机器学习系统

第二章 机器学习与框架基础

张燕咏 讲席教授 yanyongz@ustc.edu.cn

张午阳 特任教授 wuyangz@ustc.edu.cn

中国科学技术大学
University of Science and Technology of China

致谢

本章内容PPT使用

中科院陈云霁老师《智能计算系统》

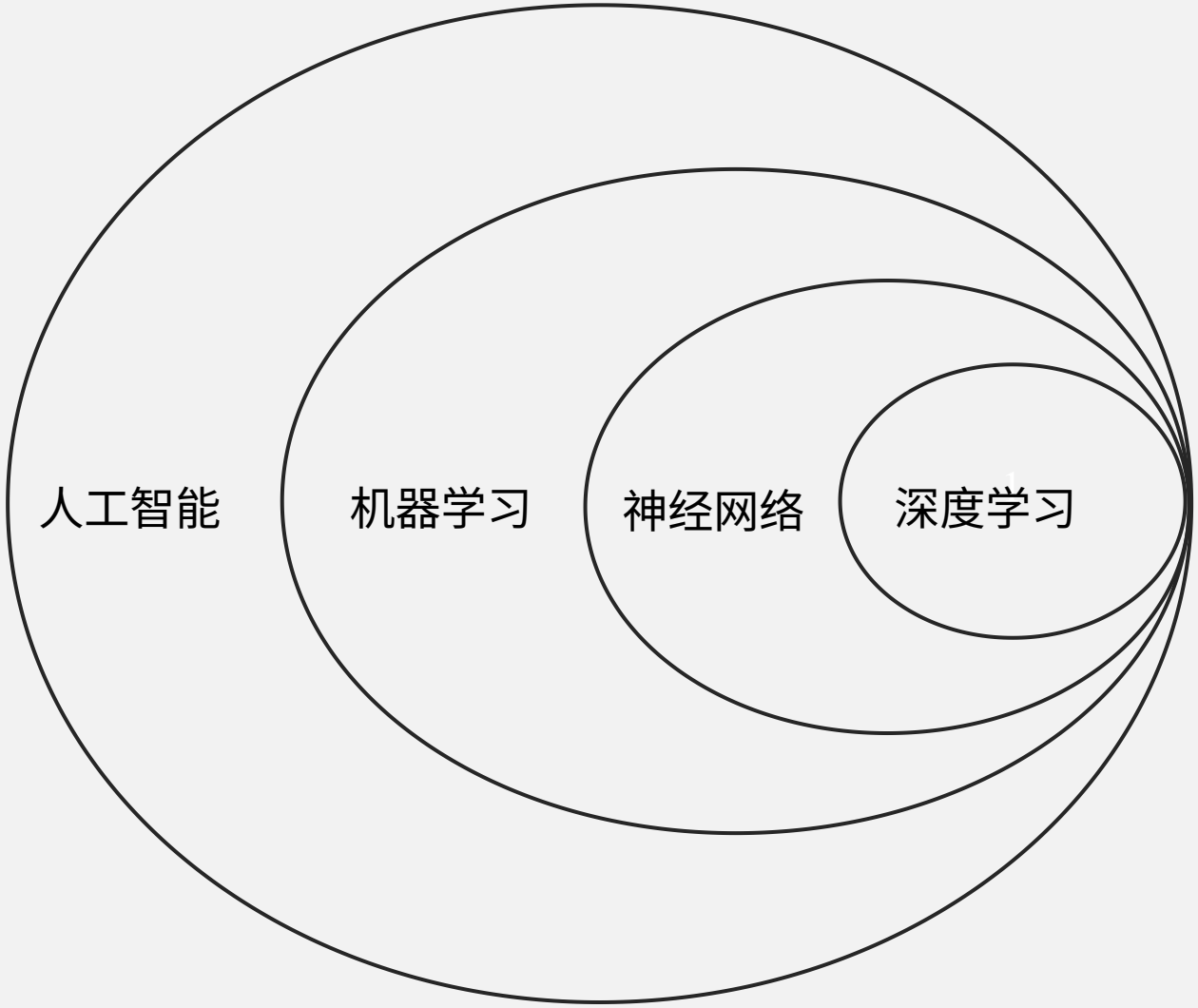
课程PPT

本节课内容

- **机器学习**
- 神经网络
- 神经网络训练方法
- 编程框架设计
- 计算图构建
- 计算图执行
- 本章小结

包含关系

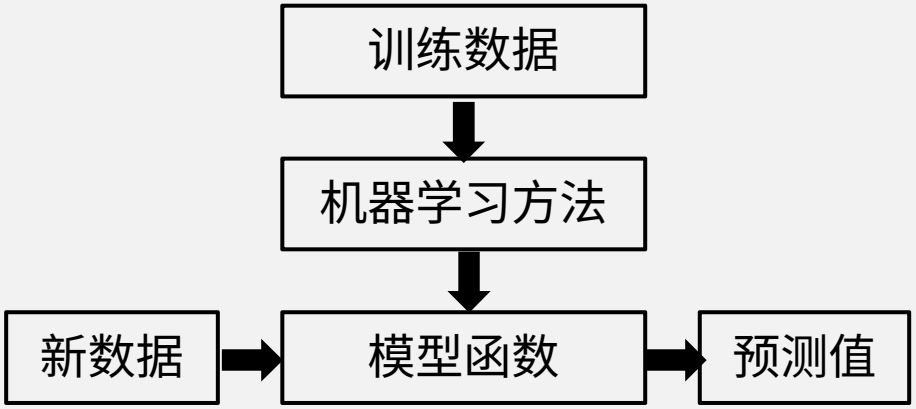
- 人工智能
- 机器学习
- 神经网络
- 深度学习



机器学习相关概念

- 机器学习是对能通过经验自动改进的计算机算法的研究（Mitchell）
- 机器学习是用数据或以往的经验，以此提升计算机程序的能力（Alpaydin）
- 机器学习是研究如何通过计算的手段、利用经验来改善系统自身性能的一门学科（周志华）

典型机器学习过程



Mitchell定义：任务T · 经验E · 性能P — 思考ChatGPT的T/E/P

机器学习系统视角：为什么理解基础很重要

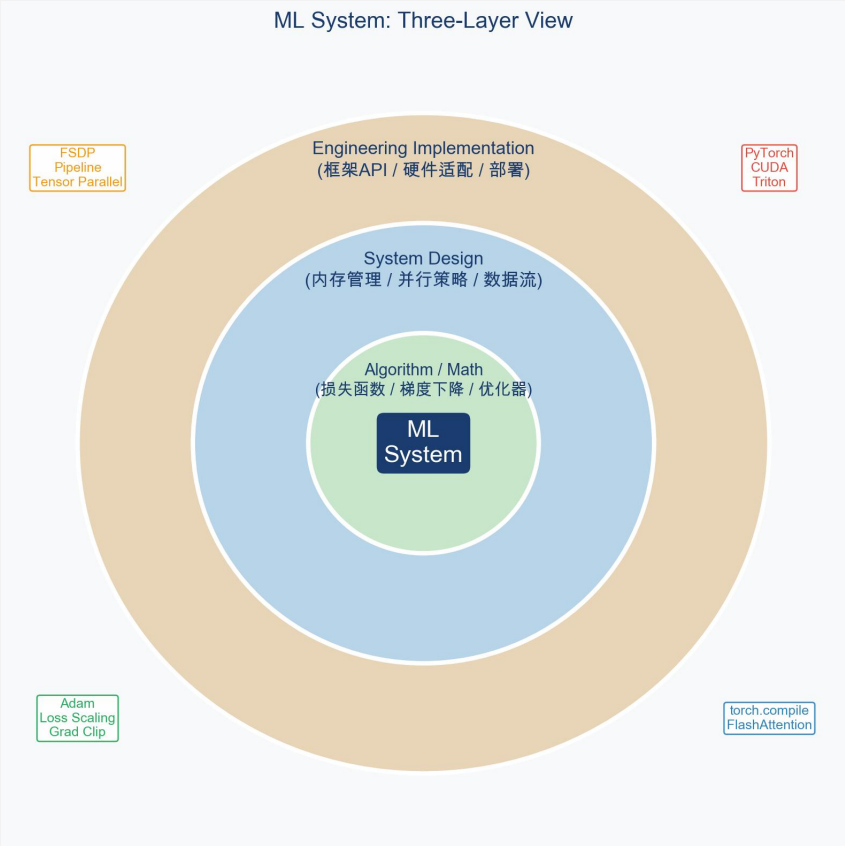
ML系统 = 算法 + 系统 + 工程

模型选择决定系统架构设计

损失函数形状影响优化器实现

数据流模式决定内存与计算策略

理解数学本质才能做好系统优化



符号说明

输入数据： x

真实值（实际值）： y

计算值（模型输出值）： $\hat{y}$

模型函数： $H(x)$

激活函数： $G(x)$

损失函数： $L(x)$

标量：斜体小写字母 *a*、*b*、*c*

向量：黑斜体小写字母 ***a***、***b***、***c***

矩阵：黑斜体大写字母***A***、***B***、***C***

x=输入特征 y=标签 θ/w=参数 h=假设函数 J=损失函数

→ 不同教材符号不同（θ vs w，J vs L）

→ 重点是直觉理解，不是数学推导

→ 读论文时先看符号定义表

符号约定：x=输入 y=标签 θ=参数 h=假设 J=损失

如何学习？如何理解？如何学会？

从最简单的线性回归模型开始，直至搭建出一个完整的神经网络架构

线性回归 = ML的Hello World，包含 模型·损失·优化 全部核心元素

线性回归

什么是回归（regression）和线性回归?

为什么从线性回归开始?

问题引入：假设房屋销售中心有这样一组关于房屋面积和房屋位置与销售价格的数据，用x_1表示房屋面积， x_2表示房屋楼层，y表示售价（万元）。

x_1	50	47	60	55	...	65
x_2	2	1	4	3	...	10
...					...	
y	50	42	80	52	...	?

设计一个回归程序进行预测

核心计算 ŷ = w·x + b — 从线性回归到GPT，基本单元都是矩阵乘法

房屋面积x_1，与售价y的关系

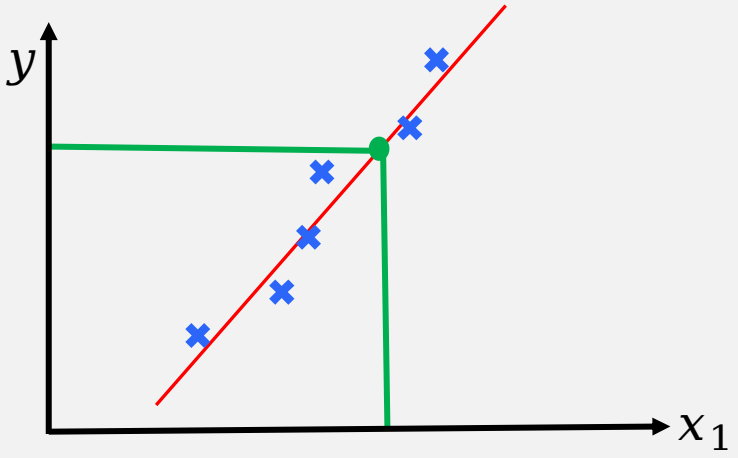
x_1	50	47	60	55	...	65
y	50	42	80	52	...	?

如果给出一个新的房屋面积，在销售记录中没有的，如何确定在该面积下的房屋售价y的值呢?

如预测$x_1 = 65$时的售价?

解决办法->寻找x_1和y的之间的关系，使用已有记录数据进行训练

MSE衡量预测偏差，J(θ) 的极小值点 = 最优参数

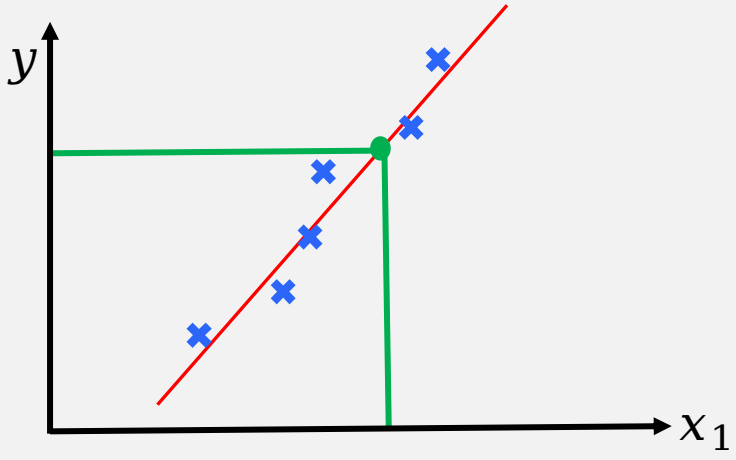


单变量线性回归模型（一元回归模型）

- 线性回归可以找到一些点的集合背后的规律：一个点集可以用一条直线来拟合，这条拟合出来的直线的参数特征，就是线性回归找到的点集背后的规律。

单变量线性模型

$$H_w(x) = w_0 + wx$$



x：Feature；$H(x)$：hypothesis

多变量线性回归模型

假设影响售价y的特征不仅仅只有房屋面积x_1，还有楼层x_2，房屋朝向x_3， x_4，…， x_n

单变量线性模型

$$H_w(x) = w_0 + wx$$

多变量线性模型

$$H_w(x) = w_0 + w_1 x_1 + w_2 x_2$$

2个特征

$$H_w(x) = \sum_{i=0}^{n} w_i x_i = \widehat{\boldsymbol{w}}^T \boldsymbol{x},$$

n个特征

$$\widehat{\boldsymbol{w}} = [w_0; w_1; \ldots w_n],$$

$$\boldsymbol{x} = [x_o; x_1; \ldots x_n],\ x_0 = 1$$

数据维度爆炸：图像$224^2 \times 3 = 150$K维，文本序列可达128K tokens

线性函数拟合得好不好?

