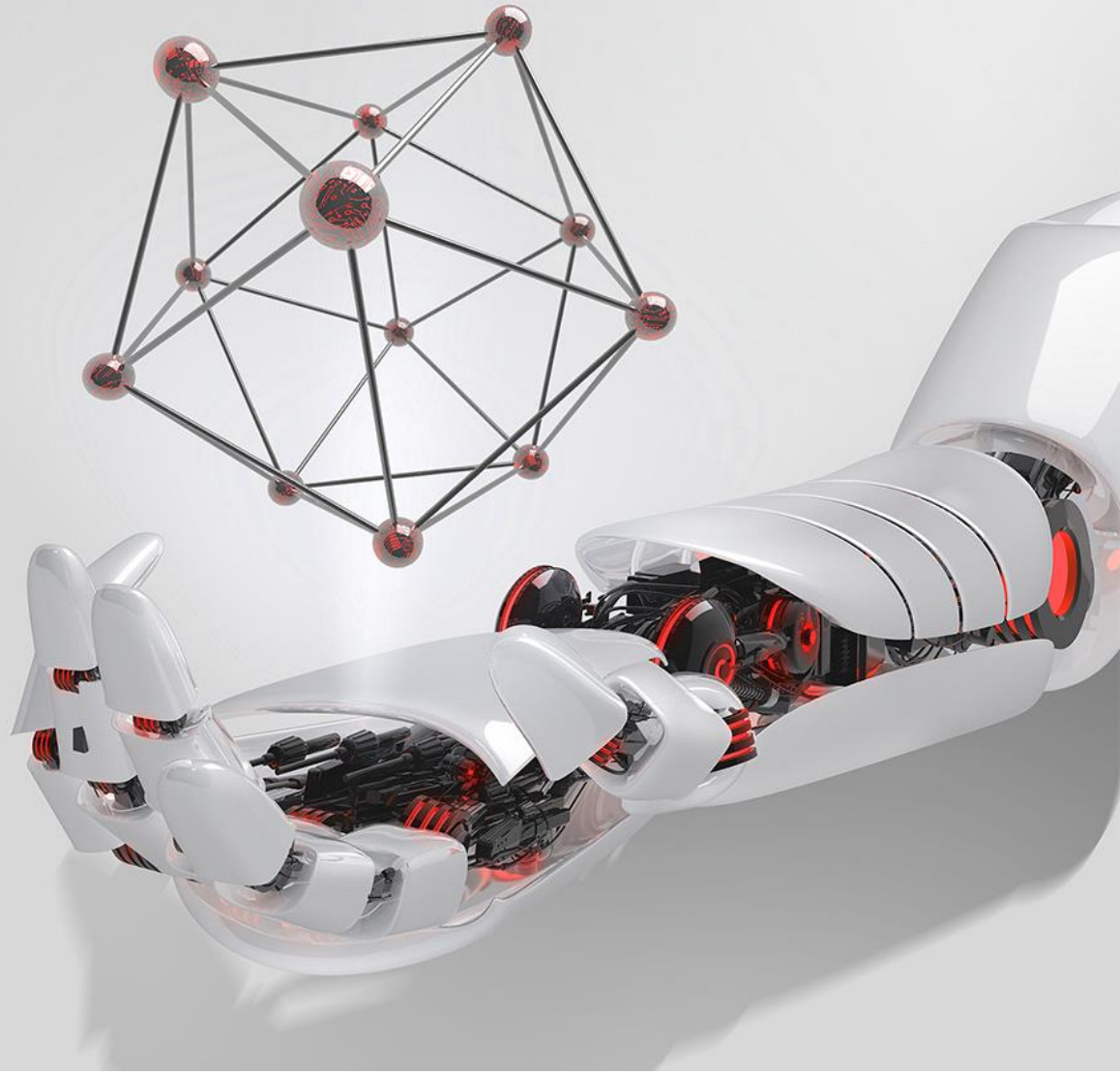
华为昇腾AI处理器及应用初识课程



Chapter 1：人工智能基础

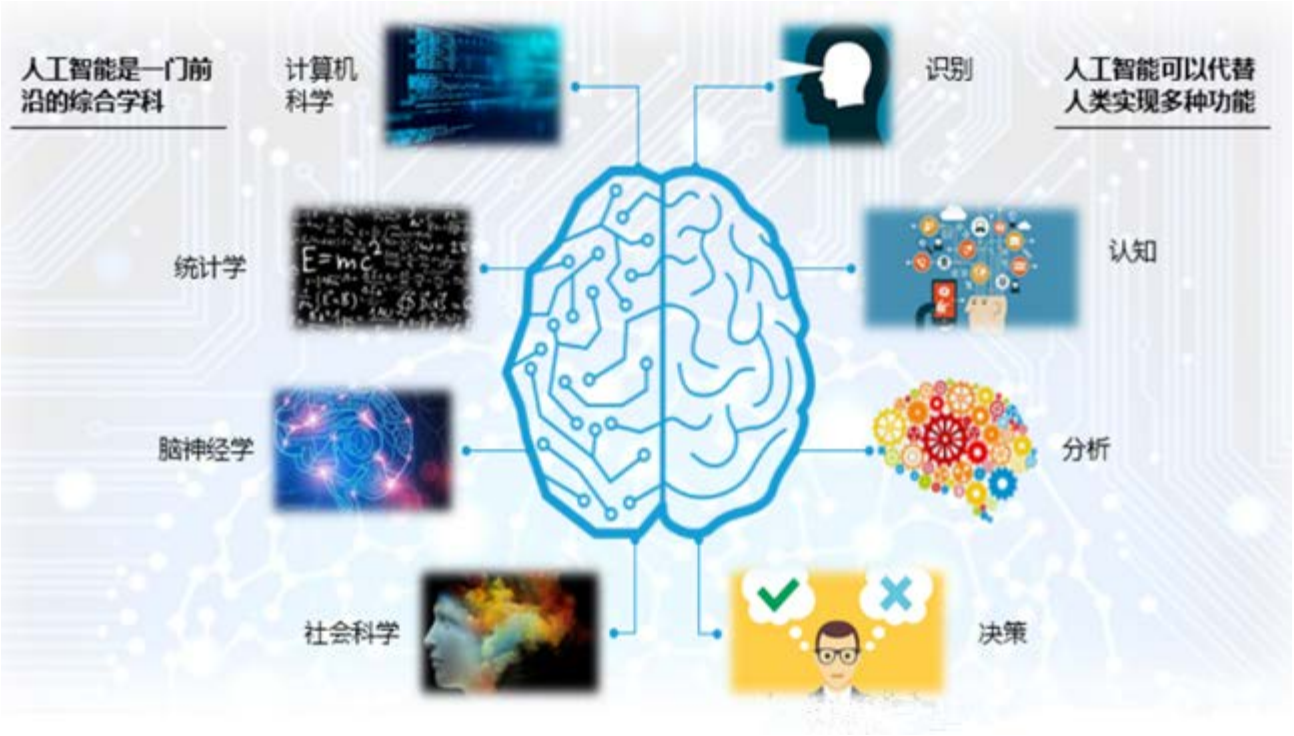
HUAWEI

目录

人工智能基础概览

人工智能（Artificial Intelligence）：

研究、开发用于模拟、延伸和扩展人的智能的理论、方法、技术及应用系统的一门新的技术科学。1956年由约翰.麦卡锡首次提出，当时的定义为“制造智能机器的科学与工程”。人工智能的目的就是让机器能够像人一样思考，让机器拥有智能。时至今日，人工智能的内涵已经大大扩展，是一门交叉学科。



2006-至今
第三次浪潮

深度强化学习+云计算大数据

- 卷积神经网络模型与参数训练技巧的进步；
- 硬件的进步：摩尔定律，可观的计算能力，云计算；
- 互联网+海量大数据集，深度学习的准确度是随着数据的增长而增加。

1976-2006
第二次浪潮

知识工程+机器学习

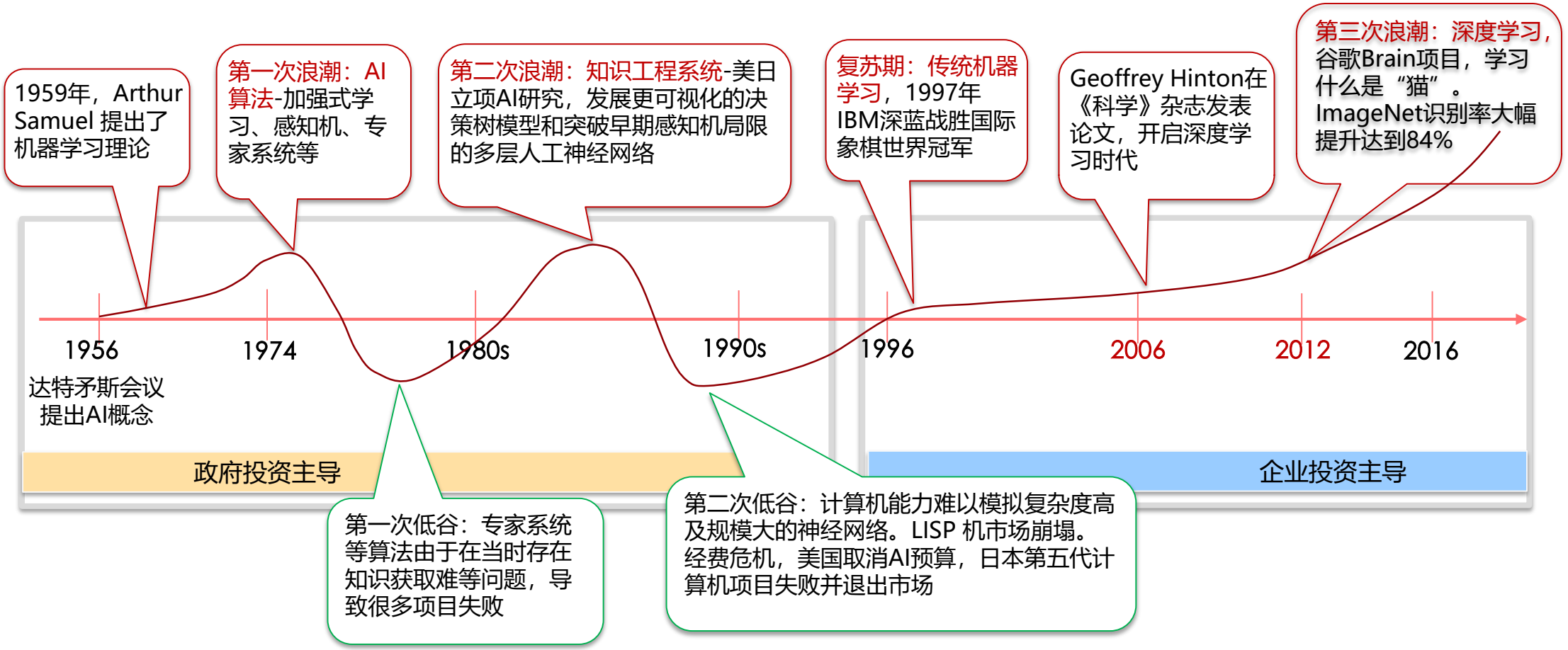
- 知识工程体现为各种专家系统，比如医学、工程学，知识的总结与获取较难，部分专家不愿意分享经验；
- 将各种机器学习算法引入人工智能，让机器从数据中自动学习，获得知识。

