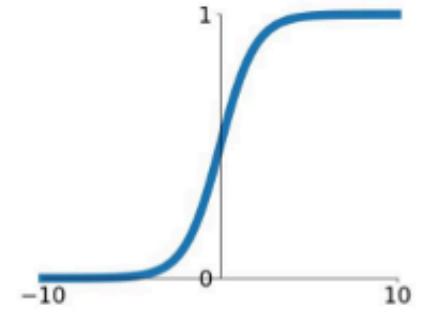


神经元

Activation Functions

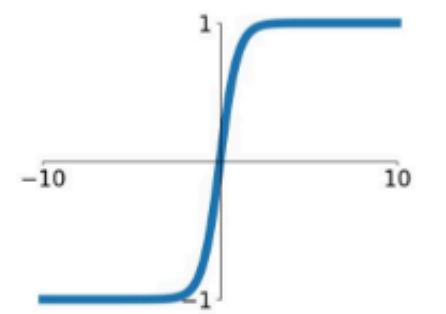
Sigmoid

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



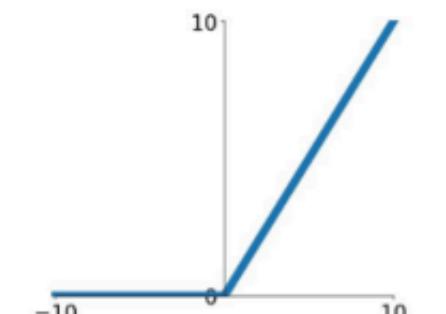
tanh

$$\tanh(x)$$



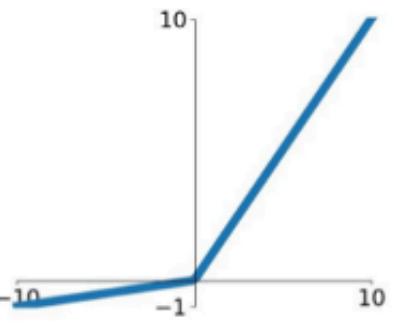
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

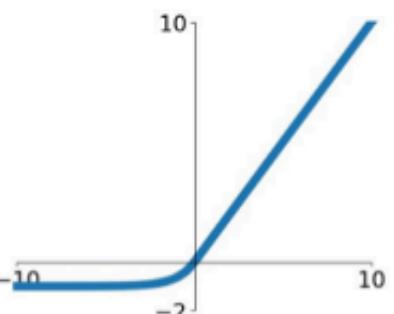


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

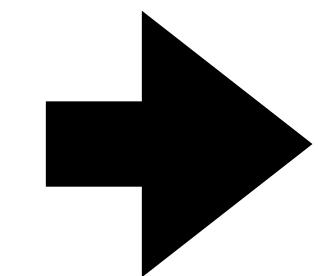
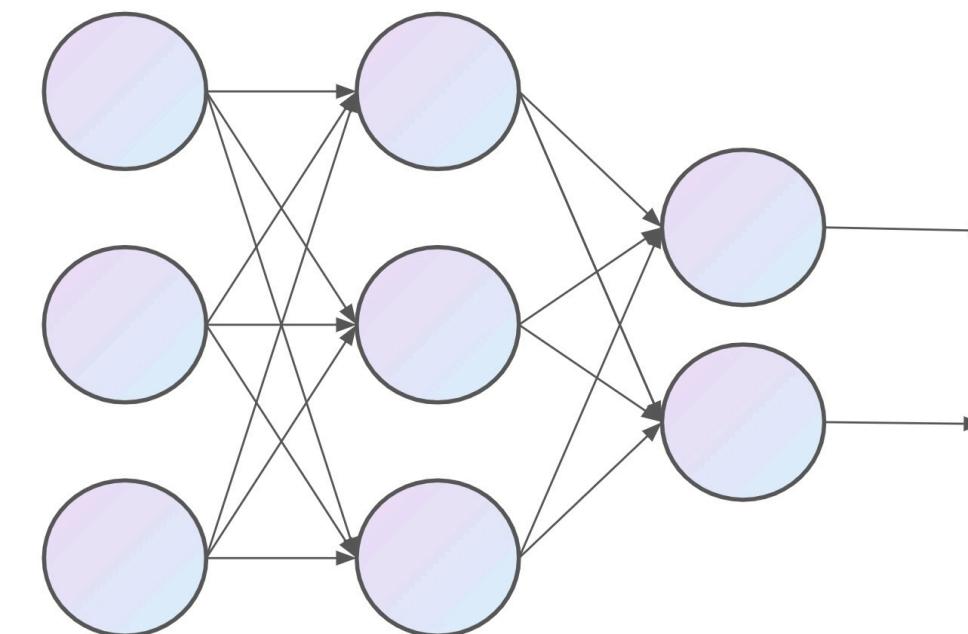
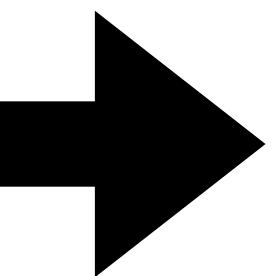
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



