

# 深度学习

人工智能，深度学习，机器学习——无论你在做什么，如果你对它不是很了解的话——去学习它，否则的话不用三年你就跟不上时代的潮流了。

——马克·库班

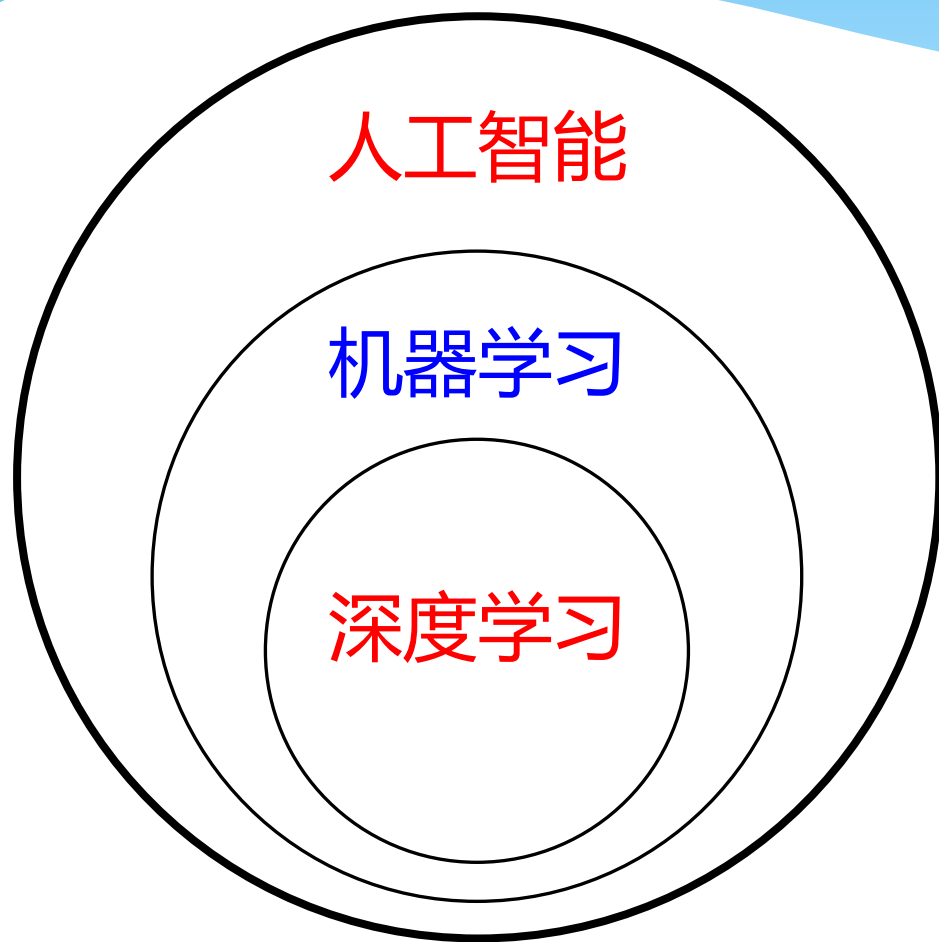
# 内容

- 一、深度学习发展历史
- 二、深度学习的应用
- 三、三位牛人
- 四、深度学习工具介绍和对比
- 五、感知器
- 六、线性单元
- 七、梯度下降优化算法
- 八、神经网络和反向传播算法

# 内容

- 一、深度学习发展历史
- 二、深度学习的应用
- 三、三位牛人
- 四、深度学习工具介绍和对比
- 五、感知器
- 六、线性单元
- 七、梯度下降优化算法
- 八、神经网络和反向传播算法

# 一、深度学习发展历史

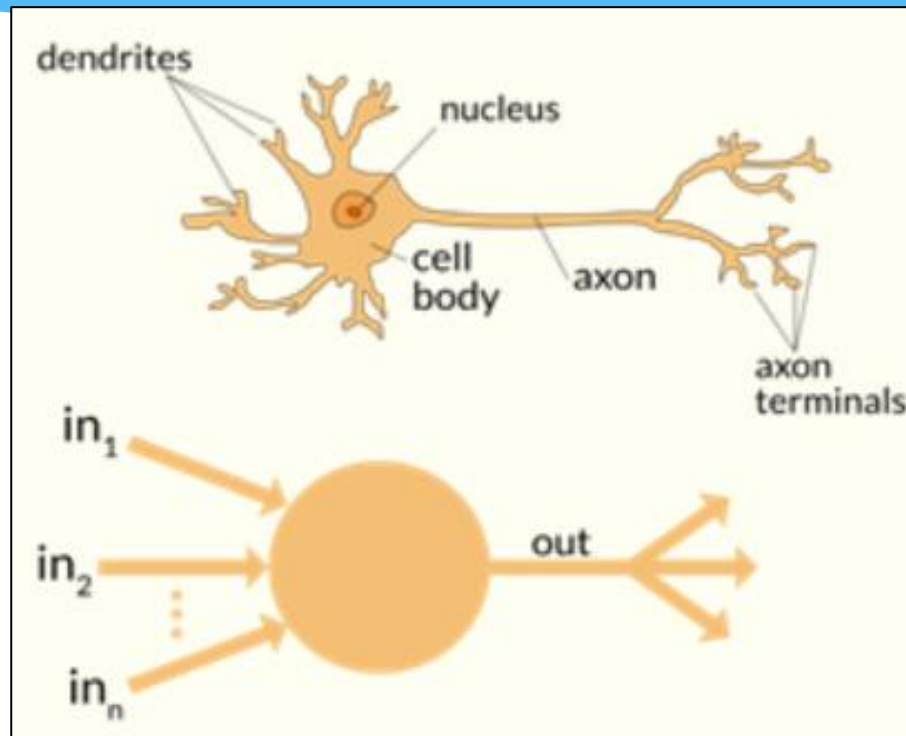


人工智能：  
Artificial Intelligence

机器学习：  
Machine Learning

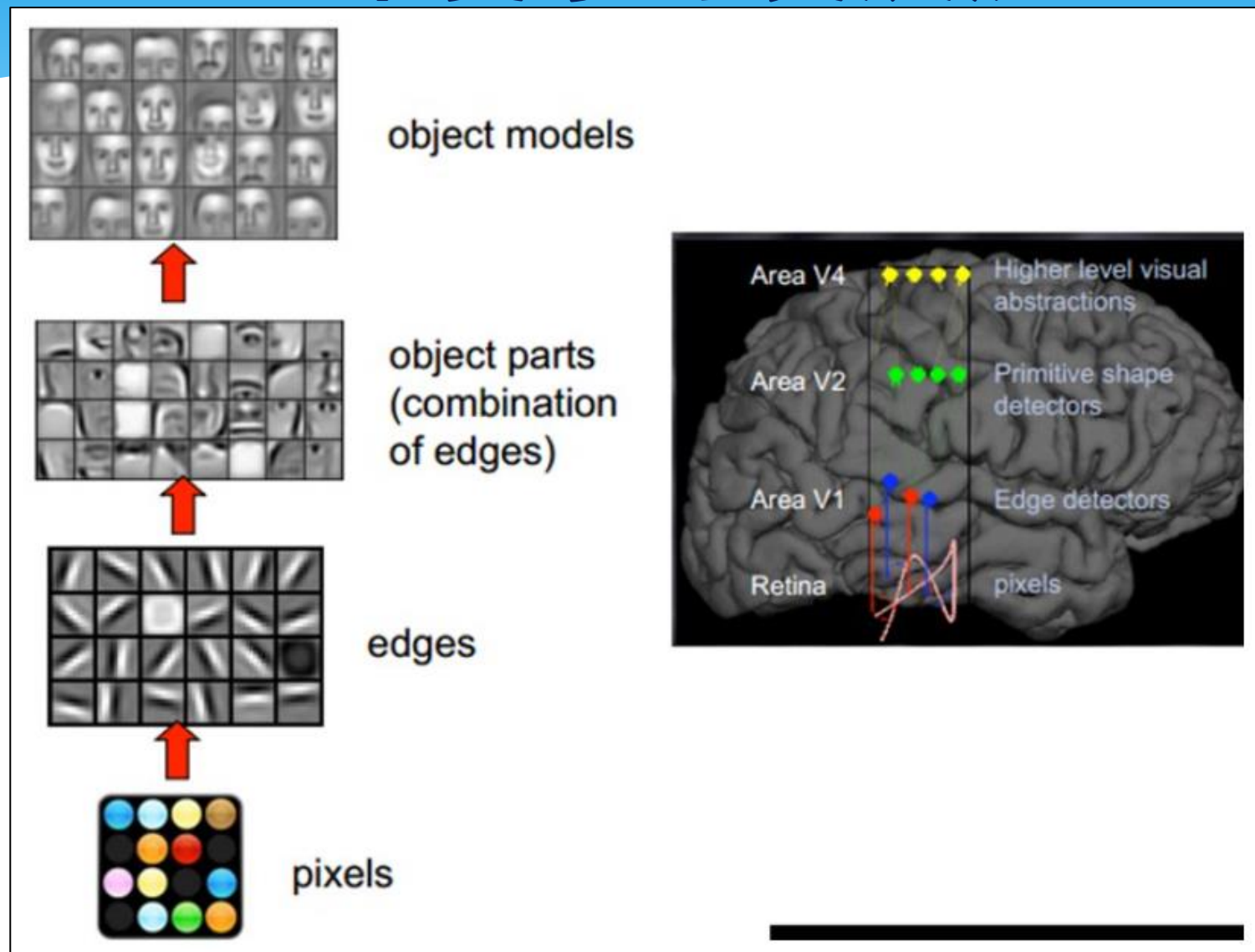
深度学习：  
Deep Learning

# 一、深度学习发展历史



1943年, Warren McCulloch教授和Walter Pitts教授在论文A logical calculus of the ideas immanent in nervous activity 提出神经网络的数学模型。

# 一、深度学习发展历史



# 一、深度学习发展历史

1958年，Frank Rosenblatt教授提出感知机模型（Perception）。感知机是首个可以根据样例数据来学习特征权重的模型。

1969年由Marvin Minsky教授和Seymour Paper教授出版的Perceptrons: An Introduction to Computational Geometry一书中，证明了感知机模型只能解决线性可分问题。

在书中，Marvin Minsky教授和Seymour Paper教授做出了“基于感知机的研究注定将失败”的结论。这导致了神经网络的第一次重大低潮期，在之后的十多年内，基于神经网络的研究几乎处于停滞状态。

# 一、深度学习发展历史

20世纪80年代末，第二波神经网络研究因分布式知识表达（distributed representation）和神经网络反向传播算法的提出而兴起。研究人员在降低训练神经网络的计算机复杂度上也取得了突破性成就。

David Everett Rumelhart教授、Geoffrey Everest Hinton教授和Ronald J. Williams教授于1986年在《自然》杂志上发表的Learning Representations by Back-propagating errors文章中首次提出了反向传播的算法（back propagation）。



# 一、深度学习发展历史

随着计算机性能的进一步提高，以及云计算、GPU的出现，到2010年左右，计算量已经不再是阻碍神经网络发展的问题。随着互联网+的发展，获取海量数据也不再困难。

2012年ImageNet举办的图像分类竞赛（ImageNet Large Scale Visual Recognition Challenge, ILSVRC）中，由Alex Krizhevsky教授实现的深度学习系统AlexNet赢得了冠军。

自此之后，深度学习（Deep Learning）作为深度神经网络的代名词被大家所熟知。

# 内容

- 一、深度学习发展历史
- 二、深度学习的应用
- 三、三位牛人
- 四、深度学习工具介绍和对比
- 五、感知器
- 六、线性单元
- 七、梯度下降优化算法
- 八、神经网络和反向传播算法

## 二、深度学习的应用

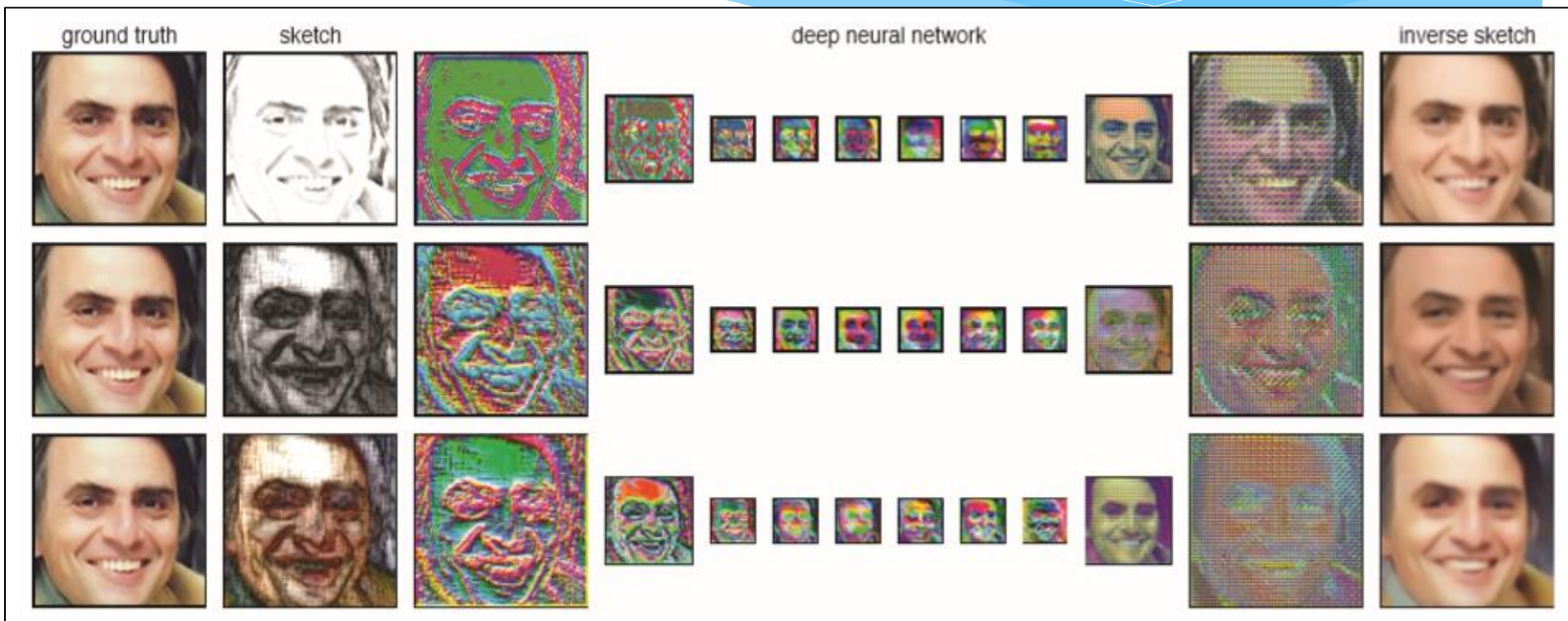


Face2Face: Real-time Face Capture and Reenactment of RGB Videos.