模型预测值$\hat{y}$与真实值y之间存在误差 $\varepsilon = y - \hat{y} = y - \widehat{\boldsymbol{w}}^T \boldsymbol{x}$

ε 满足N$(0, \sigma^2)$的高斯分布： $p(\varepsilon) = \frac{1}{\sqrt{2\pi}\sigma} exp(-\frac{(\varepsilon)^2}{2\sigma^2})$

似然函数

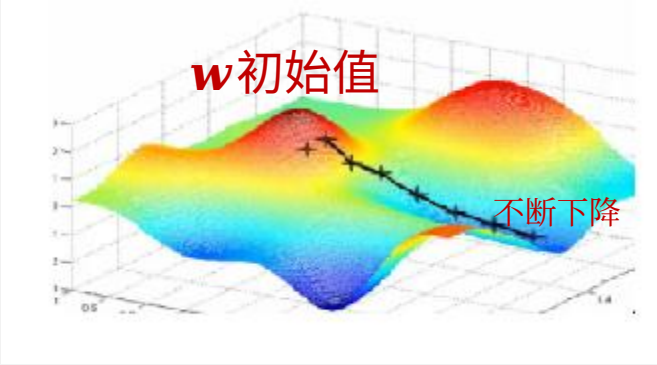
$$p(y|\boldsymbol{x}; \widehat{\boldsymbol{w}}) = \frac{1}{\sqrt{2\pi}\sigma} exp(-\frac{(y - \widehat{\boldsymbol{w}}^T \boldsymbol{x})^2}{2\sigma^2})$$

通过求最大似然函数，得到预测值与真实值之间误差尽量小的目标函数

损失函数 $L(\widehat{\boldsymbol{w}}) = \frac{1}{2}\sum_{j=1}^{m}(H_w(\boldsymbol{x}_j) - \boldsymbol{y}_j)^2 = \frac{1}{2}\sum_{j=1}^{m}(\widehat{\boldsymbol{w}}^T \boldsymbol{x}_j - \boldsymbol{y}_j)^2$

目标: 求出参数$\widehat{\boldsymbol{w}}$，使得损失函数$L(\widehat{\boldsymbol{w}})$取值最小

寻找参数$\widehat{w}$->使得$L(\widehat{w})$ 最小



迭代法（梯度下降法）寻找参数

- 初始先给定一个$\widehat{\boldsymbol{w}}$，如$\mathbf{0}$向量或随机向量
- 沿着梯度下降的方向进行迭代，使更新后的$L(\widehat{\boldsymbol{w}})$ 不断变小

$$\widehat{\boldsymbol{w}} = \widehat{\boldsymbol{w}} - \eta \frac{\partial \boldsymbol{L}(\widehat{\boldsymbol{w}})}{\partial \widehat{\boldsymbol{w}}}$$

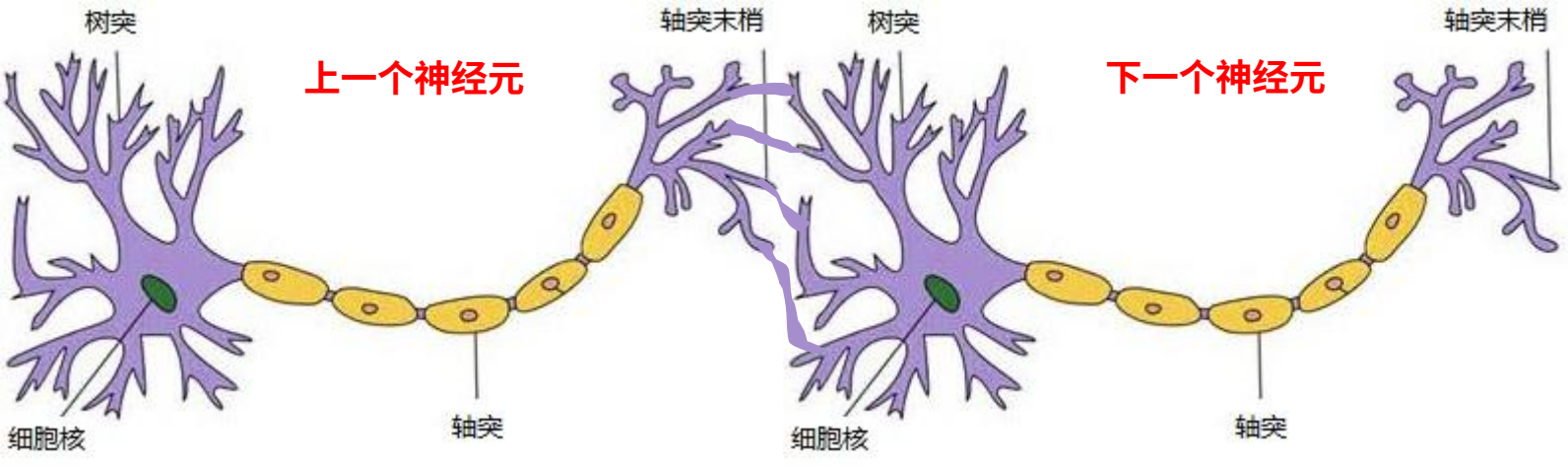
η称为学习率或步长

迭代至找到使得$L(\widehat{\boldsymbol{w}})$最小的$\widehat{\boldsymbol{w}}$值停止，从而得到回归模型参数

Scaling Laws: 更大模型+更多数据 → 可预测的性能提升

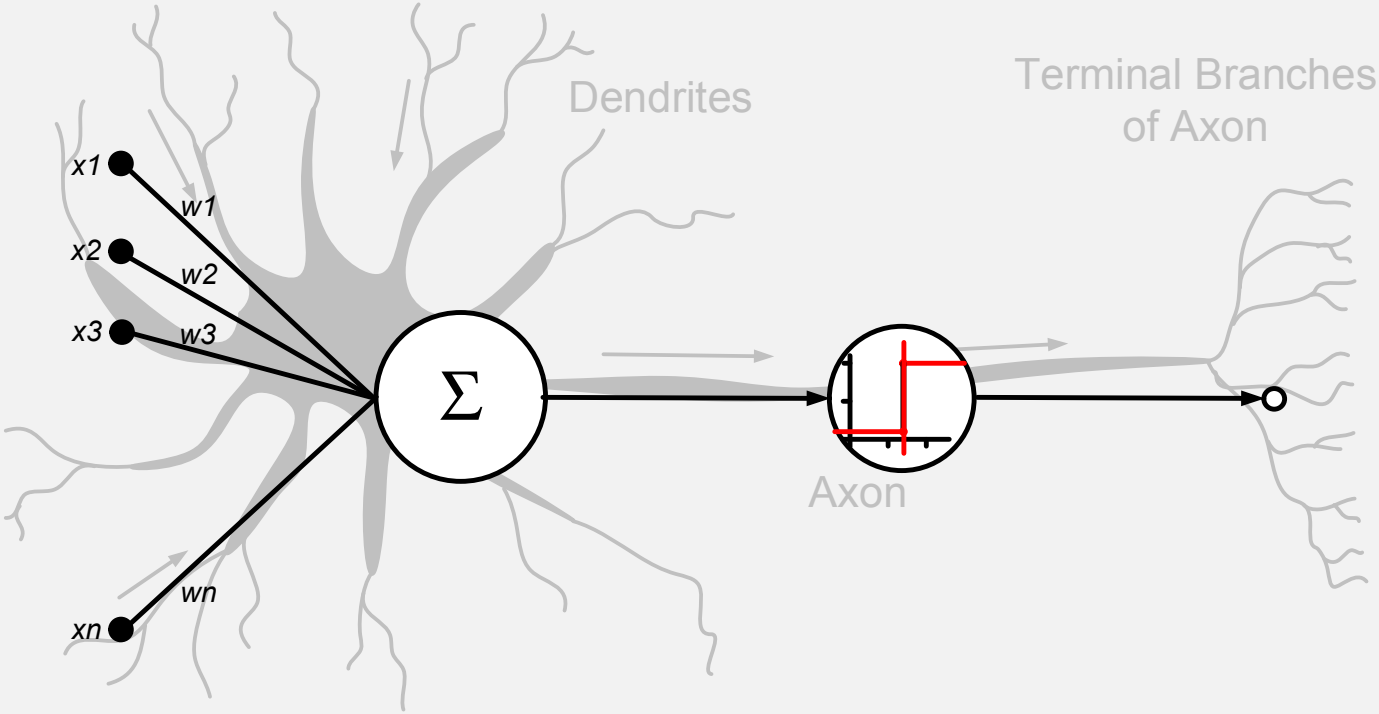
生物神经元

生物学领域，一个生物神经元有多个树突（dendrite，接受传入信息）；有一条轴突（axon），轴突尾端有许多轴突末梢（给其他多个神经元传递信息）。轴突末梢跟其它生物神经元的树突产生连接的位置叫做“突触”（synapse）。



人工神经元

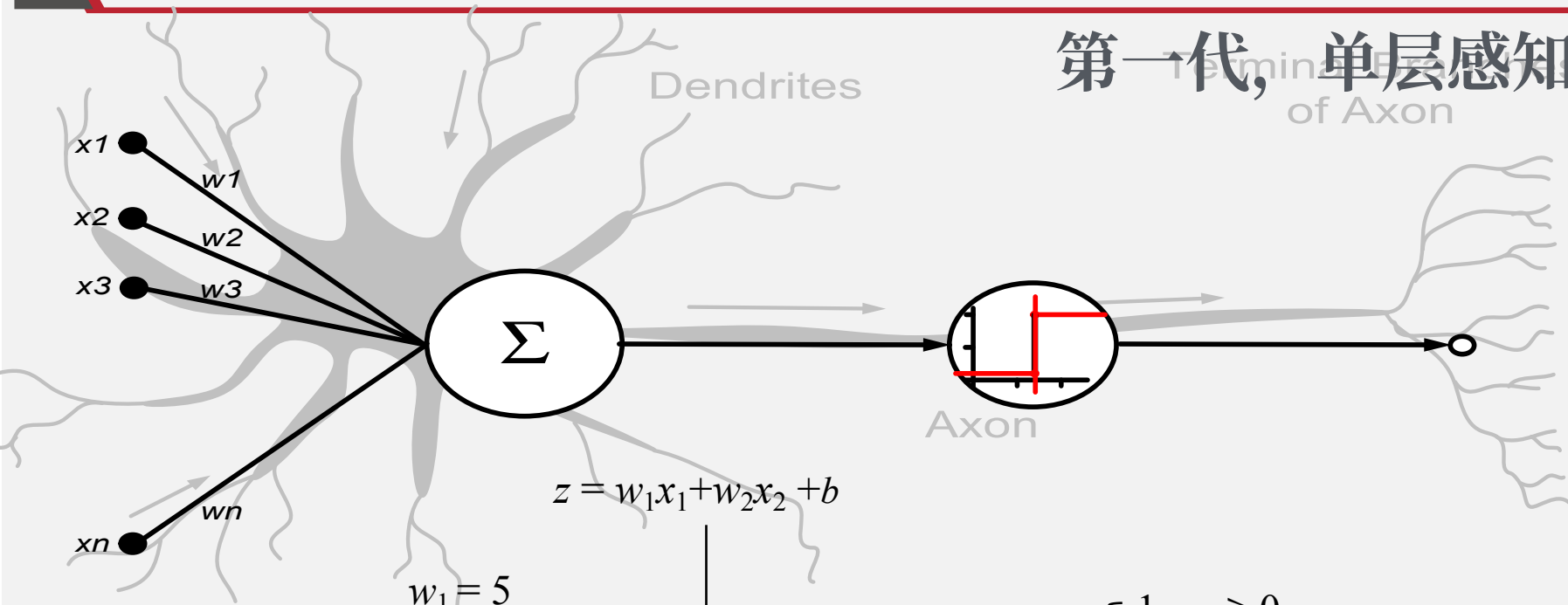
机器学习领域，人工神经元是一个包含输入，输出与计算功能的模型。不严格地说，其输入可类比为生物神经元的树突，其输出可类比为神经元的轴突，其计算可类比为细胞体。



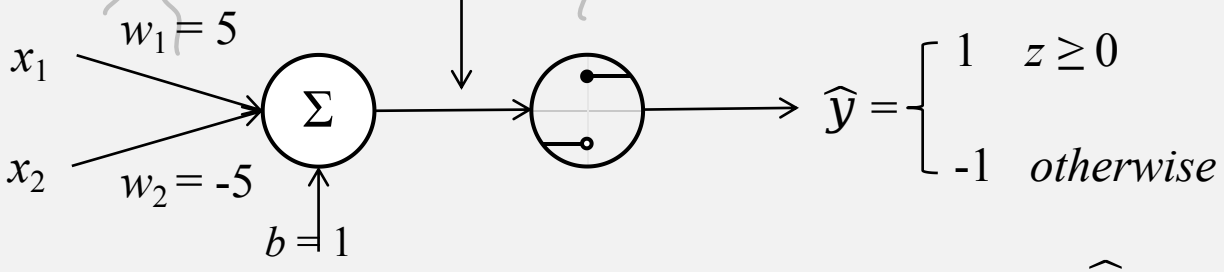
生物神经元：人工神经元=老鼠：米老鼠

一个神经元的单层感知机

第一代，单层感知机



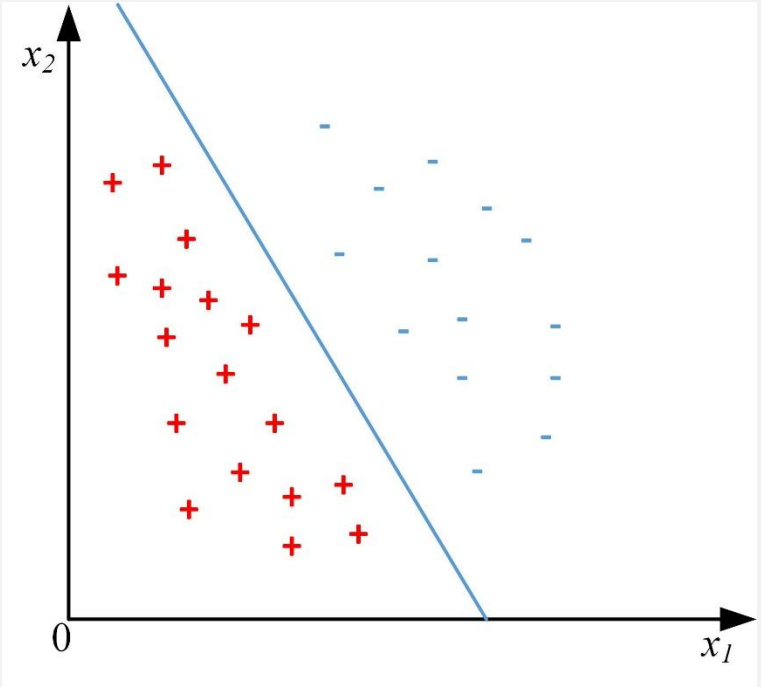
$$z = w_1 x_1 + w_2 x_2 + b$$



x_1	x_2	z	$\hat{y}$
1	-1	11	1
-1	1	-9	-1

感知机（Perceptron）模型

感知机模型$H(\boldsymbol{x}) = \text{sign}(\boldsymbol{w}^T \boldsymbol{x} + b)$对应一个超平面$\boldsymbol{w}^T \boldsymbol{x} + b = 0$，模型参数是$(\boldsymbol{w}, b)$。感知机的目标是找到一个$(\boldsymbol{w}, b)$，将线性可分的数据集T中的所有的样本点正确地分为两类。



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

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

寻找损失函数，并

将损失函数最小化

生物神经元 → 数学模型：加权求和 + 激活函数

➢ 寻找损失函数

考虑一个训练数据集$D=\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$，其中，$\boldsymbol{x}_j \in R^n$，$y_j \in \{+1, -1\}$。如果存在某个超平面S: $(\boldsymbol{w}^T \boldsymbol{x} + b = 0)$，能将正负样本分到S两侧，则说明数据集可分，那么，如何求出这个超平面S的表达式?