1956-1976
第一次浪潮

符号主义+逻辑方法

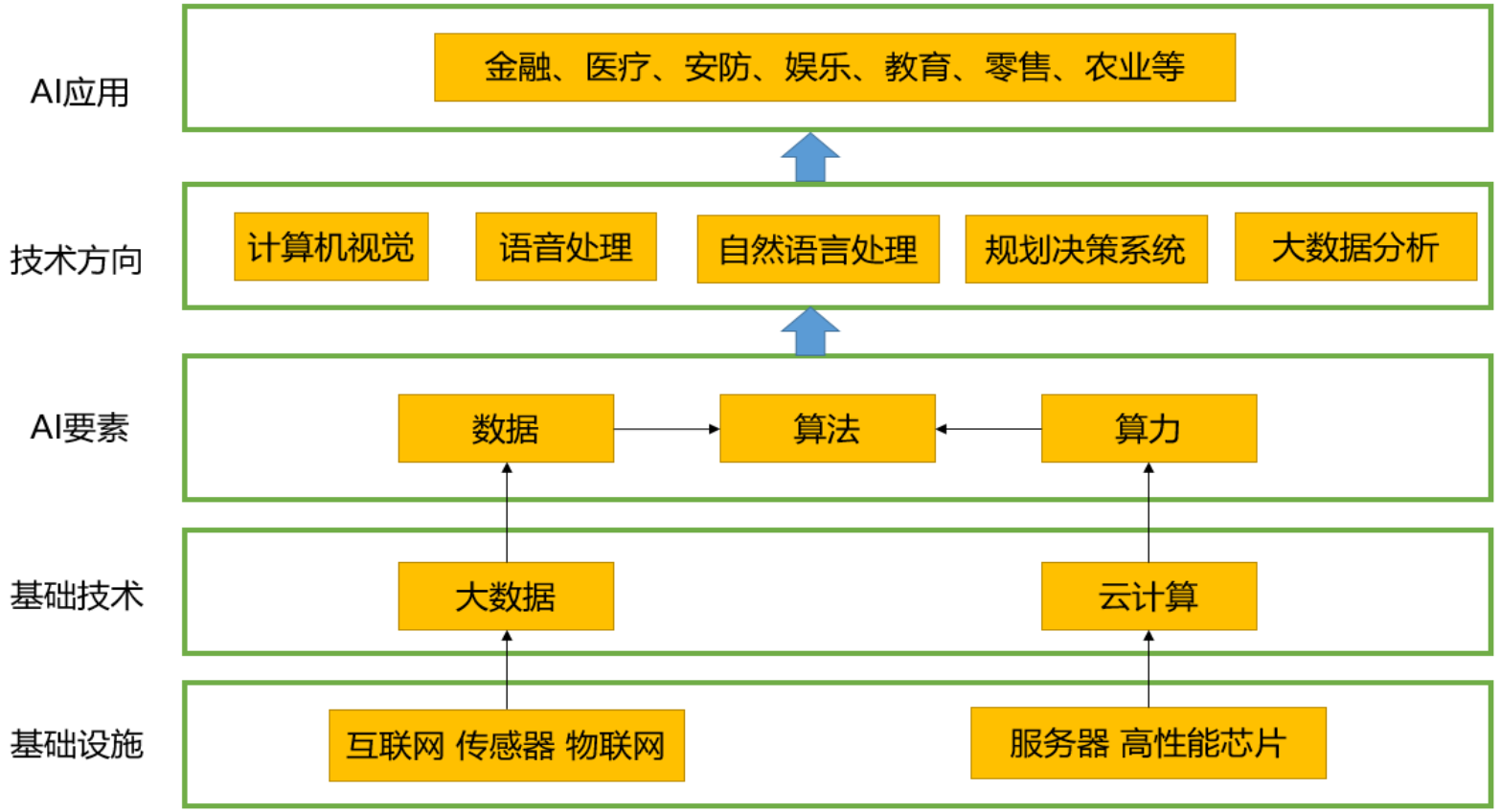
- 基于决策论的逻辑主义推理方法；
- 用机器证明的办法去推理知识，如通过逻辑程序语言证明数学定理；
- 统计方法中，引入符号主义进行语义处理，实现人机交互。

人工智能发展历史


1959年，Arthur Samuel 提出了机器学习理论
第一次浪潮：AI算法-加强式学习、感知机、专家系统等
第二次浪潮：知识工程系统-美日立项AI研究，发展更可视化的决策树模型和突破早期感知机局限的多层人工神经网络
复苏期：传统机器学习，1997年IBM深蓝战胜国际象棋世界冠军
Geoffrey Hinton在《科学》杂志发表论文，开启深度学习时代
第三次浪潮：深度学习，谷歌Brain项目，学习什么是“猫”。ImageNet识别率大幅提升达到84%
1956
达特矛斯会议提出AI概念
1974
1980s
1990s
1996
2006
2012
2016
政府投资主导
企业投资主导
第一次低谷：专家系统等算法由于在当时存在知识获取难等问题，导致很多项目失败
第二次低谷：计算机能力难以模拟复杂度高及规模大的神经网络。LISP 机市场崩塌。经费危机，美国取消AI预算，日本第五代计算机项目失败并退出市场

人工智能产业生态

人工智能的四要素是数据、算法、算力、场景。要满足这四要素，我们需要将AI与云计算、大数据和物联网结合以服务于智能社会。



人工智能应用技术方向

- 现在AI的应用技术方向主要分为：

图像识别（ >96.5% ）：

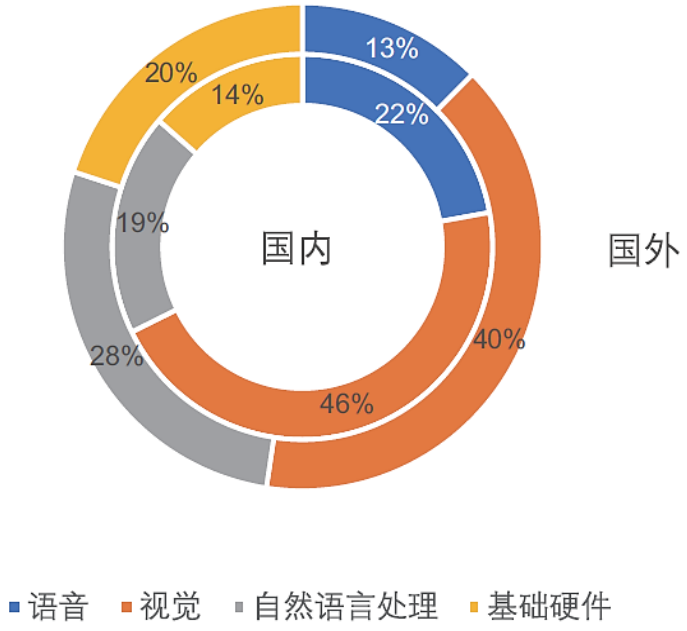
- 人脸识别（99.8%，人眼97.5%）
- 车牌（99.7%） 、商标识别等
- 图片搜索（>80%）、分类等
- 唇语（92%，远超人类专家）

语音，语言，语义识别：

- **语音识别**(97% ~99%)
- 语音-文本转换 （>95%）
- 自动翻译 (>90% 英文<->德、法等)
- 个人助理/ChatBot
- 语调情感识别

其他应用：

- 智能库存管理
- 疾病辅助诊疗
- DC自动运维
- 商品智能推荐
- ……



国内外人工智能企业应用技术分布
2018中国人工智能发展报告

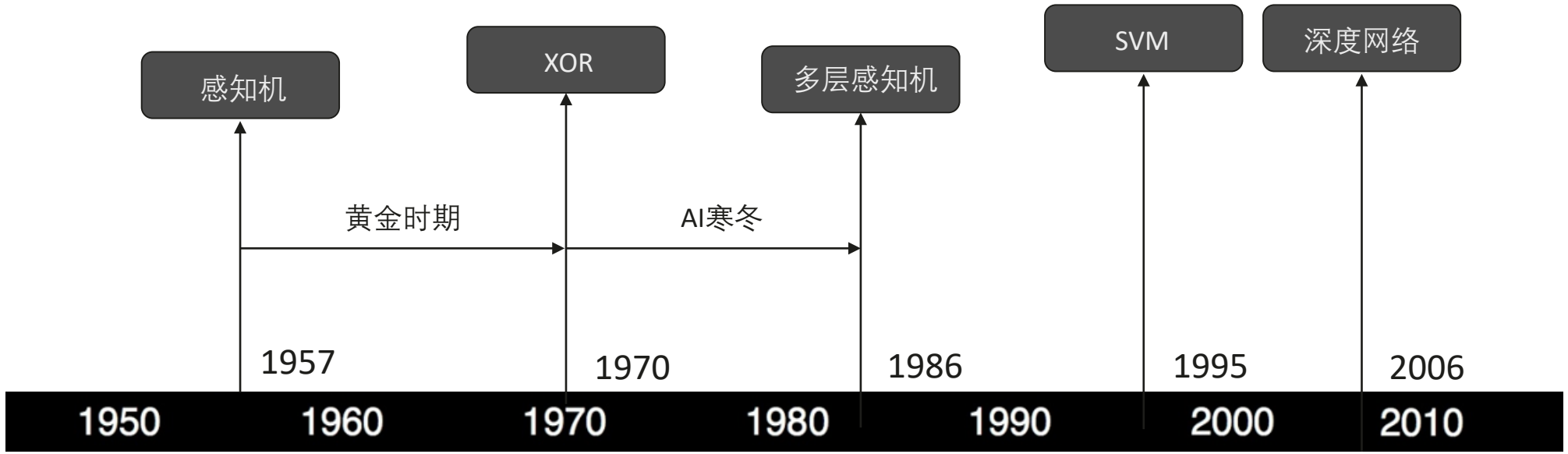
人工智能当前所处的阶段 —— 感知智能的初级阶段

	表现	示例	价值
计算智能	**能存会算**：机器开始像人类一样会计算，传递信息	例：分布式计算、神经网络	价值：能够帮助人类存储和快速处理海量数据，是感知和认知的基础
感知智能	**能听会看**：机器开始看懂和听懂，做出判断，采取一些简单行动	例：可以识别人脸的摄像头、可以听懂语言的音箱	价值：能够帮助人类高效地完成“看”和“听”相关的工作
认知智能	**能理解会思考**：机器开始像人类一样能理解、思考与决策	例：完全独立驾驶的无人驾驶汽车、自主行动的机器人	价值：可以全面辅助或替代人类部分工作

AI的发展目前处于**感知智能**的初级阶段

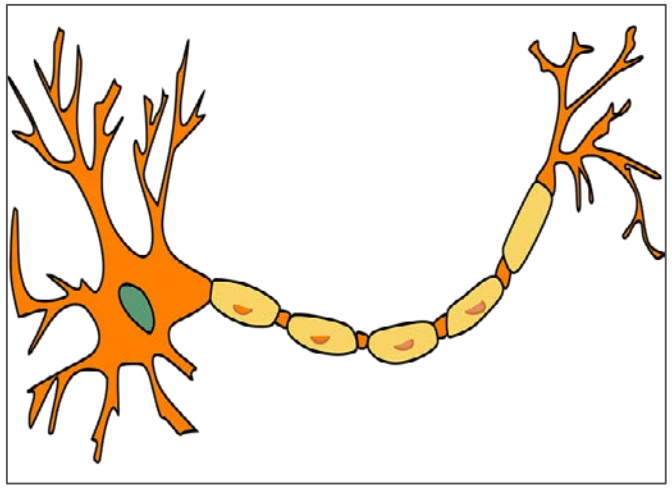
目录

深度学习发展

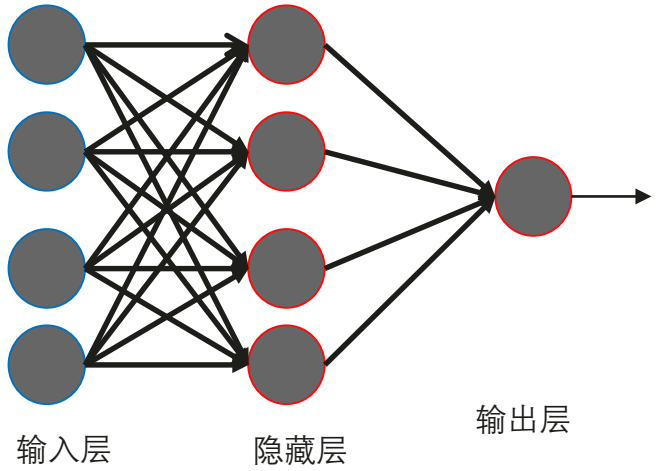

感知机
XOR
多层感知机
SVM
深度网络
黄金时期
AI寒冬
1957
1970
1986
1995
2006
1950
1960
1970
1980
1990
2000
2010