## 二、深度学习的应用



## 二、深度学习的应用

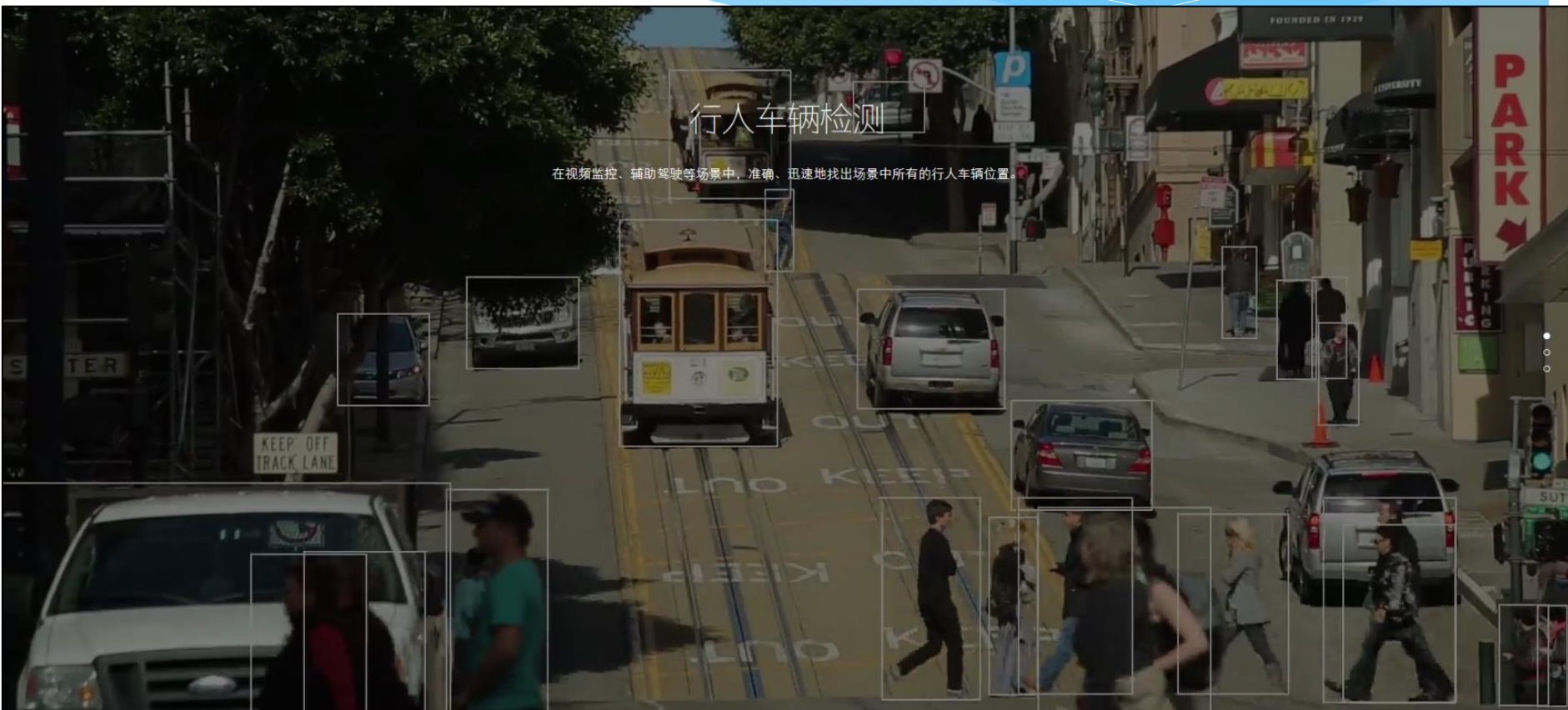




## 二、深度学习的应用

### 行人车辆检测

在视频监控、辅助驾驶等场景中，准确、迅速地找出场景中所有的行人车辆位置。



## 二、深度学习的应用

### 人脸关键点定位

微秒级别眼，口，鼻轮廓等人脸106个关键点定位。该技术可适应大角度侧脸，表情变化，遮挡，模糊，明暗变化等各种实际环境。



## 二、深度学习的应用

### 服装属性识别

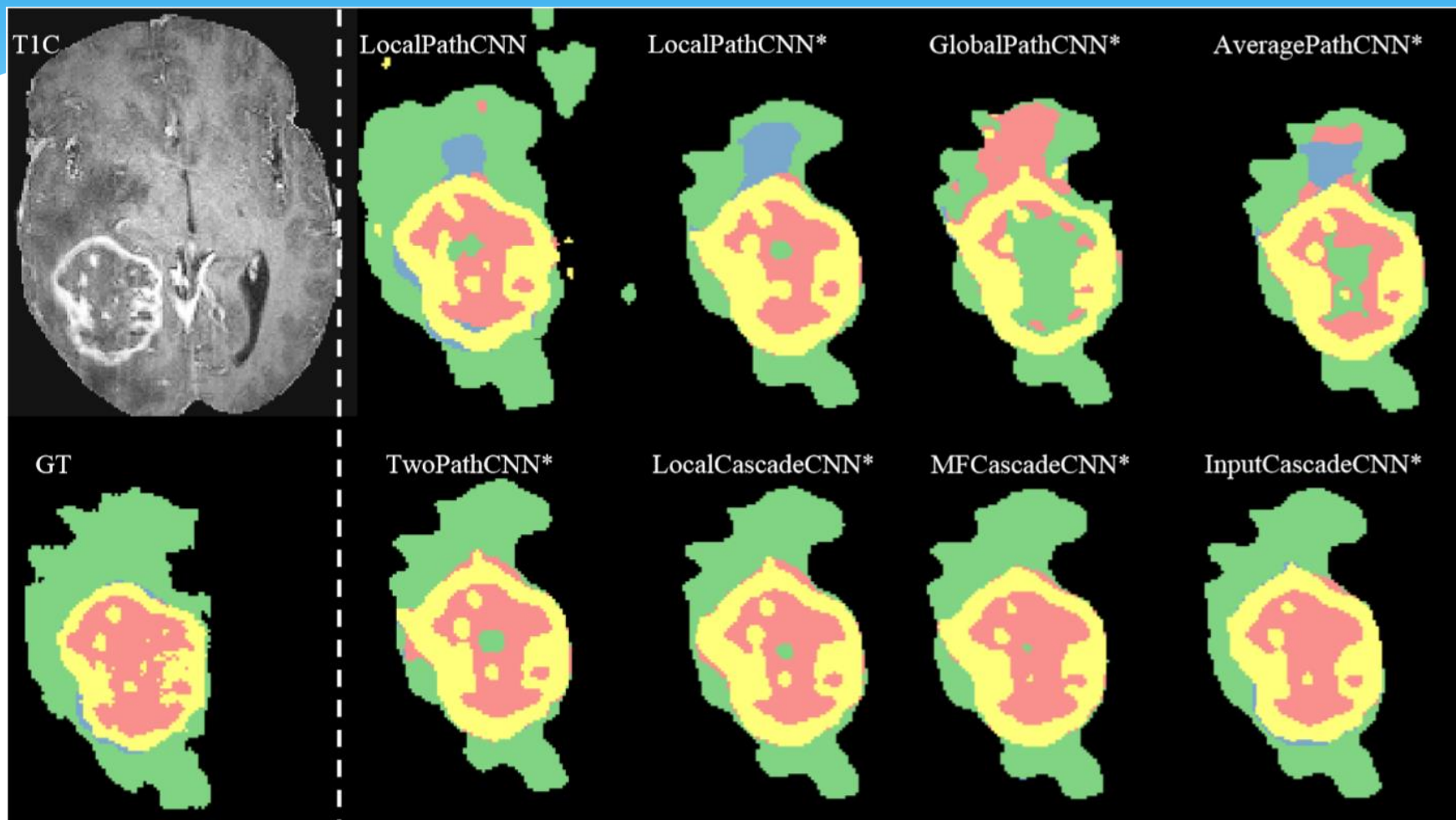
自动检测和识别图片、视频中的服饰，准确识别服饰品类、花纹、袖型、领型等特征，显著抵抗光照与姿态变化等干扰因素的影响。



衬衫、白色、长袖、尖领



## 二、深度学习的应用



Brain Tumor Segmentation

## 二、深度学习的应用

Speech Recognition:

Speech Recognition



## 二、深度学习的应用

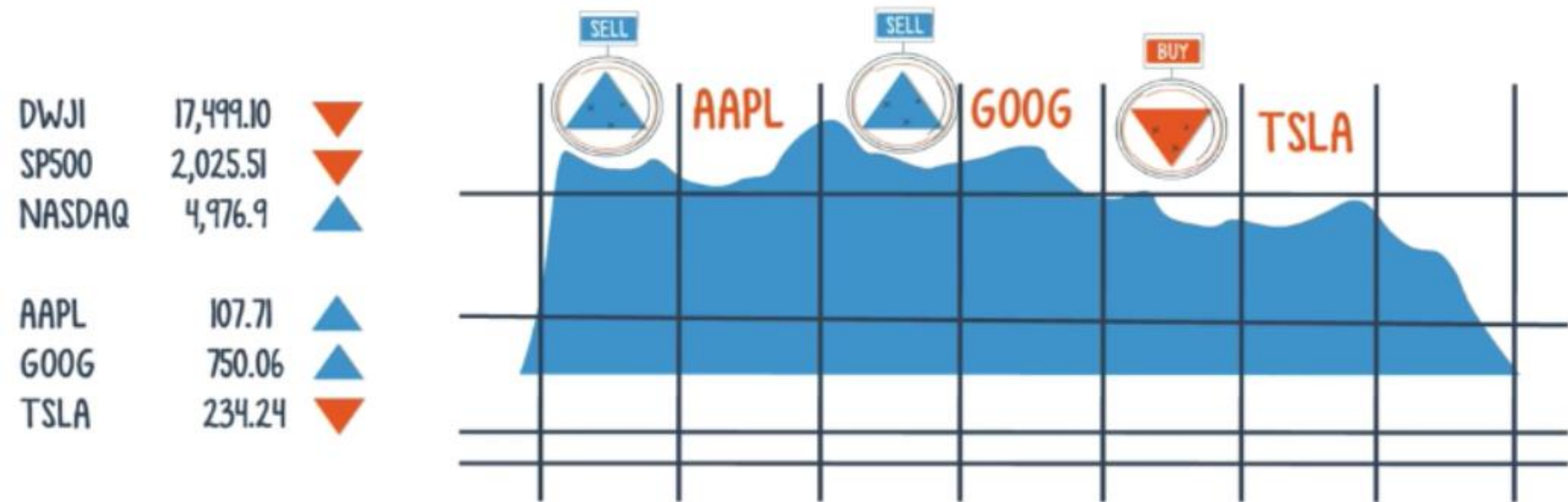
### Machine Translation



自然语言处理 (Natural Language Processing, NLP)