策略：假设误分类的点为数据集M，使用误分类点到超平面的总距离来寻找损失函数（直观来看，总距离越小越好）

梯度下降 = 闭眼下山，每步沿最陡方向走 lr 步

样本点$\boldsymbol{x}_j$到超平面S的距离：$$d = \frac{1}{\|\boldsymbol{w}\|} |\boldsymbol{w}^T \boldsymbol{x}_j + b|$$

$\|\boldsymbol{w}\|$是$\boldsymbol{w}$的L^2范数

➢ 寻找损失函数

数据集中误分类点满足条件：$-y_j(\boldsymbol{w}^T\boldsymbol{x}_j+b)>0$

去掉点 $\boldsymbol{x}_j$ 到超平面S的距离的绝对值符号：

$$d=-\frac{1}{\|\boldsymbol{w}\|}y_j(\boldsymbol{w}^T\boldsymbol{x}_j+b)$$

所有误分类点到超平面S的总距离为

$$d=-\frac{1}{\|\boldsymbol{w}\|}\sum_{\boldsymbol{x}_j\in M}y_j(\boldsymbol{w}^T\boldsymbol{x}_j+b)$$

由此寻找到感知机的损失函数

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

感知机算法

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

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

➡

损失函数

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

问题转化为寻找（ $\boldsymbol{w}, b$ ）使得损失函数极小化的最优化问题

SGD → Adam = Momentum + RMSProp，但显存开销是SGD的3倍

损失函数极小化的最优化问题可使用：随机梯度下降法

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

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}, b) = -\sum_{\boldsymbol{x}_j \in M} y_j \boldsymbol{x}_j \Rightarrow \boldsymbol{w} \leftarrow \boldsymbol{w} + \eta y_j \boldsymbol{x}_j$$

$$\nabla_b L(\boldsymbol{w}, b) = -\sum_{\boldsymbol{x}_j \in M} y_j \Rightarrow b \leftarrow b + \eta y_j$$

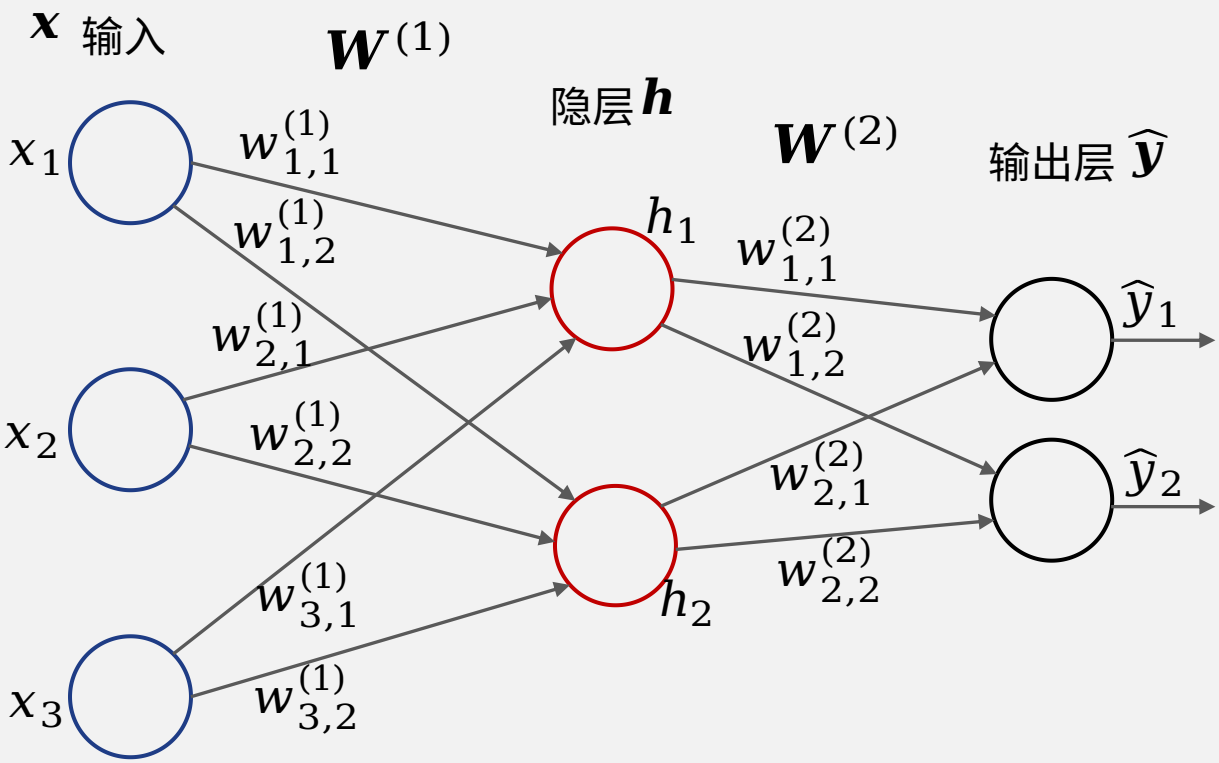
随机选取误分类点$(\boldsymbol{x}_j, y_j)$对$\boldsymbol{w}, b$以η为步长进行更新，通过迭代可以使得损失函数$L(\boldsymbol{w}, b)$不断减小，直到为0 $\boldsymbol{L(w, b) \to 0}$

两层神经网络-多层感知机

- 将大量的神经元模型进行组合，用不同的方法进行连接并作用在不同的激活函数上，就构成了人工神经网络模型

万能近似定理：足够宽的2层网络可逼近任意函数

- 全连接的两层神经网络模型也称为多层感知机（MLP）



浅层神经网络特点

需要数据量小、训练快，其局限性在于对复杂函数的表示能力有限，针对复杂分类问题其泛化能力受到制约

Why Not Go Deeper?

- Kurt Hornik证明了理论上两层神经网络足以拟合任意函数
- 过去也没有足够的数据和计算能力

- MLP = 多个全连接层堆叠，每层做 $h = \sigma(Wx + b)$

深度学习（深层神经网络）

2006年，Hinton在Science发表了论文（Reducing the dimensionality of data with neural networks. Science, Vol. 313. no. 5786），给多层神经网络相关的学习方法赋予了一个新名词--“深度学习”。他和LeCun以及Bengio三人被称为深度学习三位开创者

Geoffery Hinton

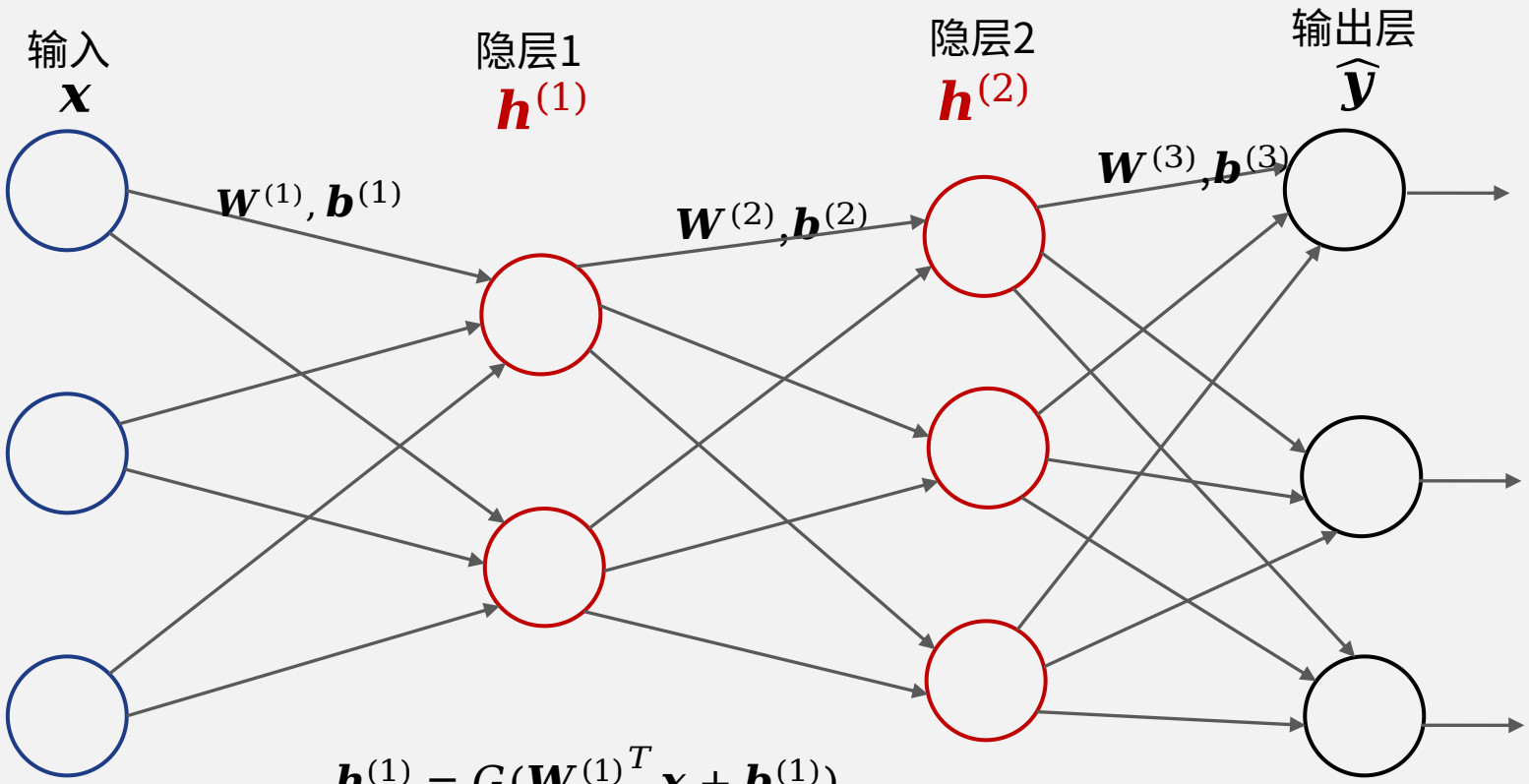
深度神经网络的成功：ABC

深度神经网络不断发展不仅依赖于自身的结构优势，也依赖于如下一些外在因素

- Algorithm：算法日新月异，优化算法层出不穷（学习算法->BP 算法-> Pre-training，Dropout等方法）
- Big data：数据量不断增大（10-> 10 k ->100M）
- Computing：处理器计算能力的不算提升（晶体管->CPU -> 集群/GPU ->智能处理器）

反向传播 + GPU → 深度学习的两大使能技术

多层神经网络



推导公式

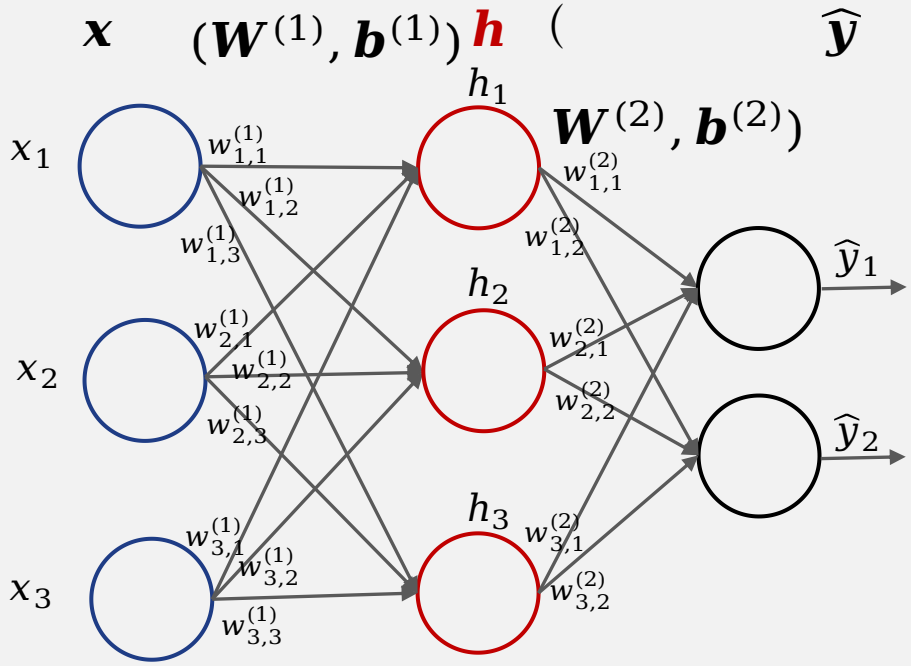
$$\boldsymbol{h}^{(1)} = G(\boldsymbol{W}^{(1)^T}\boldsymbol{x} + \boldsymbol{b}^{(1)})$$

$$\boldsymbol{h}^{(2)} = G(\boldsymbol{W}^{(2)^T}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)})$$

$$\hat{\boldsymbol{y}} = G(\boldsymbol{W}^{(3)^T}\boldsymbol{h}^{(2)} + \boldsymbol{b}^{(3)})$$

CNN处理网格数据（图像），RNN处理序列数据（文本/语音）

正向传播



输入 $\boldsymbol{x} = [x_1;\ x_2;\ x_3]$

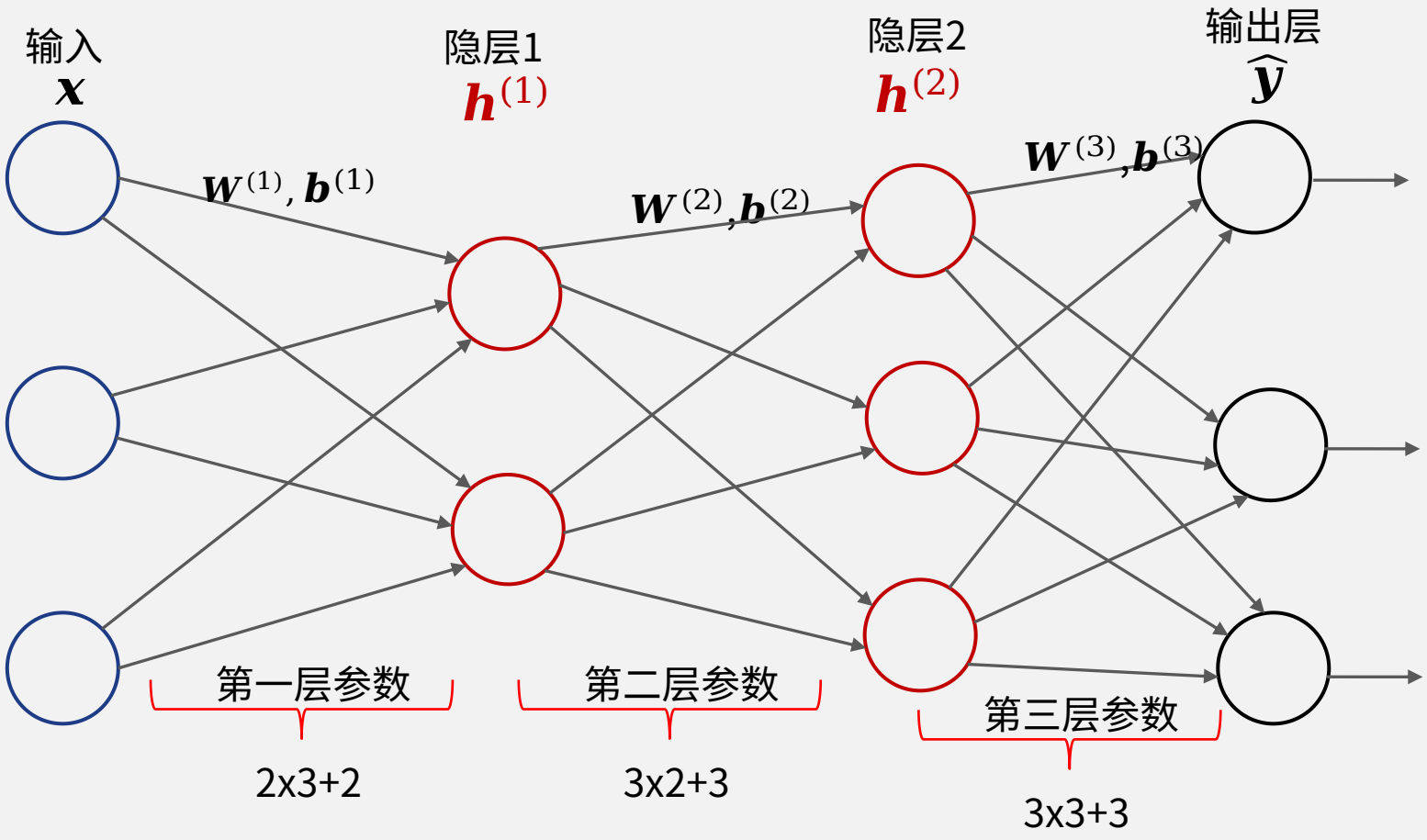
输入 $\boldsymbol{h} = [h_1;\ h_2;\ h_3]$

权重 $\boldsymbol{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} \end{bmatrix}$