- 激活函数：想象的翅膀
- 通过对每个神经元的线性运算结果过一次激活函数，使其非线性化。进而使得神经网络理论上可以拟合任意计算过程
 - 毕竟 w 足够多
 - 关键要素：计算简单、求导简单
 - 最常用：Sigmoid 和 ReLU

手写字符识别

目标

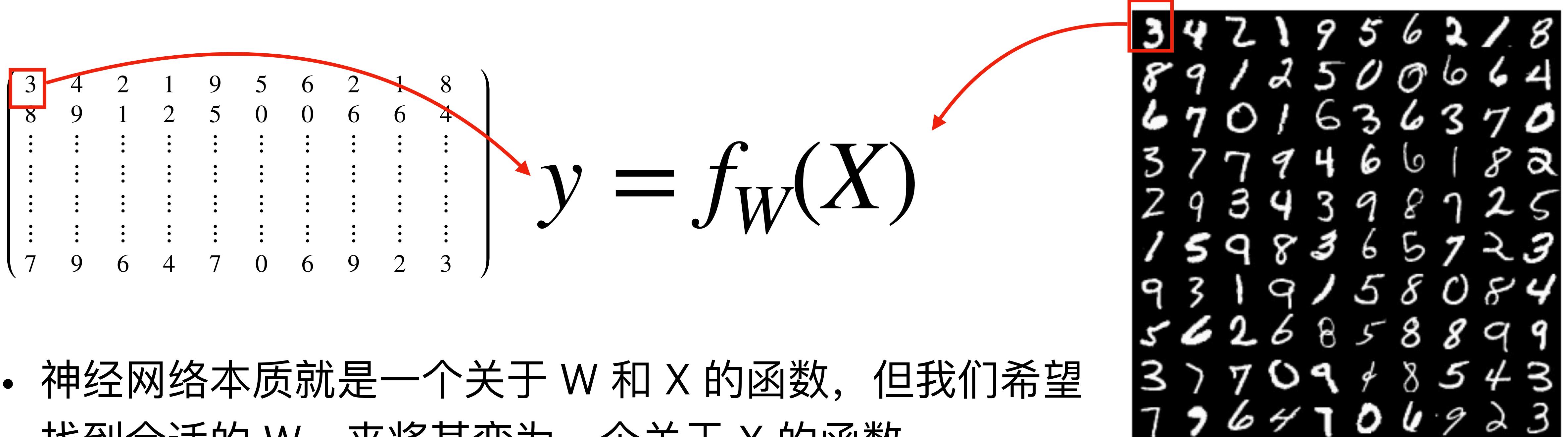


5

$$\begin{pmatrix} W_{11} & \cdots & W_{m1} \\ \vdots & \ddots & \vdots \\ W_{1n} & \cdots & W_{mk} \end{pmatrix}$$

如何找到合适的 W ?

找 W – 神经网络的训练



- 神经网络本质就是一个关于 W 和 X 的函数，但我们希望找到合适的 W ，来将其变为一个关于 X 的函数
- X 和 y 很容易构造
- 问题转化为：在有很多的 X 和 y 的情况下，如何估计 W

