CS7345 智能计算系统 课程笔记

2025 Fall, UNIkeEN

全部课程笔记

1. 绪论

人工智能的定义

人工智能是指由人制造出来的机器系统所表现出来的智能

- 弱人工智能：完成某种特定具体任务（比如目标检测、语音识别）
- 强人工智能（通用人工智能）：具备或超越人类，表现正常人类所具有的所有智能行为

三个流派

符号主义

基于符号逻辑的方法，用逻辑表示知识和求解问题（就像离散数学考试的超长表达式）

- 困难：
  - 逻辑：未找到能表述世间所有知识的简洁逻辑体系
  - 常识：无穷无尽的常识
  - 求解器：命题逻辑判定 NP 完全，一阶谓词逻辑（引入了量词 $\forall \exists$、谓词、关系）不可判定

> 符号主义推理常需要在巨大离散空间搜索，推理时间爆炸
>
> 可判定问题是指存在程序在有限步数可以输出正确答案；但显然一阶逻辑无法解决停机问题

- 本质问题：只考虑理性认识，没有考虑人类智能的感性认识

行为主义

基于**控制论**，环境感知/作用反馈 - 动作控制，类似小脑。

- 代表：波士顿机器狗，人形机器人

连接主义

基于大脑中神经元细胞连接的计算模型，**人工神经网络**拟合

- 代际：单层感知机 -> MLP -> 深度神经网络

单层感知机无法表示 XOR，是人工智能第一次寒冬的重要导火索之一

三次浪潮

深度学习

- 成功三要素：迭代的优化算法，更大的数据量，更强的算力
  - 对应 LLM 里的 **Scaling Law**：大参数规模、大数据集、大算力
    - 模型在测试集上的 Loss 和算力、数据、参数规模这三者的大小呈幂律关系，随着三者数量级增加，Loss 呈反比下降
    - 目前 LLM 在数据集量上达到瓶颈，高质量数据已基本被完全使用，模型生成数据难以学习更强模型；转向强化学习
- 局限性（18年之前）：泛化能力有限（常需要高质量样本监督，与人类成长的类强化学习不同），缺乏推理能力，缺乏可解释性（为什么这么设计？为什么这个维度？...），鲁棒性欠佳

大语言模型

- 大语言模型三结构：Encoder-decoder，仅编码器（BERT），仅解码器（GPT）

大模型加速科技演进，迭代速度显著快于传统机器学习和传统深度学习

智能计算系统：智能的物质载体

现阶段智能计算系统：集成 CPU 和智能芯片的 **异构** 系统，包含一套智能计算编程环境（编程框架和编程语言）

- 异构原因：CPU 摩尔定律放缓，GPU 等性能指数增长，形成剪刀差

- 异构困难：提高性能的同时带来编程上困难（编程语言不同、层次性变高）

三代智能计算系统

- 第一代：1980 年代，面向符号主义专用计算机（巅峰，日本五代机计划，失败）
  - 逻辑编程语言，用 facts、rule、query 由解释器得出答案
  - 编程语言和硬件统一化，只针对特定语言优化
  - 由于符号主义本身局限性，需求低（逻辑命题难以面对实际问题），成本高（不如摩尔定律全盛时期的 CPU）
- 第二代：2010 年代，面向连接主义专用计算机（深度学习计算机 / 超算，📍 We are here.）
  - 优势：实际工业应用，专用处理器（GPU 等）发展
- 第三代：未来强人工智能的载体

2. 神经网络基础

人工智能 ⊃ 机器学习 ⊃ 神经网络 ⊃ 深度学习

机器学习是研究如何通过经验（数据）改善系统性能的一门学科

线性回归

单变量：$H_w(x) = w_0 + wx$

多变量：$H_w(x) = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n = \mathbf{w}^T\mathbf{x}$

损失函数

- 首先引入误差项：$y = \mathbf{w}^T\mathbf{x} + \varepsilon$
- 假设误差满足高斯分布：$\varepsilon \sim \mathcal{N}(0, \sigma^2)$

$$p(\varepsilon) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(\varepsilon)^2}{2\sigma^2}\right) \tag{1}$$

  - 经验层面：实际问题中测量误差、环境噪声、人为扰动通常满足高斯分布
  - 理论层面：**中心极限定理** 证明很多独立小扰动的和近似高斯分布
  - 工程层面：假设为高斯数学可解，损失函数易于优化
- 那么等价于：给定 x 和参数 w 情况下，y 满足高斯分布

$$p(y \mid x; \mathbf{w}) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y - \mathbf{w}^Tx)^2}{2\sigma^2}\right) \tag{2}$$

- 由于数据已经给定（训练集），需要选 w 使数据最可能生成，有似然函数

$$\mathcal{L}(\mathbf{w}) = p(\mathcal{D} \mid \mathbf{w}) = \prod_{j=1}^{m} p(y_j \mid x_j; \mathbf{w}) \tag{3}$$

- 最大似然估计：$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \mathcal{L}(\mathbf{w})$
- 代入高斯，取对数

$$\log \mathcal{L}(\mathbf{w}) = \sum_{j=1}^{m} \left[-\log(\sqrt{2\pi}\sigma) - \frac{(y_j - \mathbf{w}^T x_j)^2}{2\sigma^2} \right] \tag{4}$$

- 去掉无关项后，最大化似然函数（取反）变为最小化损失函数：

$$L(\mathbf{w}) = \frac{1}{2} \sum_{j=1}^{m} (\mathbf{w}^T x_j - y_j)^2 \tag{5}$$

- 是为平方误差函数的真正来源

参数优化：梯度下降

为使损失函数最小，常使用梯度下降法（沿着损失函数下降最快的方向）更新以寻找参数

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial L}{\partial \mathbf{w}} \tag{6}$$

α 是学习率，具体细分为：

- 梯度下降（GD）：每一步用**全体样本**算梯度：

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta) = \theta - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \ell_i(\theta) \tag{7}$$

  - 更新方向稳定，loss 通常平滑下降，但单步计算成本高
  - 在凸问题里（比如线性回归）可稳定收敛到最优点
- 随机梯度下降（SGD）：每一步**随机抽 1 个或一小批样本**算梯度（工程实际使用）：

$$\theta \leftarrow \theta - \eta \nabla_\theta \ell_j(\theta) \quad (j \sim \text{Uniform}\{1, \dots, m\}) \tag{8}$$

  - 梯度是“带噪声的估计”，方向会抖动，loss 曲线颠簸，但常常更快到达还不错的区域
  - 单步便宜，能快速频繁更新，适合大数据/在线学习
  - 由于噪声，通常会在最优点附近“震荡”；需要学习率衰减（schedule）才能更接近最优

单层感知机

单层感知机

从回归任务变化到了 **分类** 任务，对应一个超平面 $\mathbf{w}^T\mathbf{x}+b=0$

$$H(x)=\text{sign}(\mathbf{w}^T\mathbf{x}+b) \tag{9}$$

目标是找到 $\mathbf{w},b$ 将线性可分数据集正确分类

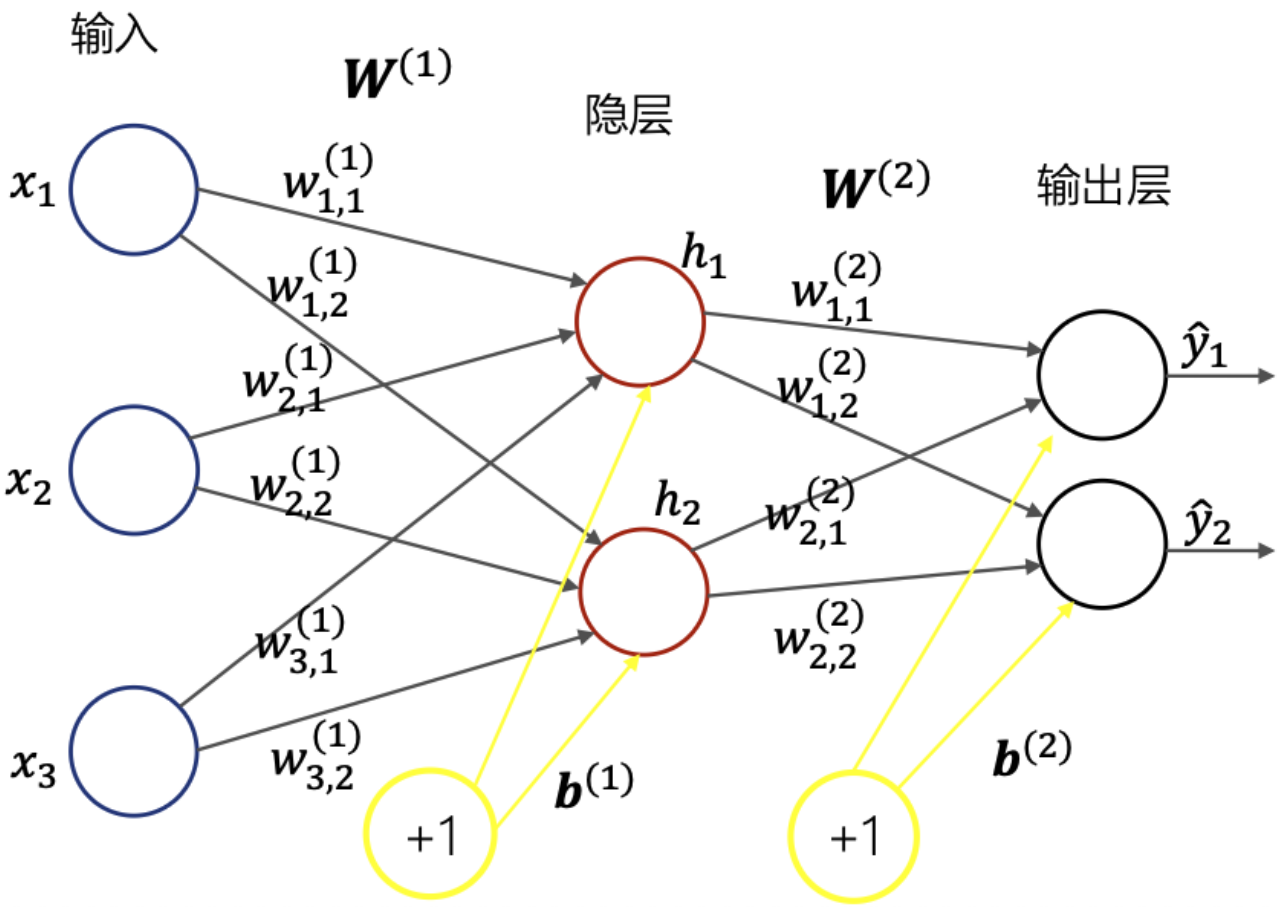
损失函数

$$L(w,b)=-\sum_{i\in M}y_i(w^Tx_i+b) \tag{10}$$

只关心分错的点（y 和 $\hat{y}$ 不同号），且直观地看、错误的点离平面距离 $\gamma_i=\frac{y_i(w^Tx_i+b)}{\|w\|}$ 越远越需要惩罚