神经网络基础

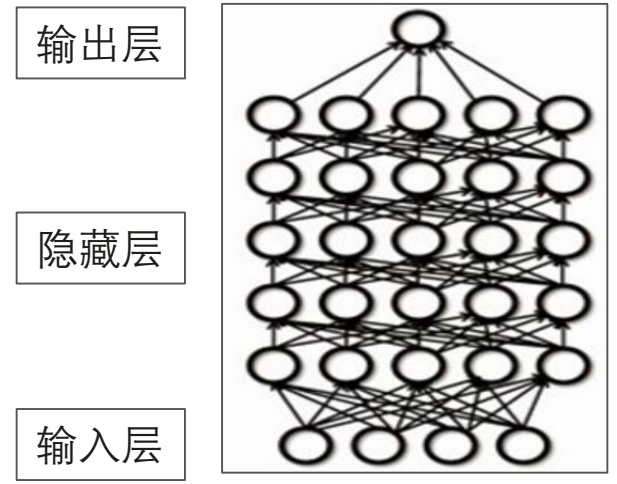
- 人工神经网络，简称神经网络（Artificial Neural Network，ANN）：是由人工神经元互连组成的网络，它是从微观结构和功能上对人脑的抽象、简化，是模拟人类智能的一条重要途径，反映了人脑功能的若干基本特征，如并行信息处理、学习、联想、模式分类、记忆等。
- 隐藏层比较多（大于2）的神经网络叫做深度神经网络（Deep Neural Network，DNN）。而深度学习，就是使用深度神经网络架构的机器学习方法。



人类神经网络



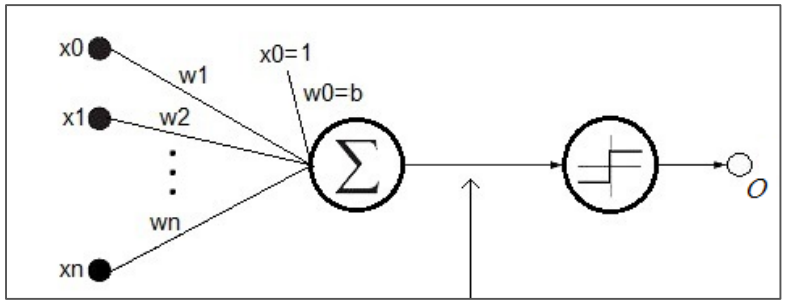
感知器



深度神经网络

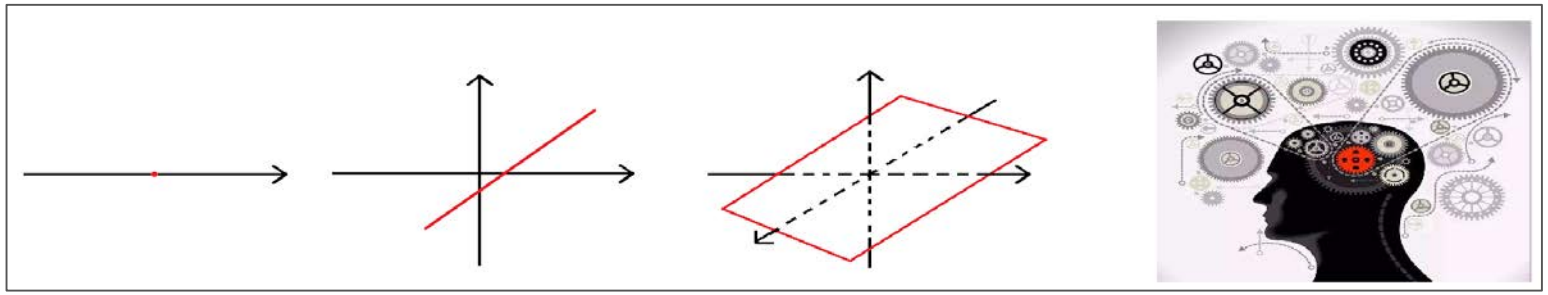
感知器

- **输入向量**：$X = [x_0, x_1, \ldots, x_n]^T$.
- **权值**：$W = [\omega_0, \omega_1, \ldots, \omega_n]^T$，其中$\omega_0$称为偏置。
- **激活函数**：$O = sign(net) = \begin{cases} 1, net > 0, \\ -1, otherwise. \end{cases}$



$$net = \sum_{i=0}^{n} \omega_i x_i = \boldsymbol{W^T X}$$

- 上面的感知器，相当于一个线性分类器，它使用高维X向量做输入，在高维空间对输入的样本进行二分类：当$\boldsymbol{W^T X} > 0$时，$o = 1$，相当于样本被归类为其中一类。否则，$o = -1$，相当于样本被归类为另一类。这两类的边界在哪里呢？就是$\boldsymbol{W^T X} = 0$，这是一个高维超平面。

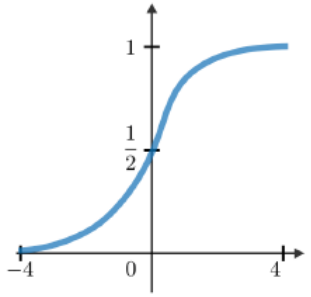
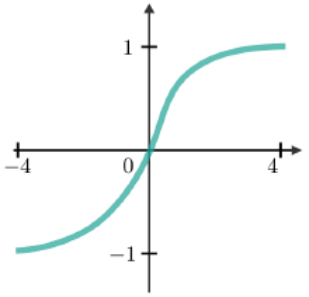
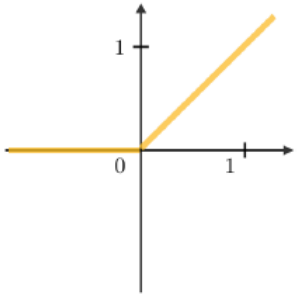
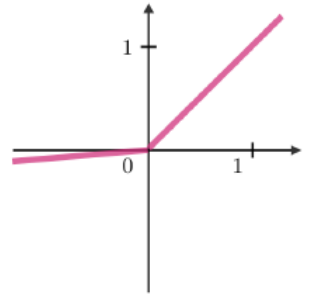


分割点	分割直线	分割平面	分割超平面
$Ax + B = 0$	$Ax + By + C = 0$	$Ax + By + Cz + D = 0$	$W^T \mathrm{X} + b = 0$

激活函数

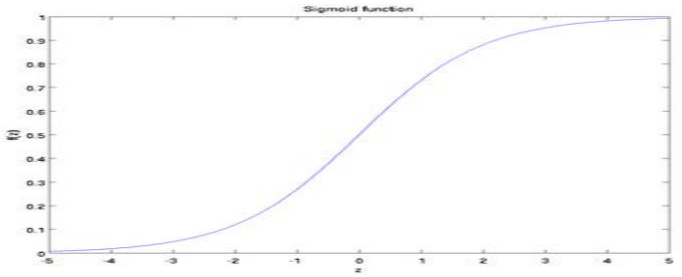
- 激活函数（Activation functions）对于神经网络模型去学习、理解非常复杂和非线性的函数来说具有十分重要的作用，激活函数的存在将非线性特性引入到我们的网络中。
- 如果不用激活函数，每一层输出都是上层输入的线性函数，无论神经网络有多少层，输出都是输入的线性组合。
- 如果使用的话，激活函数给神经元引入了非线性因素，使得神经网络可以任意逼近任何非线性函数，这样神经网络就可以应用到众多的非线性模型中。

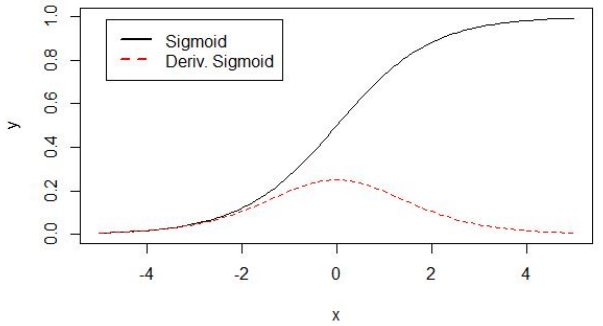
$$output = f(w_1x_1 + w_2x_2 + w_3x_3 \ldots) = f(W^t \bullet X)$$

逻辑函数(Sigmoid)	双曲正切函数(Tanh)	ReLU	Leaky ReLU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0,z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

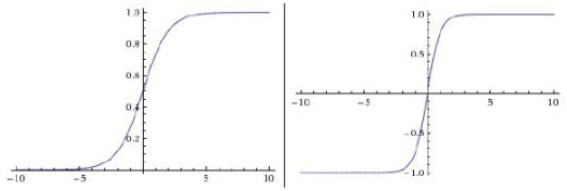
激活函数

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



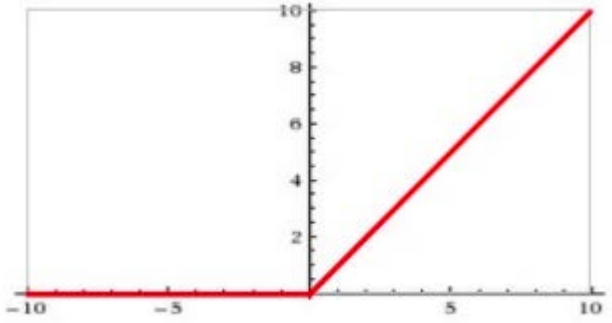


Tanh函数 $f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$

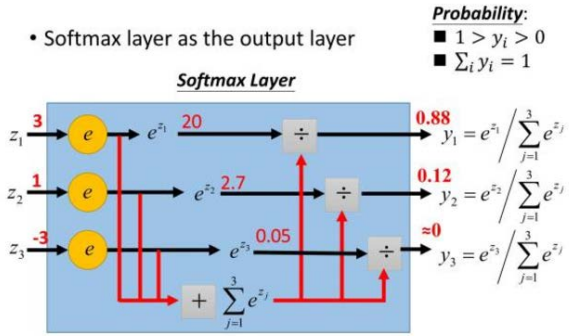


Left: Sigmoid non-linearity squashes real numbers to range between [0,1] **Right:** The tanh non-linearity squashes real numbers to range between [-1,1].

ReLU $\phi(x) = \max(0, x)$



softmax函数 $\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$



梯度下降与损失函数

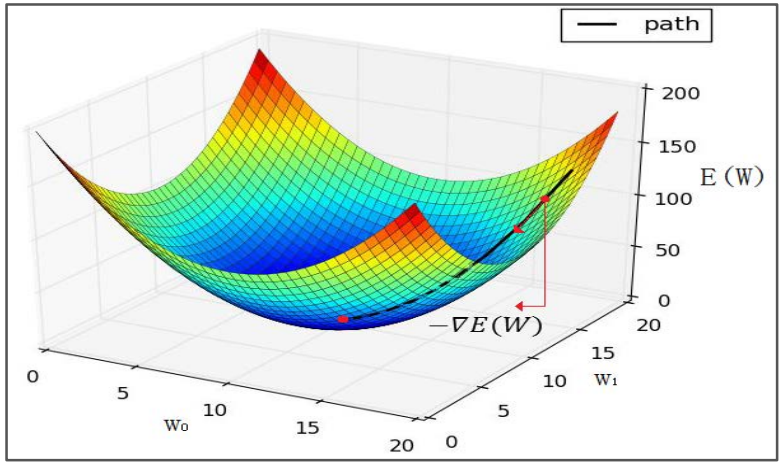
- 在训练深度学习网络的时候，为了让错误最小，我们首先要参数化描述目标分类的错误，这就是**损失函数（误差函数）**，它反映了感知器目标输出和实际输出之间的误差。最常用的误差函数为**均方误差**：

$$E(w) = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2,$$

其中，d为训练样例，D为训练样例集，t_d为目标输出，o_d为实际输出。

- 梯度下降法的思想是让损失函数沿着负梯度的方向进行搜索，不断迭代更新参数，最终使得损失函数最小化。

- **目的**：损失函数$E(W)$是定义在权值空间上的函数。我们的目的是搜索使得$E(W)$**最小**的权值向量W。
- **限制**：$E(W) = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$的复杂高维曲面，在**数学上，尚没有求极值解有效方法**。
- **解决思路**：负梯度方向是函数下降最快的方向，那我们可以从某个点开始，沿着$-\nabla E(W)$方向一路前行，期望最终可以找到$E(W)$的极小值点，这就是梯度下降法的核心思想。