找 W – 神经网络的训练

如何评价某个 W 的好坏？

- 对于已知的 Ground Truth (X, y) , 如果 $f_{W'}(X) \approx y$, 我们就认为这个 W' 是好的。 $f_{W'}(X)$ 的结果我们称为预测值
- 如果能找到一个 W , 来对所有已知的 X 做出预测并使得预测值和实际值的差异最小, 这个 W 就是我们想要的
 - “差异”本身可以建模成一个关于 W 的可导函数。即可求出能令“差异”变小的 W 的迭代方向

$$Error(W) = (y - f_W(X))^2$$



Gradient Descent

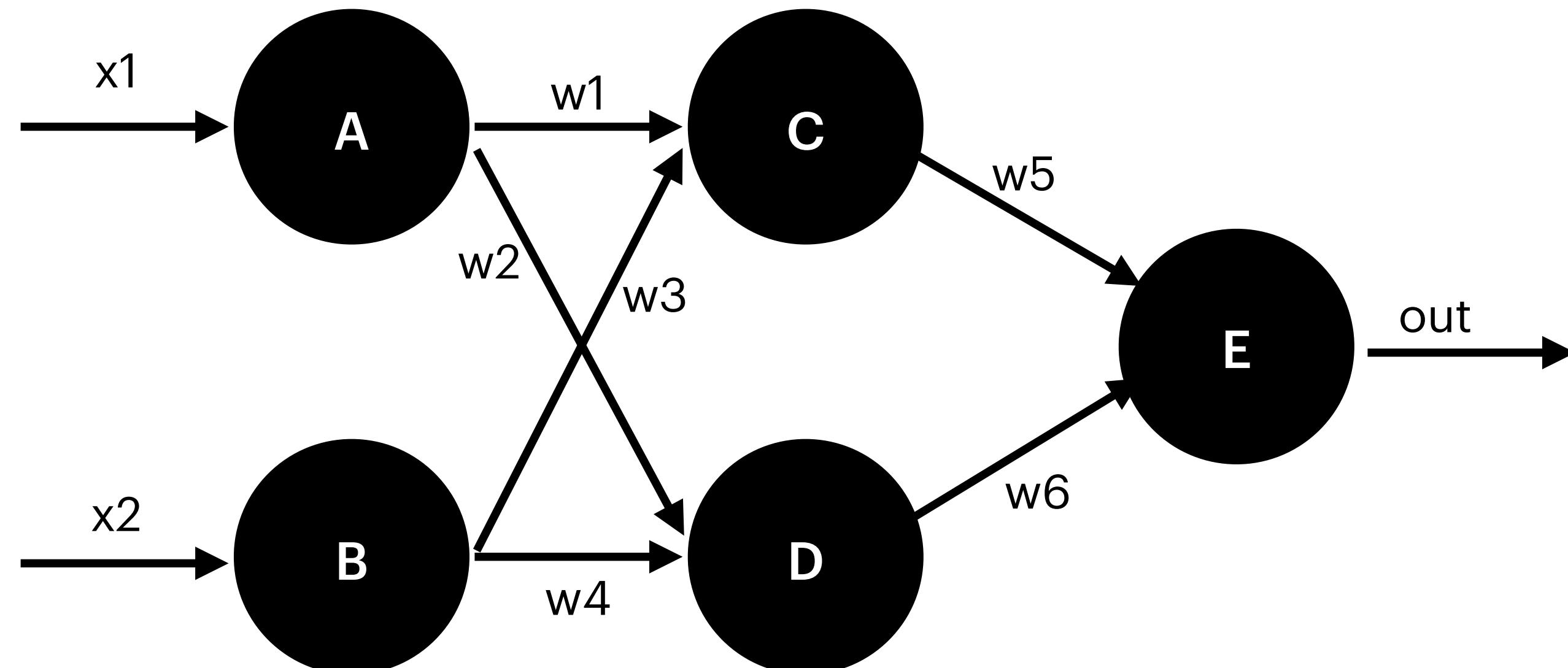
for X, y in Samples :

$$W = W + \alpha \frac{\partial Error(W)}{\partial W}$$

找 W – 神经网络的训练

- 神经网络是无数函数的非线性组合
- 如何求导?