## 二、深度学习的应用

### Finance



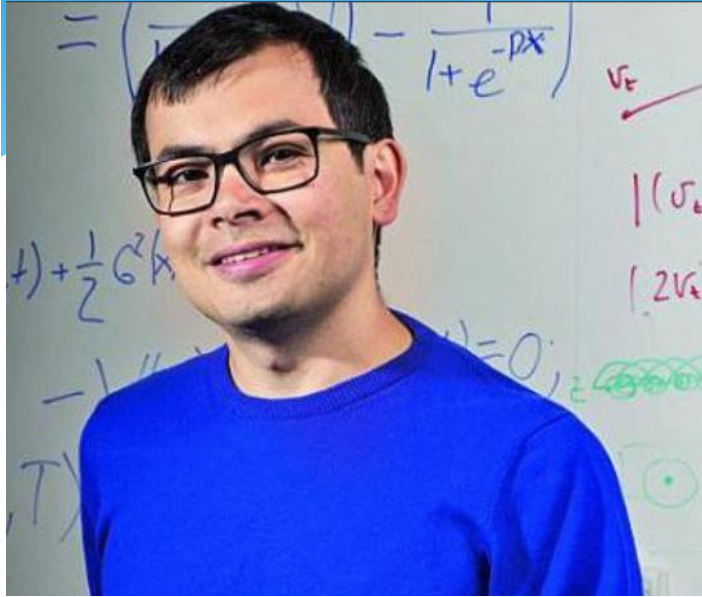
识别欺诈

## 二、深度学习的应用



人机博弈





戴密斯·哈萨比斯（Demis Hassabis），人工智能企业家，DeepMind Technologies公司创始人，人称“阿尔法围棋之父”。4岁开始下国际象棋，8岁自学编程，13岁获得国际象棋大师称号。17岁进入剑桥大学攻读计算机专业。

在大学里，他开始学习围棋。2005年进入伦敦大学学院攻读**神经科学博士**，选择大脑中的海马体作为研究对象。两年后，他证明了5位因为海马体受伤而患上健忘症的病人，在畅想未来时也会面临障碍，并凭这项研究入选《科学》杂志的“年度突破奖”。2011年创办DeepMind Technologies公司，以“解决智能”为公司的终极目标。

# 内容

- 一、深度学习发展历史
- 二、深度学习的应用
- 三、三位牛人
- 四、深度学习工具介绍和对比
- 五、感知器
- 六、线性单元
- 七、梯度下降优化算法
- 八、神经网络和反向传播算法

## 三、三位牛人



中文名:杨立昆  
Facebook 人工智能研究院院长

Yann LeCun, 计算机科学家, 被誉为“卷积网络之父”, 为**卷积神经网络** (CNN, Convolutional Neural Networks) 和图像识别领域做出了重要贡献, 以手写字体识别、图像压缩和人工智能硬件等主题发表过 190 多份论文, 拥有14项相关的美国专利。

他同Léon Bottou和Patrick Haffner等人一起创建了**DjVu图像压缩技术**, 同Léon Bottou一起开发了一种开源的**Lush语言**, 也是一位Lisp高手。

(Backpropagation, 简称BP) **反向传播**这种现阶段常用来训练人工神经网络的算法, 就是 LeCun 和其老师“神经网络之父”**Geoffrey Hinton** 等科学家于 20 世纪 80 年代中期提出的, 而后 LeCun 在贝尔实验室将 BP 应用于卷积神经网络中。



## 三、三位牛人



约书亚·本吉奥

Yoshua Bengio 是加拿大蒙特利尔大学 (Université de Montréal) 的终身教授，任教超过22年，是蒙特利尔大学机器学习研究所 (MILA) 的负责人。他的研究工作主要聚焦在高级机器学习方面，致力于用其解决人工智能问题，他是仅存的几个仍然全身心投入在学术界的深度学习教授之一。

Bengio 的主要贡献在于他对循环神经网络 (RNN, Recurrent Neural Networks) 的一系列推动，包括经典的 neural language model, gradient vanishing 的细致讨论, word2vec的雏形, 以及 machine translation。

Bengio是Deep Learning (《深度学习》) 一书的合著者 (另两位作者是Ian Goodfellow与Aaron Courville)，这本书被Elon Musk评价为“深度学习领域的权威教科书”。

## 三、三位牛人

Geoffrey Everest Hinton, 计算机学家、心理学家, 被称为“**神经网络之父**”、“**深度学习鼻祖**”。他研究了使用神经网络进行机器学习、记忆、感知和符号处理的方法, 并在这些领域发表了超过200篇论文。他是将**反向传播算法**引入多层神经网络训练的学者之一, 他还联合发明了**波尔兹曼机(Boltzmann machine)**。他对于神经网络的其它贡献包括: 分散表示 (distributed representation)、时延神经网络、专家混合系统 (mixtures of experts)、亥姆霍兹机 (Helmholtz machines)等。



**杰弗里·辛顿**  
多伦多大学计算机  
科学系教授

1970年, Hinton在英国剑桥大学获得文学学士学位, 主修实验**心理学**; 1978年, 他在爱丁堡大学获得哲学博士学位, 主修**人工智能**。此后Hinton曾在萨塞克斯大学、加州大学圣迭戈分校、剑桥大学、卡内基梅隆大学和伦敦大学学院工作。2013年3月, 谷歌收购Hinton的公司DNNResearch后, 他便随即加入谷歌, 直至目前一直在**Google Brain中担任要职**。

# 三、三位牛人



Yoshua Bengio (Yann LeCun在AT&T时的同事)



研究方向:  
(1)deep learning; (2)relational  
regression; (3)mobile robotics;  
(4)energy-based models;  
(5)invariant object recognition

## 三、三位牛人



**吴恩达**(1976-，英文名：**Andrew Ng**)，华裔美国人，是斯坦福大学计算机科学系和电子工程系副教授，人工智能实验室主任。吴恩达是人工智能和机器学习领域国际上最权威的学者之一。吴恩达也是在线教育平台Coursera的联合创始人(with Daphne Koller)。2014年5月16日，吴恩达加入百度，担任百度公司首席科学家，负责百度研究院的领导工作，尤其是Baidu Brain计划。

博士生导师是**Michael I. Jordan**

# 内容

- 一、深度学习发展历史
- 二、深度学习的应用
- 三、三位牛人
- 四、深度学习工具介绍和对比
- 五、感知器
- 六、线性单元
- 七、梯度下降优化算法
- 八、神经网络和反向传播算法



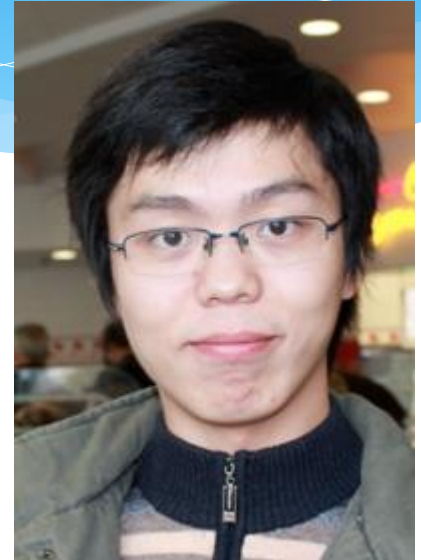
# 四、深度学习工具介绍和对比

## 1. Caffe

Caffe 的全称是 Convolutional Architecture for Fast Feature Embedding, 它是开源的, 核心语言是C++, 支持命令行、Python和Matlab接口。既可以在CPU上运行, 也可以在GPU上运行。

Caffe可以应用在视觉、语音识别、机器人、神经科学和天文学领域。

Caffe具有模块化、表示和实现分离、测试覆盖全面、接口丰富和预训练参考模型等特点。



贾扬清博士, 现就职于Facebook。博士毕业于UC Berkeley, 硕士和本科毕业于清华大学。开发Caffe, Caffe2等。

# 四、深度学习工具介绍和对比

## 2. Torch

Torch是一个广泛支持机器学习算法的科学计算框架，易于使用且高效，具有简单和快速的脚本语言LuaJIT和底层C/CUDA。

Torch广泛使用在许多学校的实验室，以及Google(DeepMind)、Twitter、NVIDIA、AMD、英特尔和许多其他公司。Torch能用于计算机视觉、信号处理、并行处理、图像、音视频。

Torch的核心是流行的神经网络，它使用简单的优化库，同时具有最大的灵活性，实现复杂的神经网络和拓扑结构。

## 四、深度学习工具介绍和对比

### 3. Theano

Theano在深度学习框架中是祖师级的存在。它的开发始于2007年。早期开发者包括传奇人物 Yoshua Bengio 和 Ian Goodfellow。

Theano是由LISA(现MILA)在加拿大魁北克的蒙特利大学开发的基于Python的深度学习框架，专门用于定义、优化、求值数学表达式，其效率比较高，适用于多维数组。

随着 Tensorflow 在谷歌的支持下强势崛起，Theano 日渐式微，使用的人越来越少。**这过程中的标志性事件是：创始者之一的 Ian Goodfellow 放弃 Theano 转去谷歌开发 Tensorflow。**



# 四、深度学习工具介绍和对比

## 4. Keras

Keras是由纯Python编写的基于Theano/Tensorflow的深度学习框架。

Keras是一个高层神经网络API，支持快速实验，能够把你的idea迅速转换为结果，如果有如下需求，可以优先选择Keras：

- a) 简易和快速的原型设计（Keras具有高度模块化、极简和可扩充特性）；
- b) 支持CNN和RNN，或二者的结合；
- c) 无缝CPU和GPU切换。

## 四、深度学习工具介绍和对比

### 5. MXNet

MXNet是亚马逊（Amazon）选择的深度学习库。它拥有类似于 Theano 和 TensorFlow 的数据流图，为多 GPU 提供了良好的配置。

MXNet是一个轻量化分布式可移植的深度学习计算平台，可以在你可以想象的任何硬件上运行（包括手机）。

MXNet使用C++实现，并提供C风格的头文件。支持Python、Scala、C++和R。

# 四、深度学习工具介绍和对比

## 6. TensorFlow

TensorFlow是谷歌基于DistBelief进行研发的第二代人工智能学习系统，其命名来源于本身的运行原理。

Tensor(张量)意味着N维数组，Flow(流)意味着基于数据流图的计算，TensorFlow为张量从流图的一端流动到另一端计算过程。

TensorFlow是将复杂的数据结构传输至人工智能神经网络中进行分析 and 处理过程的系统。

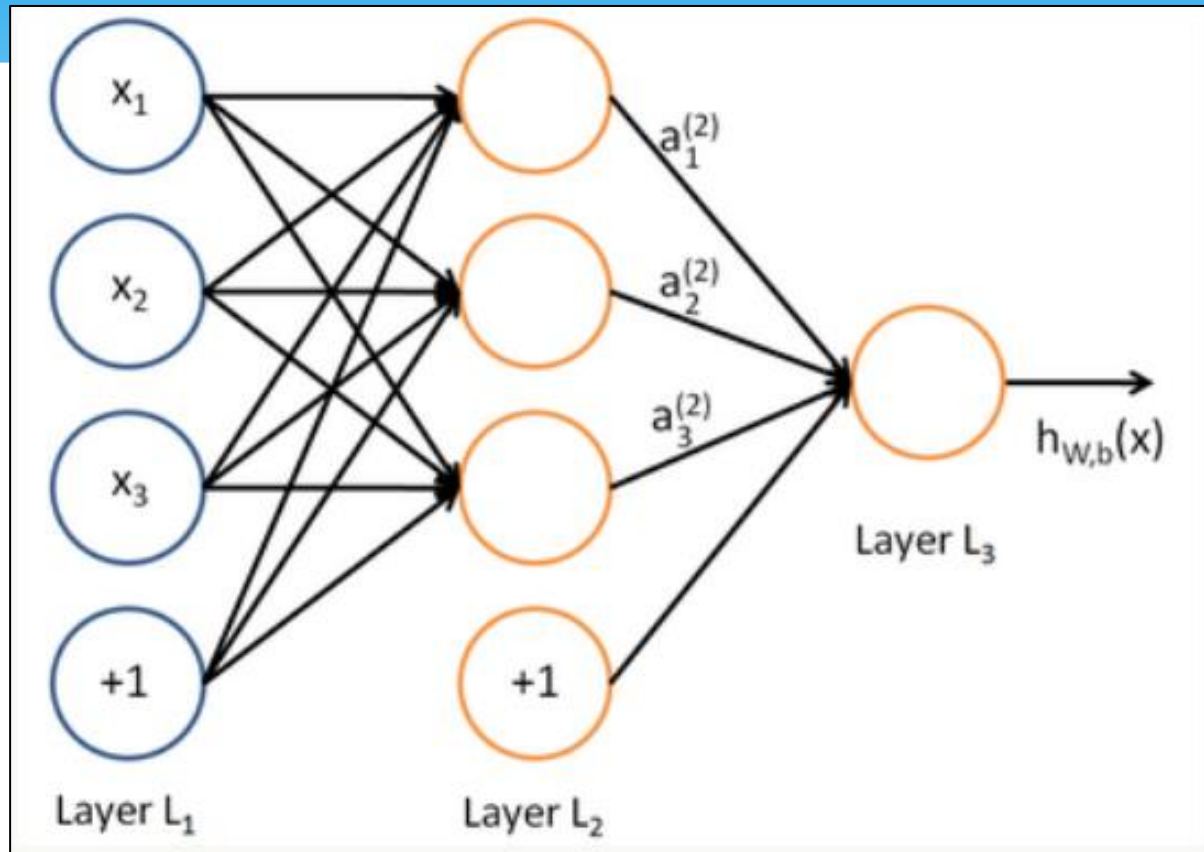
TensorFlow有合理的C++使用界面，也有易用的Python使用界面来构建和执行你的Graphs。可以使用Go, Java, Lua, Javascript, 或者是R。