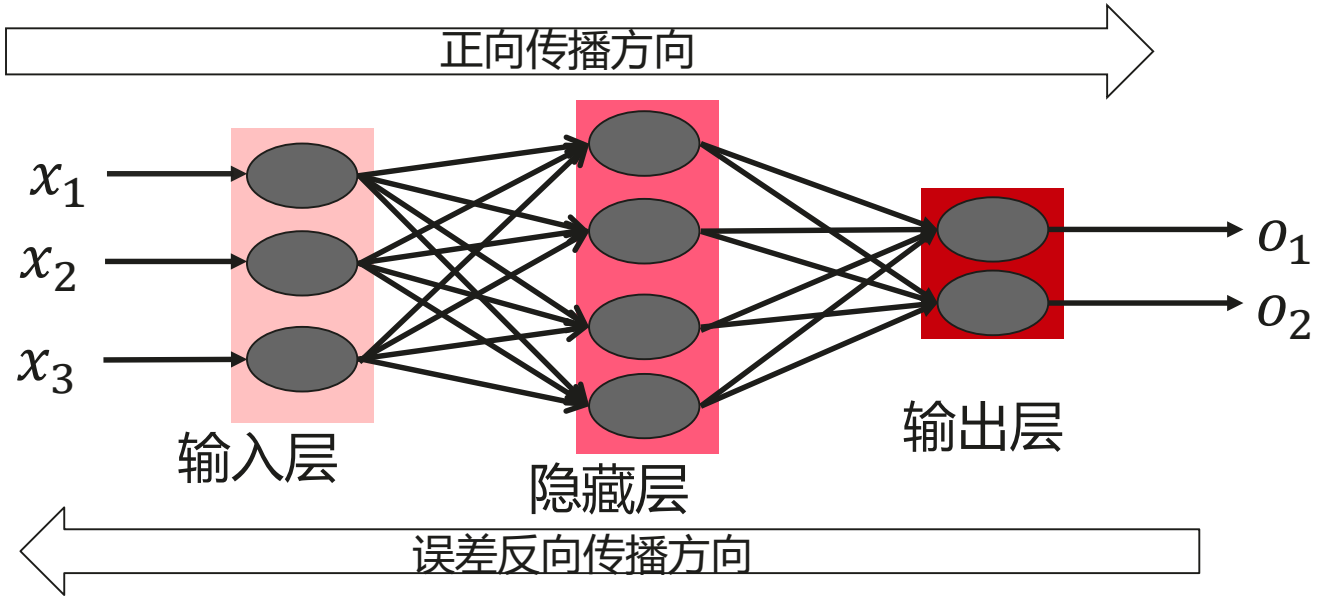


反向传播

- 信号正向传播，误差反向传播
- 对于训练样例集D中的每一个样例记为$<X, t>$，X是输入值向量，t为目标输出，o为实际输出，w为权重系数
- 损失函数：

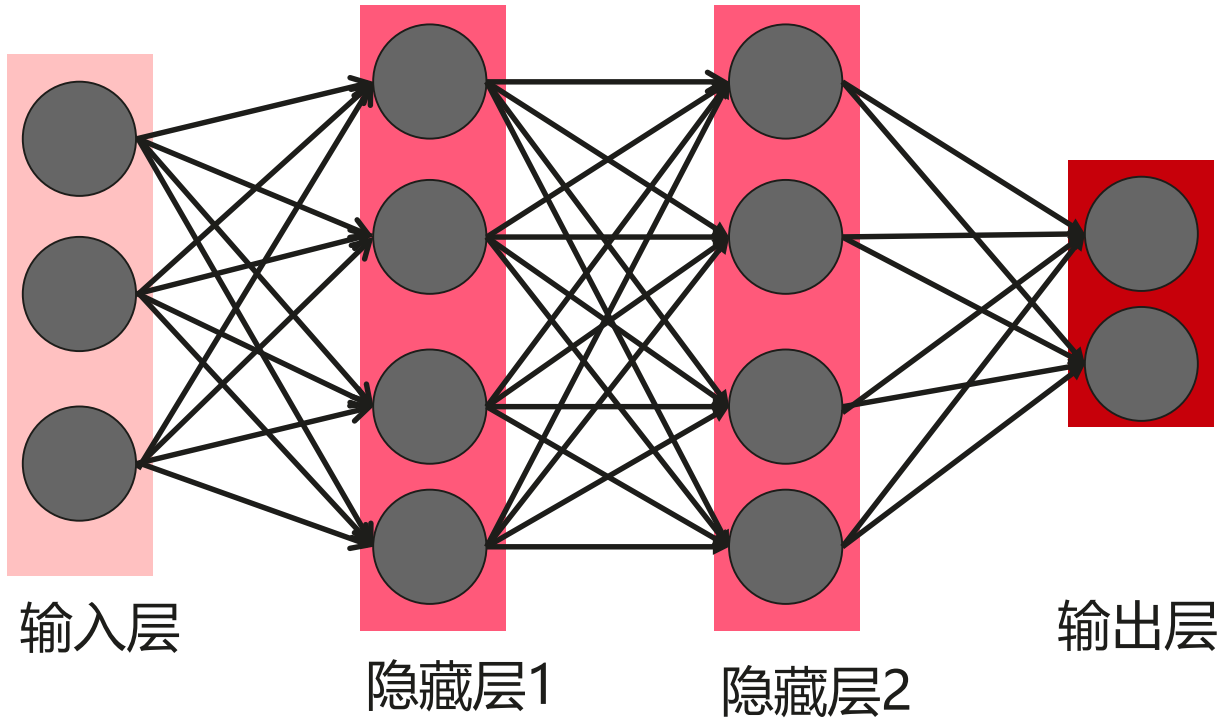
$$E(w)=\frac{1}{2}\sum_{(d\in D)}(t_d-o_d)^2$$

- BP算法训练网络的步骤如下：
  - 取出下一个训练样例$<X, T>$，将X输入网络，得到实际输出O。
  - 根据输出层误差公式求取输出层δ，并更新权值。
  - 对于隐层，根据隐层误差传播公式从输出往输入方向反向、逐层、迭代计算各层的δ，每计算好一层的δ，更新该层权值，直至所有权值更新完毕。
  - 返回第1步继续。



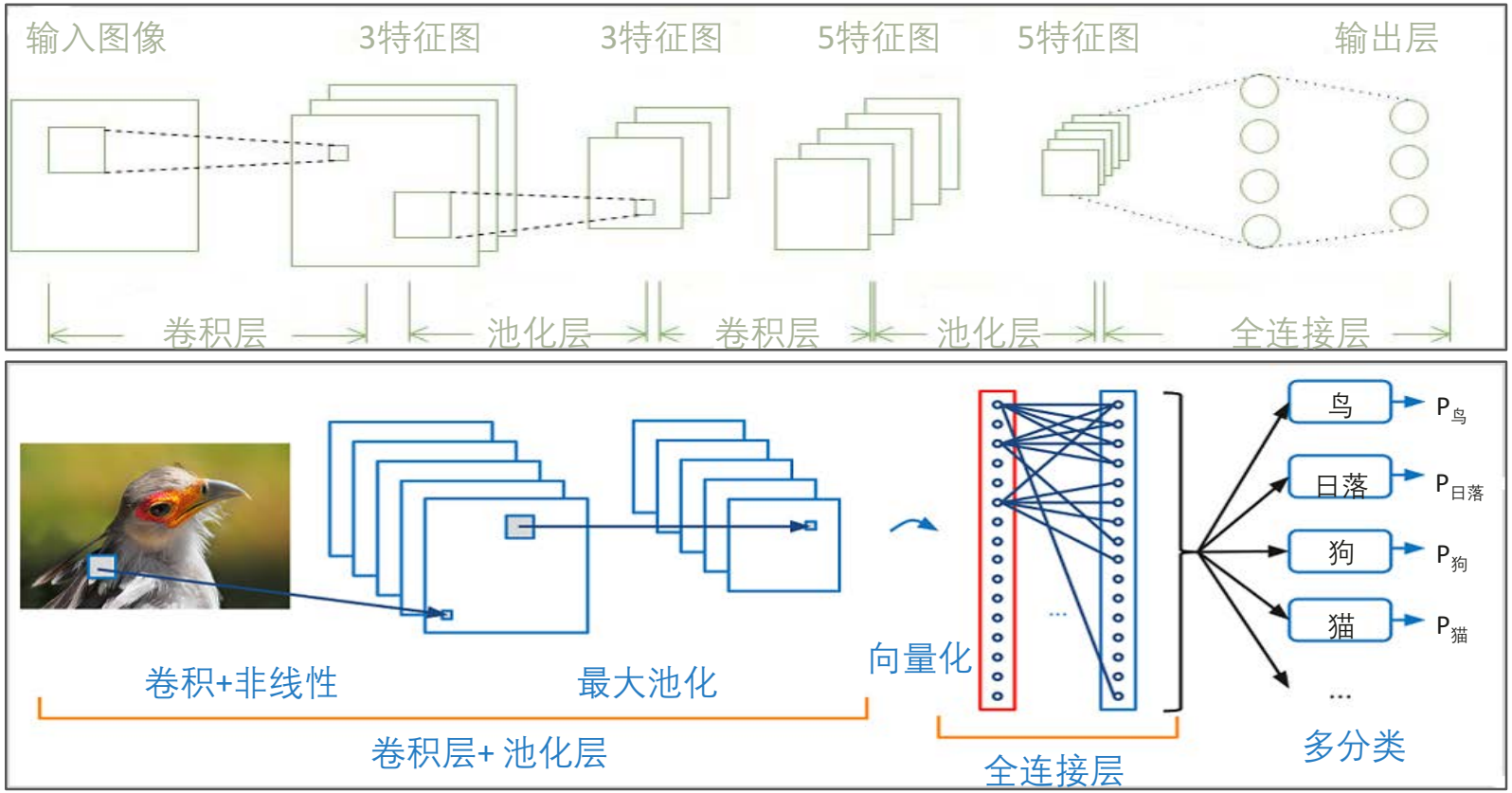
前馈神经网络

- 前馈神经网络是一种最简单的神经网络，各神经元分层排列。每个神经元只与前一层的神经元相连，同一层的神经元之间没有互相连接，层间信息的传送只沿一个方向进行。



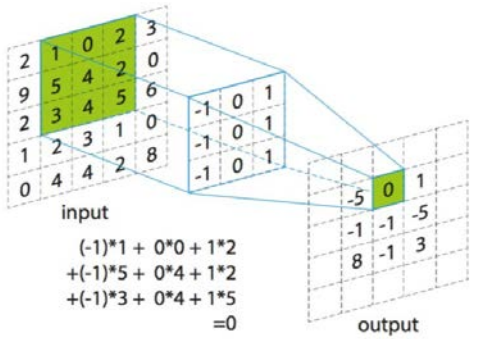
卷积神经网络

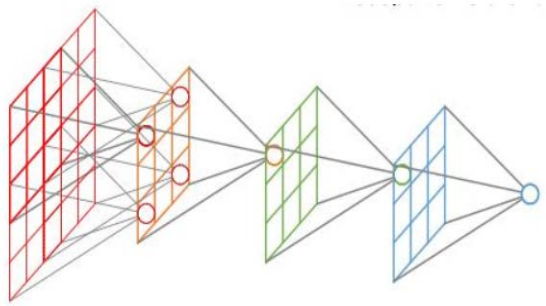
- **卷积神经网络（Convolutional Neural Network, CNN）**是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元，对于图像处理有出色表现。它包括卷积层(convolutional layer)，池化层(pooling layer)和全连接层(fully_connected layer)。