$$Error = (y - out)^2$$



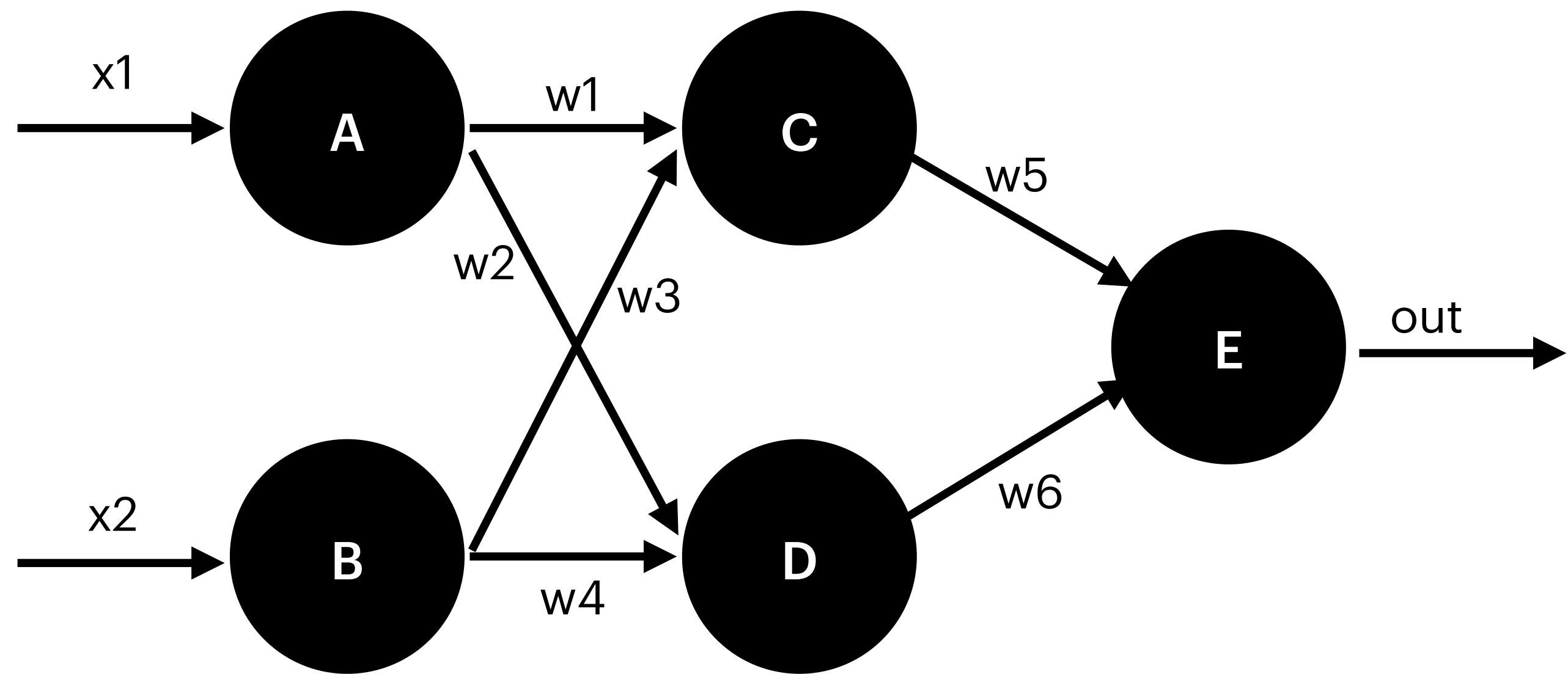
$$out = Sigmoid(C_{out} * w5 + D_{out} * w6)$$

$$\text{derivative}_{w5} = \frac{\partial Error}{\partial w5} = \frac{\partial Error}{\partial out} * \frac{\partial out}{\partial w5}$$

$$\text{derivative}_{w1} = \frac{\partial Error}{\partial w1} = \frac{\partial Error}{\partial out_E} * \frac{\partial out_E}{\partial out_C} * \frac{\partial out_C}{\partial w1}$$

找 W — 神经网络的训练

- 神经网络是无数函数的非线性组合
- 如何求导?



- 1. 从右向左，依次计算每一层节点边的权重偏导数
 - 2. 从左向右，依次更新每条边的权重
- $$w_i = w_i + \alpha * \frac{\partial Error}{\partial w_i}$$