多层感知机

单层神经元能力有限（比如解决不了 XOR），组合多个神经元 + **非线性激活函数** 组成人工神经网络



- 没有激活函数，多层全连接网络再深也只是线性模型，等价于一层线性回归/单层感知机，只能学到线性可分的边界

学到线性可分的边界

神经网络

输入的节点数与特征的维度匹配，输出层的节点数与目标的维度匹配，神经网络设计者只需要设计隐层的节点数和层数

- 隐层节点数设计方法：网格搜索（预先设定可选值，选择效果最好的组合）

浅层神经网络

- 需要数据量小、训练快
- 对于复杂函数表示能力有限，针对复杂问题泛化能力受到制约

Kurt Hornik证明了 **理论上两层神经网络足以拟合任意函数**。但是隐层节点数量可能会爆炸

多层神经网络（深度学习）

> Hinton 提出，和 LeCun、Bengio 合称深度学习开创者

深度学习成功因素：ABC（见第一节）

- 随着网络的层数增加，每一层对于前一层次的抽象表示更深入，每一层神经元学习到的是前一层神经元更抽象的表示
    - 拟合非线性分界不断增强

全连接 MLP 参数量计算

单层：从上一层 n_{in} 个神经元 $\rightarrow$ 下一层 n_{out} 个神经元：

- 权重矩阵 W 形状：$n_{\text{in}} \times n_{\text{out}}$
- 偏置向量 b 长度：n_{out}

多层总参数量（n_0 输入维度，n_L 输出维度）：

$$\#\text{total} = \sum_{l=1}^{L} (n_{l-1} n_l + n_l) = \sum_{l=1}^{L} (n_{l-1} + 1) n_l \tag{11}$$

神经网络训练

目的：调整参数使模型计算值 $\hat{y}$ 尽可能接近真实值 y

过程：正向传播 -> 反向传播（计算损失函数，链式求导，梯度回传，更新参数）

神经网络设计原则

设计较准确的神经网络，通常有调整网络设计、选择合适的激活与损失函数、调整超参数等方法

网络拓扑结构

调整隐层数量和隐层神经元的个数（也就是参数量）。过少欠拟合（难以提取信息）、过多过拟合（泛化能力变差）。

选择激活函数

激活函数需要具备的性质：

- **可微性**：优化方法基于梯度时必须
- **输出值范围**：激活函数输出也是下一层输入，输出值有限时基于梯度的优化方法会更稳定，激活函数输出无限时模型训练更高效、但通常需要较小学习率

sigmoid 函数

最常见的非线性激活函数

- 公式：$\sigma(x) = \frac{1}{1+e^{-x}}$
- 导数：$\sigma'(x) = \sigma(x)\big(1-\sigma(x)\big)$
- 输出范围：$(0,1)$，常用于“概率”解释
- 缺点：
  - **非 0 均值输出**：输出始终为正，导致梯度更新方向容易“同号偏移”，收敛慢
  - **存在指数运算**：计算机进行指数运算速度慢
  - **饱和性问题**：绝对值较大的值得到梯度趋近于0；多层 sigmoid **梯度消失**

Tanh 函数

解决了 sigmoid 神经元输出均值非 0 的问题

- 公式：$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
- 导数：$\tanh'(x) = 1-\tanh^2(x)$
- 输出范围：$(-1,1)$
- 缺点：
  - **饱和性问题还存在**，输入很大/小时输出几乎平滑，梯度很小

ReLU 函数

- 公式：$f(x) = \max(0,x)$
- 特点：在 (x>0) 时梯度不衰减（恒为 1），因此能**缓解梯度消失问题**；计算简单（比较 + 取值）、输出范围无限
- 缺点：

    - **ReLU 死掉**问题，学习率很大时、反向传播后的参数为负数，下一轮正向传播输入为负数，ReLU 不被激活（死掉）
- 缓解方案：Leaky ReLU（负数部分较小斜率）、PReLU、ELU（融合 sigmoid 和 ReLU，负数部分软包合、对噪声更鲁棒）

选择损失函数

一个典型问题：常用的均方差损失函数与 sigmoid 函数 σ 一起使用时，梯度里将包含一系列 $\sigma'(\cdot)$，在神经元输出接近 1 时梯度趋于 0，出现梯度消失，参数更新缓慢

交叉熵

- 解决上述问题的方案：**交叉熵** 损失函数，在计算梯度时会把 $\sigma'(\cdot)$ 消掉
- 二分类下的交叉熵：把二分类看成伯努利分布（Bernoulli）：

$$p(y \mid x) = \hat{y}^y (1-\hat{y})^{1-y}$$

取负对数似然（NLL）得到交叉熵（单样本）：

$$L_{\text{CE}} = -\log p(y \mid x) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$$

这就是二分类交叉熵（Binary Cross-Entropy）。

➢ 以二分类为例，则使用Sigmoid激活函数时的交叉熵损失函数为：

$$L = -\frac{1}{m}\sum_{x \in D}(y \ln(\hat{y}) + (1-y)\ln(1-\hat{y}))$$

$$\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-(w^T x+b)}}$$

$$\frac{\partial L}{\partial w} = -\frac{1}{m}\sum_{x\in D}\left[\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right]\cdot\frac{\partial\sigma(z)}{\partial w} = -\frac{1}{m}\sum_{x\in D}\left[\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right]\cdot\sigma'(z)\cdot x = \frac{1}{m}\sum_{x\in D}\frac{\sigma'(z)\cdot x}{\sigma(z)(1-\sigma(z))}\cdot(\sigma(z)-y)$$

$$\sigma'(z) = \big(1-\sigma(z)\big)\cdot\sigma(z)$$

$$\frac{\partial L}{\partial w} = -\frac{1}{m}\sum_{x\in D}(\sigma(z)-y)\cdot x$$

同理得：$$\frac{\partial L}{\partial b} = -\frac{1}{m}\sum_{x\in D}(\sigma(z)-y)$$

sigmoid的导数被约掉，这样最后一层的梯度中就没有$\sigma'(z)$

- 交叉熵通用形式：

$$L = -\frac{1}{m}\sum_{x\in D}\sum_{i} y_i \ln(\hat{y}_i) \tag{12}$$

m 为训练样本的总数量, i 表示分类类别索引

过拟合与正则化

- **泛化**：提升模型在测试集上的预测效果
- **过拟合**：模型过度接近训练数据（模型复杂性大于实际问题），训练误差低但验证集误差大
- 抑制过拟合的方法是 **正则化**

正则化常用方法

- 参数范数惩罚：在损失函数中增加对权重的惩罚
- 稀疏化：让网络很多权重/神经元/激活单元为 0，降低正向传播计算量
- Bagging 集成：训练不同模型共同决策测试样例的输出，模型平均时减小泛化误差的可靠方法
- Dropout：训练时随机关掉一部分神经元（不更新参数），**推理**时不关闭、也没有让参数变为 0（注意与稀疏化的区分）
  - 直觉：相当于训练了很多子网络做集成
- 其他：提前终止（在验证误差再次上升时提前终止）、多任务学习、数据增强、参数共享

交叉验证

交叉验证的方式给每个样本作为测试集和训练集的机会，充分利用样本信息，保证鲁棒性，防止过度拟合

> 选择多种模型进行训练时，使用交叉验证能够评判各模型的鲁棒性。

- 普通验证方式：一次性划分训练集、测试集
  - 结果依赖这次划分，只有部分数据参与训练

- Leave-one-out 留一交叉验证：每次取 1 个样本做测试集，重复 n（样本个数）次并对误差取平均
    - 每个样本都被验证一次，数据利用率高；但是计算量大、耗时长
- K 这交叉验证：分 K 份轮流做测试集
    - 较第二种方法计算成本更低、耗时更少
    - 第二种方法是 $K = n$ 的特殊情况

3. 深度学习

卷积神经网络

在做图像处理任务时，相较于全连接层有效 **减少权重数量**

重要特征：局部连接、权重共享

卷积层输出尺寸公式

$$w_o = \left\lfloor \frac{w_i + 2p - f}{s} \right\rfloor + 1, \quad h_o = \left\lfloor \frac{h_i + 2p - f}{s} \right\rfloor + 1 \tag{13}$$

- p：边界扩充 padding（公式里是单侧扩充宽度）
    - 防止深度网络中图像被动持续减小
    - 强化图像边缘信息
- s：卷积步长 stride
- f：卷积核大小（通常为方形）

卷积层运算次数

c_i - 输入通道数，c_o - 卷积核个数

- 每个输出像素：$f \times f \times c_i$ 次乘法
- 总乘法次数：

$$\mathrm{Mul} = w_o \times h_o \times c_o \times f^2 \times c_i \tag{14}$$

- 每个输出像素：$(f \times f \times c_i - 1)$ 次加法（对应到硬件实现）
- 总加法次数：

$$\mathrm{Add} = w_o \times h_o \times c_o \times (f^2 \times c_i - 1) \tag{15}$$

卷积层参数量

包含偏置：

$$\#\text{params} = f^2 \times c_i \times c_o + c_o \tag{16}$$

池化层

Max pooling / Avg pooling / L2 pooling

主动减小图片尺寸，减少参数数量和计算量，控制过拟合，不引入额外参数

- max pooling 可保留特征最大值，提高提取特征的鲁棒性

池化层输出公式与卷积层 **相同**，单常见设置 $s = f$ 所以做到下采样

典型卷积神经网络

结构: [卷积层（激活）- 池化层] - [全连接层] - [softmax]

卷积层和池化层做特征提取，全连接层作为分类器。全连接层将高维特征图映射成一维特征向量，转化为各个类别概率（softmax 归一化，凸显最大值）

浅层卷积学习局部特征，深层学习整体特征，可以从局部到整体理解图像

AlexNet

- 使用 ReLU，训练收敛快
- LRN 局部归一化，提升较大响应（后续研究发现作用不大）
- 使用 maxpool，避免特征被平均池化模糊，提升特征鲁棒性
- 使用 dropout，避免过拟合