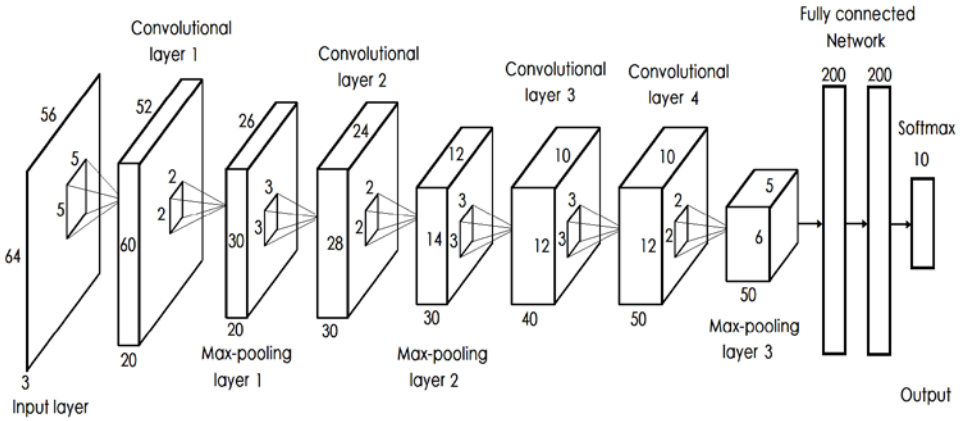


卷积神经网络的重要概念

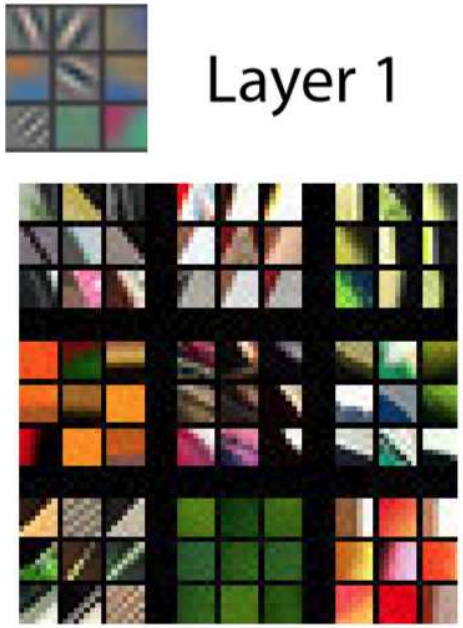
- **卷积核（Convolution Kernel ）**，根据一定规则进行图片扫描并进行卷积计算的对象称为卷积核。卷积核可以提取局部特征。
- **卷积核尺寸（Kernel Size）**，卷积核是一个3维的矩阵，可以用一个立方体图示，宽w，高h，深度d。深度d由输入的通道数决定，一般描述卷积核尺寸时，可以只描述宽w和高h。
- **特征图（Feature Map）**，经过卷积核卷积过后得到的结果矩阵就是特征图。每一个卷积核会得到一层特征图，有多个卷积核则会得到多层的卷积图。
- **特征图尺寸（Feature Map Size）**，特征图也是一个3维的矩阵，可以用一个立方体图示，宽w，高h，深度d。深度d由当前层的卷积核个数决定，一般描述特征图尺寸时，可以只描述宽w和高h
- **步长（stride）**，卷积核在输入图像上滑动的跨度。如果卷积核一次移动一个像素，我们称其步长为 1。
- **零填充（padding）**，为了提取图像边缘的信息，并且保证输出特征图的尺寸满足要求，可以对输入图像边缘填充一个全为0的边框，边框的像素宽度就是padding。
- local receptive field，权值共享，多卷积核（滤波器）
- **结果：** 通过卷积获得了特征 (features) 。



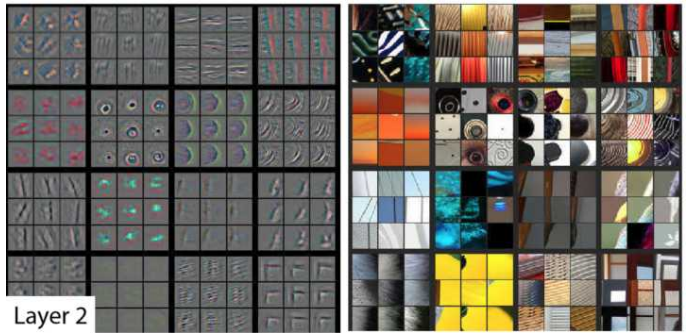




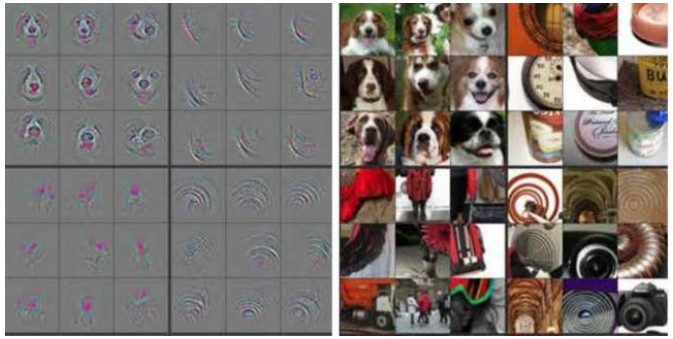
卷积层效果



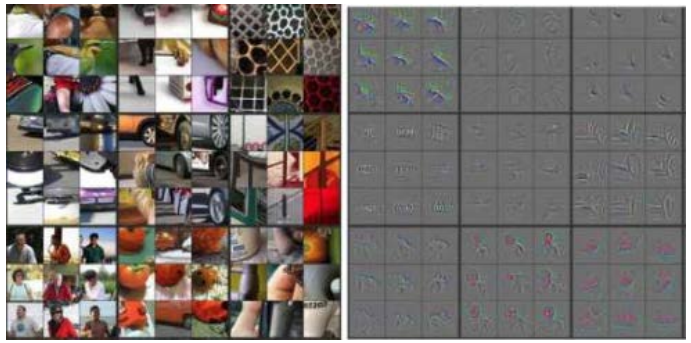
2层



4层



3层



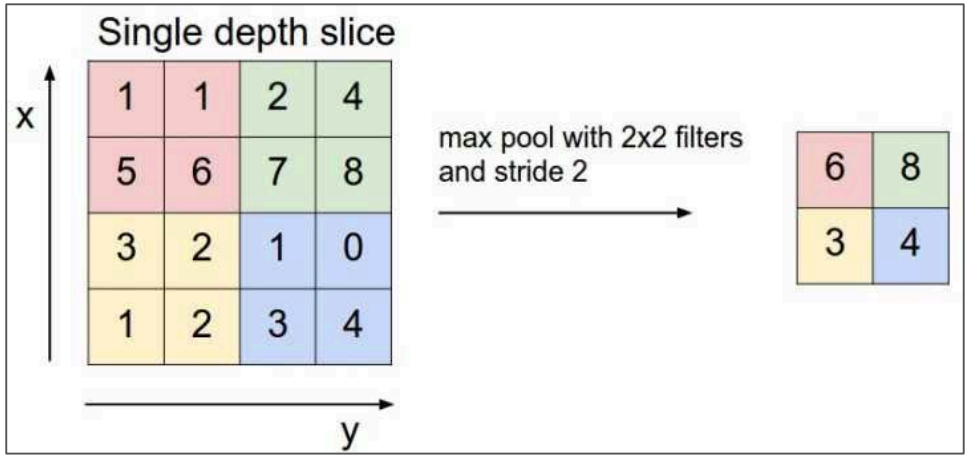
5层

池化

- 有时图像太大，我们需要减少训练参数的数量，它被要求在随后的卷积层之间周期性地引进池化层。池化层一般分为最大池化（max pooling）和平均池化（mean pooling）。
- 目的：不仅具有低得多的维度 (相比使用所有提取得到的特征)，同时还会改善结果(不容易过拟合)

260*200

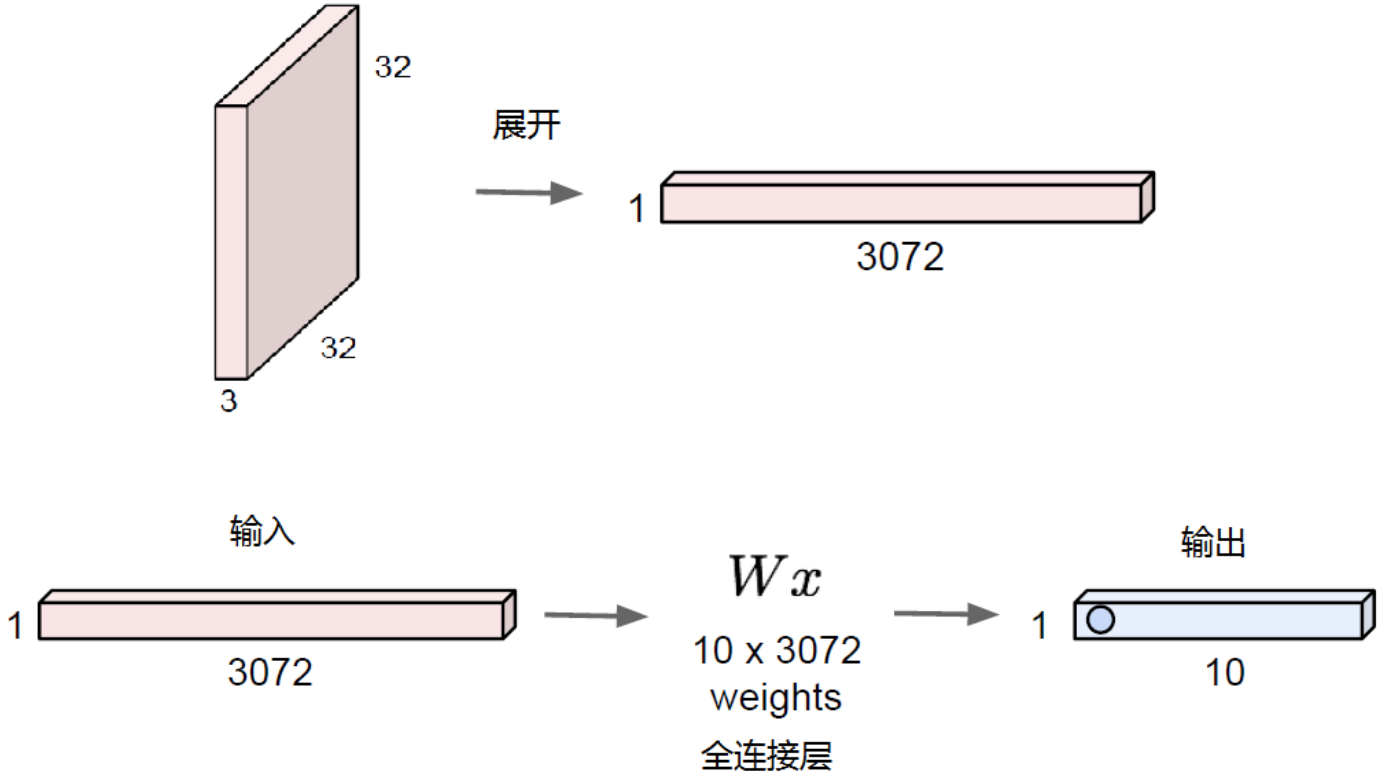
130*100



池化的实现

全连接

- 全连接层可以用来将最后得到的特征映射到线性可分的空间，通常，卷积神经网络会将末端得到的特征图平摊成一个长的列向量，经过全连接层的计算得到最终的输出层。



目录

AI芯片分类

从技术架构来看，大致分为四个类型：

- **CPU（Central Processing Unit，中央处理器）**：是一块超大规模的集成电路，是一台计算机的运算核心（Core）和控制核心（Control Unit）。它的功能主要是解释计算机指令以及处理计算机软件中的数据。
- **GPU（Graphics Processing Unit，图形处理器）**：又称显示核心、视觉处理器、显示芯片，是一种专门在个人电脑、工作站、游戏机和一些移动设备（如平板电脑、智能手机等）上图像运算工作的微处理器。
- **ASIC（Application Specific Integrated Circuit，专用集成电路）**：适合于某一单一用途的集成电路产品。
- **FPGA（Field Programmable Gate Array，现场可编程门阵列）**：其设计初衷是为了实现半定制芯片的功能，即硬件结构可根据需要实时配置灵活改变。

从功能来看，可以分为Training（训练）和Inference（推理）两个类型：

- Training环节通常需要通过大量的数据输入，或采取增强学习等非监督学习方法，训练出一个复杂的深度神经网络模型。训练过程涉及海量的训练数据和复杂的深度神经网络结构，运算量巨大，需要庞大的计算规模，对于处理器的计算能力、精度、可扩展性等性能要求很高。常用的例如NVIDIA的GPU集群、Google的TPU等。
- Inference环节指利用训练好的模型，使用新的数据去“推理”出各种结论，如视频监控设备通过后台的深度神经网络模型，判断一张抓拍到的人脸是否属于特定的目标。虽然Inference的计算量相比Training少很多，但仍然涉及大量的矩阵运算。在推理环节，GPU、FPGA和ASIC都有很多应用价值。

AI芯片生态

CPU

- ❑ **增加指令（修改架构）**的方式提升AI性能
  - Intel（CISC架构）加入AVX512等指令，在ALU计算模块加入矢量运算模块（FMA）
  - ARM（RISC架构）加入Cortex A等指令集并计划持续升级
- ❑ **增加核数**提升性能，但带来功耗和成本增加
- ❑ **提高频率**提升性能，但提升空间有限，同时高主频会导致芯片出现功耗过大和过热问题