Back Propagation

本质是基于导数链式法则的梯度下降

Design

训练数据

MNIST DataSet

行业常用的手写数字数据集

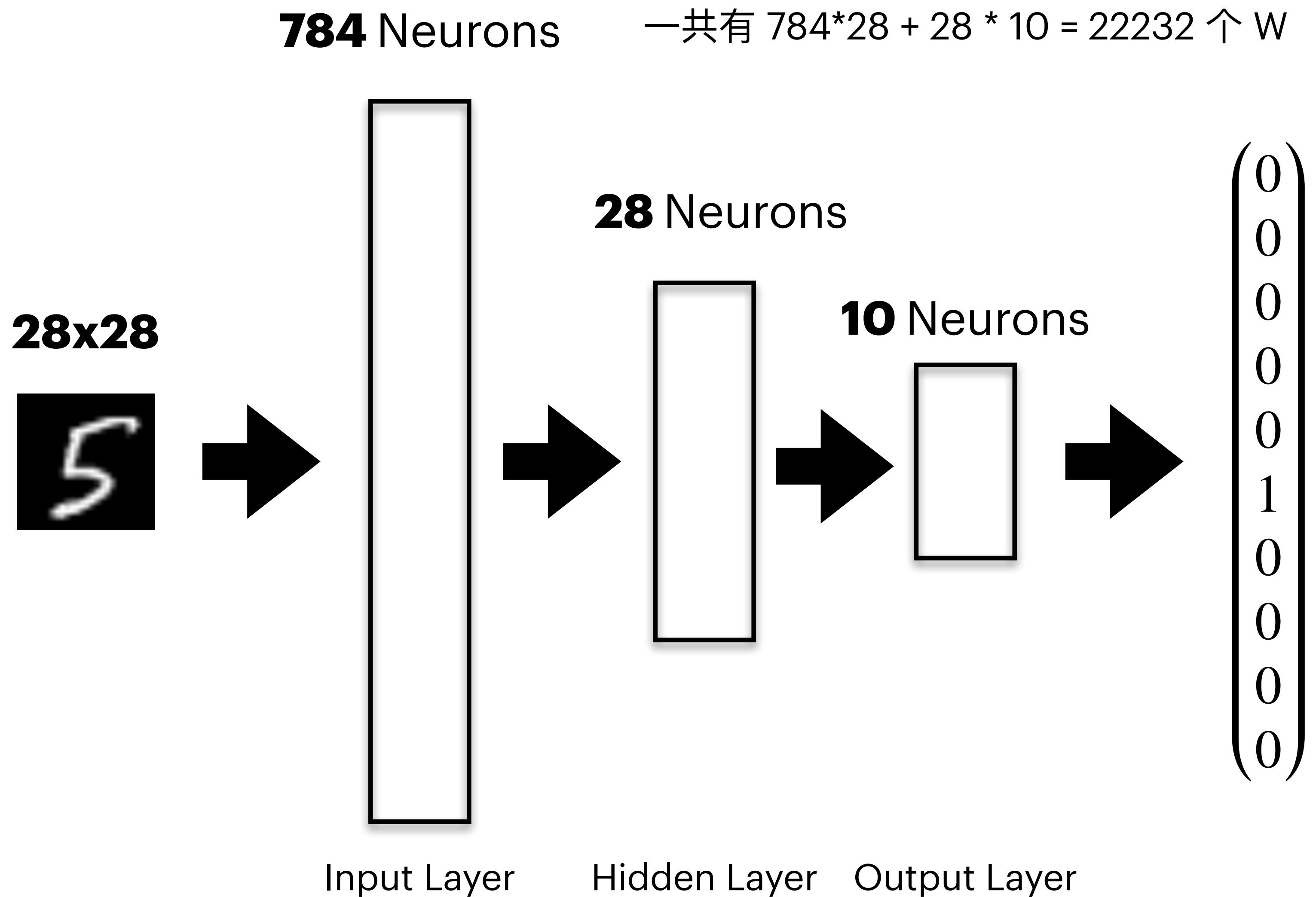
- train-images.idx3-ubyte
 - 60000张训练图片(28x28)
- train-labels.idx1-ubyte
 - 60000 个标签，对应以上训练图片
- t10k-images.idx3-ubyte
 - 10000 张测试图片(28x28)
- t10k-labels.idx1-ubyte
 - 10000 个标签，对应以上测试图片