权重 $\boldsymbol{W}^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} \\ w_{2,1}^{(2)} & w_{2,2}^{(2)} \\ w_{3,1}^{(2)} & w_{3,2}^{(2)} \end{bmatrix}$

Attention机制：让模型动态聚焦相关信息，Q和K→权重→加权V

多层神经网络



需 8+9+12=29个参数

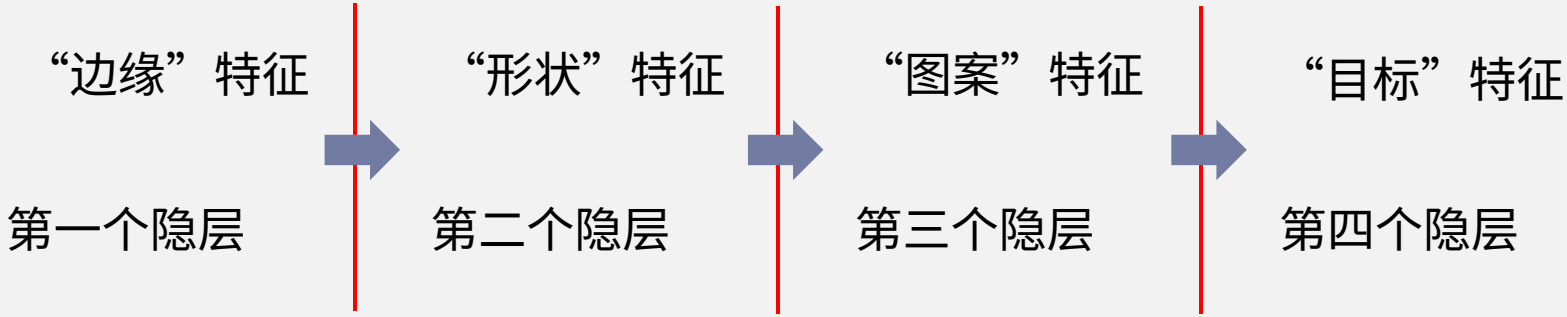
层次化表示：边缘→纹理→部件→物体（CNN）/ 词法→语义→推理

多层神经网络

- 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

  层次化表示学习 = 深度学习的核心优势

- 通过抽取更抽象的特征来对事物进行区分，从而获得更好的区分与分类能力



从传统ML到Foundation Model的范式转变

传统：每个任务训练专用模型

Foundation Model：预训练+微调/提示

关键转变：从Task-specific到General

数据需求：从标注数据到海量无标注数据

2020-2025：FM论文数量从500到9000+

NEW

Scaling Laws：规模即性能?

Kaplan 2020: Loss与模型、数据、算力的幂律关系

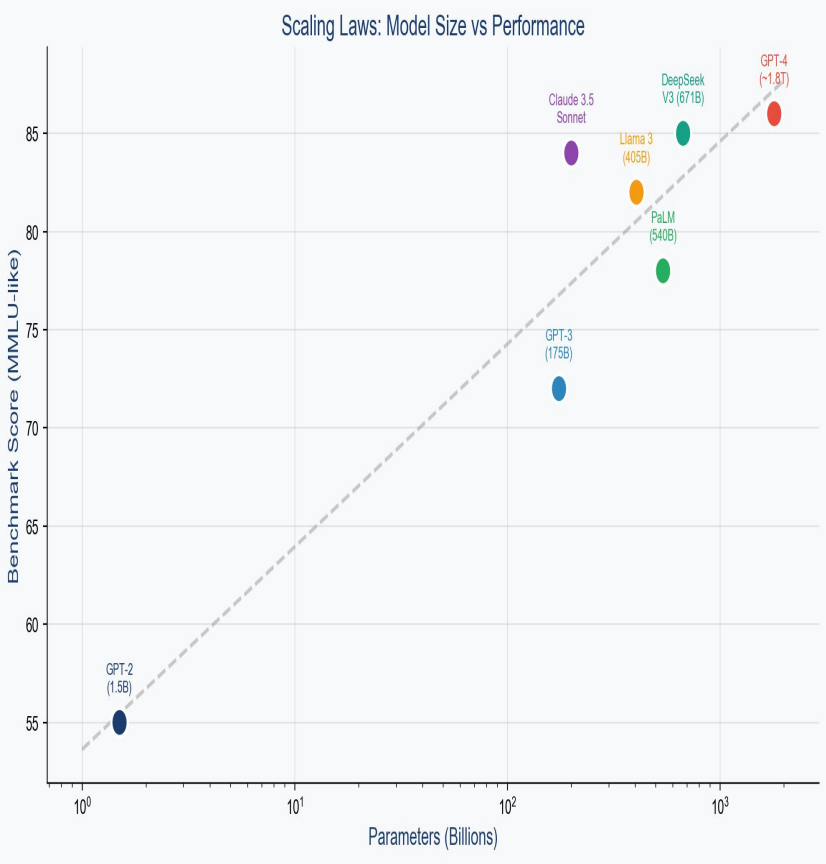
Chinchilla 2022: 模型与数据应同步扩展

实践演进: Llama-3 token/param比达200:1

2024新方向: Test-time Compute Scaling

系统启示: 规模增长驱动系统创新

Scaling Laws：Loss ∝ N^(-0.076)·D^(-0.095)·C^(-0.050)

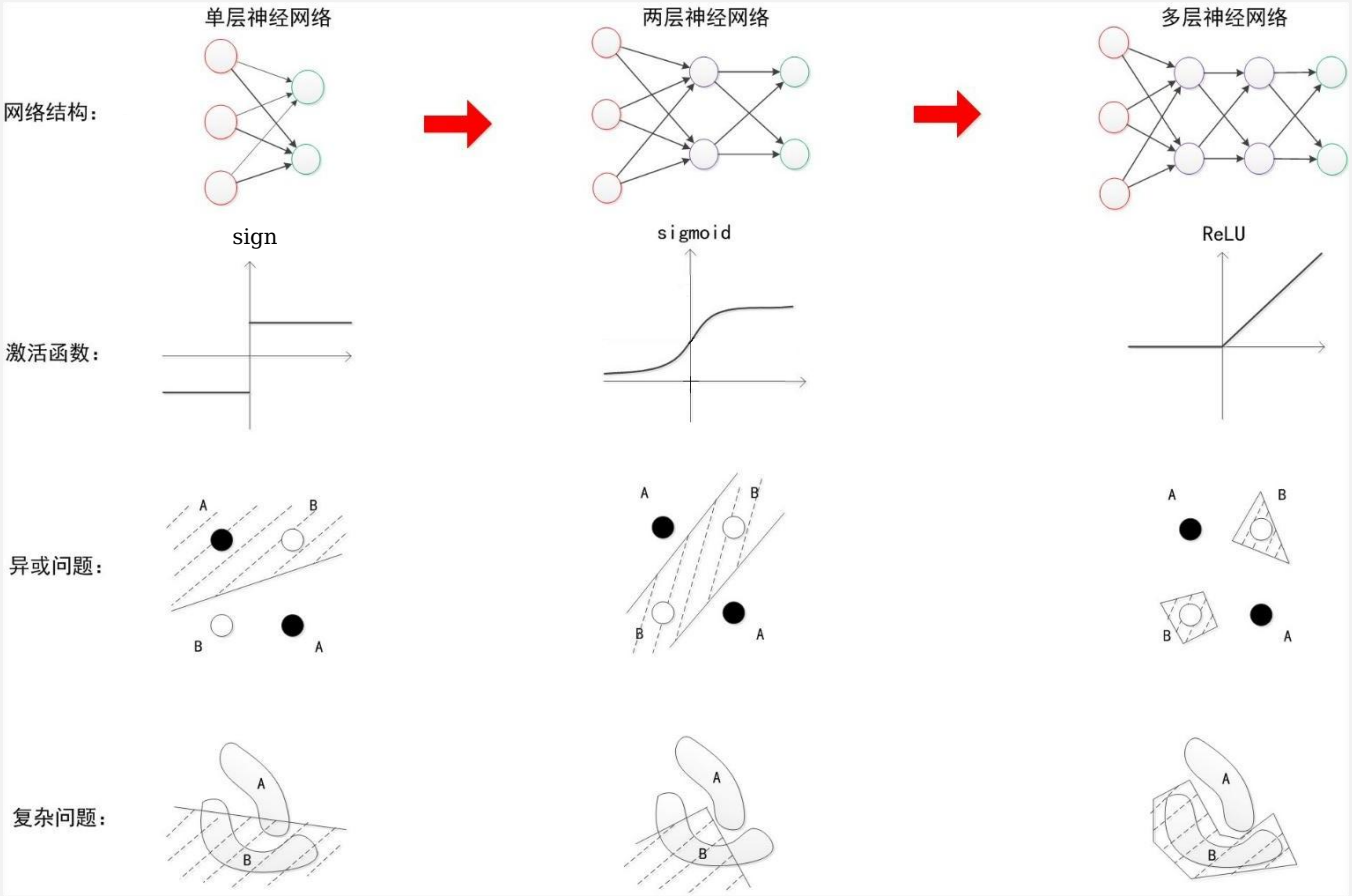


本节课内容

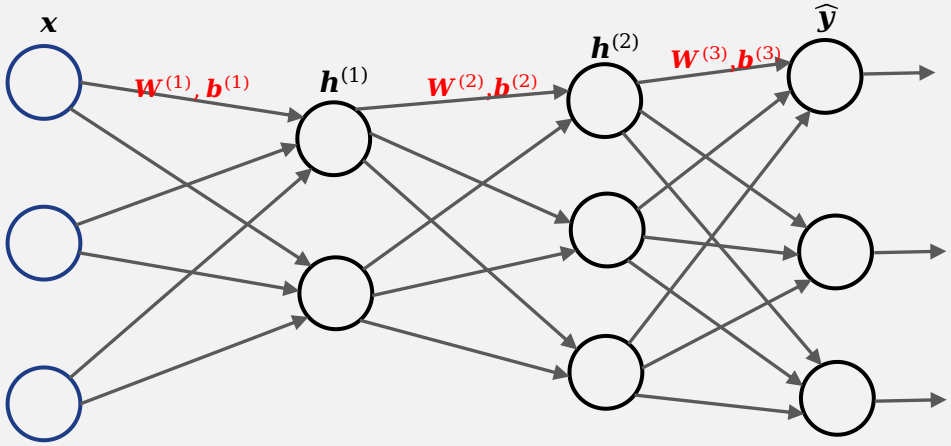
- 机器学习
- 神经网络
- **神经网络训练方法**
- 编程框架设计
- 计算图构建
- 计算图执行
- 本章小结