GPU

- ❑ 异构计算的主力，AI计算兴起之源，**生态成熟**
- ❑ Nvidia沿用GPU架构，对深度学习主要向两个方向发力：
  - **丰富生态**：推出cuDNN针对神经网络的优化库，提升易用性并优化GPU底层架构
  - **提升定制性**：增加多数据类型支持（不再坚持float32，增加int8等）；添加深度学习专用模块（如V100的TensorCore）
- ❑ 当前主要问题在于：**成本高，能耗比低，延迟高**

FPGA

- ❑ 采用HDL可编程方式，灵活性高，**可重构**（烧），**可深度定制**
- ❑ 可通过多片FPGA联合将DNN模型加载到片上进行低延迟计算，**计算性能优于GPU**，但由于需考虑不断擦写，性能达不到最优（冗余晶体管和连线，相同功能逻辑电路占芯片面积更大）
- ❑ 由于可重构，**供货风险和研发风险较低**，成本取决于购买数量，相对自由
- ❑ 设计、流片过程解耦，**开发周期较长**（通常半年），**门槛高**

ASIC

- ❑ 不考虑带宽、功耗约束，**可按需定制**运算单元数目和物理空间分布，**计算性能和能耗效率极高，业务范围窄（需求明确）**
- ❑ 成本受应用规模约束，**制造成本高**（单片成本低，但厂商只接收一定数量规模定制），发生错误只能重头开始，**风险高**
- ❑ **开发周期长**，18-24月（TPU团队15月），立项到上线时间长
- ❑ **完整的产业链下，价值不在于本身，而是领域扩张走向上下游必经之路**

GPU、CPU设计比对

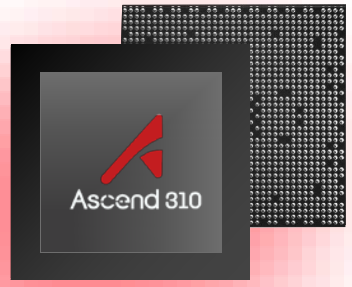
- **CPU需要很强通用性处理不同数据类型，同时需要逻辑判断，还会引入大量分支跳转和中断处理**
  - 由专为串行处理而优化的**几个核心**组成
  - 基于**低延时设计**
    - 强大的ALU单元，可在很短时钟周期完成计算
    - 大量缓存降低延时
    - 高时钟频率
    - 复杂逻辑控制单元，多分支程序可通过分支预测能力降低时延
    - 对于依赖之前指令结果的部分指令，逻辑单元决定指令在pipeline中的位置实现数据快速转发
  - 擅长逻辑控制、串行运算

- **GPU主要面对类型高度统一、相互无依赖的大规模数据和不需打断的纯净计算环境**
  - 拥有若干由**数以千计的更小的核心**（专为同时处理多重任务而设计）组成的大规模并行计算架构
  - 基于大吞吐量设计
    - 有很多ALU和很少cache（和CPU目的不同，为thread提高服务），缓存合并访问DRAM，带来时延问题
    - 控制单元合并访问
    - 大量ALU实现大量thread并行掩盖时延问题
  - 擅长计算密集和易于并行的程序

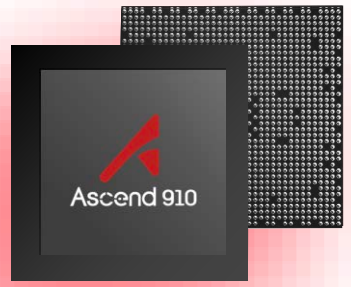
昇腾（Ascend）AI处理器

- **NPU（Neural-Network Processing Units，神经网络处理器）**：在电路层模拟人类神经元和突触，并且用深度学习指令集直接处理大规模的神经元和突触，一条指令完成一组神经元的处理。

NPU的典型代表 —— 华为昇腾AI芯片（Ascend）、寒武纪芯片、IBM的TrueNorth。



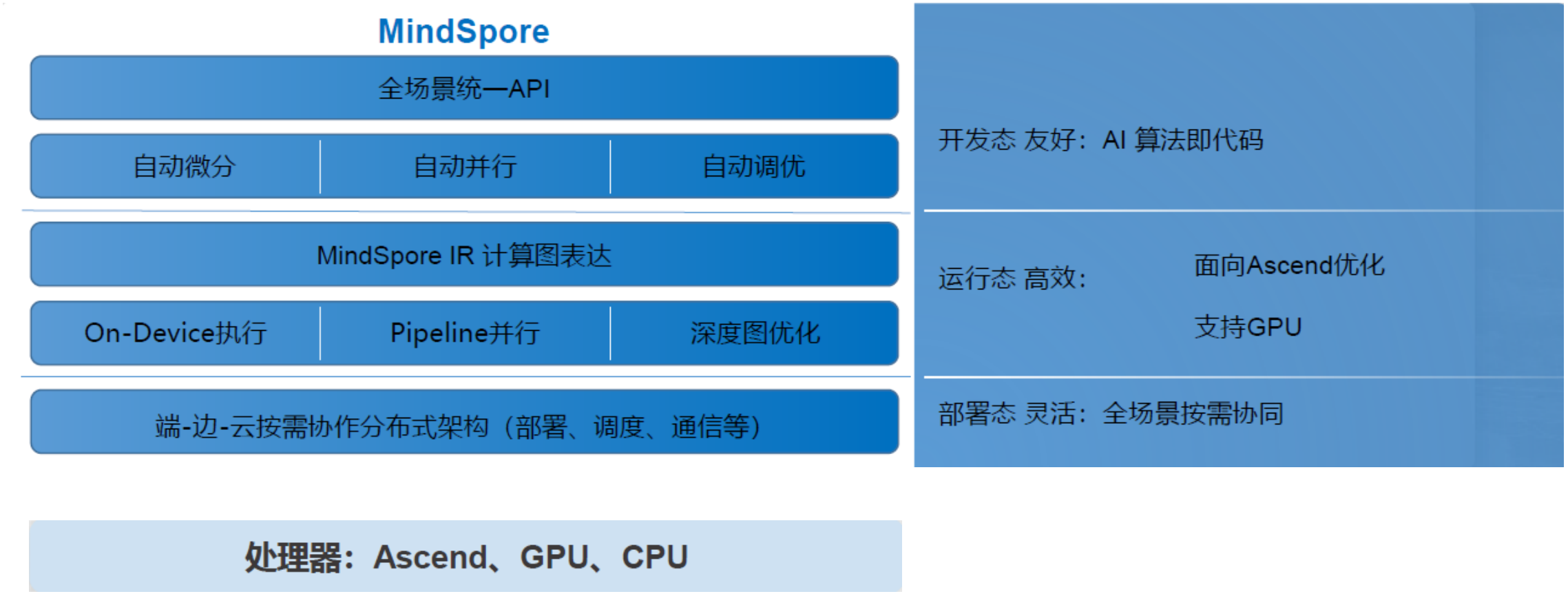
- Ascend-Mini
- 架构: 达芬奇
- 半精度 (FP16): 8 Tera-FLOPS
- 整数精度 (INT8) : 16 Tera-OPS
- 16 通道 全高清 视频解码器 – H.264/265;
- 1 通道 全高清 视频编码器 – H.264/265;
- **最大功耗: 8W**
- 12nm FFC



- Ascend-Max
- 架构: 达芬奇
- 半精度 (FP16): 256 Tera-FLOPS
- 整数精度 (INT8) : 512 Tera-OPS
- 128 通道 全高清 视频解码器 – H.264/265;
- 最大功耗: 350W
- 7nm

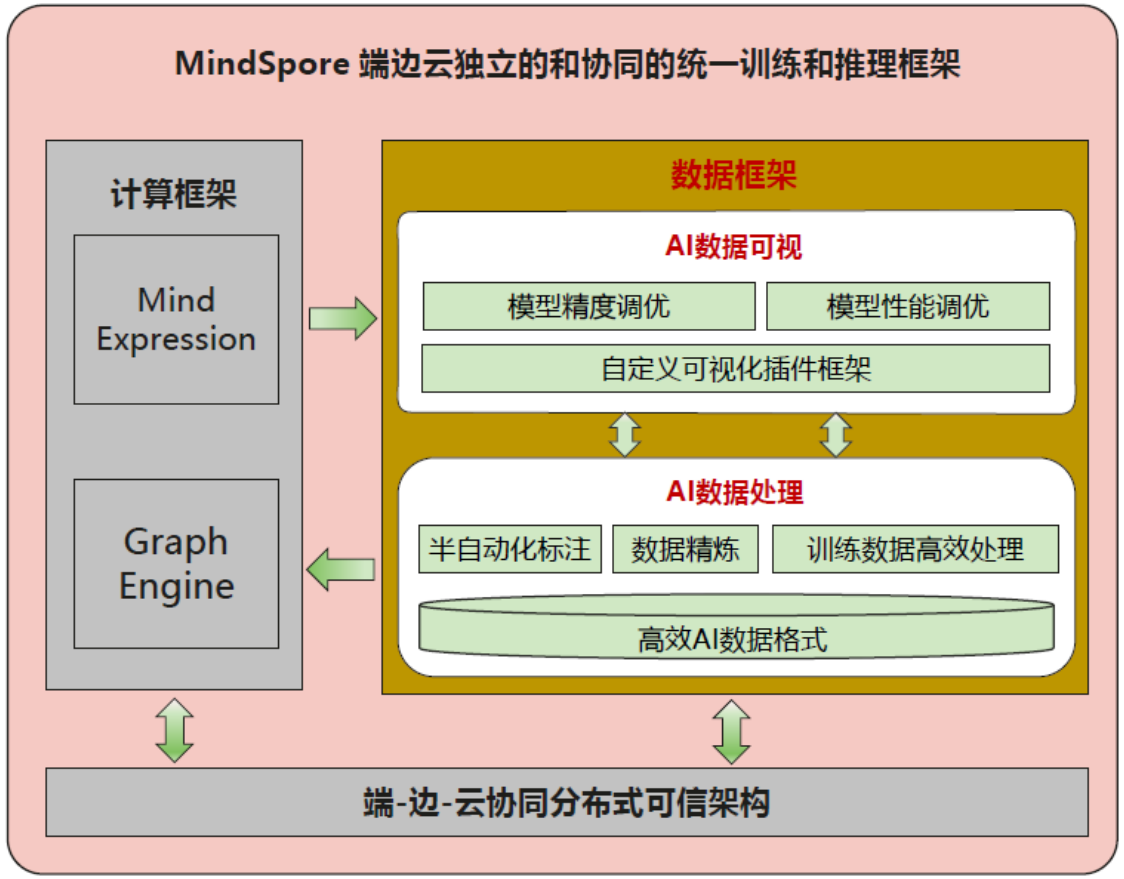
目录

MindSpore —— 核心架构



MindSpore —— 数据框架

- 直面 AI 数据问题与挑战，构建端到端 AI 数据处理与可视化能力。



高效训练数据准备与处理

提供端到端AI数据处理，有效降低数据准备成本，缩短模型训练周期

- **人机协同的半自动标注框架**：快速搭建数据标注系统，加速训练数据准备
- **统一高效的自研数据格式**：自描述可检索的AI数据格式，让训练数据处理更高效
- **数据精炼**：训练时间缩短10%，精度不下降

加速模型精度和性能调优

训练过程可视、软硬件全栈信息可视能力，显著提升模型调优效率和开发者体验。

- **模型精度调优**：错例可视化解释、模型溯源与比对
- **模型性能调优**：数据集可视、鞍点可视、学习率曲线

MindSpore —— 设计理念

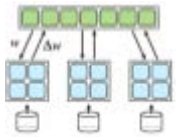
（1）新编程范式，AI算法即代码，降低AI开发门槛

开发态主要挑战

模型开发：　技能要求高

需要 懂数学、懂AI、懂计算机系统

模型调优：　黑盒调优难



模型并行：　并行规划难

严重依赖人的经验，既需要懂数据、模型，也要懂分布式系统架构等

基于数学原生表达的AI编程新范式

让算法专家聚焦AI创新和探索

AI算法原函数　　**AI算法代码**　　**自动生成微分函数**

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

```
B= compute((n,m), Lambda i,j:
    (exp(A[i][j]) -exp(A[i][j]*(-1))/
    (exp(A[i][j]) + exp(A[i][j]*(-1))),
    name='B')
```