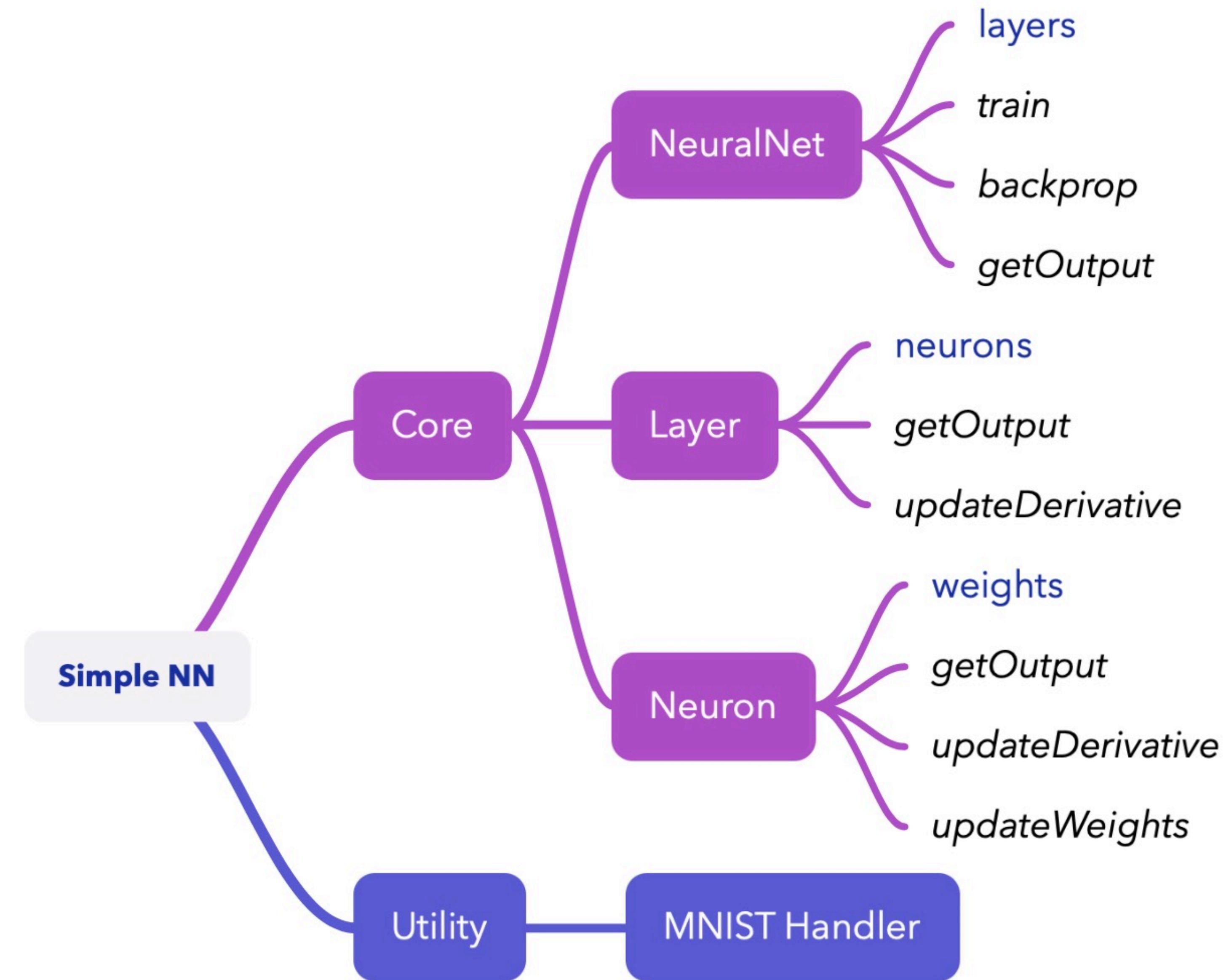
网络结构

原则

- 简单
 - 教学目的
 - Swift 闭环
- 有用
 - 网络确实能学到东西
 - 还 OK 的准确率



代码结构



Code

处理 MNIST

Image

```
stream.open()
var empty = [UInt8](repeating: 0, count: 16)
stream.read(&empty, maxLength: 16)
var image = [UInt8](repeating: 0, count: 784)
while stream.read(&image, maxLength: 784) == 784 {
    result.append(image)
}
```

Label

```
stream.open()
var empty = [UInt8](repeating: 0, count: 8)
stream.read(&empty, maxLength: 8)
var label: UInt8 = 0
while stream.read(&label, maxLength: 1) == 1 {
    result.append(label)
}
```

NeuralNet

```
class NeuralNet {  
    var layers : [Layer] = []  
  
    init(shape : [Int]) {  
        var prevLayer:Layer? = nil  
        for item in shape{  
            let layer = Layer(neuronConut: item, prevL: prevLayer)  
            prevLayer?.nextLayer = layer  
            prevLayer = layer  
            layers.append(layer)  
        }  
    }  
  
    func getOutputs(input: [Double]) → [Double] {  
        return layers.reduce(input) { r, l in  
            l.getOutputs(input: r);  
        }  
    }  
}
```