多层神经网络

从单层神经网络，到两层神经网络，再到多层神经网络，随着网络层数的增加，以及激活函数的调整，神经网络拟合非线性分界不断增强



神经网络的模型训练

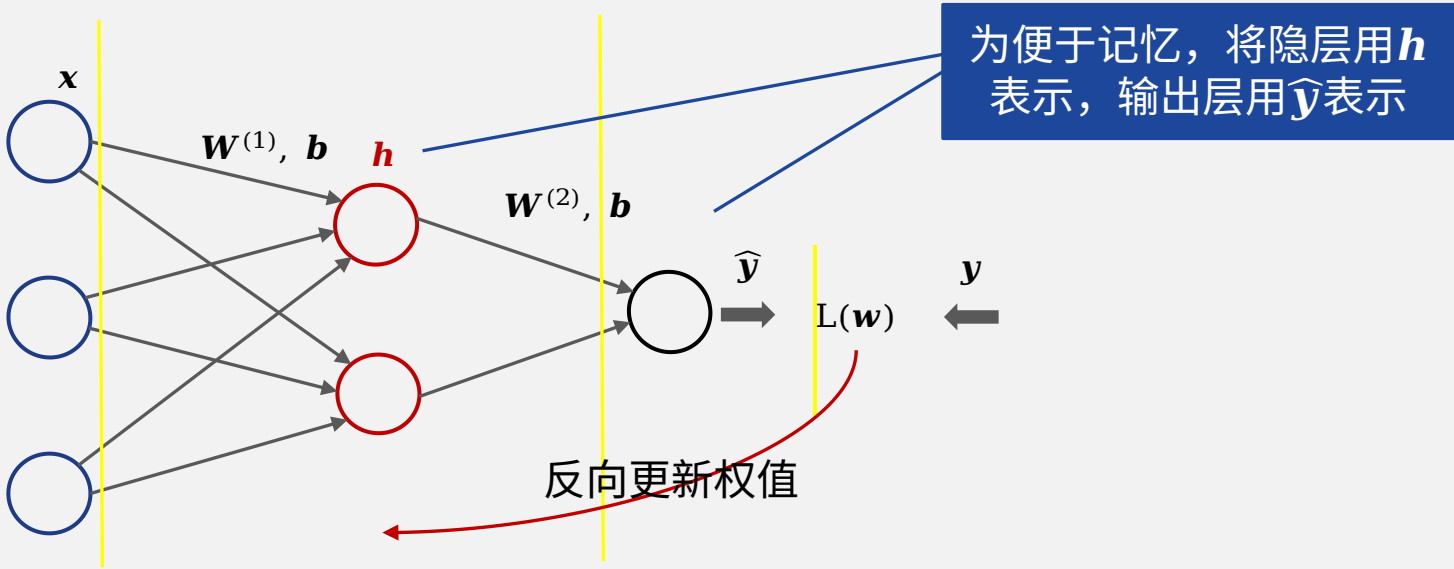


- 模型训练的目的，就是调整参数使得模型计算值$\hat{y}$尽可能的与真实值y逼近

训练三步循环：前向传播 → 计算损失 → 反向传播更新参数

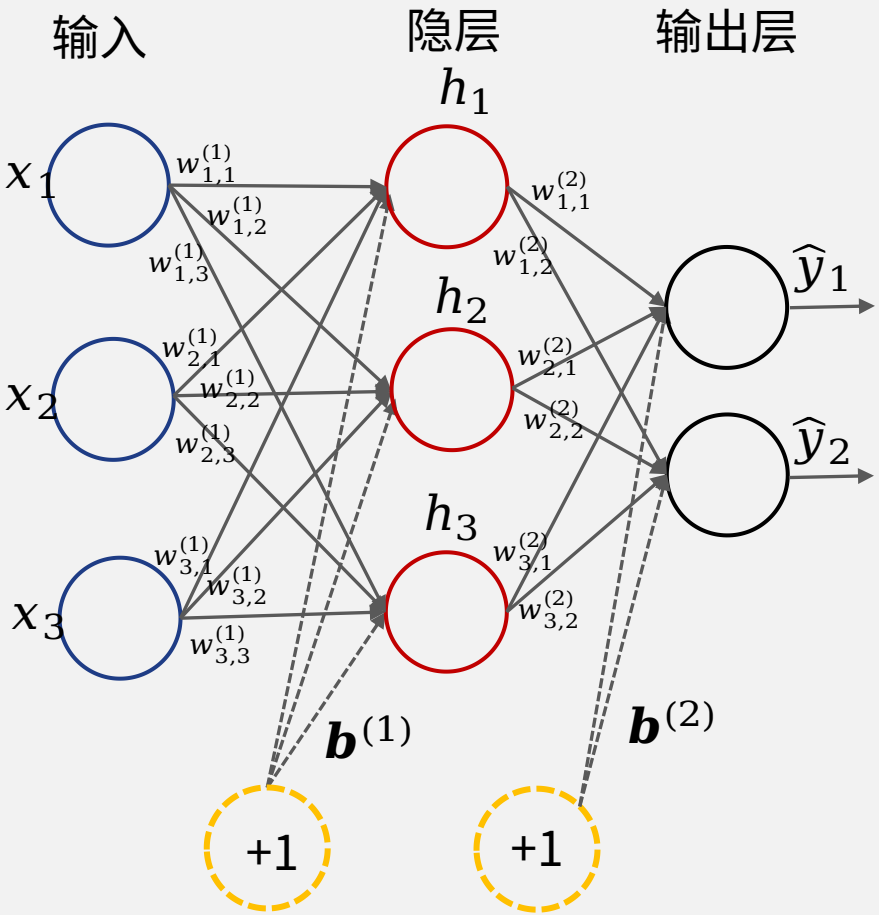
神经网络训练

- 正向传播是根据输入，经过权重、激活函数计算出隐层，将输入的特征向量从低级特征逐步提取为抽象特征，直到得到最终输出结果的过程



- 反向传播是根据正向传播的输出结果和期望值计算出损失函数，再通过链式求导，最终从网络后端逐步修改权重使输出和期望值的差距变到最小的过程

前向传播 = 逐层计算 y = f(Wx+b)，信号从输入到输出



- 输入：包含神经元 x_1，x_2，x_3
- 隐层：包含 h_1, h_2, h_3
- 输出层：包含$\hat{y}_1$， $\hat{y}_2$

- 输入和隐层之间

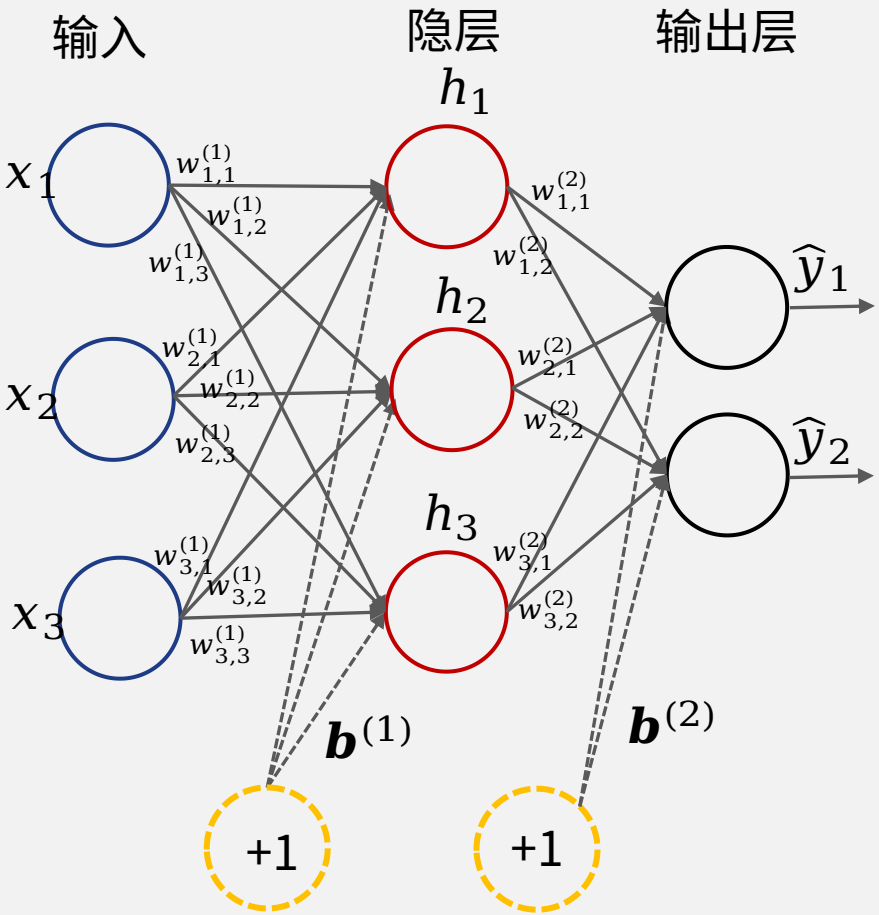
  偏置：$\boldsymbol{b}^{(1)}$

  权重：$\boldsymbol{W}^{(1)}$

- 隐层和输出层之间

  偏置：$\boldsymbol{b}^{(2)}$

  权重：$\boldsymbol{W}^{(2)}$



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

正向传输：输入到隐层

$$\boldsymbol{v} = \boldsymbol{W}^{(1)^T}\boldsymbol{x} + \boldsymbol{b}^{(1)}$$

$$= \begin{bmatrix} w_{1,1}^{(1)} & w_{2,1}^{(1)} & w_{3,1}^{(1)} \\ w_{1,2}^{(1)} & w_{2,2}^{(1)} & w_{3,2}^{(1)} \\ w_{1,3}^{(1)} & w_{2,3}^{(1)} & w_{3,3}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \boldsymbol{b}^{(1)}$$

$$\boldsymbol{h} = \frac{1}{1+e^{-\boldsymbol{v}}}$$

示例

假定输入数据 $x_1 = 0.02$ 、$x_2 = 0.04$ 、$x_3 = 0.01$

固定偏置$\boldsymbol{b}^{(1)} = [0.4; 0.4; 0.4]$、$\boldsymbol{b}^{(2)} = [0.7; 0.7]$

期望输出$y_1 = 0.9$、$y_2 = 0.5$

未知权重 $\boldsymbol{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} \end{bmatrix}$, $\boldsymbol{W}^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} \\ w_{2,1}^{(2)} & w_{2,2}^{(2)} \\ w_{3,1}^{(2)} & w_{3,2}^{(2)} \end{bmatrix}$

目的是为了能得到 $y_1 = 0.9$、$y_2 = 0.5$的期望的值，需计算出合适的 $\boldsymbol{W}^{(1)}$、$\boldsymbol{W}^{(2)}$ 的权重值

- 初始化权重值

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} \end{bmatrix} = \begin{bmatrix} 0.25 & 0.15 & 0.30 \\ 0.25 & 0.20 & 0.35 \\ 0.10 & 0.25 & 0.15 \end{bmatrix}$$

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} \\ w_{2,1}^{(2)} & w_{2,2}^{(2)} \\ w_{3,1}^{(2)} & w_{3,2}^{(2)} \end{bmatrix} = \begin{bmatrix} 0.40 & 0.25 \\ 0.35 & 0.30 \\ 0.01 & 0.35 \end{bmatrix}$$

隐层→输出层：o=w5·a1+w6·a2+b3→sigmoid→ŷ

→ 完整数据流：x→线性→激活→线性→激活→ŷ

→ GPT-3：96层，~1750亿参数

一次前向≈3500亿FLOPs，A100上需几百ms

→ 下一步：ŷ和y差多少？→损失函数

系统视角：反向传播的计算量 ≈ 2×前向传播

- 输入到隐层计算

$$\boldsymbol{v}=\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \boldsymbol{W}^{(1)^T}\boldsymbol{x}+\boldsymbol{b}^{(1)} = \begin{bmatrix} 0.25 & 0.25 & 0.10 \\ 0.15 & 0.20 & 0.25 \\ 0.30 & 0.35 & 0.15 \end{bmatrix}\begin{bmatrix} 0.02 \\ 0.04 \\ 0.01 \end{bmatrix}+\begin{bmatrix} 0.4 \\ 0.4 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.4160 \\ 0.4135 \\ 0.4215 \end{bmatrix}$$

$$\boldsymbol{h}=\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \frac{1}{1+e^{-\boldsymbol{v}}} = \begin{bmatrix} \frac{1}{1+e^{-0.416}} \\ \frac{1}{1+e^{-0.4135}} \\ \frac{1}{1+e^{-0.4215}} \end{bmatrix} = \begin{bmatrix} 0.6025 \\ 0.6019 \\ 0.6038 \end{bmatrix}$$

- 隐层到输出层计算

$$\boldsymbol{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \boldsymbol{W}^{(2)^T}\boldsymbol{h} + \boldsymbol{b}^{(2)} = \begin{bmatrix} 0.40 & 0.35 & 0.01 \\ 0.25 & 0.30 & 0.35 \end{bmatrix} \begin{bmatrix} 0.6025 \\ 0.6019 \\ 0.6038 \end{bmatrix} + \begin{bmatrix} 0.7 \\ 0.7 \end{bmatrix} = \begin{bmatrix} 1.1577 \\ 1.2425 \end{bmatrix}$$

$$\widehat{\boldsymbol{y}} = \begin{bmatrix} \widehat{y}_1 \\ \widehat{y}_2 \end{bmatrix} = \frac{1}{1+e^{-\boldsymbol{z}}} = \begin{bmatrix} \frac{1}{1+e^{-1.1577}} \\ \frac{1}{1+e^{-1.2425}} \end{bmatrix} = \begin{bmatrix} 0.7609 \\ 0.7760 \end{bmatrix}$$

距离期望输出 $y_1 = 0.9$、$y_2 = 0.5$ 还有差距，通过反向传播修改权重

前向传播必须保存中间激活值 → 这是显存的主要消耗

模型计算输出

$$\begin{bmatrix}\hat{y}_1\\ \hat{y}_2\end{bmatrix} = \begin{bmatrix}0.7609\\ 0.7760\end{bmatrix}$$

期望输出

$$y_1 = 0.9$$

$$y_2 = 0.5$$

- 计算误差

$$L(\boldsymbol{W}) = L_1 + L_2 = \frac{1}{2}(y_1 - \hat{y}_1)^2 + \frac{1}{2}(y_2 - \hat{y}_2)^2$$

$$= \frac{1}{2}(0.9 - 0.7609)^2 + \frac{1}{2}(0.5 - 0.7760)^2 = 0.0478$$

计算值与真实值之间还有很大的差距，如何缩小计算值与真实值之间的误差?

通过反向传播进行反馈，调节权重值

实例：2层网络手动推导反向传播的完整过程

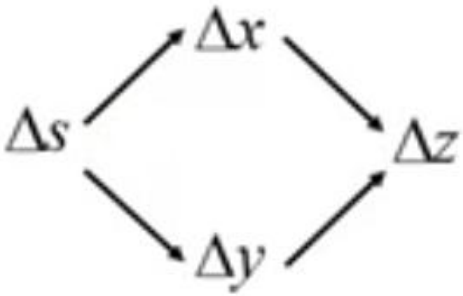
链式法则

Case 1 $y = g(x) \quad z = h(y)$

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

Case 2

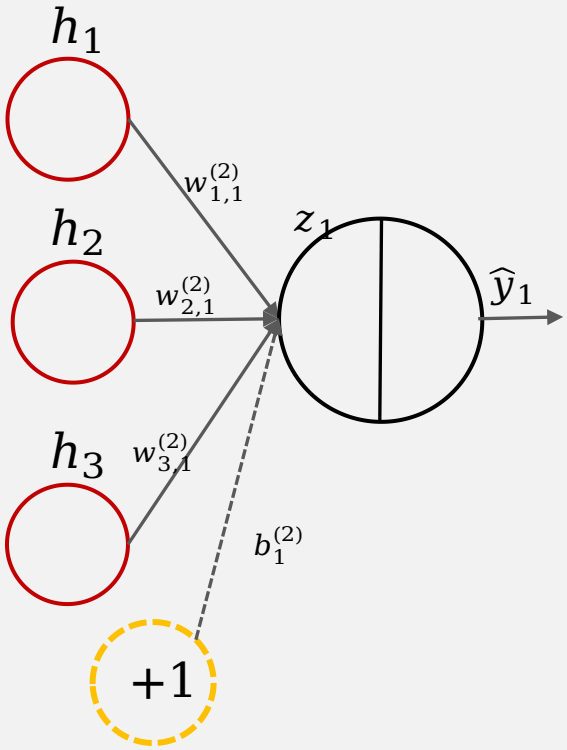
$$x = g(s) \qquad y = h(s) \qquad z = k(x, y)$$



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

反向传播

- 隐层到输出层的权重$\boldsymbol{W}^{(2)}$的更新



$$L(\boldsymbol{W}) = L_1 + L_2 = \frac{1}{2}(y_1 - \hat{y}_1)^2 + \frac{1}{2}(y_2 - \hat{y}_2)^2$$

- 以 $w_{2,1}^{(2)}$(记为 ω) 参数为例子，计算 ω 对整体误差的影响有多大，可以使用整体误差对 ω 参数求偏导

梯度消失：sigmoid导数≤0.25 → 10层后梯度≈10^{-6}

权重梯度 = 该层误差项 × 该层输入

→ dL/dw5 = delta_output × a1

→ 梯度通过权重矩阵'反向'传播

→ 前向用W，反向用W^T

→ 反复应用链式法则→所有权重梯度算出

- 根据偏导数的链式法则推导

$$\frac{\partial L(\boldsymbol{W})}{\partial \omega} = \frac{\partial L(\boldsymbol{W})}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial z_1} \frac{\partial z_1}{\partial \omega}$$

$$L(\boldsymbol{W}) = \frac{1}{2}(y_1 - \hat{y}_1)^2 + \frac{1}{2}(y_2 - \hat{y}_2)^2$$

梯度累积：小显存也能模拟大batch训练

$$\frac{\partial L(\boldsymbol{W})}{\partial \hat{y}_1} = -(y_1 - \hat{y}_1) = -(0.9 - 0.7609) = -0.1391$$

$$\hat{y}_1 = \frac{1}{1 + e^{-z_1}}$$

$$\frac{\partial \hat{y}_1}{\partial z_1} = \hat{y}_1(1 - \hat{y}_1) = 0.7609 * (1 - 0.7609) = 0.1819$$