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

MindSpore —— 设计理念

（2）新执行模式，Ascend Native 的执行引擎

运行态主要挑战

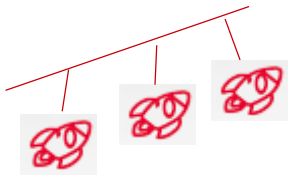
AI计算的复杂性和算力的多样性

① CPU核，CUBE单元、Vector计算单元

② 标量、向量、张量的运算

③ 混合精度计算

④ 稠密矩阵、稀疏矩阵计算

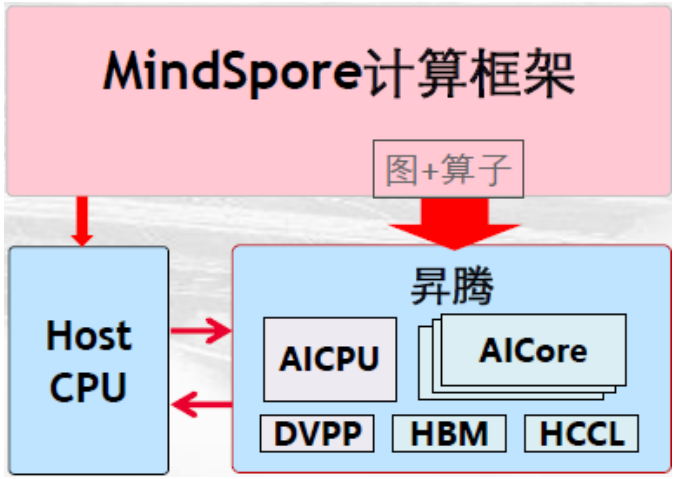


多卡运行： 并行控制开销大

随着节点增加，性能难以线性增加

On-Device执行

整图卸载执行，充分发挥昇腾大算力

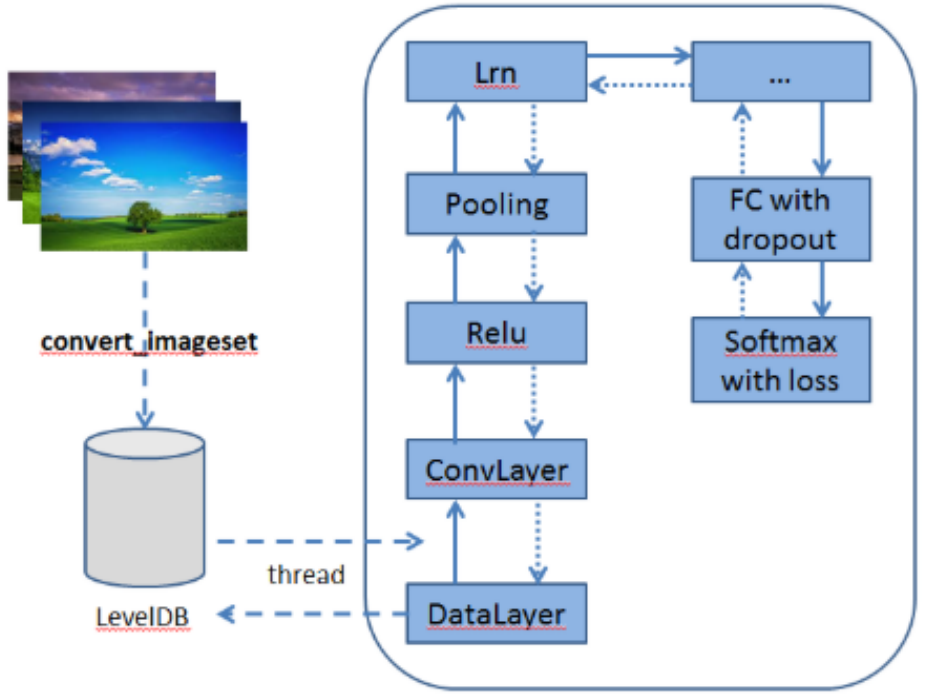


MindSpore —— 设计理念

（3）全场景按需协同，更好的资源效率和隐私保护

Caffe —— 简介

- Caffe，全称Convolutional Architecture for Fast Feature Embedding，是一个兼具表达性、速度和思维模块化的深度学习框架。
- 由伯克利人工智能研究小组和伯克利视觉和学习中心开发。
- 虽然其内核是用C++编写的，但Caffe有Python和Matlab 相关接口。
- Caffe支持多种类型的深度学习架构，面向图像分类和图像分割，还支持CNN、RCNN、LSTM和全连接神经网络设计。
- Caffe支持基于GPU和CPU的加速计算内核库，如NVIDIA cuDNN和Intel MKL。
- Caffe实现了前馈卷积神经网络架构（CNN），在一个n层的神经网络中，通过调整其中的参数，使任何一层的输入和输出都是相等的，任何一层都是输入的另一种表示。深度学习是一种特征学习方法，把原始数据通过一些简单的非线性的模型转化为更高层次、更抽象的表达，高层次的表达能强化输入数据的区分能力，同时削弱不相关因素。



Caffe架构

Caffe —— 组成

Caffe的组成模块包括4个部分：Blob（caffe的数据表示，在layer上流动），layer（不仅可以表示神经网络层，也可以表示输入输出层），Net（神经网络结构，将layer层叠关联起来），slover（协调神经网络的训练和测试，定义一些参数）。

- **Blob** 四维连续数组，通常表示为（n, c, w, h）是基础的数据结构，可表示输入输出数据，也可表示参数数据
- **Layer** 网络基本单元，每一层类型定义了3种计算: 1. 初始化网络参数。2. 前向传播的实现。 3. 后向传播。
- **Net** 无回路有向图，有一个初始化函数，主要有两个作用： 1. 创建blobs和layers。2. 调用layers的setup函数来初始化layers。还有两个函数 Forward和Backward，分别调用layers的 forward 和 backward。
- **Solver** 的作用是：a. 创建用于学习的训练网络和用于评估的测试网络; b. 周期性的评估测试网络; c. 通过调用前馈和后馈函数进行的迭代优化和参数更新。solver每轮迭代都会通过前馈函数计算输出和损失（loss）,还用后馈传播来计算梯度。通过更新学习率等方法更新solver。
- 训练好的Caffe model 是用于保存和恢复网络参数，后缀为 .caffemodel; solver保存和恢复运行状态，后缀为 .solverstate

Caffe —— 组成（续）

Blob

是Caffe作为数据传输的媒介，无论是网络权重参数，还是输入数据，都是转化为Blob数据结构来存储，网络，求解器等都是直接与此结构打交道的。

1. 从数学上来说, Blob就是一个N维数组。它是caffe中的数据操作基本单位，就像matlab中以矩阵为基本操作对象一样，矩阵是2维，Blob是N维的，N可以是2，3，4等
2. 对于图片数据来说，Blob可以表示为（N*C*H*W）这样一个4D数组：

   N：图片的数量，

   C：图片的通道数，

   H：图片的高度，

   W：图片的宽度。
3. 当Blob用于非图片数据，比如传统的多层感知机，就是比较简单的全连接网络，用2D的Blob，调用innerProduct层来计算就可以了。
4. 在模型中设定的参数，也是用Blob来表示和运算。它的维度会根据参数的类型不同而不同。比如：

   1）在一个卷积层中，输入一张3通道图片，有96个卷积核，每个核大小为11*11，因此这个Blob是96*3*11*11.

   2）在一个全连接层中，输入1024通道图片，输出1000个数据，则Blob为1000*1024

Blob 通过类 *SynecdMemory* 来保存数据, 并提供CPU和GPU数据同步的统一接口:

```
class SyncedMemory {
public:
  // 获取保存在cpu/gpu中的常量数据
  const void* cpu_data();
  const void* gpu_data();
  // 获得保存在cpu/gpu的可修改的数据
  void* mutable_cpu_data();
  void* mutable_gpu_data();
  void set_cpu_data(void* data);
  void set_gpu_data(void* data);
  size_t size() { return size_; }
}
```

Caffe —— 组成（续）

Layer

Caffe十分强调网络的层次性，层是网络模型的组成要素和计算的基本单位。比如Data, Convolution, Pooling, ReLU, Softmax-loss等。

Layer主要包含5类：

- **NeuronLayer类** 定义于neuron_layers.hpp中，其派生类主要是元素级别的运算（比如Dropout运算，激活函数ReLu，Sigmoid等）
- **LossLayer类** 定义于loss_layers.hpp中，其派生类会产生loss，只有这些层能够产生loss。
- **数据层** 定义于data_layer.hpp中，作为网络的最底层，主要实现数据格式的转换。
- **特征表达层**定义于vision_layers.hpp 实现特征表达功能，更具体地说包含卷积操作，Pooling操作
- **网络连接层和激活函数**定义于common_layers.hpp，Caffe提供了单个层与多个层的连接，并在这个头文件中声明。这里还包括了常用的全连接层InnerProductLayer类。

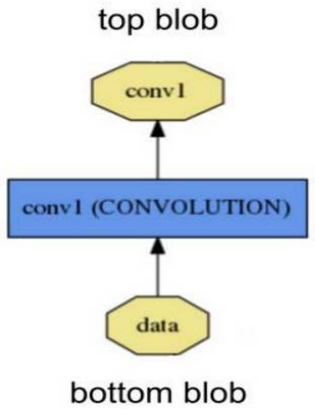
Layer中数据传递：数据主要有两种传递方式：

正向传导（Forward） —— 是根据bottom计算top的过程

反向传导（Backward） —— 根据top计算bottom

Forward和Backward有CPU和GPU（部分有）两种实现。

Caffe中所有的Layer都要用这两种方法传递数据。

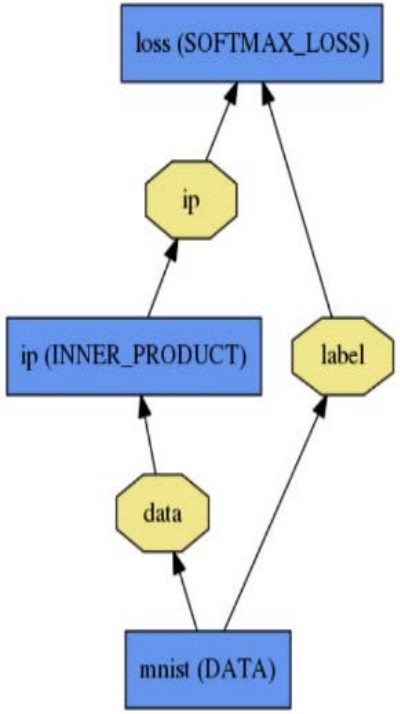


Caffe —— 组成（续）

Net

Net用容器的形式将多个Layer有序地放在一起，其自身实现的功能主要是对逐层Layer进行初始化，以及提供Update( )的接口（更新网络参数），本身不能对参数进行有效地学习过程。

一个简单的2层神经网络的模型定义(加上loss 层就变成三层了)，拓扑如下：



- **第一层**：name为mnist, type为Data，没有输入（bottom)，只有两个输出（top),一个为data,一个为label
- **第二层**：name为ip，type为InnerProduct, 输入数据data, 输出数据ip
- **第三层**：name为loss, type为SoftmaxWithLoss，有两个输入，一个为ip,一个为label，有一个输出loss,没有画出来。
- 对应的配置文件prototxt如右边所示：

```
name: "LogReg"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "input_leveldb"
    batch_size: 64
  }
}
layer {
  name: "ip"
  type: "InnerProduct"
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 2
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
```

Caffe —— 组成（续）

Solver

Solver用于协调着整个模型的运作。交替调用前向（forward)算法和后向（backward)算法来更新参数，从而最小化loss.

caffe程序运行必带的一个参数就是solver配置文件。

```
# caffe train --solver=*_slover.prototxt
```

Solver的流程：

1. 设计好需要优化的对象，以及用于学习的训练网络和用于评估的测试网络。（通过调用另外一个配置文件prototxt来进行）
2. 通过forward和backward迭代的进行优化来跟新参数。
3. 定期的评价测试网络。（可设定多少次训练后，进行一次测试）
4. 在优化过程中显示模型和solver的状态