VGG

- 深而规整的卷积-池化结构
- 多层小卷积比单层大卷积效果好，相同感受野下多层网络权值更少
    - 7x7 卷积 49 个参数；相同感受野可以是三个 3x3 卷积（27个参数）
- 对于深层网络，靠近输入的可能训练不动；VGG 使用部分网络层参数的预初始化，提高训练收敛速度

Inception

- 一个卷积-池化块中有多尺寸卷积核，做特征并行提取
    - 1x1 卷积做跨通道融合 + 降维
- softmax 辅助分类网络：中间某一层输出旁路经过 softmax 得到分类结果，按较小权重加到最后结果（模型融合），只从中间结果回传梯度可防止梯度消失
    - 推理时去除

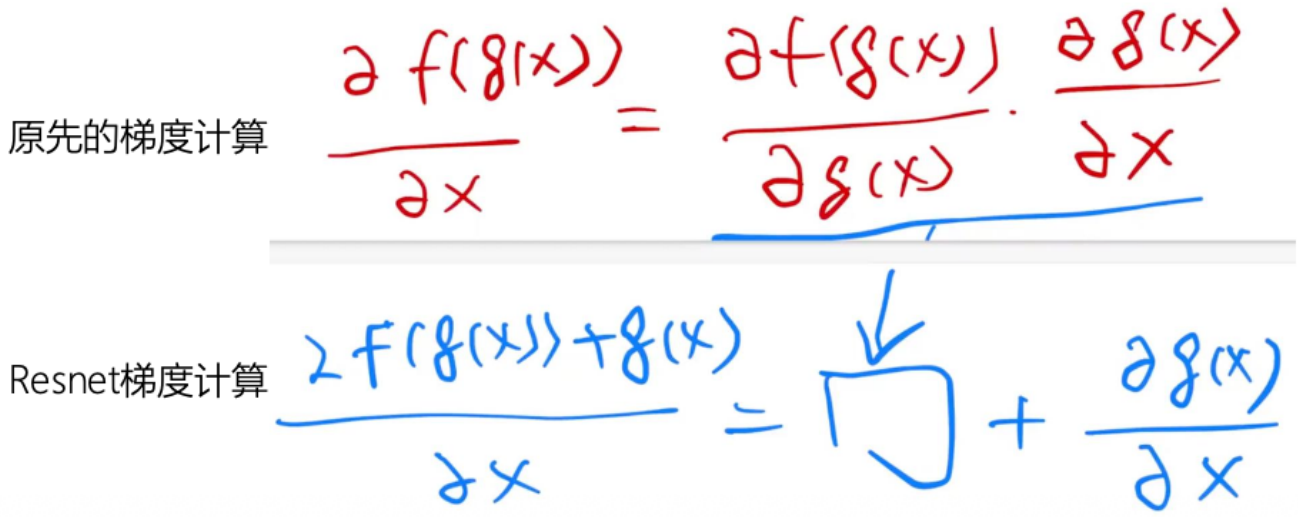
- Factorization 思想（Inception-v3）：将 3x3 卷积拆分成 1x3 和 3x1 卷积，减少参数数量，同时通过非对称卷积结构拆分增加特征多样性

ResNet

问题背景：

- 网络加深后出现**退化问题**（错误率不降反升）
- 不是过拟合（深层网络训练集误差同样高），也不是梯度消失（使用 BatchNorm 可有效缓解梯度消失）

解决方法：残差连接
* 训练比较快 -> 梯度保持的较好 -> 训练得动



循环神经网络

输入具有时间相关性，需要记忆历史信息

RNN 状态更新：

$$h^{(t)} = f(Wh^{(t-1)} + Ux^{(t)} + b) \tag{17}$$

特点：

- 参数时间上共享
- 理论上可处理任意长度序列

反向传播时问题：

- 梯度中包含多次矩阵连乘（多时间步），梯度易爆炸或消失，导致只能学习到短期依赖关系
    - 改进1: 梯度截断解决爆炸问题
    - 改进2: 模型改进（LSTM、GRU）解决消失问题

LSTM（Long Short-Term Memory）

LSTM（Long Short-Term Memory）

核心思想：引入一条**显式的记忆通道**：单元状态 c_t，用多个门来**精细控制信息流动**

- **遗忘门（forget gate）**：决定旧记忆 c_{t-1} 保留多少

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{18}$$

- **输入门（input gate）**：决定新信息写入多少

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{19}$$

- **候选记忆**

$$\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \tag{20}$$

- **更新单元状态**

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{21}$$

- **输出门（output gate）**

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{22}$$

- **隐状态输出**

$$h_t = o_t \odot \tanh(c_t) \tag{23}$$

优点：显式记忆单元，长期依赖建模能力强；梯度近似线性传播、稳定
缺点：门多、参数多、计算复杂

GRU（Gated Recurrent Unit）

- **简化版 LSTM**
- 不区分 c_t 和 h_t
- 用更少的门实现类似效果
- **更新门（update gate）**：决定“保留旧状态 vs 更新新状态”

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z) \tag{24}$$

- **重置门（reset gate）**：决定旧状态在生成新信息时的参与程度

$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r) \tag{25}$$

- **候选隐状态**

$$\tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h) \tag{26}$$

- **状态更新**

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{27}$$

优点：结构更简单、参数更少，实践中性能接近 LSTM

Seq2Seq

传统 Seq2Seq 的缺陷：编码器把输入编码成固定长度语义编码

- 固定长度语义向量，难以存储较长序列的所有信息
- 语义编码每个元素权重相同，模型无法区分各个元素重要程度

注意力机制

- 对输入部分赋予不同权重来聚焦重要信息、忽略不重要信息
- 参数少，计算和存储开销小
- 可以并行运算

基本形式：

$$\alpha_{ij} = \mathrm{softmax}(a(q_i, h_j)), \quad e_i = \sum_j \alpha_{ij} h_j \tag{28}$$

（1）计算查询和所有键相似度，利用相似度计算每个键值对的注意力权重
（2）利用注意力权重计算所有值的注意力汇聚结果，比如加权求和

缩放点积注意力

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{29}$$

- 除以 $\sqrt{d_k}$ 用于控制数值尺度（分子 QK^T 是相关关系权重，这个内积的方差为 d_k，具体可以用方差公式推导），稳定梯度

多头注意力

类似多通道卷积，对 QKV 做多组线性变换，并行关注不同子空间特征

自注意力

QKV来自同一组输入，捕捉数据内部的相关关系

> 以翻译例子为例，训练和推理的区别，输入输出是什么、训练为什么可以并行

Transformer

编码器：

- Self-Attention

- Self-Attention
- Feed Forward
- 残差连接 + LayerNorm

解码器：

- Masked Self-Attention
- Cross Attention（Encoder–Decoder）

位置编码：余弦公式

例子（翻译“机器学习改变世界”）

- 源序列（中文）：`[机器，学习，改变，世界]`，长度 $n = 4$
- 目标序列（英文）：`<bos> Machine learning changes the world <eos>`，长度 $m = 7$

训练目标：在每个位置 t，预测目标序列的下一个词（next token prediction）。

编码器前向

得到源端上下文表示：

1. 源端 token → id → embedding，得到

$$X_{enc} \in \mathbb{R}^{n \times d_{model}} \ (n = 4) \tag{30}$$

2. 加位置编码（positional encoding）
3. 经过 L 层 encoder block（self-attention + FFN），得到

$$H_{enc} \in \mathbb{R}^{n \times d_{model}} \tag{31}$$

后续供解码器查询使用

解码器输入训练构造

这里是监督学习，已知真实翻译

目标序列：

- `y = [<bos>, Machine, learning, changes, the, world, <eos>]`

训练时解码器输入（右移一位）：

- `y_in = [<bos>, Machine, learning, changes, the, world]`

训练标签（要预测的下一个词）：

- `y_out = [Machine, learning, changes, the, world, <eos>]`

即：第 t 个位置输入 $y_{<t}$，预测 y_t

解码器前向：Masked Self-Attn + Cross-Attn

对每个目标位置 t（并行计算）：

- Masked Self-Attention（自注意力 + 因果掩码）
  - 输入：Y_{in} 的 embedding（加位置编码）
  - 用 mask 保证位置 t **只能看见** $0 \sim t-1$ 的 token（不能看未来）

作用：学习“目标语言内部”的上下文依赖（类似语言模型）。

- Cross-Attention（交叉注意力）
  - Query 来自解码器当前位置表示
  - Key/Value 来自编码器输出 H_{enc}

作用：学习源端到目标端的对齐与信息提取（翻译对应关系）。

训练时目标序列 embedding 都存在（右移构造训练输入），可以并行；推理时不能并行，要一个一个生成 token

归一化

- BatchNorm（BN）：Batch 维度上做跨样本某个维度的归一
- LayerNorm（LN）：样本内部归一

时序模型更常用 LN：

- 序列长度可变、padding 多，BN 的 batch 间抖动大、统计不稳定
- 遇到分布差异大的新样本，BN 的历史统计可能失效；LN 每个样本自统计，不依赖历史
- LN 梯度更稳

自然语言处理大模型

GPT-1

解码器 only

训练阶段：预训练 + 微调

- 预训练：在大规模无标记语料库进行无监督学习（给定前 N-1 词、预测第 N 个词）
- 微调：对每个下游任务利用标记数据微调（面向的不只是垂直领域，面向各种自然语言处理任务）
  - 具体任务时，把任务输入拼成文本序列，再从模型某个位置抽取表示+线性 head，输出标签或分数

BERT

编码器 only

预训练方式是完形填空、判断两句话是否有上下文关系

从 GPT-1 到 GPT-4

- GPT2
  - 增加数据集、参数、词汇表
  - 利用无监督预训练做有监督任务，将任何有监督任务都形式化成根据输入估计输出序列
  - 不再针对不同任务微调
- GPT3
  - 更大参数量、数据集（scaling law 发力）
  - few-shot 能力
- CodeX
  - GitHub 数据微调 GPT3
- InstructGPT
  - 来自人类反馈的强化学习，来微调预训练大模型
- ChatGPT
  - 主动记忆对话内容
  - 强化学习（训练流程与 InstructGPT 相同），标注数据和原有数据改为对话形式
- GPT4
  - 多模态能力