➢ 根据偏导数的链式法则推导

混合精度训练：FP16计算+FP32主权重 → 速度翻倍、显存减半

$$\frac{\partial L(\boldsymbol{W})}{\partial \omega} = \frac{\partial L(\boldsymbol{W})}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial z_1} \frac{\partial z_1}{\partial \omega}$$

$$z_1 = w_{1,1}^{(2)} \times h_1 + \omega \times h_2 + w_{3,1}^{(2)} \times h_3 + b_1^{(2)}$$

$$\frac{\partial z_1}{\partial \omega} = h_2 = 0.6019$$

$$\frac{\partial L(\boldsymbol{W})}{\partial \omega} = -(y_1 - \hat{y}_1) \times \hat{y}_1(1 - \hat{y}_1) \times h_2$$
$$= -0.1391 \times 0.1819 \times 0.6019$$
$$= -0.0152$$

➢ 更新$w_{2,1}^{(2)}$（记为ω）的值

初始值

$$\boldsymbol{W}^{(2)}=\begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} \\ w_{2,1}^{(2)} & w_{2,2}^{(2)} \\ w_{3,1}^{(2)} & w_{3,2}^{(2)} \end{bmatrix}=\begin{bmatrix} 0.40 & 0.25 \\ 0.35 & 0.30 \\ 0.01 & 0.35 \end{bmatrix}$$

$$\frac{\partial L(\boldsymbol{W})}{\partial \omega}=-0.0152$$

$$\omega=\omega-\eta\times\frac{\partial L(\boldsymbol{W})}{\partial \omega}=0.35-(-0.0152)=0.3652$$

➢ 同理，可以计算新的$\boldsymbol{W}^{(2)}$的其他元素的权重值

- 反向传播的作用是将神经网络的输出误差反向传播到神经网络的输入端，并以此来更新神经网络中各个连接的权重

- 当第一次反向传播法完成后，网络的模型参数得到更新，网络进行下一轮的正向传播过程，如此反复的迭代进行训练，从而不断缩小计算值与真实值之间的误差

- 反向传播是现代深度学习框架的核心功能。PyTorch、TensorFlow都实现了自动微分（Autograd），你只需要写前向传播的代码，框架自动帮你算梯度

混合精度训练: FP16/BF16/FP8

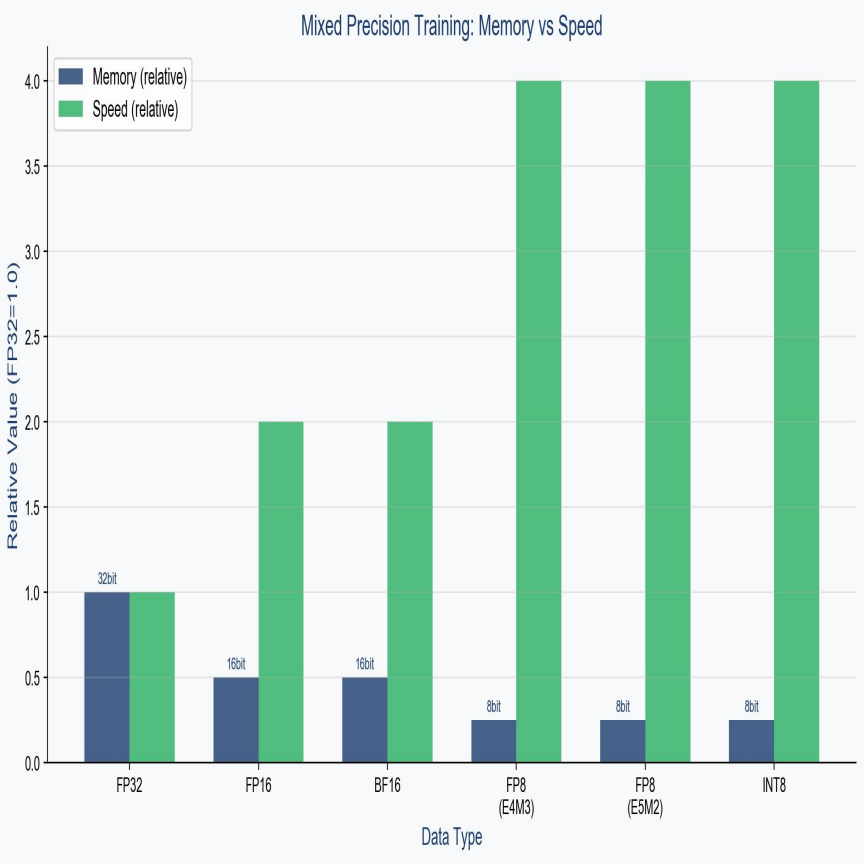
FP32 -| FP16/BF16: 内存减半，速度翻倍

Loss Scaling防止梯度下溢

BF16: 动态范围同FP32，更稳定

FP8 (H100+): 吞吐量再翻倍

实践: PyTorch AMP一行代码开启



优化器演进: 从SGD到Sophia

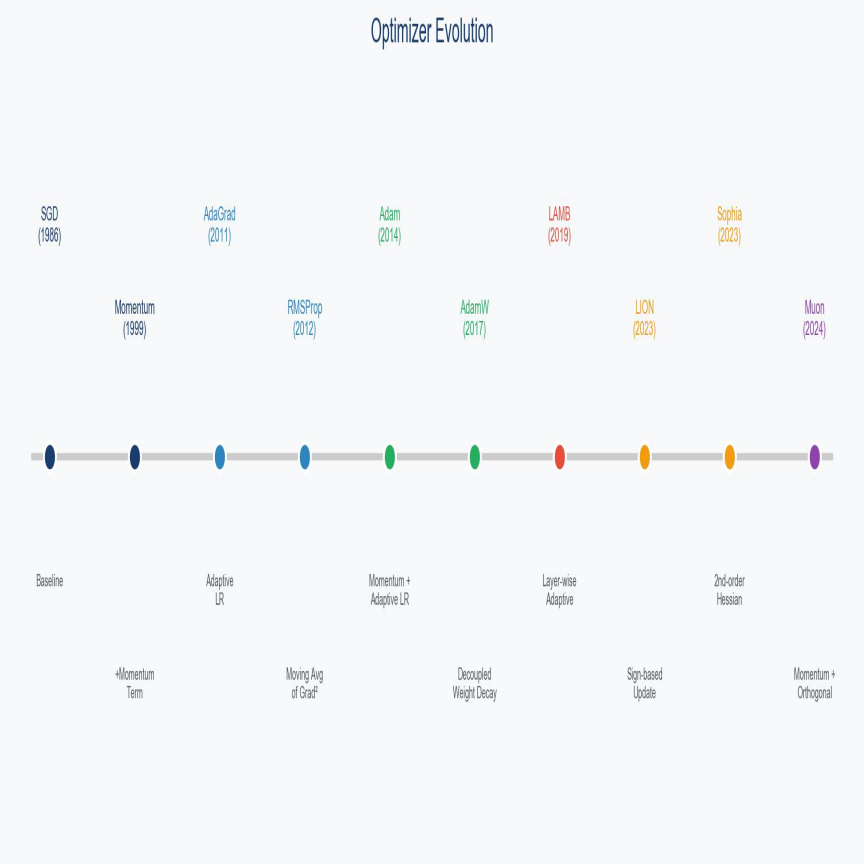
SGD+Momentum: 经典但需精调学习率

Adam: 自适应学习率，深度学习标配

AdamW: 解耦权重衰减，大模型首选

LION: 进化搜索发现，内存省一半

Sophia: 二阶信息，收敛快50%



NEW

梯度累积与梯度检查点

梯度累积: 小显存模拟大Batch训练

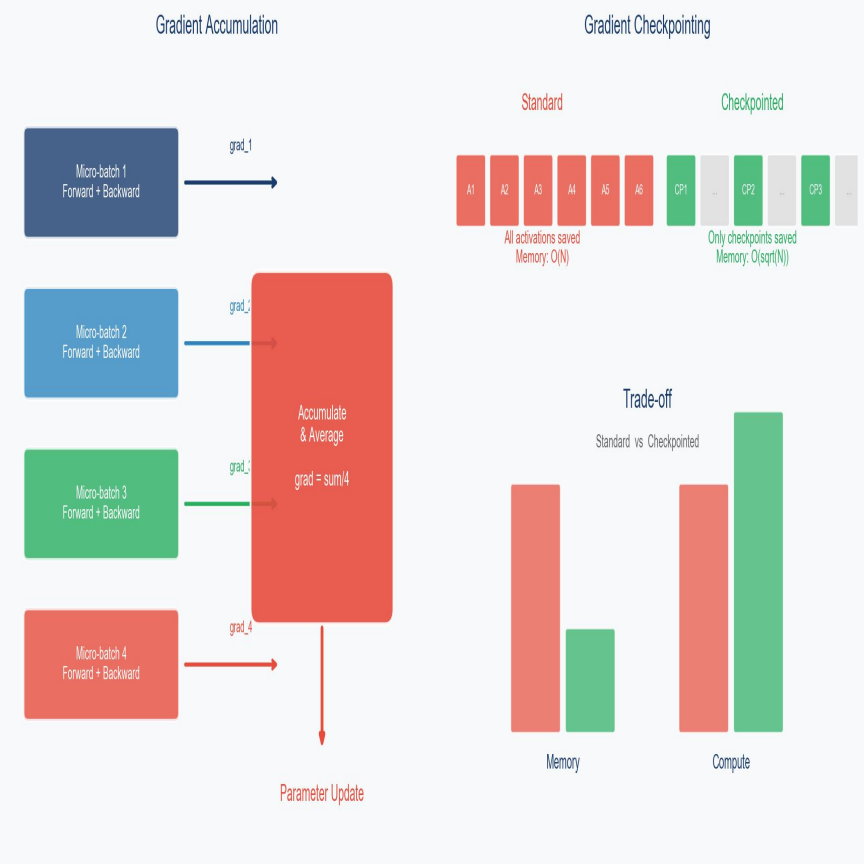
多步micro-batch累加后统一更新

梯度检查点: 用时间换空间

只存部分激活值，反向时重新计算

节省40-60%显存，代价~20-30%速度

三者组合: 混合精度+累积+检查点



主流框架对比: PyTorch vs TensorFlow vs JAX vs MindSpore

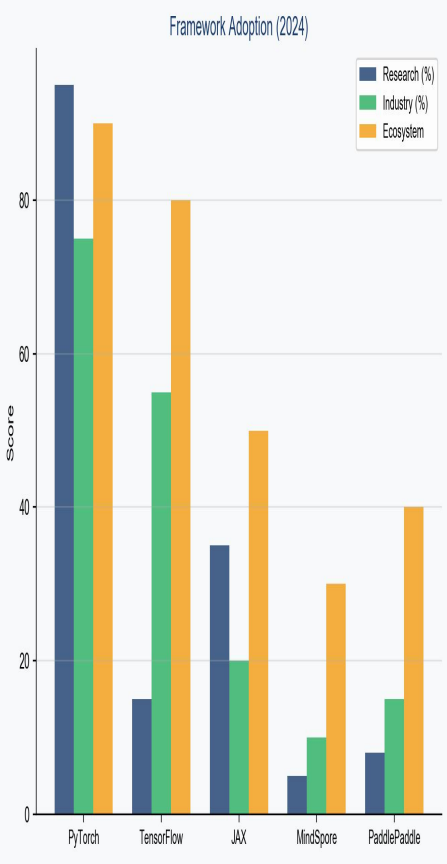
PyTorch: 动态图+Pythonic，研究界185%使用

TensorFlow: 企业部署强，Keras高层接口

JAX: 函数式+XLA编译，科学计算首选

MindSpore: 华为生态，Ascend芯片深度适配

趋势: 编译优化成为各框架竞争焦点



Feature Comparison

Feature	PyTorch	JAX	TensorFlow
Eager Mode	Yes	No*	TF2 Yes
Graph Mode	torch.compile	XLA JIT	tf.function
Auto Diff	Autograd	Func. AD	GradientTape
Distributed	FSDP	pjit	Strategy
Compiler	Inductor	XLA	XLA
Mobile	ExecuTorch	Limited	TFLite

从数学推导到系统实现

手动实现反向传播

```
# 20+ lines of code
def backward(y, y_hat, W2, W1, h, x):
    dL = 2*(y_hat - y)
    dW2 = h.T @ dL
    db2 = dL.sum(axis=0)
    dh = dL @ W2.T
    dh[h <= 0] = 0  # ReLU grad
    dW1 = x.T @ dh
    db1 = dh.sum(axis=0)
    W2 -= lr * dW2
    W1 -= lr * dW1
    ...
```

PyTorch 等价实现

```
# Just 3 lines!

loss = criterion(y_hat, y)
loss.backward()
optimizer.step()
```

框架帮我们做了什么?