# 内容

- 一、深度学习发展历史
- 二、深度学习的应用
- 三、三位牛人
- 四、深度学习工具介绍和对比
- 五、感知器
- 六、线性单元
- 七、梯度下降优化算法
- 八、神经网络和反向传播算法

## 五、感知器

最左边的层叫做输入层，这层负责接收输入数据。



最右边的层叫输出层，可以从这层获取神经网络输出数据。

每个圆圈都是一个神经元，每条线表示神经元之间的连接。层与层之间的神经元有连接，而层内之间的神经元没有连接。输入层和输出层之间的层叫做隐藏层。

## 五、感知器

隐藏层比较多（大于2）的神经网络叫做深度神经网络。

深度学习，就是使用深层架构（比如，深度神经网络）的机器学习方法。

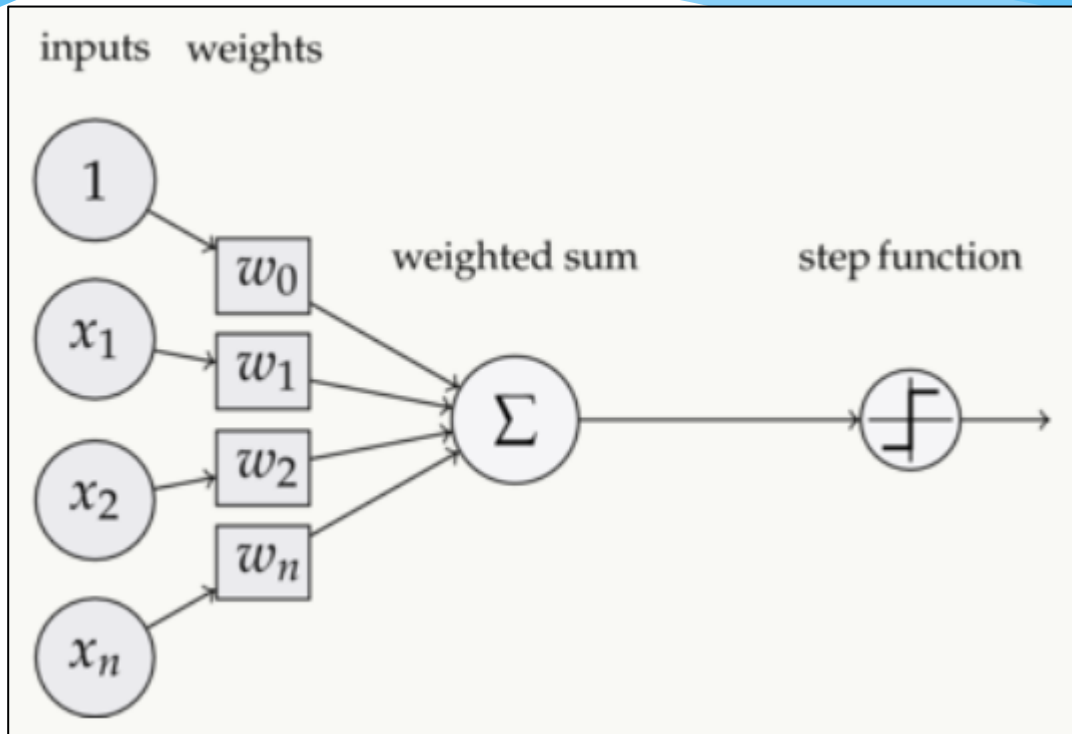
仅有一个隐藏层的神经网络就能拟合任何一个函数，但是它需要很多很多的神经元。（浅层网络）

深层网络用少得多的神经元就能拟合同样的函数。

为了拟合一个函数，要么使用一个浅而宽的网络，要么使用一个深而窄的网络。而后者往往更节约资源。

深层网络也有劣势，就是它不太容易训练。

# 五、感知器



感知器有如下组成部分：

- 1) 输入权值；
- 2) 激活函数；
- 3) 输出。

# 五、感知器

**输入权值** 一个感知器可以接收多个输入 $(x_1, x_2, \dots, x_n \mid x_i \in \mathfrak{R})$ ，每个输入上有一个**权值** $w_i \in \mathfrak{R}$ ，此外还有一个**偏置项** $b \in \mathfrak{R}$ ，就是上图中的 $w_0$ 。

**激活函数** 感知器的激活函数可以有很多选择，比如我们可以选择下面这个**阶跃函数** $f$ 来作为激活函数：

$$f(z) = \begin{cases} 1 & z > 0 \\ 0 & otherwise \end{cases}$$

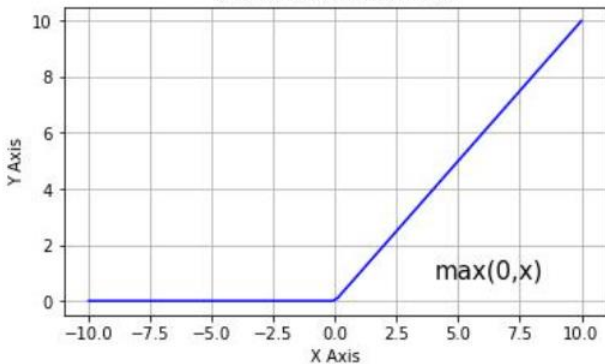
**输出** 感知器的输出由下面这个公式来计算

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

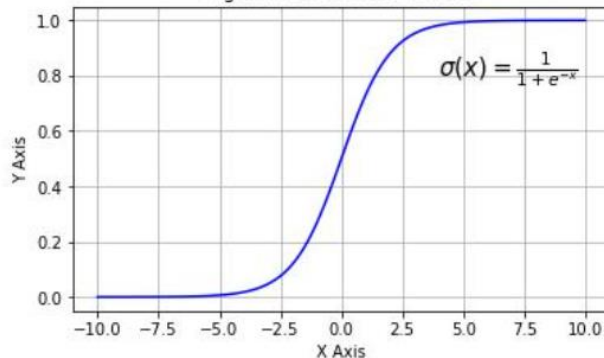


## 常见激活函数:

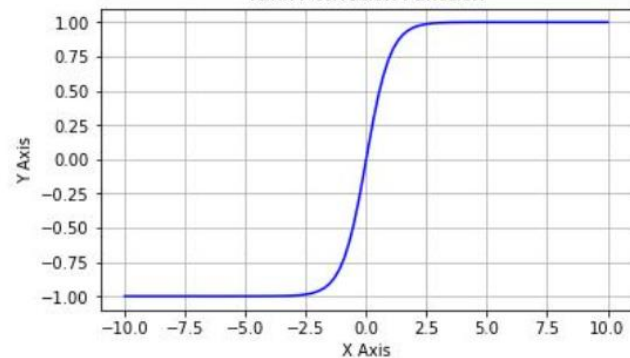
ReLU Activation Function



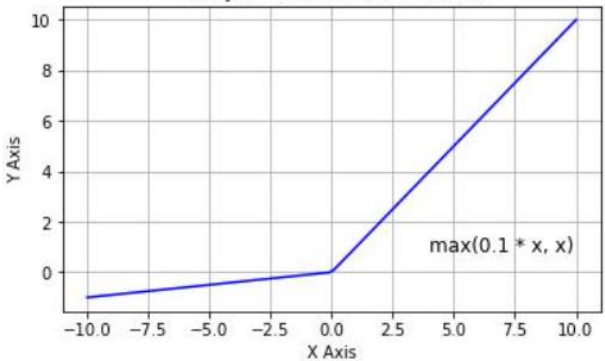
Sigmoid Activation Function



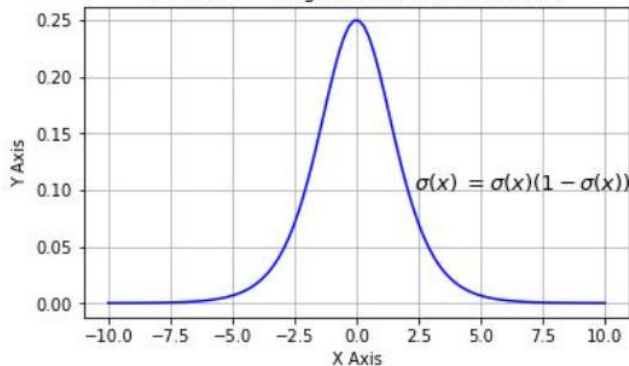
Tanh Activation Function



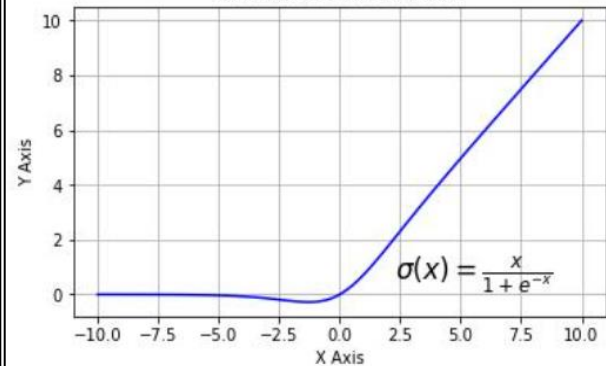
Leaky ReLU Activation Function



Derivative of Sigmoid Activation Function



Swish Activation Function



## 五、感知器

用感知器实现and函数

X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

## 五、感知器

令  $w_1 = 0.1$ ,  $w_2 = 0.2$ ,  $b = -0.2$ , 激活函数  $f$  采用前面的阶跃函数。

验证如下：

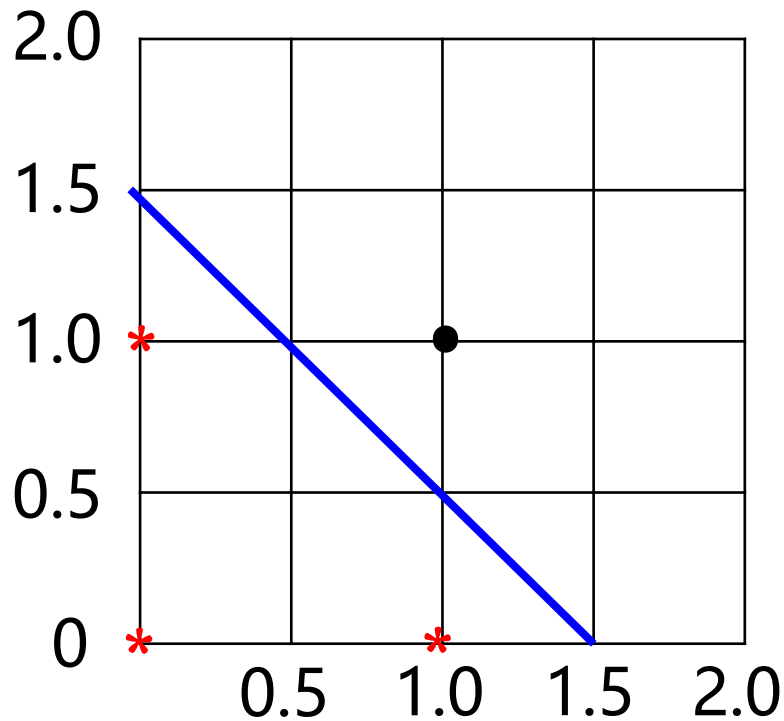
1) 当  $x_1 = 0$ ,  $x_2 = 0$  (第一行), 计算如下:

$$\begin{aligned} y &= f(w * x + b) = f(w_1 * x_1 + w_2 * x_2 + b) \\ &= f(0.1 * 0 + 0.2 * 0 - 0.2) = f(-0.2) = 0 \end{aligned}$$

2) 当  $x_1 = 1$ ,  $x_2 = 1$  (第四行), 计算如下:

$$\begin{aligned} y &= f(w * x + b) = f(w_1 * x_1 + w_2 * x_2 + b) \\ &= f(0.1 * 1 + 0.2 * 1 - 0.2) = f(0.1) = 1 \end{aligned}$$