NeuralNet

反向计算所有
Neuron 的梯度

正向更新所有
Neuron 的 weight

```
func backprop(labels : [Double]){
    for i in (1..
```

Layer

```
class Layer {
    var neurons : [Neuron] = []
    var prevLayer : Layer?
    var nextLayer : Layer?
    init(neuronConut : Int, prevL : Layer?) {
        prevLayer = prevL
        for _ in 0...neuronConut-1{
            let n = Neuron()
            if let pl = prevLayer{
                n.inputWeights = [Double](repeating: 0, count:
pl.neurons.count)
                n.makeWeightsRandom()
            }
            neurons.append(n)
        }
    }

    func getOutputs(input:[Double])→[Double]{
        if prevLayer ≠ nil{
            return neurons.map { n in
                n.getOutputs(input: input)
            }
        } else {
            for i in (1..
```

Neuron - 基础结构

- **inputWeights**: 前一层所有 Neuron 连接这个 Neuron 的边的权重
- **grad**: 反向传播时计算的导数
- **learningRate**: 学习率
- **a**: 前向传播时, 点乘计算后的结果
- **s**: 前向传播时, 经过 sigmoid 之后的结果

```
class Neuron{  
    var inputWeights : [Double] = []  
    var grad : Double = 0.0  
    var learningRate : Double = 0.006  
    var a : Double = 0.0  
    var s : Double = 0.0  
  
    func makeWeightsRandom(){  
        inputWeights = inputWeights.map{_ in drand48() *  
2 - 1}  
    }  
  
    func getOutputs(input : [Double]) → Double{  
        a = dotProduct(x: input, y: inputWeights)  
        s = sigmoid(x: a)  
        return s  
    }  
}
```

Neuron - 计算单元

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

$$\frac{\partial \text{sigmoid}}{\partial x} = \text{sigmoid}(x) * (1 - \text{sigmoid}(x))$$

```
func dotProduct(x : [Double], y : [Double]) → Double{
    var result:Double = 0.0
    vDSP_dotprD(x, 1, y, 1, &result, vDSP_Length(x.count))
    return result
}

func sigmoid(x : Double) → Double{
    return 1.0 / (1.0 + exp(-x))
}

func dSigmoid(x : Double) → Double{
    let sig = sigmoid(x: x)
    return sig * (1 - sig)
}
```

Neuron - 反向传播

隐藏层梯度更新

$$grad_{l,i} = sigmoid'(a) * \sum_{j=0}^k w_{l+1,j,i} * grad_{l+1,j}$$

- $grad_{l,i}$: l层, 第 i 个神经元的梯度
- $w_{l+1,j,i}$: l+1层, 第 j 个神经元, 的第 i 个权重
- $grad_{l+1,j}$ l+1层, 第 j 个神经元的梯度

输出层梯度更新

$$grad_i = sigmoid'(a) * (label - s)$$

```
func updateGrad(truth : Double, nextLayer : Layer?, idx : Int) {  
    if let nl = nextLayer{  
        let weights = nl.neurons.map { n in  
            n.inputWeights[idx]  
        }  
        let grads = nl.neurons.map { n in  
            n.grad  
        }  
        let sum = dotProduct(x: weights, y:  
grads)  
        grad = dSigmoid(x:a) * sum  
    } else {  
        grad = dSigmoid(x:a) * (truth - s)  
    }  
}
```

Neuron - 正向更新权重

$$w_{l,j,i} = w_{l,j,i} + \alpha * output_{l-1,i} * grad_{l,j}$$

- $w_{l,j,i}$: l层, 第 j 个神经元的第 i 个权重
- $output_{l-1,i}$: l-1 层, 第 i 个神经元的输出(sigmoid 之后), 也就是我们的成员 s
- $grad_{l,j}$ |层, 第 j 个神经元的梯度

```
func updateWeight(prevLayer : Layer?){
    for i in (0..
```

Demo

Thanks

代码与 Demo:

https://github.com/aaaron7/light_nn

参考资料

Artificial Intelligence, A Modern Approach
Classic Computer Science Problems in Swift

UC(阿里智能信息事业群)急招各 level
iOS 工程师。
联系我内推, 光速安排面试。



End