自动构建计算图

自动求导 (AutoGrad)

内存管理与优化

多设备调度执行

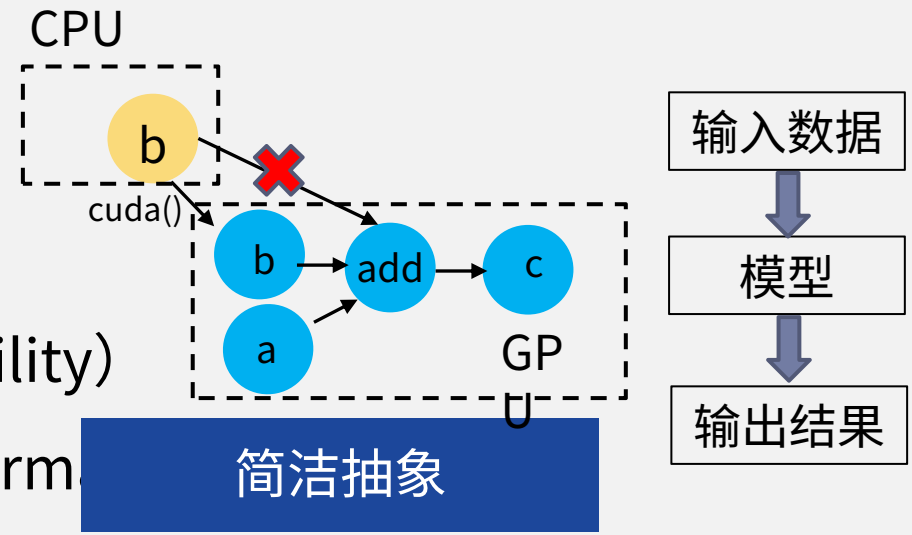
接下来，我们深入框架内部 —— 看看这3行代码背后的系统设计

本节课内容

- 机器学习
- 神经网络
- 神经网络训练方法
- **编程框架设计**
- 计算图构建
- 计算图执行
- 本章小结

1、设计原则

- 简洁性（Simplicity）
  - 框架提供一套抽象机制，用户仅需关心算法本身和部署策略



- 易用性（Usability）
- 高效性（Perform

- 框架四大核心模块：计算图、编译优化、运行时、分布式

1、设计原则

- 简洁性（Simplicity）
- 易用性（Usability）
  - 直观且用户友好的接口：如PyTorch提供了命令式的动态图编程方法
- 高效性（Performance）

- 硬件多样性 → 框架需要统一编程接口 + 多后端支持

1、设计原则

- 简洁性（Simplicity）
- 易用性（Usability）
- 高效性（Performance）
  - 如采用静态图编程方式，可以生成完整的计算图并进行全局优化，从而尽量提高用户应用程序的运行效率
  - 支持深度学习编译技术，多层级表示优化，充分利用用户硬件的计算能力
  - 支持多机多卡条件的分布式训练，从而高效支持大规模深度学习任务

- 框架设计三大目标：灵活性 · 高性能 · 可扩展

torch.compile: 动态图的编译革命

PyTorch 2.0核心特性 (2023)

TorchDynamo: Python字节码级别图捕获

TorchInductor: 生成Triton/C++ kernel

一行代码加速: model = torch.compile(model)

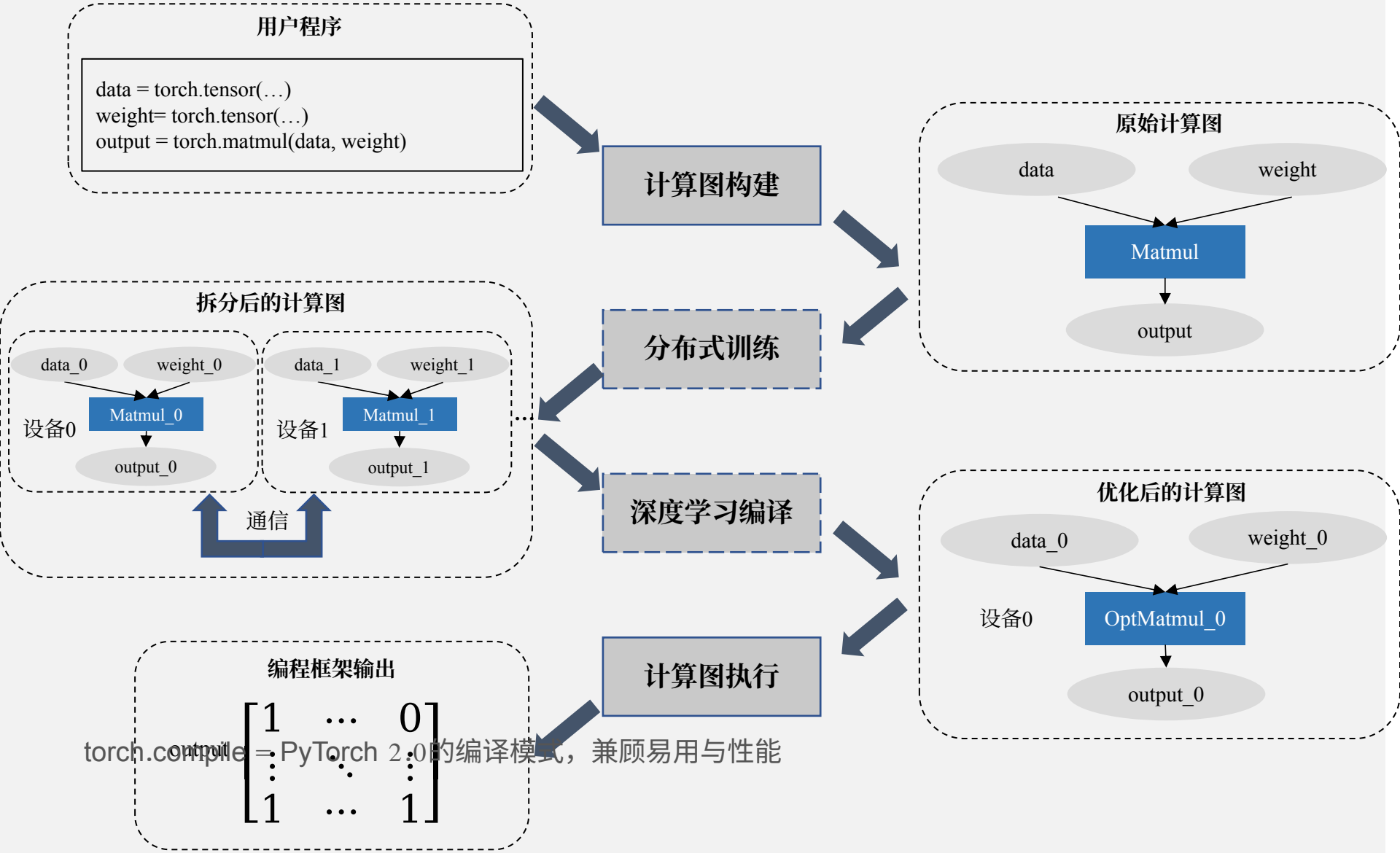
实测: 复杂模型加速2-5x，推理1.3-3.5x

从Theano到PyTorch 2.0：框架演进 = 灵活性与性能的平衡

2、整体架构

- 四大模块
  - **计算图构建**模块：完成从输入的用户程序到编程框架内部原始计算图的转换过程，编程框架的入口模块
  - **分布式训练**模块：应对更大规模的神经网络，将训练、推理任务从一台设备扩展到多台设备
  - **深度学习编译**模块：对计算图分别进行图层级和算子层级的编译优化，从而提升单设备上的执行效率
  - **计算图执行**模块：将优化后的计算图中的张量和操作映射到指定设备上进行具体执行，并给出编程框架的输出结果

2、整体架构


用户程序
data = torch.tensor(…)
weight= torch.tensor(…)
output = torch.matmul(data, weight)
计算图构建
原始计算图
data
weight
Matmul
output
分布式训练
拆分后的计算图
data_0
weight_0
Matmul_0
output_0
设备0
data_1
weight_1
Matmul_1
output_1
设备1
通信
深度学习编译
优化后的计算图
data_0
weight_0
OptMatmul_0
output_0
设备0
计算图执行
编程框架输出
matmul
torch.compile：PyTorch 2.0的编译模式，兼顾易用与性能

框架设计的系统视角: 易用性与性能的平衡

用户视角: Python接口、自动微分、动态图

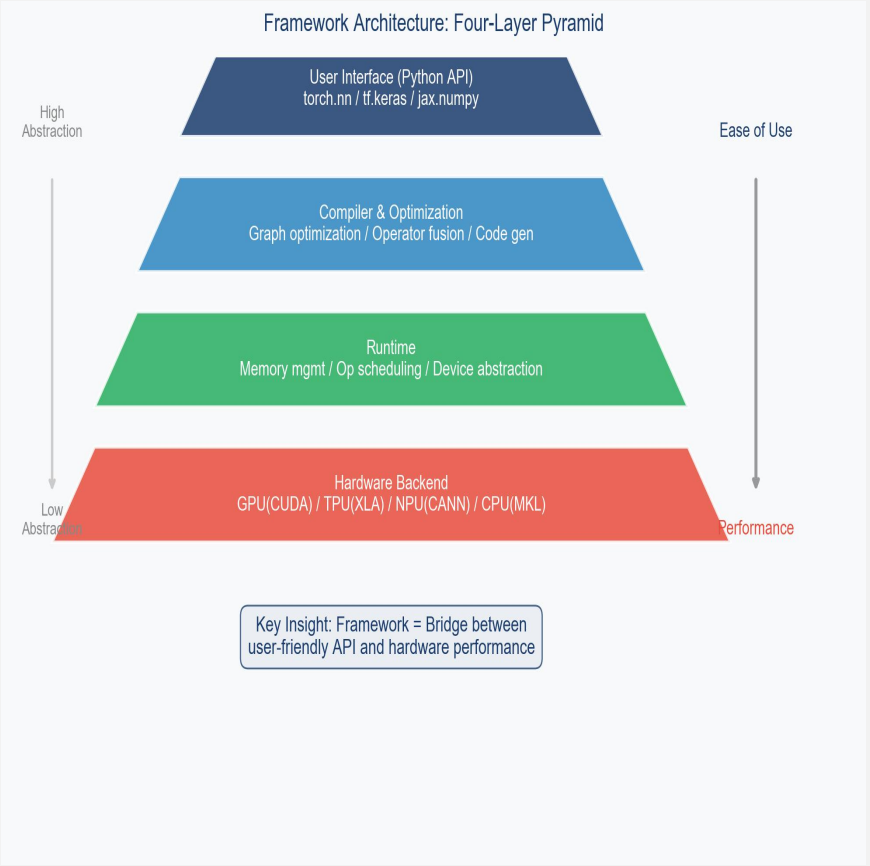
系统视角: 内存管理、算子调度、编译优化

关键矛盾: 灵活性 vs 可优化性

解决思路: 渐进式编译(Eager -| Compiled

未来方向: AI编译器自动化优化

算子是框架的原子操作:
MatMul、Conv、Softmax等

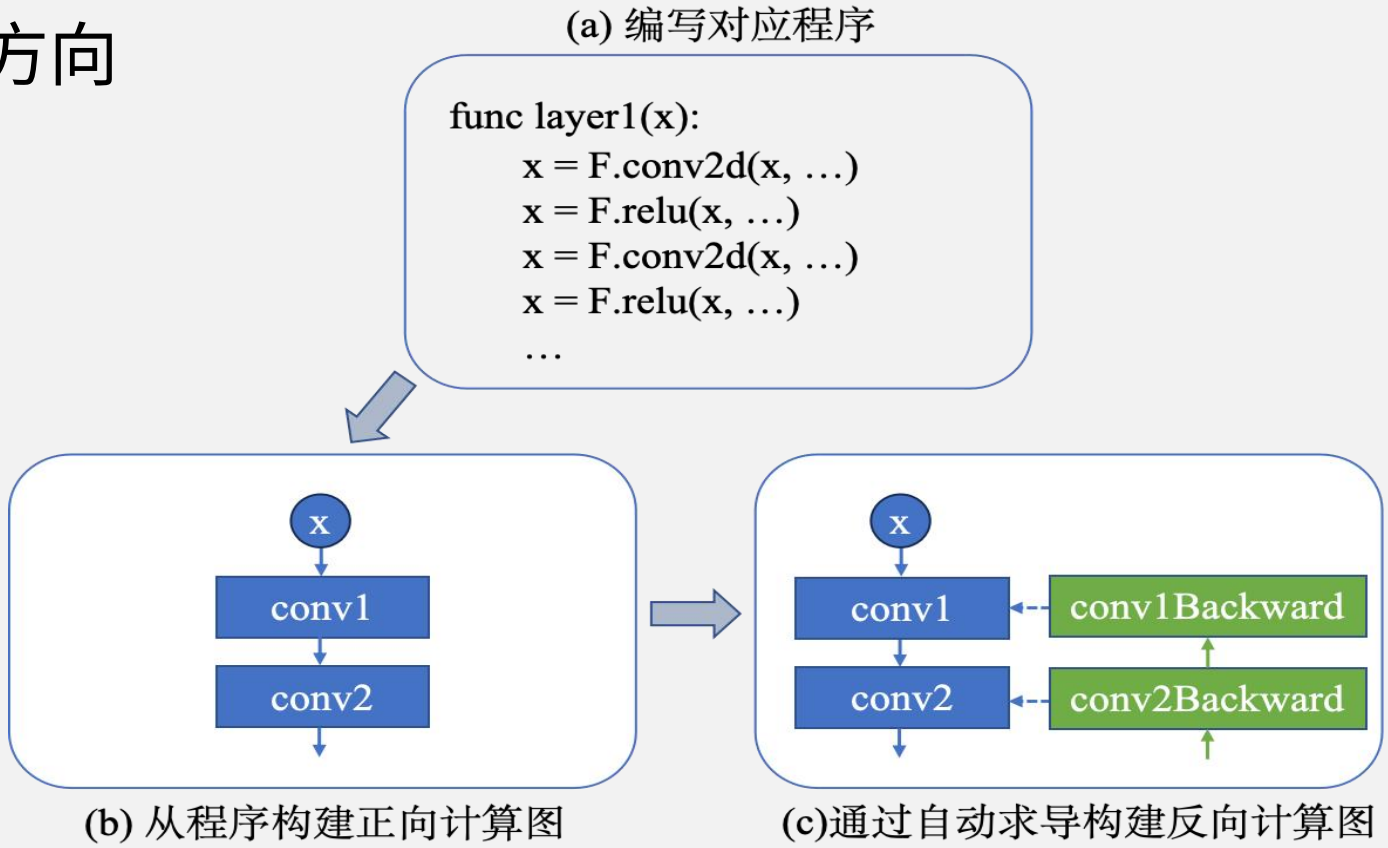


本节课内容

- 机器学习
- 神经网络
- 神经网络训练方法
- 编程框架设计
- **计算图构建**
- 计算图执行
- 本章小结

正向图与反向图构建

- 计算图由两个基本元素构成：张量（Tensor）和张量操作（Operation）。计算图是有向图，有向边指明了张量的流动方向



1、正向传播

- 输入张量经过搭建的神经网络层层计算传递，并最终获得计算结果的过程
- 构建形式
  - **动态图**：在执行函数时，按照函数顺序逐条语句地生成节点，立即计算并返回结果；易调试但性能优化空间有限
  - **静态图**：在执行计算之前构建好所有图上的节点，在图运行时才计算整个计算图并返回最终结果；不易调试但性能好

- 可微分编程：程序本身就是可求导的计算图

动态图

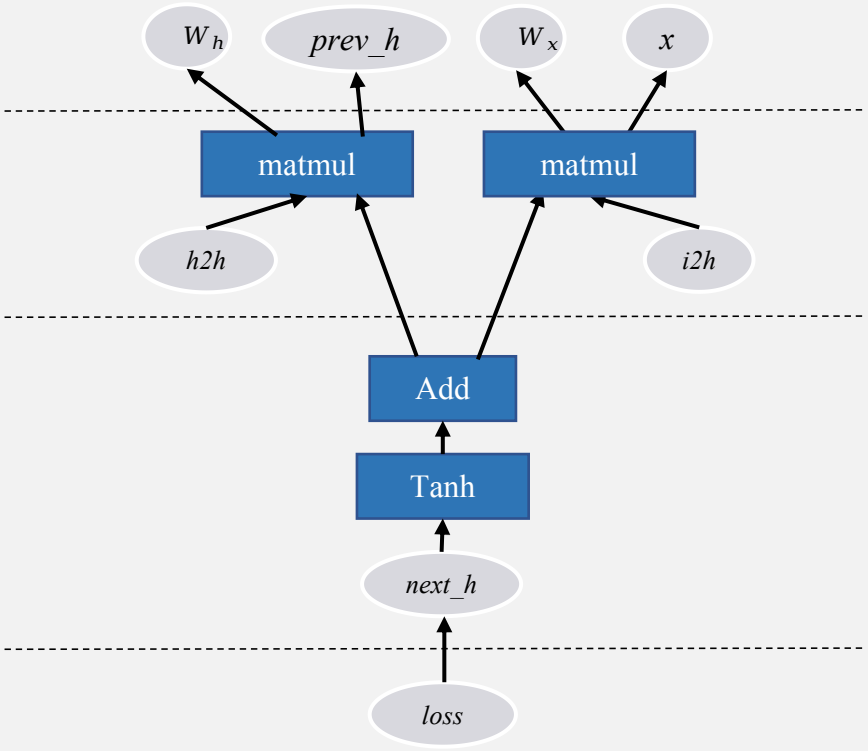
- 计算图在函数运行过程中逐步构建的（On-the-fly）
- 立即（eager）模式：每次调用语句就立刻执行计算
- PyTorch中的动态图实现：每次执行，都会重新被构建

```
W_h = torch.randn(20, 20, requires_grad = True)
W_x = torch.randn(20, 10, requires_grad = True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)

h2h = torch.matmul(W_h, prev_h.t())
i2h = torch.matmul(W_x, x.t())

next_h = h2h + 12h
next_h = next_h.tanh()

loss = next_h.sum()
```