# 五、感知器



感知器不仅仅能实现简单的布尔运算，它可以拟合任何的线性函数。任何线性分类或线性回归问题都可以用感知器来解决。

布尔运算可以看作是二分类问题。

感知器不能实现异或运算。因为异或运算不是线性的，无法用一条直线把0和1分开。

# 五、感知器

## 如何计算每个变量的权重和偏置量？

将权重项和偏置项初始化为0，然后，利用下面的感知器规则迭代的修改 $w_i$ 和 $b$ ，直到训练完成。

$$w_i \leftarrow w_i + \Delta w_i$$

$$b \leftarrow b + \Delta b$$

其中，

$$\Delta w_i = \eta(t - y)x_i$$

$$\Delta b = \eta(t - y)$$

$$y = f(w \cdot x + b)$$

$t$ 是训练样本的实际值，一般称之为label。

$y$ 是感知器的输出值。

$\eta$ 是一个称为学习速率的常数，其作用是控制每一步调整权的幅度。

```
from functools import reduce
```

```
class Perceptron( object ):
```

```
    def __init__(self, input_num, activator):
```

```
        """
```

```
        初始化感知器，设置输入参数的个数，以及激活函数。
```

```
        激活函数的类型为idouble -> double
```

```
        """
```

```
        self.activator = activator
```

```
        #权重向量初始化为0
```

```
        self.weights = [0.0 for _ in range( input_num )]
```

```
        #偏置项初始化为0
```

```
        self.bias = 0.0
```

```
    def __str__(self):
```

```
        """
```

```
        打印学习到的权重、偏置项
```

```
        """
```

```
        return 'weights\t:%s\nbias\t:%f\n' % (self.weights, self.bias)
```

```
def predict(self, input_vec):
```

```
    """ 输入向量, 输出感知器的计算结果 """
    #把input_vec[x1, x2, x3,...] 和 weights[w1, w2, w3, ...]打包在一起
    #变成[(x1, w1), (x2, w2), (x3, w3), ...]
    #然后利用map函数计算[x1 * w1, x2 * w2, x3 * w3]
    #最后利用reduce求和
    return self.activator (
        reduce(lambda a, b: a + b,
            list(map(lambda tp: tp[0] * tp[1],
                zip(input_vec, self.weights))),
            0.0) + self.bias)
```

```
def train(self, input_vecs, labels, iteration, rate):
```

```
    """
    输入训练数据:一组向量、与每个向量对应的label;以及训练迭代次数、学习率
    """
    for i in range(iteration):
        self._one_iteration(input_vecs, labels, rate)
```



```
def _one_iteration(self, input_vecs, labels, rate):
```

```
    """
```

```
    一次迭代，把所有的训练数据过一遍
```

```
    """
```

```
    #把输入和输出打包在一起，成为样本的列表[(input_vec, label), ...]
```

```
    #而每个训练样本是(input_vec, label)
```

```
    samples = zip(input_vecs, labels)
```

```
    #对每个样本，按照感知器规则更新权重
```

```
    for (input_vec, label) in samples:
```

```
        #计算感知器在当前权重下的输出
```

```
        output = self.predict(input_vec)
```

```
        #更新权重
```

```
        self._update_weights(input_vec, output, label, rate)
```

```
def _update_weights(self, input_vec, output, label, rate):
    """
    按照感知器规则更新权重
    """
    #把input_vec[x1, x2, x3, ...]和weights[w1, w2, w3,...]打包在一起
    #变成[(x1, w1), (x2, w2), (x3, w3), ...]
    #然后利用感知器规则更新权重
    delta = label - output
    #self.weights = map(lambda x,w: w + rate * delta * x,
    #                    zip(input_vec, self.weights))
    self.weights = list(map(lambda tp: tp[1] + rate * delta * tp[0],
                           zip(input_vec, self.weights)))

    #更新bias
    self.bias += rate * delta
    print("_update_weights()-----")
    print("label - output = delta:", label, output, delta)
    print("weights ", self.weights)
    print("bias", self.bias)
```

```
def f(x):
```

```
    """ 定义激活函数f """
```

```
    return 1 if x > 0 else 0
```

```
def get_training_dataset():
```

```
    """ 基于and真值表构建训练数据 """
```

```
    #构建训练数据
```

```
    #输入向量列表
```

```
    input_vecs = [[1, 1], [0, 0], [1, 0], [0, 1]]
```

```
    #期望的输出列表，注意要与输入一一对应
```

```
    #[1, 1] -> 1, [0, 0] -> 0, [1, 0] -> 0, [0, 1] -> 0
```

```
    labels = [1, 0, 0, 0]
```

```
    return input_vecs, labels
```

```
def train_and_perceptron():
```

```
    """ 使用and真值表训练感知器 """
```

```
    #创建感知器，输入参数个数为2（因为and是二元函数），激活函数为f
```

```
    p = Perceptron(2, f)
```

```
    #训练，迭代10次，学习速度为0.1
```

```
    input_vecs, labels = get_training_dataset()
```

```
    p.train(input_vecs, labels, 10, 0.1)
```

```
    #返回训练好的感知器
```

```
    return p
```

```
if __name__ == '__main__':
```

```
    #训练and感知器
```

```
    and_perception = train_and_perceptron()
```

```
    #打印训练获得的权重
```

```
    print (and_perception)
```

```
    #测试
```

```
    print ('1 and 1 = %d' % and_perception.predict([1, 1]))
```

```
    print ('0 and 0 = %d' % and_perception.predict([0, 0]))
```

```
    print ('1 and 0 = %d' % and_perception.predict([1, 0]))
```

```
    print ('0 and 1 = %d' % and_perception.predict([0, 1]))
```

最终结果:

```
weights    :[0.1, 0.2]  
bias       :-0.200000
```

1 and 1 = 1

0 and 0 = 0

1 and 0 = 0

0 and 1 = 0

# 内容

- 一、深度学习发展历史
- 二、深度学习的应用
- 三、三位牛人
- 四、深度学习工具介绍和对比
- 五、感知器
- 六、线性单元
- 七、梯度下降优化算法
- 八、神经网络和反向传播算法

# 六、线性单元

## 1. 线性单元的模型

当面对的数据集不是线性可分时，感知器可能无法收敛，则无法完成感知器的训练。

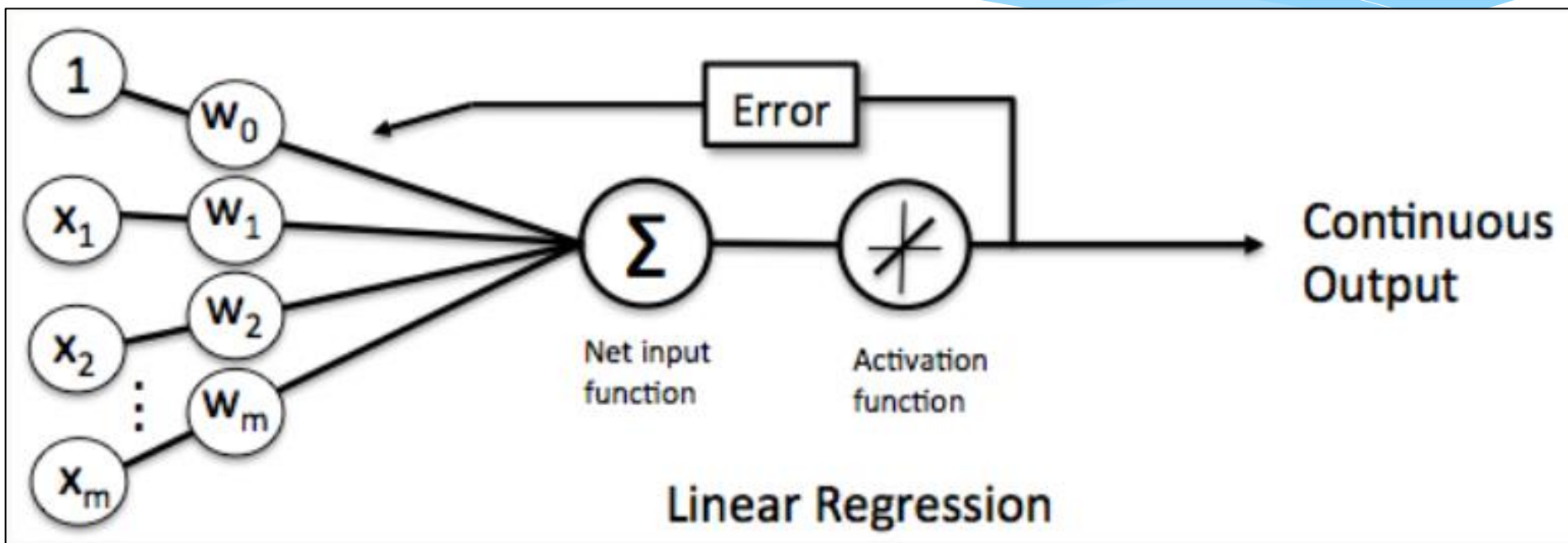
为了解决这个问题，可以采用可导的线性函数来替代感知器的阶跃函数，这样感知器就叫做线性单元。

为了简单起见，设置线性单元的激活函数如下所示：

$$f(x) = x$$



## 六、线性单元



**模型**，根据输入 $x$ 预测输出 $y$ 的算法。

## 六、线性单元

### 举例：

设某人的工作年限为 $x$ ，则他的月薪为 $y$ ，我们可以建立一个模型预测他的收入。比如： $y = h(x) = w * x + b$ 。

函数 $h(x)$ 叫做假设，而 $w$ 、 $b$ 是模型的参数。

假设参数 $w=1000$ ，参数 $b=500$ ，某人的工作年限是5年，根据模型预测他的月薪如下：

$$y = h(x) = 1000 * 5 + 500 = 5500(\text{元})$$

## 六、线性单元

如果考虑工作年限、行业、公司、职级等信息，预测出来的工资可能会更加准确。

假设某人工作了5年，IT行业，百度工作，职级T6，用特征向量来表示 $x = (5, IT, 百度, T6)$ 。每个特征都有对应的权重（参数），则4个特征对应的参数分别为 $w_1, w_2, w_3, w_4$ 。

根据模型可以得到如下公式：

$$y = h(x) = w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4 + b$$

令  $b = w_0 * x_0$  其中  $x_0 = 1$

$$y = h(x) = w_0 * x_0 + w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4$$

## 六、线性单元

$$y = h(x) = w_0 * x_0 + w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4$$



$$y = h(x) = w^T x$$

上面的模型就叫做**线性模型**，因为输出 $y$ 就是输入特征 $x_1, x_2, x_3, \dots$ 的线性组合。

# 六、线性单元

## 2. 线性单元的目标函数

对于一个样本，我们知道它的特征 $x$ ，以及标记 $y$ 。根据 $h(x)$ 计算得到输出 $\bar{y}$ 。我们希望模型计算出来的 $y$ 和 $\bar{y}$ 越接近越好。

如下公式可以表示单个样本的误差：

$$e = \frac{1}{2} (y - \bar{y})^2$$

# 六、线性单元

训练数据有N个，则所有样本的误差之和，用E表示：

$$\begin{aligned} E &= e^{(1)} + e^{(2)} + e^{(3)} + \dots + e^{(n)} = \sum_{i=1}^n e^{(i)} \\ &= \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)})^2 \\ &= \frac{1}{2} \sum_{i=1}^n (y^{(i)} - h(x^{(i)}))^2 \end{aligned}$$

$$E(w) = \frac{1}{2} \sum_{i=1}^n (y^{(i)} - w^T x^{(i)})^2$$

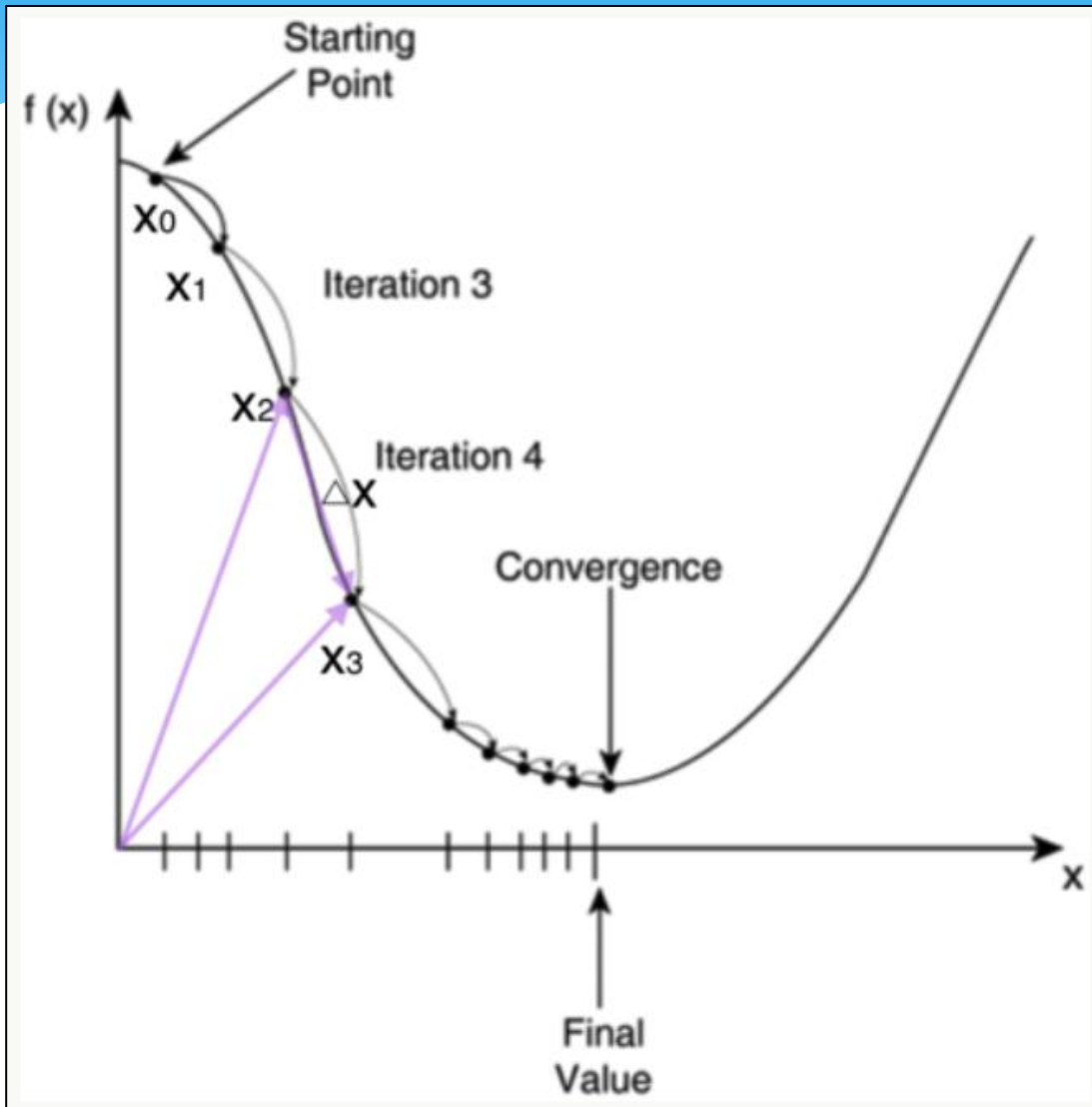
**E(w)称为目标函数。**

# 内容

- 一、深度学习发展历史
- 二、深度学习的应用
- 三、三位牛人
- 四、深度学习工具介绍和对比
- 五、感知器
- 六、线性单元
- 七、梯度下降优化算法
- 八、神经网络和反向传播算法