Gemini

不同模态的输入映射到同一序列空间，共同进行自回归训练，随后采用不同的Decoder输出多模态结果（**模态统一**）

相比之下，GPT 的每个模态都有自己的编解码器，不同模态用交叉注意力融合

基于大模型的智能体

LLM 充当大脑，分解复杂任务，分步自主决策

还包括：规划（子任务分解、反思）、记忆、工具调用

4. 编程框架使用

实践内容，略

5. 编程框架原理

3. 编程框架原理

深度学习算法理论复杂，代码工程量大；有必要将基本操作里大量的共性运算（比如卷积、池化等）封装，提高编程效率。同时面对封装操作，硬件程序员可以针对优化

设计原则

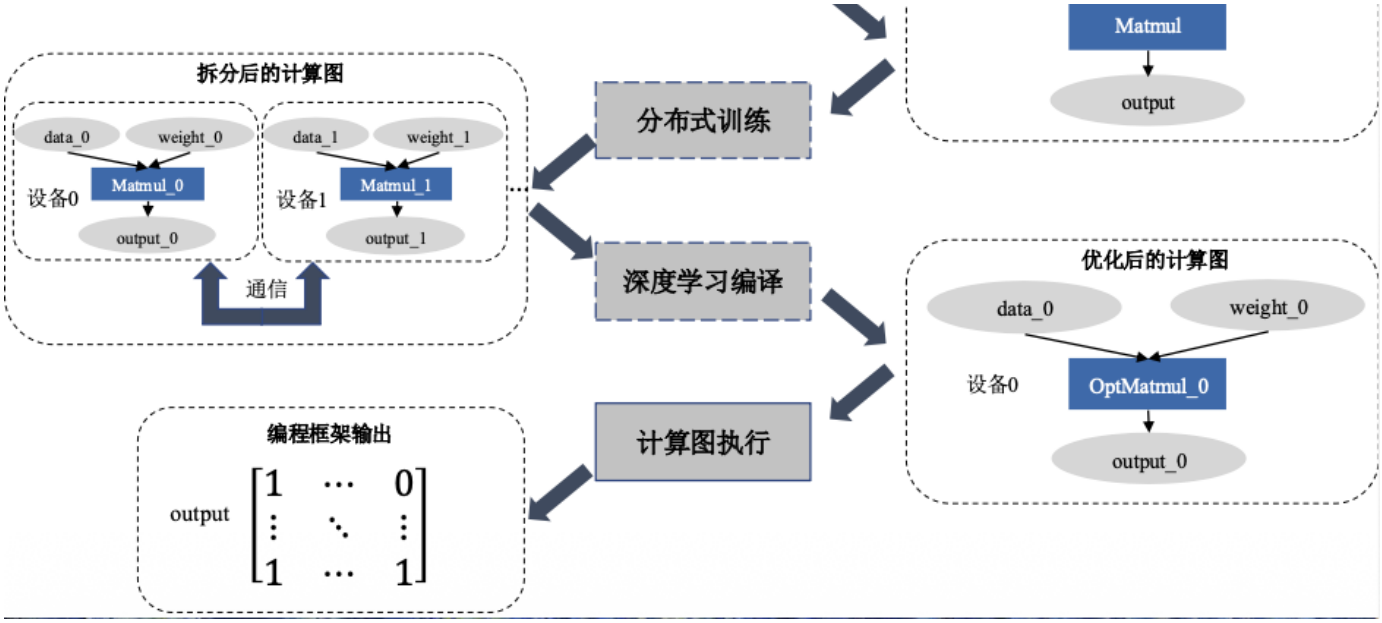
- 简洁性：提供抽象机制，用户仅需关心算法本身和部署策略
- 易用性：（1）熟悉的开发范式，比如 PyTorch 使用且忠于 python；（2）直观且用户友好的接口：如 PyTorch 提供了命令式动态图方法，易于调试
- 高效性：静态图、深度学习编译、分布式训练...

整体架构

四大模块：

1. **计算图构建**：输入用户程序，转换为编程框架内部原始计算图，入口模块
2. 分布式训练：将训练推理扩展到多台设备
3. 深度学习编译：对计算图进行图层级和算子级的编译优化
4. **计算图执行**：将（优化后）计算图张量和操作映射到指定设备执行，给出输出

> 1 和 4 已经构成基本编程框架



计算图构建

计算图是 **有向图**，边代表张量的流动方向

正向传播构建图

按构建形式可分为动态图和静态图，由 **张量** 和 **操作** 组成

动态图

在函数运行过程中逐步构建（on-the-fly）

- 命令式编程方式，没有显式建图过程
- 随着程序语句被解释器按步解释执行时隐式建图（图是副产物？）
- **立即（eager）模式**：每次调用语句就立刻执行计算

优点：易于调试；缺点：性能优化空间有限（一些基于图的优化方法无法应用）

PyTorch 中的动态图实现：**每次执行都会重新构建**

- 前向是建立了若干数据结构，形成逻辑上的计算图
- 遇到条件判断、循环等控制流语句，每次只会构建

静态图

在开始计算前，将整个网络结构先建立完成计算图。框架接收整个计算图而不是单一语句

优点：性能好；缺点：不易于调试（两者区别就类似于解释型语言和编译型语言的区别）

- TensorFlow 1.x 中的静态图：使用基本控制流算子组合来实现复杂控制流场景
- PyTorch 2.0 使用图捕获技术将用户动态图转化为静态图

反向传播求导数

对计算图做链式法则求导，把梯度沿依赖关系传回需要求导的叶子节点（参数）

求导方式与比较

常见求导方式：

- **手动求导**：手动求解梯度公式，代入数值求解
  - 对于大规模模型，手动计算并写代码很困难；模型变化后需要大量修改
- **数值求导**：用微小扰动近似导数，如有限差分
  - 前向差分（用 $f(x+\varepsilon)$ 和 $f(x)$ 近似）、中心差分（用 $f(x+\varepsilon)$ 和 $f(x-\varepsilon)$ 近似）
  - 实现简单，无需推导公式
  - 计算量巨大，速度慢；引入舍入误差和截断误差
- **符号求导**：使用求导规则对表达式进行自动操作，比如

$$\frac{d}{dx}\big(f(x)g(x)\big) = \left(\frac{d}{dx}f(x)\right)g(x) + f(x)\left(\frac{d}{dx}g(x)\right) \tag{32}$$

  - 缺点：多层函数后，表达式长度爆炸
- **自动求导（深度学习框架使用的）**：介于数值求导和符号求导。正向传播建图的时候计算出中间节点、记录计算图节点依赖关系；自动生成反向传播过程并计算梯度
  - 如果前向时一个数据连接多个输出数据，自动求导时需要累加对于该数据的导数

方法	对图的遍历次数	精度	备注
手动求解法	NA	高	实现复杂
数值求导法	$n_i + 1$	低	计算量大，速度慢
符号求导法	NA	高	表达式膨胀
自动求导法	$n_o + 1$	高	对输入维度较大的情况优势明显

其中：

- n_i：要求导的神经网络层的输入变量数，包括 w, x, b
- n_o：神经网络层的输出个数

PyTorch 的自动求导

PyTorch 核心特性即动态计算图

- 模型在进行正向传播时，PyTorch 的 Autograd 模块会记录每一次张量运算

及其依赖关系，为每个操作生成相应的梯度函数（grad_fn），而不是立
即创建反向算子

    - 每当对 `requires_grad=True` 的张量做加法、乘法、均值等，PyTorch 会自动创建 `AddBackward0`、`MulBackward0` 等对象，赋值到 Tensor 的 `grad_fn` 属性

- **这些梯度函数**共同构成有向计算图，用于描述张量间的计算依赖：下游节点的 `grad_fn` 会指向上游的 `grad_fn`（通过 `next_functions`）
- 沿着上述关系，反向逐步求出梯度

计算图执行

将计算图 **张量** 和 **操作**（算子）映射到具体 **设备** 上执行

设备管理

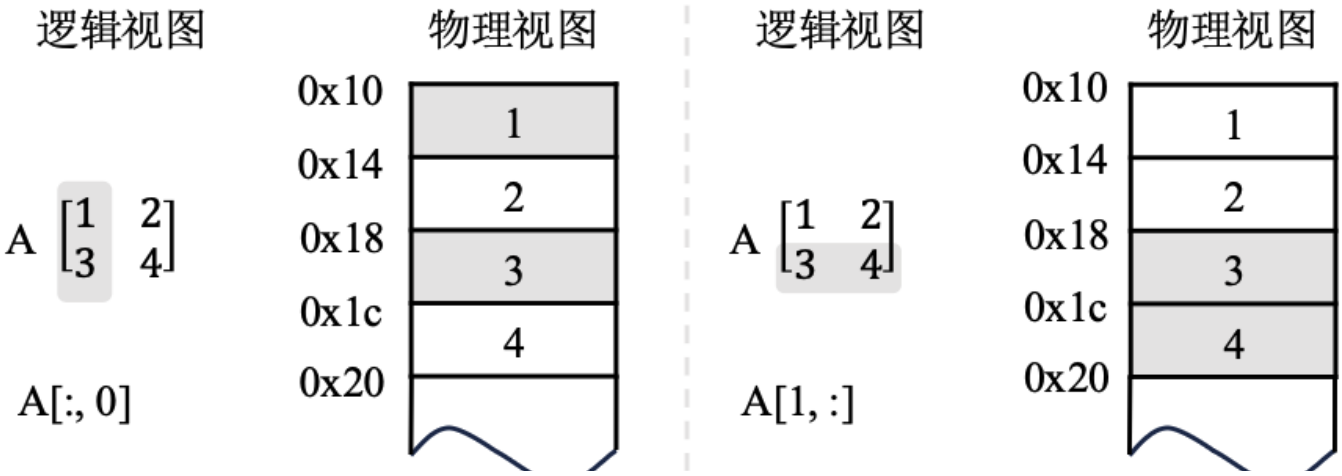
对通用处理器（如 CPU）和领域专用处理器（GPU、DLP...）管理支持（三个模块）：