在每一次的迭代过程中，solver做了这几步工作：

1. 用forward算法来计算最终的输出值，以及对应的loss
2. 调用backward算法来计算每层的梯度
3. 根据选用的slover方法，利用梯度进行参数更新
4. 记录并保存每次迭代的学习率、快照，以及对应的状态。

```
net: “examples/mnist/lenet_train_test.prototxt“
//设置网络模型，即net
test_iter: 100
//和layer中batch_size对应，minist样本10000，Batch_size=100, test_iter=100
test_interval: 500
//测试间隔，每训练500次，进行1测测试
base_lr: 0.01
// 学习率也叫步长，策略由lr_policy定，为inv时： 返回base_lr * (1 + gamma * iter) ^ (- power)
momentum: 0.9
//梯度更新的权重
type: SGD
//weight_decay: 0.0005
//权重衰减项，防止过拟合的一个参数
lr_policy: "inv"
gamma: 0.0001
power: 0.75
display: 100
//每训练100次，在屏幕上显示一次。如果设置为0，则不显示
max_iter: 20000
//最大迭代次数。设置太小，会不收敛，精确度很低。设置太大，导致震荡，浪费时间
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
//快照。将训练出来的model和solver状态进行保存,用于设置训练多少次后进行保存
solver_mode: CPU
//设置运行模式。默认为GPU,如果你没有GPU,则需要改成CPU
```

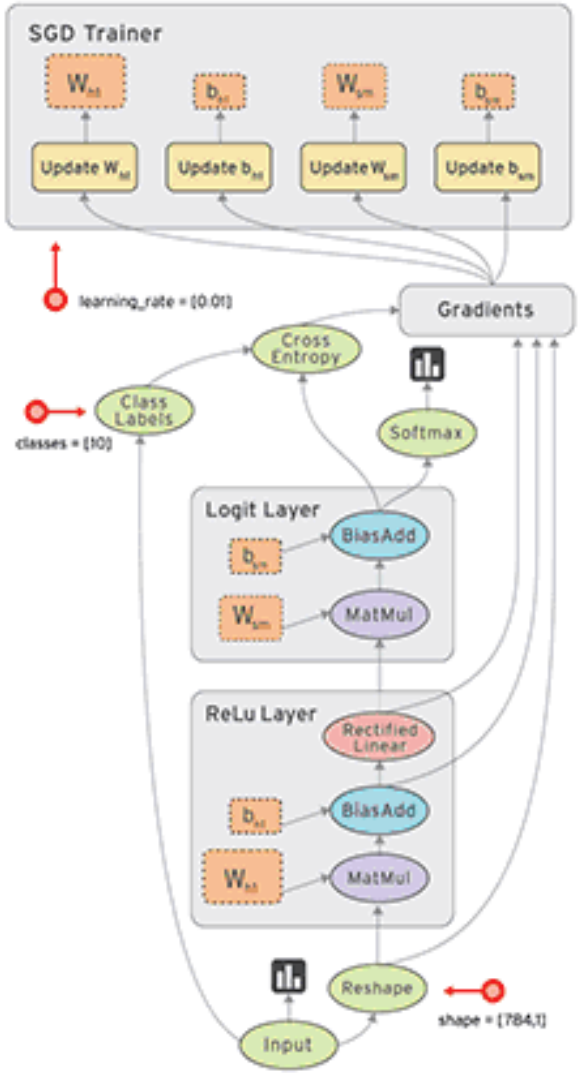
TensorFlow —— 简介

- TensorFlow是Google开发的一款神经网络的Python外部的结构包，也是一个采用数据流图来进行数值计算的开源软件库。其中 Tensor 代表传递的数据为张量（多维数组），Flow 代表使用计算图进行运算。TensorFlow 让我们可以先绘制计算结构图, 也可以称是一系列可人机交互的计算操作, 然后把编辑好的Python文件转换成更高效的C++, 并在后端进行计算。
- TensorFlow可以将复杂的数据结构传输至人工神经网中，进行分析和处理。通过使用TensorFlow可以快速的入门神经网络, 大大降低了深度学习（也就是深度神经网络）的开发成本和开发难度。
- TensorFlow支持CNN卷积神经网络算法、RNN循环神经网络算法和LSTM长短记忆算法，这都是目前在语音识别、自然语言理解、计算机视觉、广告受众分析等领域，最流行的深度神经网络。
- TensorFlow 的开源性, 让所有人都能使用并且维护, 巩固它，使它能迅速更新, 提升。

TensorFlow —— 基础架构知识

- **数据流图**

  TensorFlow是采用数据流图（data flow graphs）来计算，所以首先得创建一个数据流流图，然后再将数据（数据以张量(tensor)的形式存在）放在数据流图中计算。节点（Nodes）在图中表示数学操作，图中的线（edges）则表示在节点间相互联系的多维数据数组，即张量（tensor)。训练模型时tensor会不断的从数据流图中的一个节点flow到另一节点。



TensorFlow —— 基础架构知识（续）

- **会话Session**

  TensorFlow所有的操作都必须在Session会话命令中调用run（运行）函数才能真正起作用。

  Session会话命令（运行）完成有两种方式，具体如下：

  方法一，调用close关闭函数，释放资源

  ```
  ss=tf.Session()
  ss.run(...)
  ss.close()
  ```

  方法二，使用with...as...语句关闭

  ```
  with tf.Session() as ss:
      ss.run()
  ```

TensorFlow —— 基础架构知识（续）

- **TensorFlow数据类型**

1、张量（Tensor）：对应多维Array数组（Numpy），以及List列表。

定义语法：tensor_name=tf.placeholder(type,shape,name)

张量有多种：

零阶张量为纯量或标量 (scalar) 也就是一个数值. 比如 [1]

一阶张量为向量 (vector), 比如 一维的 [1, 2, 3]

二阶张量为矩阵 (matrix), 比如 二维的 [[1, 2, 3],[4, 5, 6],[7, 8, 9]]

以此类推, 还有 三阶 三维的 ...

2、常量（Constant）：常量是无须初始化的变量。

定义语法：name_constant=tf.constant(value)

例子:

```
import tensorflow as tf
matrix1 = tf.constant([[3,3]])          #1行2列矩阵
matrix2 = tf.constant([[2], [2]])       #2行1列矩阵
product = tf.matmul(matrix1,matrix2)
sess = tf.Session()
result = sess.run(product)
print(result)
sess.close()
```

例子:

```
import tensorflow as tf
a= tf.constant([1,3,3])
b= tf.constant([2,3,5])
result = tf.add(a,b,name="add")
print(result)
```

运行结果 `Tensor("add:0", shape=(3,), dtype=int32)`

add:0 表示result这个张量是计算节点”add”输出的第一个结果；
shape=(3,) 表示张量是一个一维数组，这个数组的长度是3；
dtype=int32 是数据类型，TensorFlow会对参与运算的所有张量进行类型的检查，当类型不匹配时会报错。

TensorFlow —— 基础架构知识（续）

- **TensorFlow数据类型**

3、变量（Variable）：由TensorFlow系统内部进行调整的动态参数。

定义语法：name_variable=tf.Variable(value,name)

如果在 Tensorflow 中设定了变量，那么初始化变量是最重要的，所以定义了变量以后, 一定要定义：

init=tf.global_variables_initializer ()

且需要再在 sess 里, sess.run(init) , 激活 init 这一步

例子:

```
import tensorflow as tf
state = tf.Variable(0, name='counter')  # 定义变量 state
one = tf.constant(1)  # 定义常量 one
new_value = tf.add(state, one)  # 定义加法步骤 (注: 此步并没有直接计算)
update = tf.assign(state, new_value)  # 将 State 更新成 new_value
init = tf.global_variables_initializer()  # 如果定义 Variable, 就一定要 initialize
# 使用 Session
with tf.Session() as sess:
    sess.run(init)
    for _ in range(3):
        sess.run(update)
        print(sess.run(state))
```

代码中assign操作是图所描绘的表达式的一部分，和add()操作一样。所以在调用run()执行表达式之前，它并不会真正执行赋值操作。

通常会将一个统计模型的参数表示为一组变量。例如，可以将一个神经网络的权重作为某个变量存储在一个tensor中。在训练过程中，通过重复运行训练图，更新这个tensor.

TensorFlow —— 基础架构知识（续）

- **占位符**

  tf.placeholder是Tensorflow中特有的一种数据结构，类似动态变量。

  Tensorflow 如果想要从外部传入data, 那就需要用到 tf.placeholder()， 然后以这种形式传输数据 sess.run(***, feed_dict={input: **})。placeholder 与 feed_dict={} 是绑定在一起出现的。

例子:

```
import tensorflow as tf
#在 Tensorflow 中需要定义placeholder的type，一般为float32形式
input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
# mul = multiply 是将input1和input2 做乘法运算，并输出为output
output = tf.multiply(input1, input2)
with tf.Session() as sess:
    print(sess.run(output, feed_dict={input1: [7.], input2: [2.]}))
# [ 14.]
```

作用：在训练神经网络时需要每次提供一个批量的训练样本，如果每次迭代选取的数据要通过常量表示，那么 TensorFlow 的计算图会非常大。因为每增加一个常量，TensorFlow 都会在计算图中增加一个节点。所以说拥有几百万次迭代的神经网络会拥有极其庞大的计算图，而占位符却可以解决这一点，它只会拥有占位符这一个节点

TensorFlow —— 基础架构知识（续）

- **Fetch&Feed**

  为了取回操作的输出内容，可以在使用Session对象的run()调用执行图时，传入一些tensor，来取回结果。当获取的多个tensor值时，是在op的一次运行中一起获得而不是逐个去获取的。

  feed使用一个tensor值临时替换一个操作的输出结果。你可以提供feed数据作为run()调用的参数。Feed只在调用它的方法内有效，方法结束，feed就会消失。

例子:

```
import tensorflow as tf
a= tf.constant(2.0)
b= tf.constant(3.0)
c= tf.constant(4.0)
insum=tf.add(a,b)
mul=tf.multiply(a,insum)
with tf.Session() as sess:
    result=sess.run([mul,insum])
    print(result)
```

结果：[10.0, 5.0]

```
import tensorflow as tf
a= tf.placeholder(tf.float32)
b= tf.placeholder(tf.float32)
output=tf.multiply(a,b)
with tf.Session() as sess:
    print(sess.run([output],feed_dict={a:[7.],b:[2.]}))
```

结果：[array([14.], dtype=float32)]

PyTorch —— 简介

PyTorch是什么?

PyTorch是Facebook开发的用于训练神经网络的Python包，也是Facebook打造的深度学习框架。Facebook用Python重写了基于Lua语言的深度学习库Torch，PyTorch不是简单的封装Torch提供Python接口，而是对Tensor上的全部模块进行了重构，新增了自动求导系统，使其成为流行的动态图框架。PyTorch继承了Torch灵活、动态的编程环境和用户友好的界面，支持以快速和灵活的方式构建动态神经网络，还允许在训练过程中快速更改代码而不妨碍其性能，即支持动态图形等尖端AI模型的能力。

PyTorch旨在服务两类场合：

- 替代numpy发挥GPU潜能
- 一个提供了高度灵活性和效率的深度学习实验性平台

PyTorch工作流程非常接近Numpy，且数据类型可以相互转换

- 可以平滑地与Python数据科学栈相结合
- 学习成本很低，几乎没有额外概念

动态计算图：计算图随着代码的执行生成。

- 自然地书写控制逻辑；
- 可以在网络执行过程中进行调试，甚至修改网络；
- 可以很容易的运行部分代码，并实时检查它，不需要等待整个代码都开发完成；
- 不需要关注图结构。

PyTorch —— 主要概念说明

- **Module**
  - 网络（子图）模型定义。
  - 用户通过重写初始化函数与forward函数，定义自己的网络模型。
  - PyTorch中有些算子也是基于此数据结构定义的。
- **Tensor**
  - 张量定义，两个重要属性：devicetype(CPU or CUDA)，datatype（int or float）
  - 会承载算子计算方法，例如：add、sub、conv2d等
- **Variable**
  - Variable是Tensor的一层封装，承载了梯度计算相关的属性，例如grad_fn等
- **Function**
  - 算子定义。Autograd的单元。

PyTorch —— 包含的组件

- **组件**

1. torch：类似于Numpy的通用数组库，可以在将张量类型转换为（torch.cuda.TensorFloat）并在GPU上进行计算。
2. torch.autograd：基于tape的自动微分库，支持torch中所有可微的Tensor运算。
3. torch.nn：一个深度整合了为最大灵活性而设计autograd的神经网络库
4. torch.optim：一个通过torch.nn使用的优化包，包含通用优化算法（如SGD，RMSProp，LBFGS，Adam等）。
5. torch.multiprocessing：微处理，在跨处理的Torch Tensors上带有内存共享，对数据加载和hogwild训练有用。
6. torch.utils：可以方便使用的DataLoader、Trainer等实用的函数。
7. torch.legacy(.nn/.ortim)：为了向后兼容而从Torch移植的旧代码。

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home, and organization for a fully connected, intelligent world.

HUAWEI