# 七、梯度下降优化算法



计算机凭借强大的  
计算能力能够计算  
得到近似最小值。

## 七、梯度下降优化算法

梯度是一个向量，它指向函数值上升最快的方向。梯度的反方向就是函数值下降最快的方向。

每次沿着梯度相反方向去修改 $x$ 的值，当然就能走到函数的最小值附近。

之所以是最小值附近而不是最小值那个点，是因为每次移动的步长不会那么恰到好处，有可能最后一次迭代走远了，越过了最小值那个点。

# 七、梯度下降优化算法

梯度下降算法的公式：

$$x_{new} = x_{old} - \eta \nabla f(x)$$

$\nabla$  是梯度算子， $\nabla f(x)$ 就是指 $f(x)$ 的梯度。 $\eta$  是步长， 也称作学习速率

# 七、梯度下降优化算法

$$E(w) = \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)})^2 \quad (\text{式 2})$$

梯度下降算法

$$w_{new} = w_{old} - \eta \nabla E(w)$$

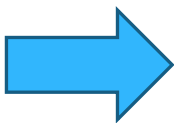
梯度上升算法

$$w_{new} = w_{old} + \eta \nabla E(w)$$

# 七、梯度下降优化算法

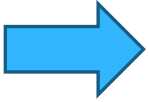
$$w_{new} = w_{old} - \eta \nabla E(w)$$

$$\nabla E(w) = - \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)}) x^{(i)}$$


$$w_{new} = w_{old} + \eta \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)}) x^{(i)} \quad (\text{式 3})$$

# 七、梯度下降优化算法

$$w_{new} = w_{old} + \eta \sum_{i=1}^n (y^{(i)} - \bar{y}^{(i)}) x^{(i)}$$



$$\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{bmatrix} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{bmatrix} + \eta \sum_{i=1}^n (y(i) - \bar{y}(i)) \begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix}$$

# 七、梯度下降优化算法

## 随机梯度下降算法(Stochastic Gradient Descent, SGD)

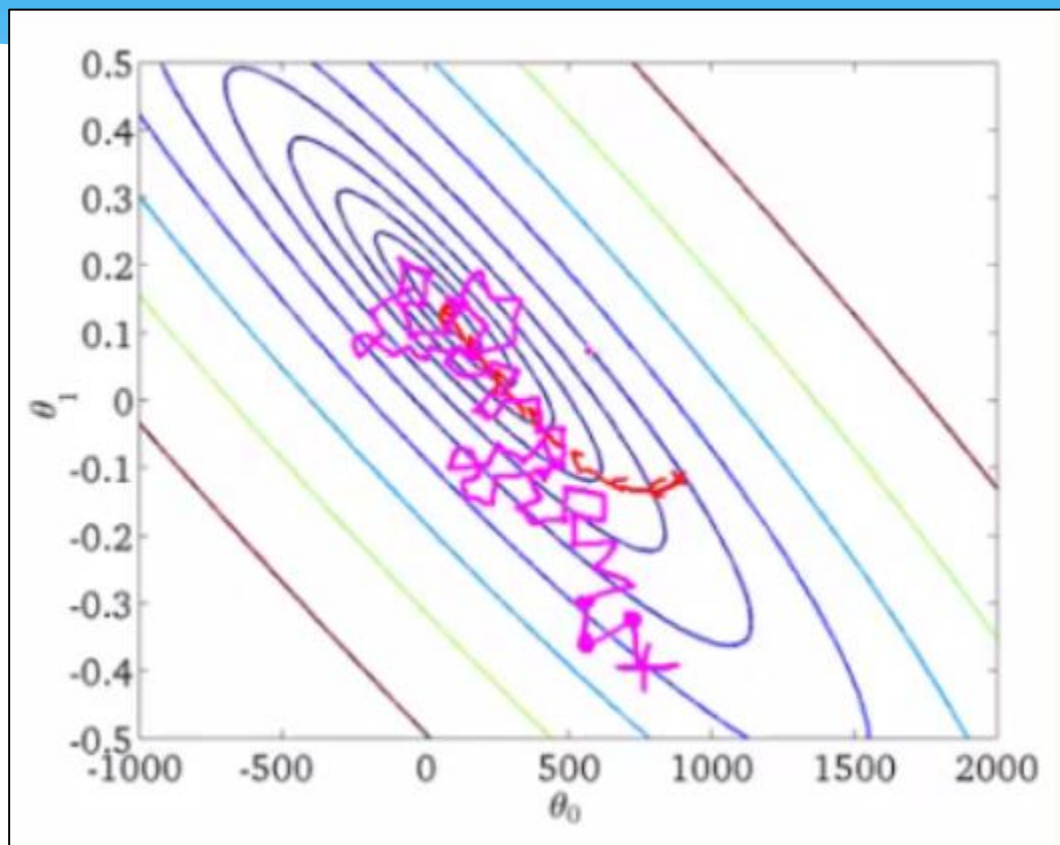
根据(式3)来训练模型，每次更新 $w$ 的迭代，都要遍历训练数据中所有的样本进行计算，这种算法叫做**批梯度下降 (Batch Gradient Descent)**。

如果样本数量非常大，比如数百万到数亿，那么计算量异常巨大。

在SGD算法中，每次更新 $w$ 的迭代，只计算一个样本。这样对于一个具有数百万样本的训练数据，完成一次遍历就会对 $w$ 更新数百万次，效率大大提升。



# 七、梯度下降优化算法



椭圆表示的是函数值的等高线，椭圆中心是函数的最小值点。  
 红色是BGD的逼近曲线，而紫色是SGD的逼近曲线。

## 示例2

```
from MyPerceptron import Perceptron
#定义激活函数
f = lambda x : x

class LinearUnit(Perceptron):
    def __init__(self, input_num):
        '''初始化线性单元，设置输入参数的个数'''
        Perceptron.__init__(self, input_num, f)

def get_training_dataset():
    '''设置5个人的收入数据'''
    #构建训练数据
    #输入向量列表，每一项是工作年限
    input_vecs = [[5], [3], [8], [1.4], [10.1]]
    #期望的输出列表，月薪，注意要与输入一一对应
    labels = [5500, 2300, 7600, 1800, 11400]
    return input_vecs, labels
```

```
def train_linear_unit():  
    """使用数据训练线性单元"""  
    #创建感知器，输入参数的特征数为1（工作年限）  
    lu = LinearUnit(1)  
    #训练，迭代10轮，学习速率为0.01  
    input_vecs, labels = get_training_dataset()  
    lu.train(input_vecs, labels, 10, 0.01)  
    #返回训练好的线性单元  
    return lu
```

```
if __name__ == '__main__':  
    '''训练线性单元'''  
    linear_unit = train_linear_unit()  
    #打印训练获得的权重  
    print(linear_unit)  
    #测试  
    print('Work 3.4 years, monthly salary = %.2f'  
          % linear_unit.predict([3.4]))  
    print('Work 15 years, monthly salary = %.2f'  
          % linear_unit.predict([15]))  
    print('Work 1.5 years, monthly salary = %.2f'  
          % linear_unit.predict([1.5]))  
    print('Work 6.3 years, monthly salary = %.2f'  
          % linear_unit.predict([6.3]))
```

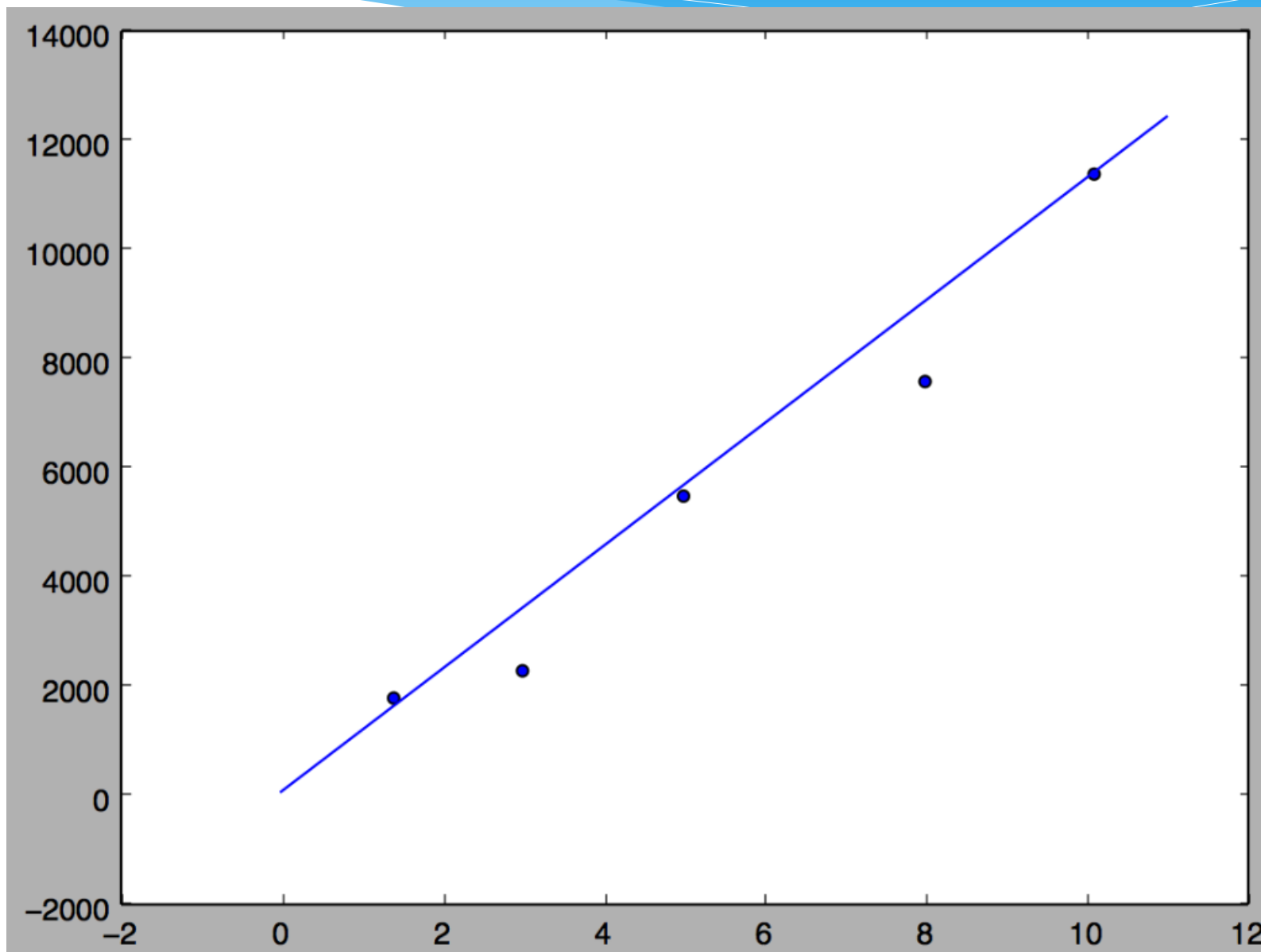
## 运行结果

Work 3.4 years, monthly salary = 3907.30

Work 15 years, monthly salary = 16946.44

Work 1.5 years, monthly salary = 1771.58

Work 6.3 years, monthly salary = 7167.09



# 内容

- 一、深度学习发展历史
- 二、深度学习的应用
- 三、三位牛人
- 四、深度学习工具介绍和对比
- 五、感知器
- 六、线性单元
- 七、梯度下降优化算法
- 八、神经网络和反向传播算法