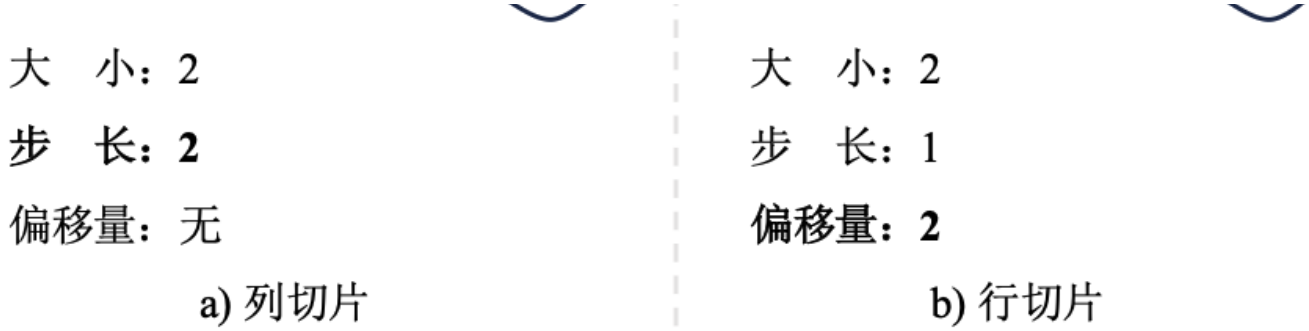
- 设备操作：初始化环境、获取句柄、释放等
- 执行流管理（抽象计算任务的管理）：异构模型下完成设备上任务执行的下发和同步
- 事件管理：设备任务运行状态，事件 CRUD 操作

张量实现

张量在框架中同时具有 **逻辑视图（用户感知）** 和 **物理视图（框架关注）**：两者分离是深度学习框架高效实现切片、转置、view 等操作的核心。

- 逻辑视图：包含张量的维度、步长、数据类型、偏移量（索引）设备类型等，框架使用者可以直接控制
- 物理视图：设备上的物理地址空间大小、指针、数据类型等，对框架使用者不可见
- **一个逻辑视图可以对应多个逻辑视图**：切片、转置等操作只会创建新的逻辑视图，不会产生新的物理内存，相当于通过 stride + offset 在同一块物理内存中“重新解释”数据
- PyTorch 里是 Tensor 和 Storage 类的分层结构保证分离的





张量的内存分配

Storage 类调用结构体 Allocator 进行张量数据空间分配：

- CPU 即时分配，因其内存分配相对便宜、可由 OS 高效管理
- GPU 内存池分配：预先分配固定大小内存池，有自我维护功能（内存块拆分合并），优点是节约设备内存、减少碎片化

算子执行

1. 计算图 -> 执行序列：拓扑排序
2. 针对每个算子进行算子实现：包含前端定义（配置算子定义、生成接口如 Python API）、后端实现（如 Cuda 侧）和前后端绑定
    - 前后端绑定：同一个算子可能有多个后端实现，PyTorch 需要用一张分派表来分派

> PyTorch 使用 native_function 模式管理整个算子实现模块

深度学习编译器

接受计算图，在指定硬件平台生成高性能代码

产生原因：

- 框架维护成本高：新硬件和新算子如果需要程序员手动开发，数量平方级增长 -> 深度学习编译器对不同平台自动生成代码，**减少开发量**
    - 算子的长尾效应
- 编译器可以从 **图层级** 和 **算子层级** 进行性能优化

图层级优化

- 子图替换：类似用交换律、结合律简化计算操作
- 常量折叠：复用常量计算结果
- 公共子表达式消除
- 代数化简：比如存在 *0 时直接把表达式置0
- 布局优化：比如 Tensor Core 对 NHWC 性能优于 NCHW

- 布局优化：比如 Tensor Core 对 NHWC 性能优于 NCHW
- 算子融合：避免多次访存开销
    - 纵向：小算子合成大算子
    - 横向：小矩阵乘合成大矩阵乘

算子层级优化

算子-计算-调度

- 算子：定义功能操作（如矩阵乘法）
- 计算：数学定义（也就是公式）
- 调度（优化目标）：调度策略、包含循环顺序、分块、缓存与并行等实现，直接影响性能
    - 绝大多数算子都是多层循环，适合算子层级优化

通过搜索的方式确定合适的调度配置，包含性能测量和评估的代价模型，有基于手工模版（需要专家设计）、基于序列构建（自动化，耗时且低效）、层次化搜索（从循环结构粗粒度到具体调度参数的细粒度，搜索空间大、需要专家设计）方法

常见深度学习编译器

TVM、XLA、TorchDynamo（PyTorch2.0）

TVM 的核心思想：计算与调度分离、图层级和算子层级各抽象中间表示

分布式训练

原因：

- 模型参数量和数据量提升，单个设备资源有限
- 提升单个设备性能成本远高于使用多个设备

基础：

- 分布式架构：组织和管理分布式训练任务的方式
- 分布式同步策略：不同节点之间协调与同步

常见架构：

- 参数服务器：中心化管理模型参数，李沐 2014
    - 中心节点（可以多个）存储参数和梯度更新，计算节点完成中心节点下发的计算任务，与中心节点通信更新和检索共享参数
    - 优点：灵活（中心节点数量）、高效参数共享
    - 缺点：单点故障（单个中心节点故障影响全部系统）、数据一致性问题（同时读写）、网

络开销（中心化瓶颈）

- 集合通信：去中心化，每个计算节点都有全局最新参数，参数同步是多次设备间点对点，对算力和通信要求高

同步策略：同步和异步（优缺点略）

并行策略：

- 数据并行：不同数据相同模型，解决单节点算力不足
- 模型并行：解决单节点内存不足
  - 算子内并行：通信量大，如用 NVLink；对单个算子按行/列切分
  - 算子间并行：通信量小，就是把前向过程分段，使用流水线缓解空泡
- 混合并行（实际实现如微软 DeepSpeed）

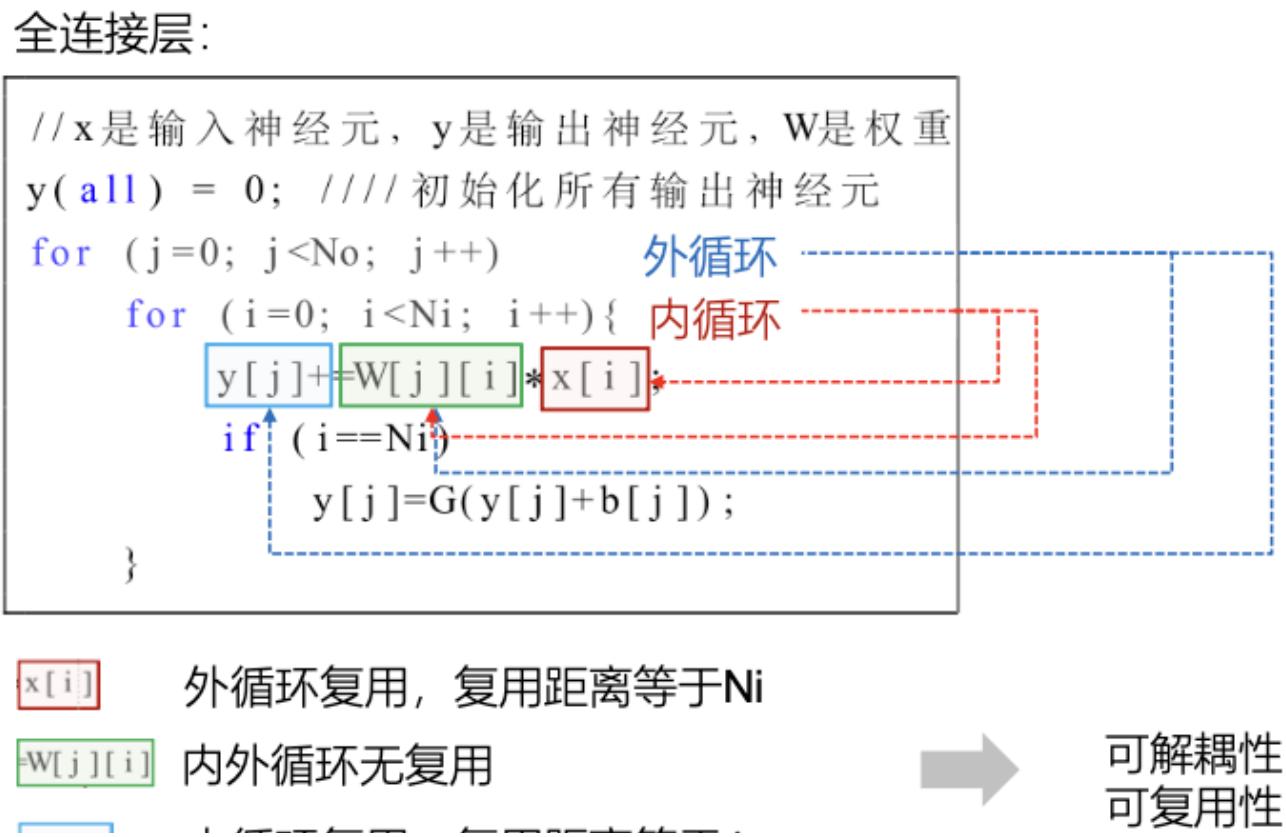
6. 深度学习处理器原理

概述

目标算法分析

分析什么：

- 计算：是否存在固定重复的计算模式
- 访存：数据的局部性、数据和计算的关系（带宽需求、是否需要循环分块）



y[j]+ 内循环复用，复用距离等于1

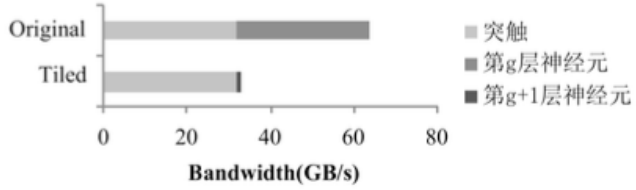
```
//x是输入神经元，y是输出神经元，W是权重
y(all) = 0; ////初始化所有输出神经元
for (j=0; j<No; j++)
    for (i=0; i<Ni; i++){
        y[j]+=W[j][i]*x[i];
        if (i==Ni)
            y[j]=G(y[j]+b[j]);
    }
```

关于X：No×Ni 次访存

```
//T是循环分块大小
y(all) = 0; //初始化所有输出神经元
for (ii=0; ii<Ni; ii+=Ti)
    for (j=0; j<No; j++)
        for (i=ii; i<ii+Ti; i++){
            y[j]+=W[j][i]*x[i];
            if (i==Ni)
                y[j]=G(y[j]+b[j]);
        }
```