# 八、神经网络和反向传播算法

神经元和感知器本质上是一样的。

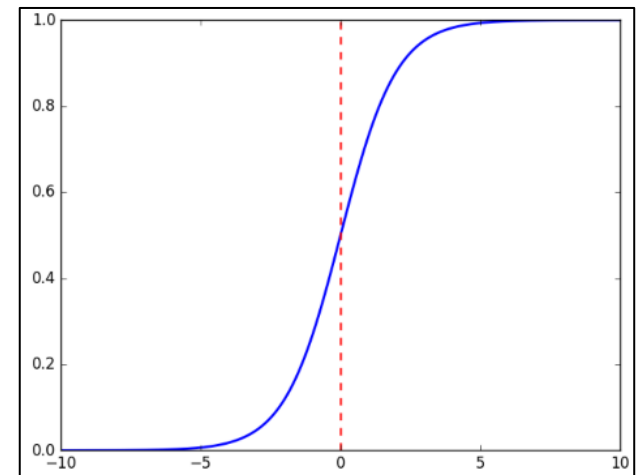
当我们说感知器的时候，它的激活函数是阶跃函数。

当我们说神经元时，激活函数往往选择为sigmoid函数或tanh函数。

$$f(z) = \begin{cases} 1 & z > 0 \\ 0 & \text{otherwise} \end{cases}$$

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

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





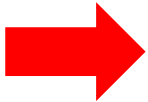
# 八、神经网络和反向传播算法

计算一个神经元的输出的方法和计算一个感知器的输出是一样的。

假设神经元的输入是向量  $\vec{x}$ ，权重向量是  $\vec{w}$ （偏置项是  $w_0$ ），激活函数是sigmoid函数，则其输出  $y$ ：

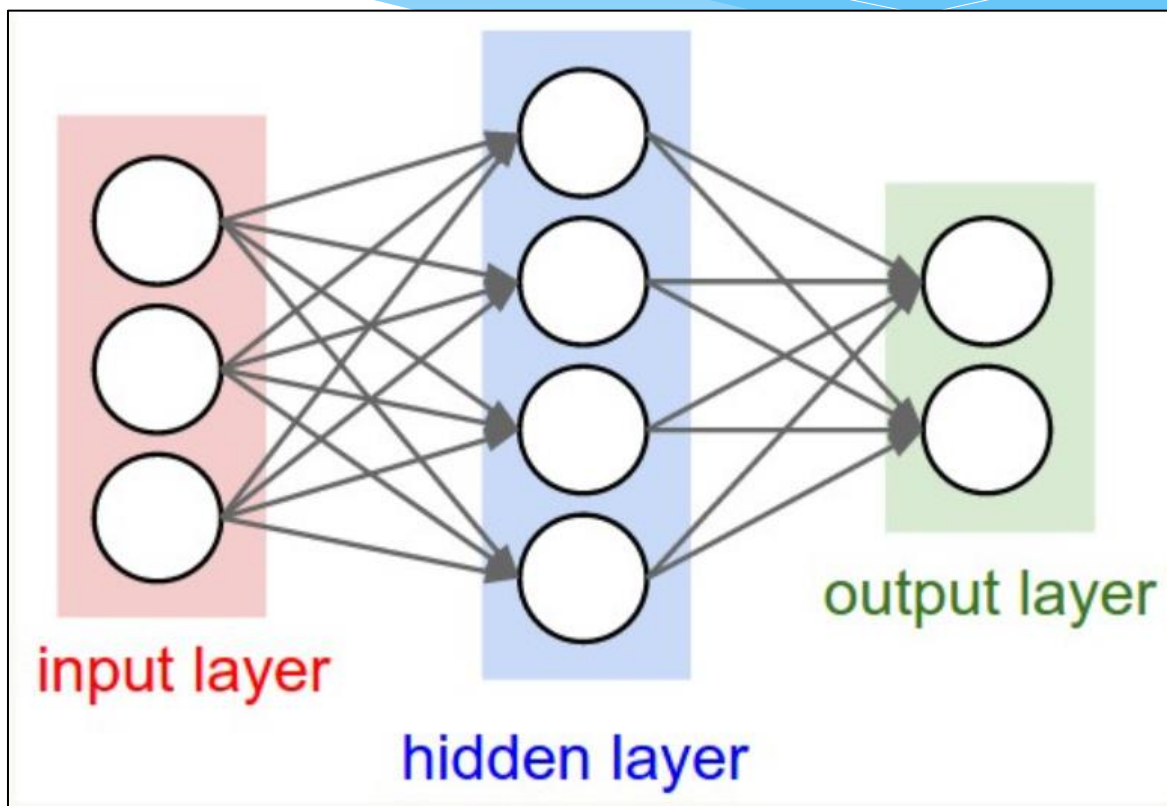
$$y = \text{sigmoid}(\vec{w}^T \cdot \vec{x})$$

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


$$y = \frac{1}{1 + e^{-\vec{w}^T \cdot \vec{x}}}$$

# 八、神经网络和反向传播算法

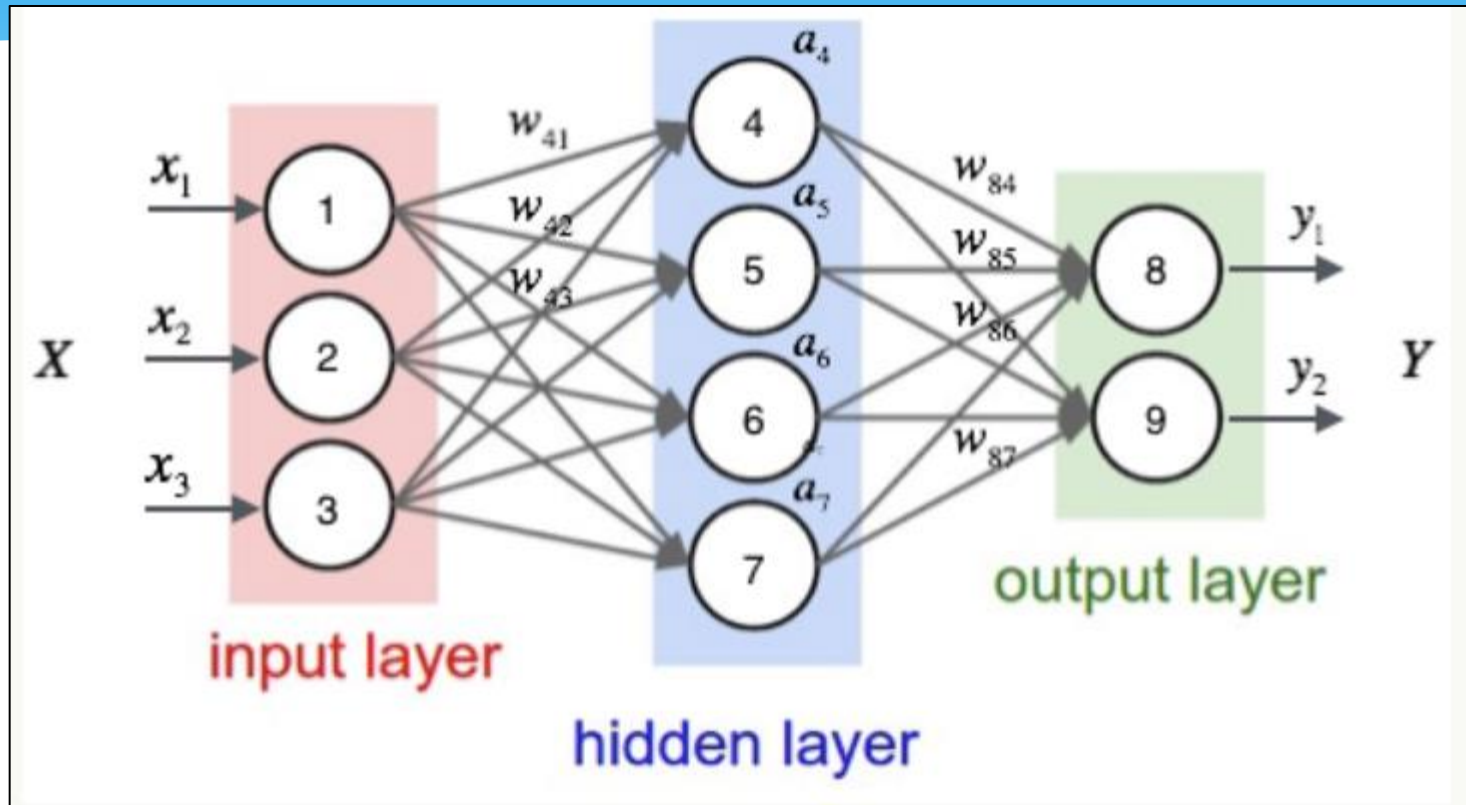
神经网络：



全连接(full connected, FC)神经网络

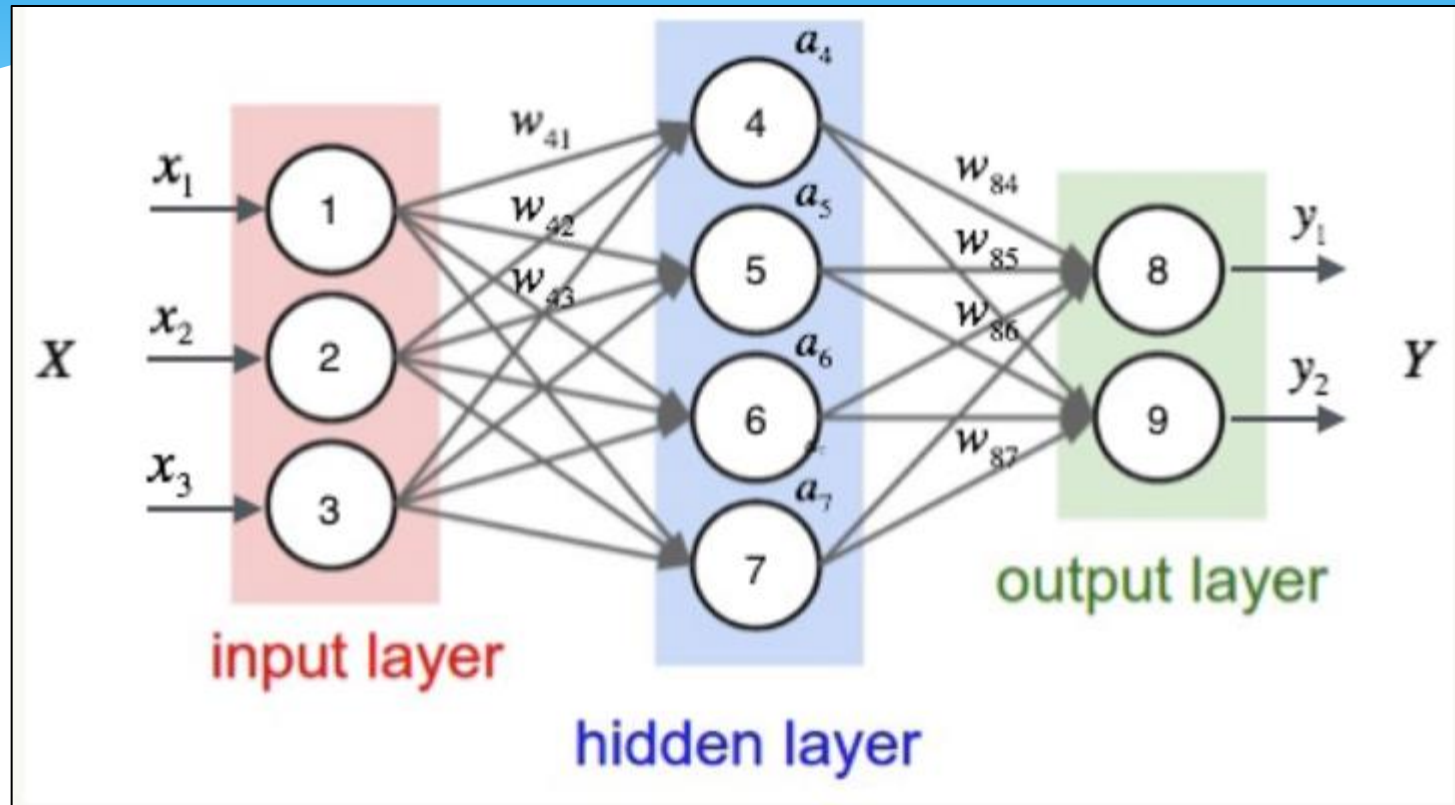
# 八、神经网络和反向传播算法

输入向量的维度和输入层神经元个数相同。



$$\begin{aligned} a_4 &= \text{sigmoid}(\vec{w}^T \cdot \vec{x}) \\ &= \text{sigmoid}(w_{41}x_1 + w_{42}x_2 + w_{43}x_3 + w_{4b}) \end{aligned}$$