关于X：Ni 次访存

假设Ni=16384
循环分块能减少46.7%的访存带宽需求



不同网络的访存特性：

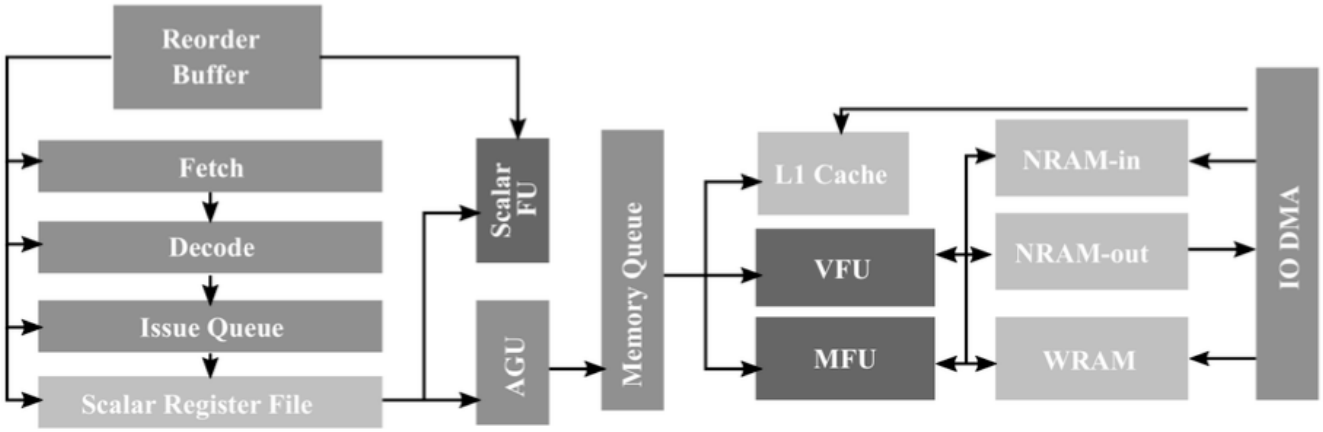
层类型	可重用	不可重用
卷积层	输入神经元、输出神经元、突触权重	无
池化层	当池化窗口 **大于步长** 时，部分输入神经元可重用	当池化窗口 **小于等于步长** 时，输入神经元、输出神经元均不可重用
全连接层	输入神经元、输出神经元	突触权重

为什么可复用：神经元被重复使用

DLP 结构

DLP 更关心向量和矩阵，专门指令集高效处理矩阵计算；CPU 通用的指令集最后都变为单个元素计算

深度学习更关心数据并行（向量化运算）、不太关心指令级并行

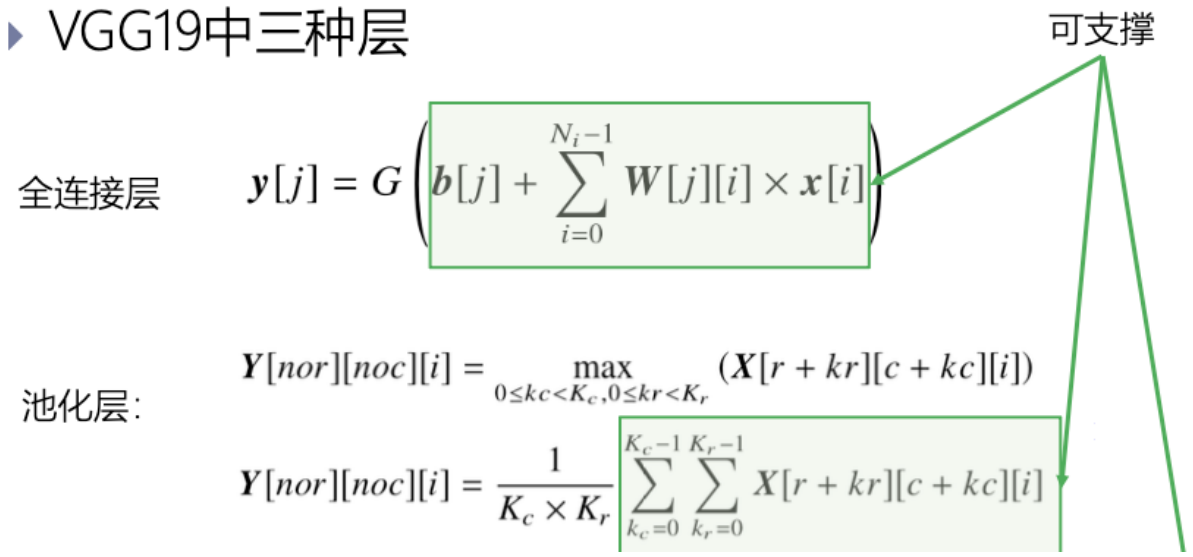


指令集

面向 vector 和 matrix 设计逻辑和 IO

MAC（Multiply-Accumulator）

相对于标量的 ALU，其可以做向量内积、相加；N 个向量 MAC 单元堆叠可以支撑 DLP 指令集



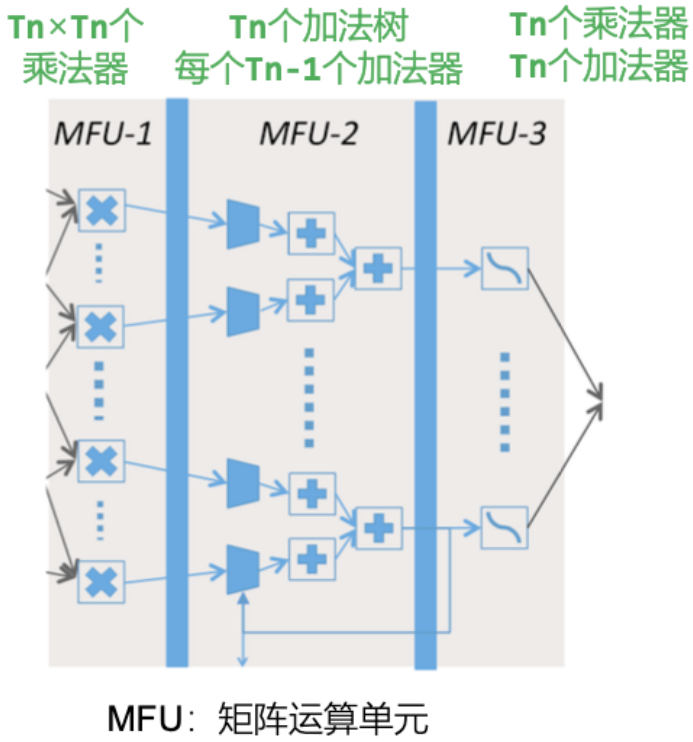
卷积层：

$$Y[nor][noc][j] = G\left(b[j] + \sum_{i=0}^{N_{if}-1}\sum_{k_c=0}^{K_c-1}\sum_{k_r=0}^{K_r-1} W[k_r][k_c][j][i] \times X[r+k_r][c+k_c][i]\right)$$

其他运算部件

- 激活函数处理单元：非线性函数单元
- 常用做法：分段查找表 + 线性插值
- 支持多激活函数、配置查找表段数可以配置京都

MFU（🌟）



MFU：矩阵运算单元

$$y[j] = G\left(b[j] + \sum_{i=0}^{N_i-1} W[j][i] \times x[i]\right)$$

- MFU-1
  - 4*4个乘法器
- MFU-2
  - 4个加法树
  - 每个（2+1）个加法器
- MFU-3
  - 4个激活函数实现

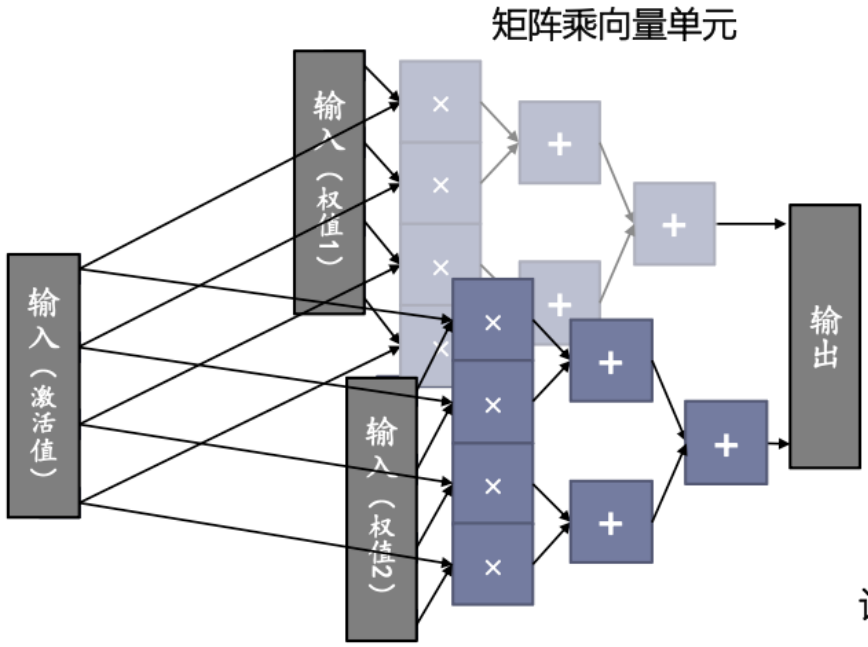
- 全连接层：需要乘法器、加法树（Tn-1）、激活函数
- 卷积层：本质可以视为全连接层
- 池化层：
  - 最大池化：不需要 MFU-1 MFU-3，只需要 MFU-2 比较器
  - 平均池化：需要 MFU-1 乘法（除以池化窗口大小）、不需要 MFU-2、MFU-3