# 八、神经网络和反向传播算法



输出向量的维度和输出层神经元个数相同。

$$\begin{aligned}
 y_1 &= \text{sigmoid}(\vec{w}^T \cdot \vec{a}) \\
 &= \text{sigmoid}(w_{84}a_4 + w_{85}a_5 + w_{86}a_6 + w_{87}a_7 + w_{8b})
 \end{aligned}$$

# 八、神经网络和反向传播算法

## 神经网络的矩阵表示

隐藏层4个节点的计算依次排列出来：

$$a_4 = \text{sigmoid}(w_{41}x_1 + w_{42}x_2 + w_{43}x_3 + w_{4b})$$

$$a_5 = \text{sigmoid}(w_{51}x_1 + w_{52}x_2 + w_{53}x_3 + w_{5b})$$

$$a_6 = \text{sigmoid}(w_{61}x_1 + w_{62}x_2 + w_{63}x_3 + w_{6b})$$

$$a_7 = \text{sigmoid}(w_{71}x_1 + w_{72}x_2 + w_{73}x_3 + w_{7b})$$

# 八、神经网络和反向传播算法

令：

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix}$$

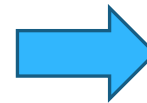
$$\vec{w}_4 = [w_{41}, w_{42}, w_{43}, w_{4b}]$$

$$\vec{w}_5 = [w_{51}, w_{52}, w_{53}, w_{5b}]$$

$$\vec{w}_6 = [w_{61}, w_{62}, w_{63}, w_{6b}]$$

$$\vec{w}_7 = [w_{71}, w_{72}, w_{73}, w_{7b}]$$

$$f = \text{sigmoid}$$



$$a_4 = f(\vec{w}_4 \cdot \vec{x})$$

$$a_5 = f(\vec{w}_5 \cdot \vec{x})$$

$$a_6 = f(\vec{w}_6 \cdot \vec{x})$$

$$a_7 = f(\vec{w}_7 \cdot \vec{x})$$

# 八、神经网络和反向传播算法

把上述计算 $a_4, a_5, a_6, a_7$ 的四个式子写到一个矩阵里面，每个式子作为矩阵的一行，就可以利用矩阵来表示它们的计算了。

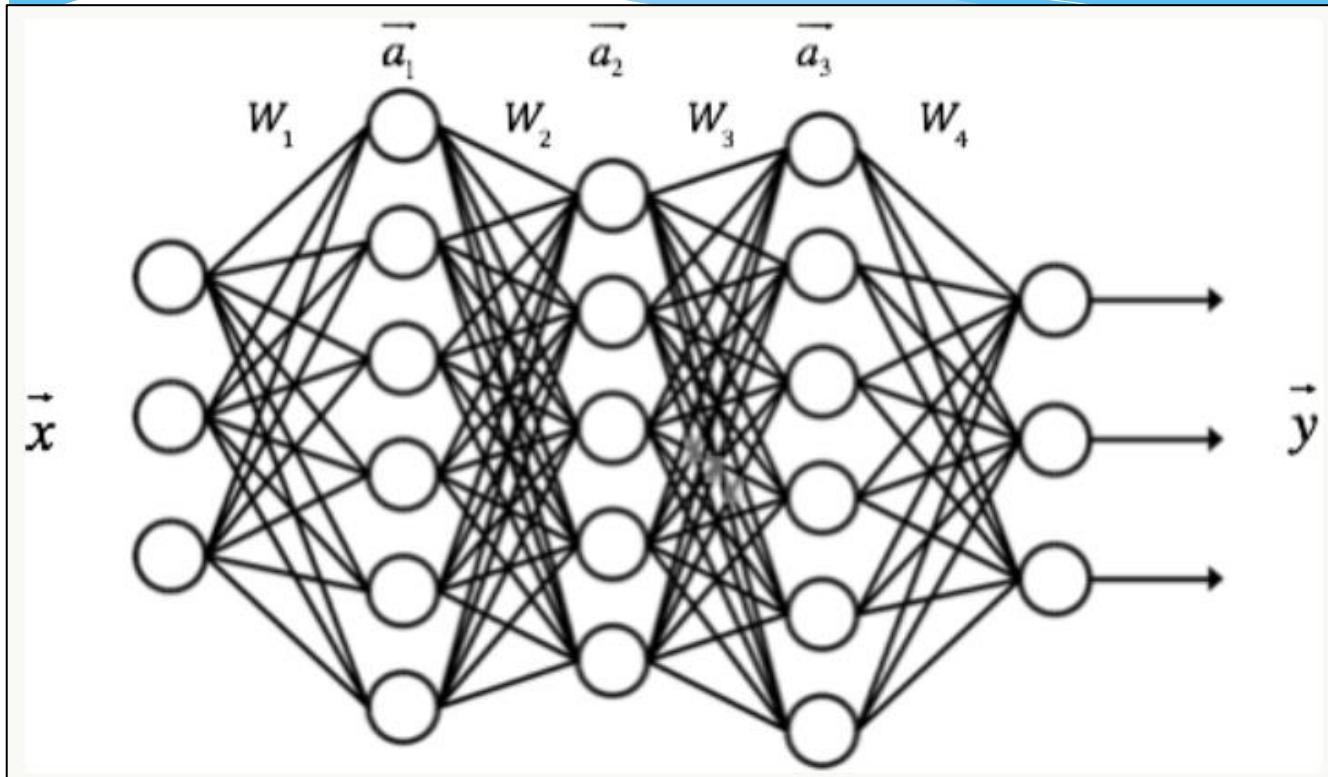
$$\vec{a} = \begin{bmatrix} a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}, \quad W = \begin{bmatrix} \vec{w}_4 \\ \vec{w}_5 \\ \vec{w}_6 \\ \vec{w}_7 \end{bmatrix} = \begin{bmatrix} w_{41}, w_{42}, w_{43}, w_{4b} \\ w_{51}, w_{52}, w_{53}, w_{5b} \\ w_{61}, w_{62}, w_{63}, w_{6b} \\ w_{71}, w_{72}, w_{73}, w_{7b} \end{bmatrix}, \quad f\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{bmatrix}\right) = \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \end{bmatrix}$$



$$\vec{a} = f(W \cdot \vec{x})$$



# 八、神经网络和反向传播算法



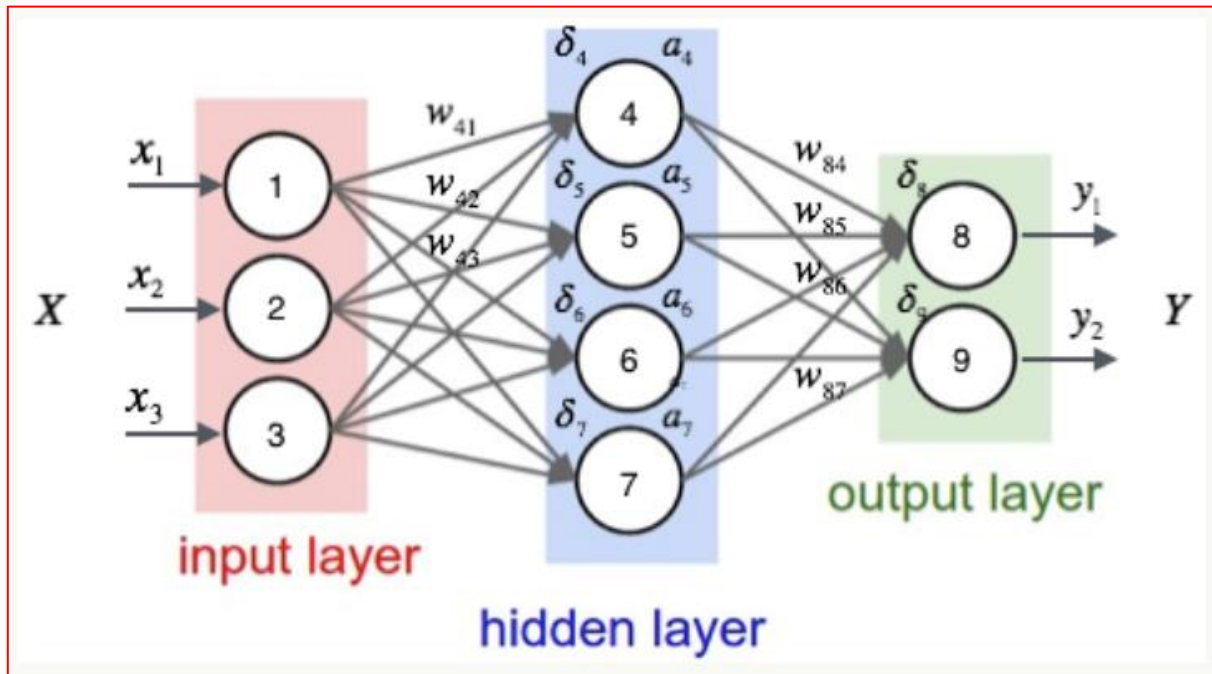
$$\begin{aligned}\vec{a}_1 &= f(W_1 \cdot \vec{x}) \\ \vec{a}_2 &= f(W_2 \cdot \vec{a}_1) \\ \vec{a}_3 &= f(W_3 \cdot \vec{a}_2) \\ \vec{y} &= f(W_4 \cdot \vec{a}_3)\end{aligned}$$

每一层的算法都是一样的。



# 八、神经网络和反向传播算法

## 反向传播算法(Back Propagation)



# 八、神经网络和反向传播算法

根据上一节介绍的算法，用样本的特征  $\vec{x}$ ，计算出神经网络中每个隐藏层节点的输出  $a_i$ ，以及输出层每个节点的输出  $y_i$ 。

按照下面的方法计算出每个节点的误差项  $\delta_i$ ：

**对于输出层节点  $i$**

$$\delta_i = y_i(1 - y_i)(t_i - y_i)$$

例如：

$$\delta_8 = y_1(1 - y_1)(t_1 - y_1)$$

# 八、神经网络和反向传播算法

对于隐藏层节点:

$$\delta_i = a_i(1 - a_i) \sum_{k \in \text{outputs}} w_{ki} \delta_k$$

对于隐藏层节点4来说，计算方法如下：

$$\delta_4 = a_4(1 - a_4)(w_{84}\delta_8 + w_{94}\delta_9)$$

# 八、神经网络和反向传播算法

更新每个连接上的权值：

$$w_{ji} \leftarrow w_{ji} + \eta \delta_j x_{ji}$$

$w_{ji}$ 是节点*i*到节点*j*的权重， $\eta$ 是一个成为学习速率的常数， $\delta_j$ 是节点*j*的误差项， $x_{ji}$ 是节点*i*传递给节点*j*的输入。

例如，权重 $w_{84}$ 的更新方法： $w_{84} \leftarrow w_{84} + \eta \delta_8 a_4$

例如，权重 $w_{41}$ 的更新方法： $w_{41} \leftarrow w_{41} + \eta \delta_4 x_1$

# 八、神经网络和反向传播算法

我们已经介绍了神经网络每个节点误差项的计算和权重更新方法。

显然，计算一个节点的误差项，需要先计算每个与其相连的下一层节点的误差项。

这就要求误差项的计算顺序必须是从输出层开始，然后反向依次计算每个隐藏层的误差项，直到与输入层相连的那个隐藏层。

这就是反向传播算法的名字的含义。

Input Volume (+pad 1) (7x7x3)

 $x[:, :, 0]$ 

0	0	0	0	0	0	0
0	0	1	1	0	2	0
0	2	2	2	2	1	0
0	1	0	0	2	0	0
0	0	1	1	0	0	0
0	1	2	0	0	2	0
0	0	0	0	0	0	0

 $x[:, :, 1]$ 

0	0	0	0	0	0	0
0	1	0	2	2	0	0
0	0	0	0	2	0	0
0	1	2	1	2	1	0
0	1	0	0	0	0	0
0	1	2	1	1	1	0
0	0	0	0	0	0	0

 $x[:, :, 2]$ 

0	0	0	0	0	0	0
0	2	1	2	0	0	0
0	1	0	0	1	0	0
0	0	2	1	0	1	0
0	0	1	2	2	2	0
0	2	1	0	0	1	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

 $w0[:, :, 0]$ 

-1	1	0
0	1	0
0	1	1

 $w0[:, :, 1]$ 

-1	-1	0
0	0	0
0	-1	0

 $w0[:, :, 2]$ 

0	0	-1
0	1	0
1	-1	-1

Bias b0 (1x1x1)

 $b0[:, :, 0]$ 

1
---

Filter W1 (3x3x3)

 $w1[:, :, 0]$ 

1	1	-1
-1	-1	1
0	-1	1

 $w1[:, :, 1]$ 

0	1	0
-1	0	-1
-1	1	0

 $w1[:, :, 2]$ 

-1	0	0
-1	0	1
-1	0	0

Bias b1 (1x1x1)

 $b1[:, :, 0]$ 

0
---

Output Volume (3x3x2)

 $o[:, :, 0]$ 

6	7	5
3	-1	-1
2	-1	4

 $o[:, :, 1]$ 

2	-5	-8
1	-4	-4
0	-5	-5

toggle movement

**谢 谢！**