矩阵运算单元的一种实现：由向量内积单元堆叠而成

- 计算的时候激活值按行输入内积单元、权值按列；计算-IO 比例和单向量时计算相同

- 优化：近端权值可以存储在内积单元附近电路（减少缓存）

- 多个内积单元组成矩阵乘向量单元

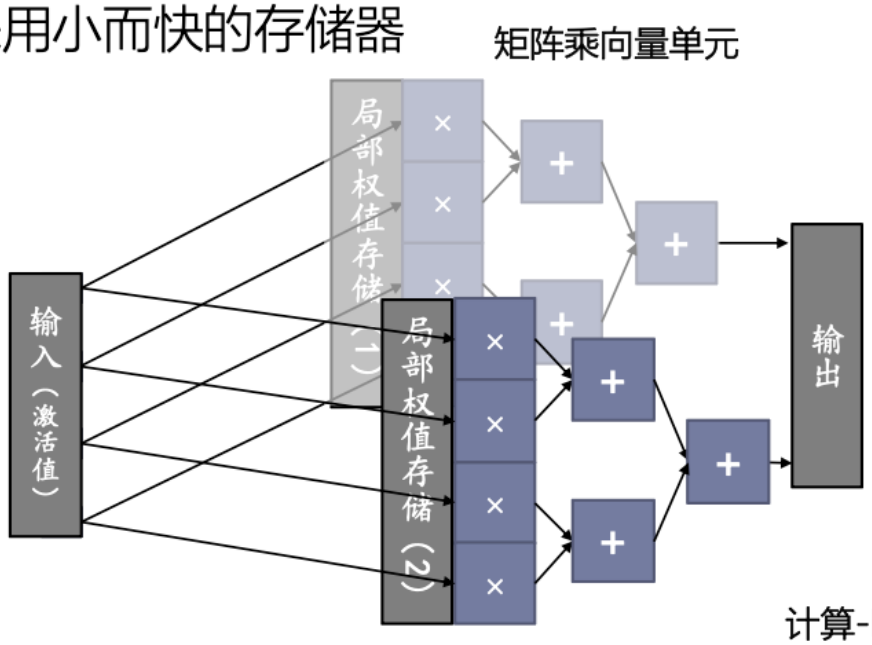


7:9

计算-I/O比例 = 1:1.3

矩阵运算单元

- 近端数据（权值）存储在内积单元附近的电路中
- 采用小而快的存储器



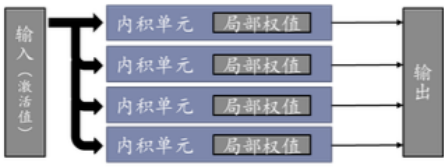
计算-I/O比例 = 7:5 = 1:0.7

引入到矩阵乘向量单元，增加内积单元数量、广播共享权值、扩大规模.....

- 注意 MFU（由 VFU 构成）的【计算-IO比例】计算时，不要忘记输出！

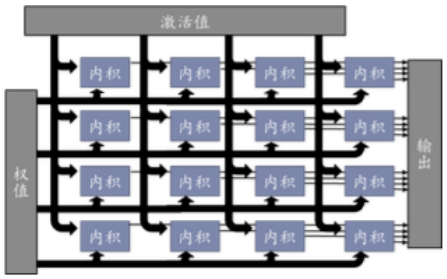
矩阵运算单元

- 矩阵乘向量单元
  - 计算密度已经较好



计算-I/O比例 = 1:0.3

- 矩阵乘矩阵单元
  - 优势：规模大时，理论上较好
    - （第六章）
  - 困难：连线复杂，距离远、扇出多
  - 规模不大时，未取得实际优势



计算-I/O比例 = 1:0.4

访存部件

分离的 NRAM-in（输入）、NRAM-out（输出）、WRAM（权重）

- 避免访存流互相干扰
- 片上缓存，提高数据复用率

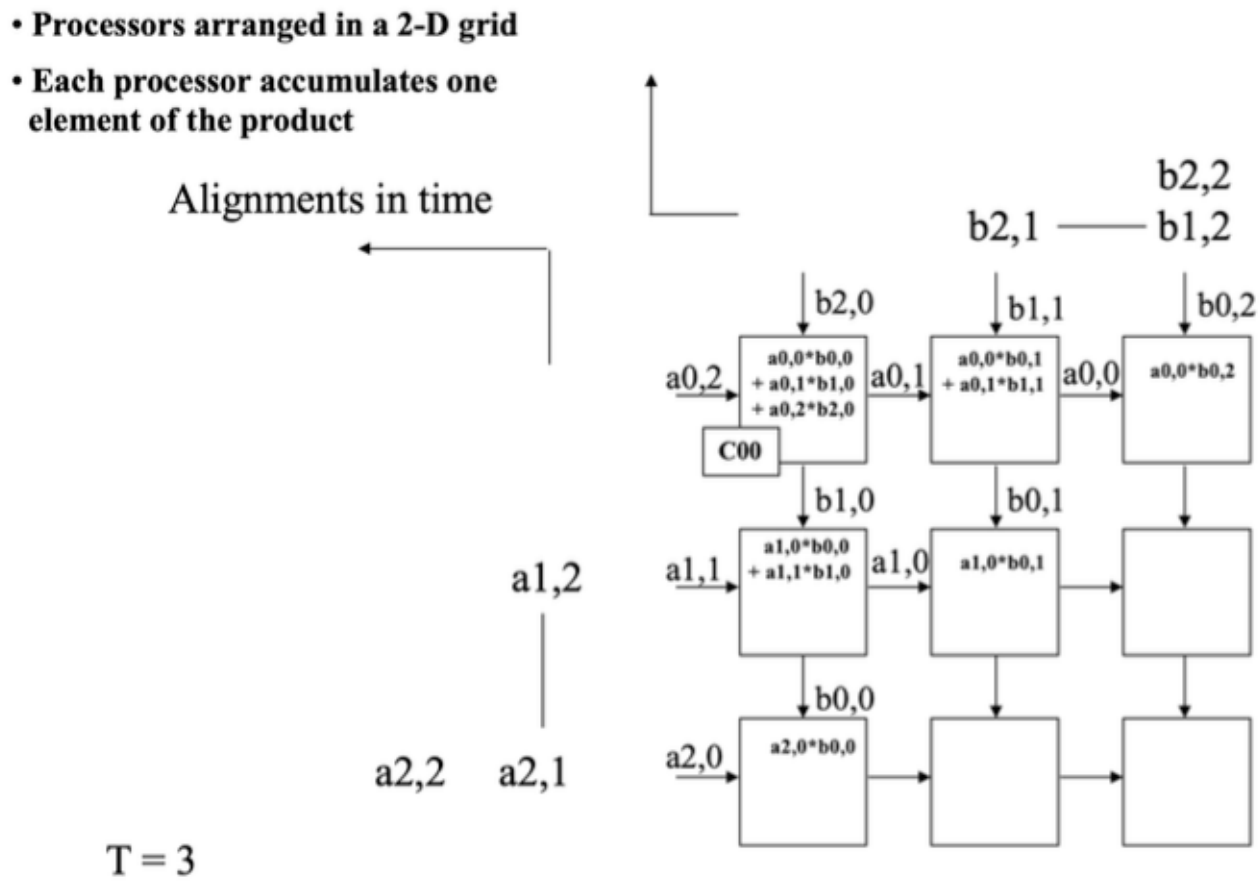
算法映射

基本思想：硬件的分时复用

优化设计：脉冲阵列（systolic array）

大规模矩阵乘单元，谷歌 TPUv1 的核心，使用的是 **标量运算单元**

数据从左上角开始流入，每个流入元素继续向右、向下流动；部分和在阵列内部累加并继续传递



	基于向量 MAC 计算单元	基于标量 MAC 计算单元
大小	16 个 16 维向量 MAC	256 个 MAC，16x16 阵列
乘法器个数	256	256
加法器个数	240（16x15）	256
每拍需求外部操作数	512	32
操作粒度	向量、矩阵	向量、矩阵
卷积层映射	输入神经元复用、输出神经元复用	输入神经元复用、输出神经元复用、权重复用
优点	高效支持矩阵向量映射，灵活性高	专用数据流高效支持卷积，带宽需求降低
缺点	依赖外部数据排布，带宽需求高	灵活性差，支持其他算子其他特性困难

性能评价

影响性能的关键因素：

$$T = \sum_i \frac{N_i \times C_i}{f_c} \tag{33}$$

其中：

- N_i：第 i 类操作的次数
- C_i：完成该类操作所需的周期数
- f_c：处理器主频

优化方向：

- **减少 C_i**（优化算子、流水线、并行）
- **减少访存开销**（数据复用、就近存储）
- **多级并行**（PE 并行、流水并行、任务并行）

TOPS（Tera Operations Per Second）

- 注意不是 TFLOPS，这里统计的是定点/整数运算次数

$$\text{TOPS} = \frac{f_c \times (N_{mul} + N_{add})}{1000} \tag{34}$$

其中：

- f_c：处理器主频（GHz）
- N_{mul}：每个时钟周期内的乘法次数
- N_{add}：每个时钟周期内的加法次数

访存带宽

$$BW = f_m \times b \times \eta \tag{35}$$

其中：

- f_m：存储器主频
- b：每次传输的数据位宽
- η：访存效率

基准测试

基准测试程序：MLPerf，业界标准，用 CNN、RNN、强化学习一系列模型的训练和推理任务评估 DLP 性能，横向对比不同 DLP / GPU / FPGA 的真实性能

其他加速器

类别	目标	速度	能效	灵活性
DLP	深度学习专用	高	高	深度学习领域通用
FPGA	通用的可编程电路	低	中	通用
GPU	SIMD 架构矩阵加速	中	低	矩阵类应用通用

7. 深度学习处理器架构

主要问题

1. 提高处理器能效比（性能/功耗）
2. 提高处理器的可编程性

单核深度学习处理器（DLP-S）

面向手机等智能终端应用，需要 **低延迟**、**高能效比**

- 需要基于 tensor 语义设计运算和存储模块
- DLP 的局限性：不支持指令级并行和低位宽运算、不支持稀疏神经网络模型、访存延迟过大

基于上述背景从 DLP 向 DLP-S 演进

总体架构

控制模块 + 运算模块 + 存储模块

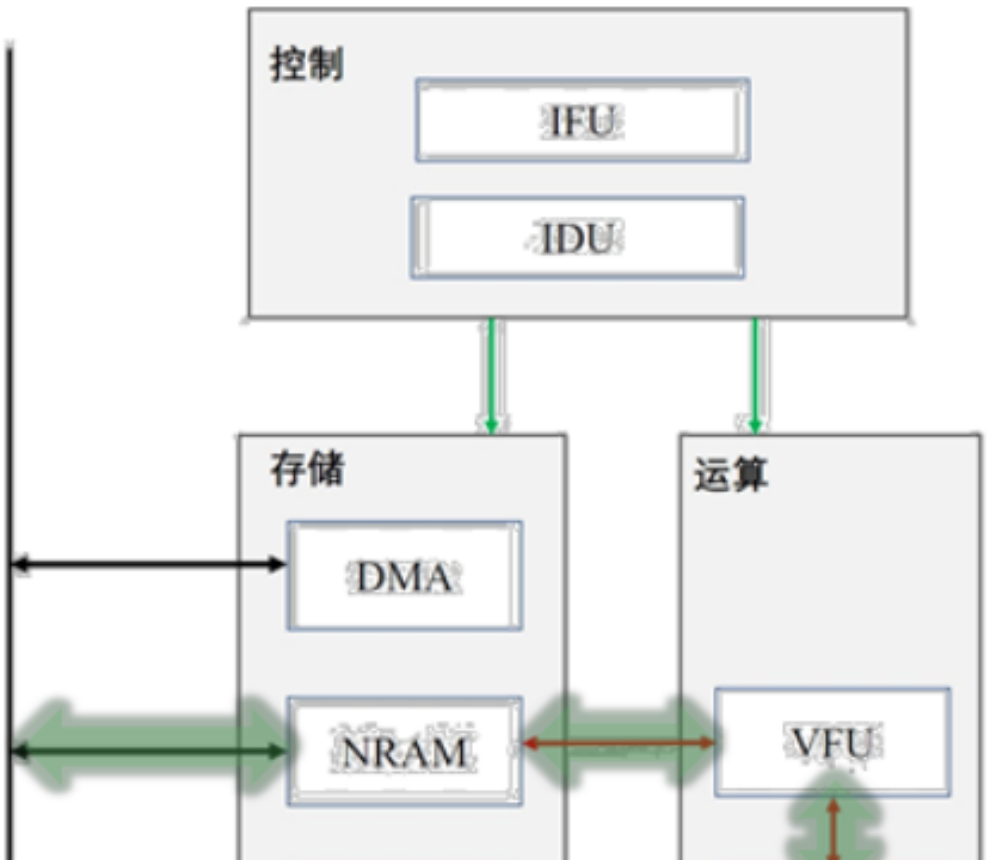
- 控制模块：提供 tensor 粒度指令
  - IFU：取指单元
  - IDU：指令译码单元
- 运算模块：基于 tensor 语义设计的运算模块
  - VFU：向量运算单元：向量计算 / 激活 / 预处理 / 后处理
  - MFU：矩阵运算单元：矩阵乘法（核心）
- 存储模块：基于 tensor 语义设计的存储模块
  - NRAM：神经元存储单元（输入、中间特征、输出）
  - WRAM：权重存储单元
  - DMA：~~开挂~~ 直接内存存取单元

从 DLP 到 DLP-S

- 控制层增强：多发射队列、支持指令级并行、寄存器重命名

- 增加更多运算器支持的操作、支持低位宽运算（推理提速）
- 稀疏数据稠密话访存、降低访存开销
- TLB：降低虚实地址转换延迟
- LLC：降低 DRAM 访问延迟

执行流程



1. IFU 通过 DMA 从 DRAM 取指令，IDU 译码发给 DMA、VFU、MFU
2. DMA 从 DRAM 读取神经元 tensor 到 NRAM；读取权重 tensor 到 WRAM
3. VFU 读取 NRAM、预处理（padding / reshape），发给 MFU
4. MFU 读取 WRAM，与 VFU 输出执行矩阵运算，发回 VFU
5. VFU 对输出 tensor 后处理（激活 / 池化）
6. VFU 将结果写回 NRAM
7. DMA 将输出 tensor 从 NRAM 写会 DRAM

数据流：

- 神经元 tensor：DRAM → NRAM → VFU → MFU → VFU → NRAM → DRAM
- 权重 tensor：DRAM → WRAM → MFU （**只参与矩阵乘，不参与 VFU）

控制模块

体系结构内容，略

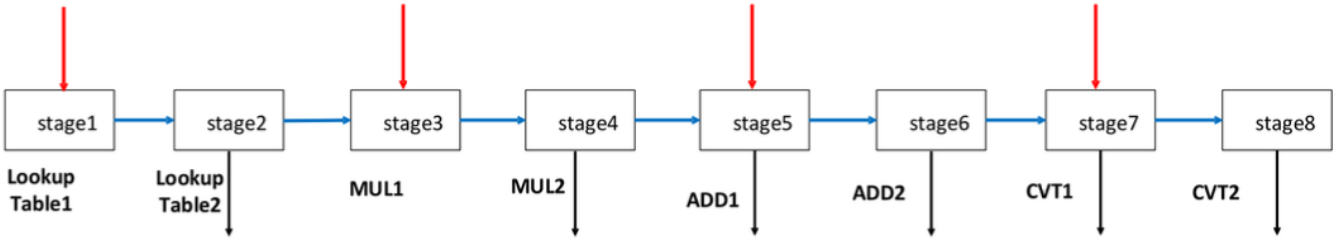
- 三个指令队列（控制、运算、访存）乱序发射，队列内顺序发射
- 同类型指令有依赖，队列内顺序发射；不同类指令有依赖，添加 SYNC 同步指令

VFU：向量运算单元

- 负责神经元 前处理 + 后处理
- 包括向量流水单元（承载向量运算功能）和转置单元（数据重新摆放）

向量流水单元

- 支持 INT8、INT16、INT32、FP16、FP32
- 相较于之前的向量运算（乘法加法）、新增运算：查找、边缘扩充、数据格式转换



完成平均池化：

- 输入数据是 INT 型：本质是 kx * ky 个向量累加
    - INT 型（加法延迟单个 cycle）：只用 stage5
        - 逐个累加（结果进入 stage5 输出寄存器再和下一个向量一起输入 stage5）
        - 总计需要 kx * ky 个 cycle
    - FLOAT 型（加法延迟 2 个 cycle）：stage5+6
        - 总计需要 2 * kx * ky 个 cycle

完成最大池化：

- 需要额外的比较器，逐个比较大小然后写回寄存器
    - INT、FLOAT 下都需要 kx * ky 个 cycle

超车问题：后执行的运算指令比先执行的指令执行的更快

- 解决规则：
    - 指令从第 n 个 stage 进入，需要等待 stage 1-n **输入**寄存器清空
    - 指令从第 m 个 stage 输出，需要等待 stage (m+1)-8 所有 **输入**寄存器清空

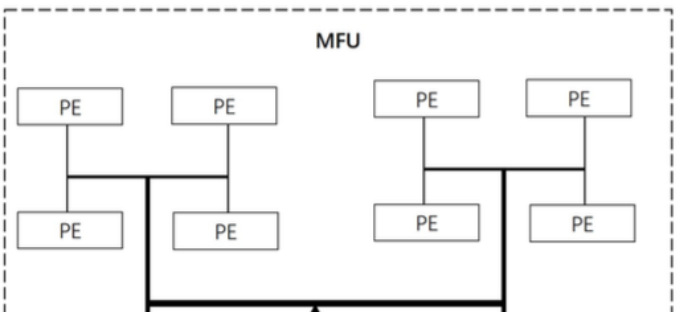
转置单元

转置、镜像、旋转灯，由读写控制逻辑对数据缓存进行多模式读写实现

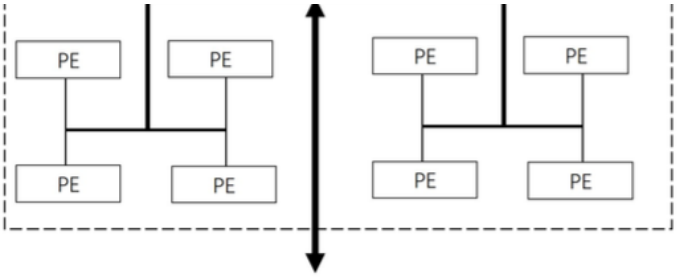
MFU：矩阵运算单元（分布式设计）

MFU 计算量占 90%，是整个芯片的功耗瓶颈

- MFU：矩阵运算单元（分布式设计）
    - H-tree互联（广播神经元）
    - 低位宽定点运算器
    - 三种模式：
        - int16 ×int16



- int8 ×int8
- int8 ×int4



> 从图中可看到，带宽逐渐减半。

- PE 是 Processing Element（最小可复用计算单元），包含定点乘法器、累加器和寄存器
- H-tree 是分层对称的广播结构，用于广播神经元 / reduction。优点在于延迟对称（源到各 PE 距离一致）
    - 不使用总线，因为带宽瓶颈明显；不使用 mesh 因为延迟不一致

三种模式：INT16 * INT16，INT8 * INT8，INT8 * INT4（注意只有权重压缩到 4，激活的位宽不能太低）

存储结构

NRAM 存神经元，WRAM 存权重，DRAM 在片外，DMA 进行数据搬运

虚拟存储：片内片外同一地址。片内无需虚实地址转换，片内外需要转换

优化：

- 降低访问延迟：TLB（缓存常用页表）、LLC（缓存经常访问的 DRAM）
- 降低访存量：稀疏化存储、数据压缩

多核深度学习处理器（DLP-M）

场景从智能终端（推理）转向云端智能（训练+推理）

继续提升 DLP-S？处理器面积增大，传输延迟大，主频降低。所以和 CPU 的发展路程一致，引入了多核 DLP-M

主要特征：

- 多层片上存储结构，逐级缓存数据
- 定义一套完整的多核协同通信与同步机制

分层结构：

```
Chip (DLP-M)
 └── Cluster (DLP-C * 4)
      └── Core (DLP-S * 4)
```

Cluster

Cluster

Cluster 层设计了 MEMCORE，包含 DLP-C 内各个 DLP-S 的共享存储 SMEM；有 GDMA 与片外 DRAM 通信；CDMA 与其他 DLP-C 通信

- 没有 SMEM，相同权重重复 4 次从 DRAM 读，片外访存数据量 4 倍
- 使用 SMEM 不使用广播总线，相同权重重复 4 次从 SMEM 读

广播总线读写请求：单播写/读、多播（一个发送多个接收）

BARRIER 指令：多核同步，解决访存冲突

互联架构

设计目标：核间延时相同（完全对等）、尽量稠密（减小单通路负载），理论上只有多核全连接拓扑满足

- 总线互联：所有核连接到总线
  - 优点：端口之间独立不易死锁
  - 缺点但随着核数增加在带宽、时钟同步和物理实现面临挑战（扩展性差、并发性差）
- 片上网络（NoC）：通信节点和互联通路构成网络，分层可扩展，逻辑上任意点对点连接
  - 优点：扩展性并发性强，功耗性能更好、物理链路复杂度低
  - 缺点：设计复杂