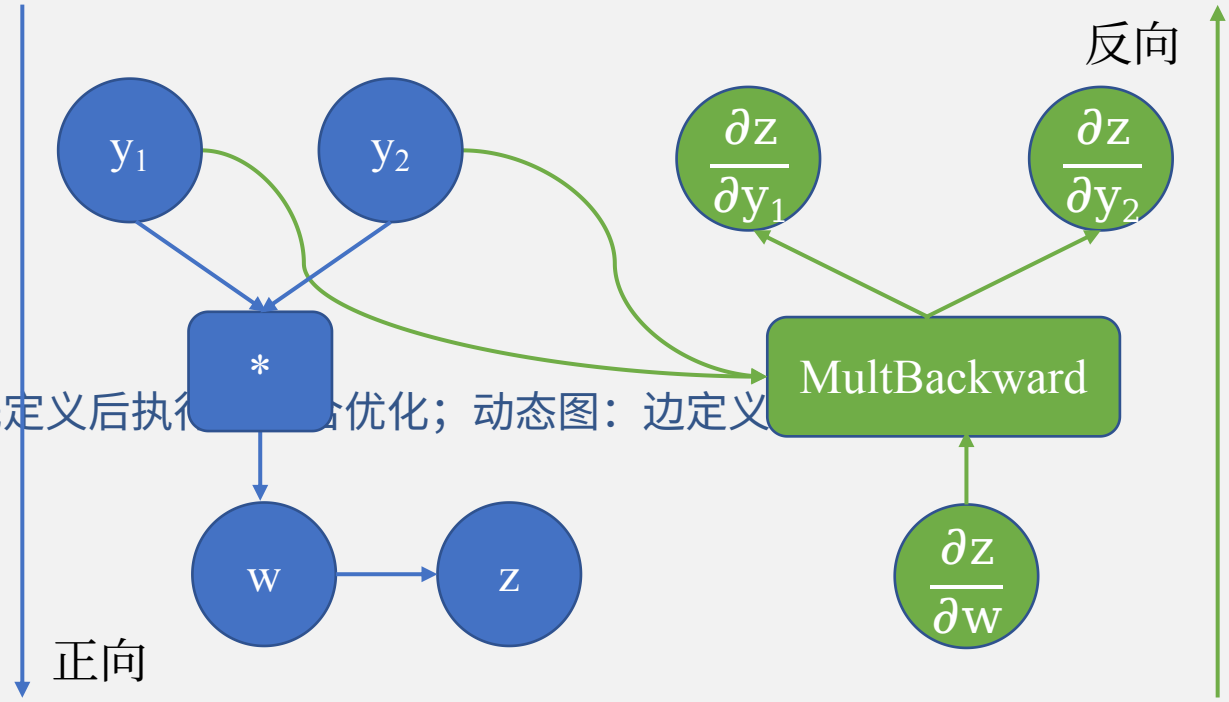
静态图

- 整个网络的结构会在开始计算前就建立完成计算图
- 框架执行时接收整个计算图而不是单一语句
- TensorFlow 1.x中的静态图
  - 使用若干基本控制流算子（Switch、Merge、Enter、Exit和NexIteration）的不同组合来实现各种复杂控制流场景
- PyTorch 2.0中的静态图
  - PyTorch 2.0中采取了图捕获（TorchDynamo）的技术将用户的动态图转化为静态图

- 计算图 = DAG：节点是运算，边是张量数据流

2、反向传播

- 正向计算得到的结果和目标结果存在损失函数值，对其求导得到梯度，并使用该梯度更新参数



- 静态图：先定义后执行，可优化；动态图：边定义

计算导数的方法

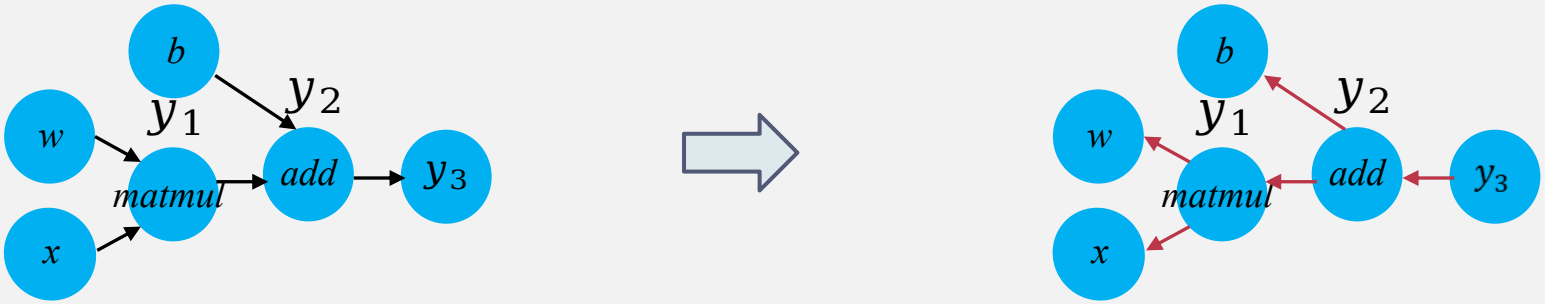
- 自动微分是一种计算导数的方法
- 常见的求导方式
  - **手动求导**：用链式法则求解出梯度公式，然后根据公式编写代码、代入数值计算得到梯度结果
  - **数值求导**：直接代入数值近似求解
  - **符号求导**：直接对代数表达式求解，最后才代入问题数字，出现表达式膨胀问题
  - **自动求导**：用户只需描述前向计算的过程，由编程框架自动推导反向计算图，先建立表达式，再代入数值计算

- 自动微分 ≠ 数值微分 ≠ 符号微分 — 精确且高效

手动求解法

- 手动用链式法则求解出梯度公式，代入数值，得到最终梯度值
- 缺点：
  - 对于大规模的深度学习算法，手动用链式法则进行梯度计算并转换成计算机程序非常困难
  - 需要手动编写梯度求解代码，且模型变化，算法也需要修改

- 前向模式AD：适合少输入多输出；反向模式AD：适合多输入少输出（DL标配）



前向传播　　反向传播

数值求导法

- 利用导数的原始定义求解

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- 优点：
  - 易操作
  - 可对用户隐藏求解过程
- 缺点：
  - 计算量大，求解速度慢
  - 可能引起舍入误差和截断误差

符号求导法

- 利用求导规则来对表达式进行自动操作，从而获得导数
- 常见求导规则：

$$\frac{d}{dx}(f(x)+g(x))=\frac{d}{dx}f(x)+\frac{d}{dx}g(x)$$

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

$$\frac{d}{dx}\frac{f(x)}{g(x)}=\frac{f'(x)g(x)-f(x)g'(x)}{g(x)^2}$$

- 缺点：表达式膨胀问题

n	l_n	$\frac{d}{dx}l_n$ 符号求导结果	$\frac{d}{dx}l_n$ 手动求导结果
1	x	1	1
2	$4x(1-x)$	$4(1-x)-4x$	$4-8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2-16x(1-2x)^2-64x(1-x)(1-2x)$	$16(1-10x+24x^2-16x^3)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$

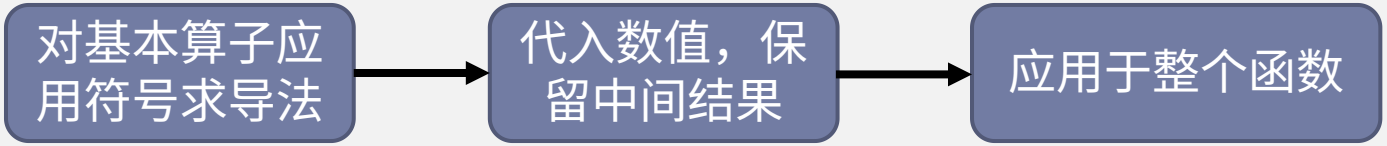
PyTorch Autograd：Tensor记录操作历史 → backward()自动求导

表达式膨胀示例

自动求导法

- 数值求导法：直接代入数值近似求解
- 符号求导法：直接对代数表达式求解，最后才代入问题数字
- 自动求导法：介于数值求导和符号求导的方法

- Autograd实战：requires_grad=True → forward → loss.backward()



JAX vs PyTorch: 自动微分实现对比

PyTorch: define-by-run动态图

JAX: 函数式变换 + JIT编译

PyTorch: loss.backward( )触发梯度

JAX: jax.grad(fn)返回梯度函数

JAX支持高阶导数和Hessian

两种范式的适用场景差异

计算图中间节点的梯度默认不保留 → 用retain_grad( )显式保存

- 计算分两步执行：
  - 1）原始函数建立计算图，数据正向传播，计算出中间节点x_i，并记录计算图中的节点依赖关系
  - 2）反向遍历计算图，计算输出对于每个节点的导数

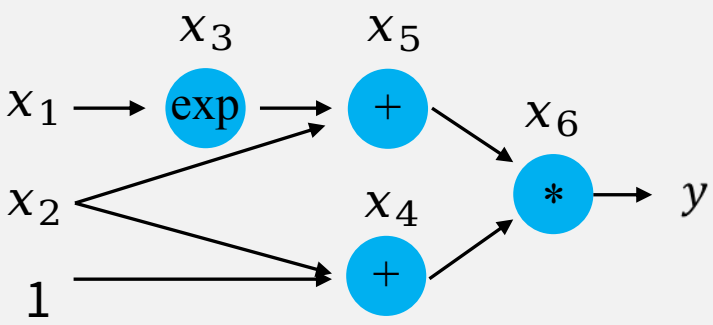
$$\overline{x}_i = \frac{\partial y_j}{\partial x_i}$$

- 对于前向计算中一个数据(x_i)连接多个输出数据(y_j、y_k)的情况，自动求导中，将这些输出数据相对于该数据的导数累加

$$\overline{x}_i = \overline{y}_j \frac{\partial y_j}{\partial x_i} + \overline{y}_k \frac{\partial y_k}{\partial x_i}$$

高阶导数：create_graph=True 让梯度本身也可求导

示例---$f(x_1, x_2) = (e^{x_1} + x_2)(x_2 + 1)$

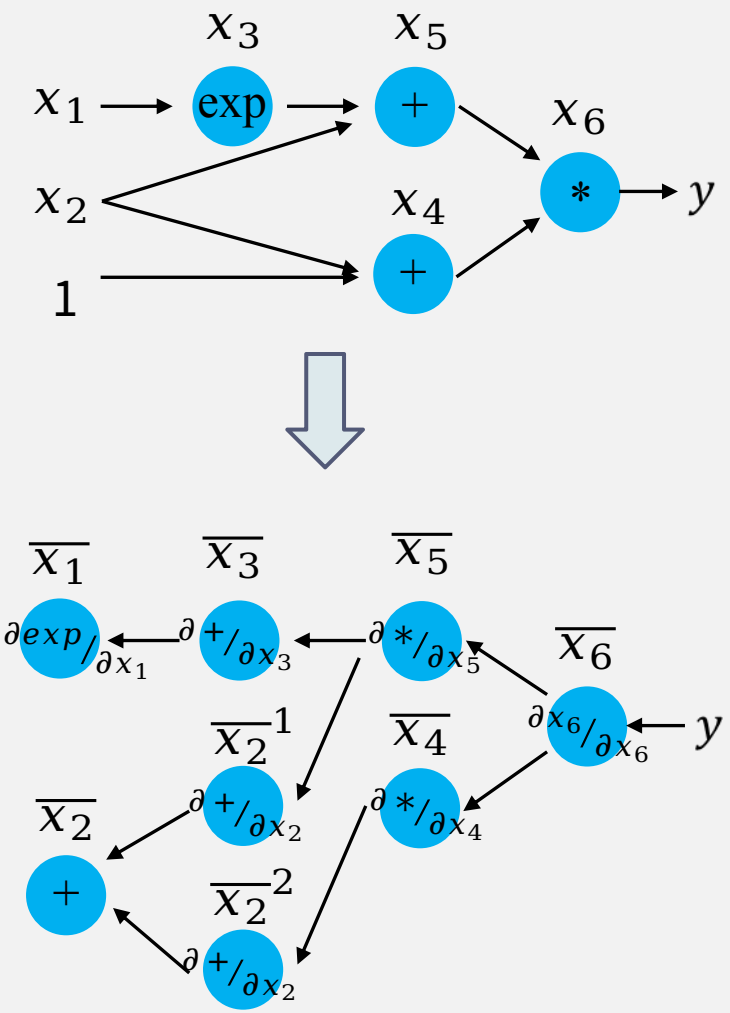


$x_1 = 3$
$x_2 = 2$
$x_3 = e^{x_1} = 20.086$
$x_5 = x_3 + x_2 = 22.086$
$x_4 = x_2 + 1 = 3$
$x_6 = x_4 * x_5 = 66.258$
$y = x_6 = 66.258$

前向计算

自定义算子：继承Function，实现forward/backward

示例---$f(x_1, x_2) = (e^{x_1} + x_2)(x_2 + 1)$



$$\overline{x_2} = \overline{x_2}^1 + \overline{x_2}^2 = 25.086$$

$$\overline{x_1} = \overline{x_3} * \frac{\partial x_3}{\partial x_1} = \overline{x_3} * x_3 = 60.258$$

$$\overline{x_2}^2 = \overline{x_4} * \frac{\partial x_4}{\partial x_2} = \overline{x_4} * 1 = 22.086$$

$$\overline{x_2}^1 = \overline{x_5} * \frac{\partial x_5}{\partial x_2} = \overline{x_5} * 1 = 3$$

$$\overline{x_3} = \overline{x_5} * \frac{\partial x_5}{\partial x_3} = \overline{x_5} * 1 = 3$$

$$\overline{x_4} = \overline{x_6} * \frac{\partial x_6}{\partial x_4} = \overline{x_6} * x_5 = 22.086$$

$$\overline{x_5} = \overline{x_6} * \frac{\partial x_6}{\partial x_5} = \overline{x_6} * x_4 = 3$$

$$\overline{x_6} = \frac{\partial y}{\partial x_6} = 1$$

反向计算

求导方式对比

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

静态图编译优化：算子融合、常量折叠、内存复用

PyTorch中的自动求导

- AutoGrad是PyTorch的自动微分引擎，用户只需要一行代码tensor.backward()，即可调用其自动计算梯度并反向传播
- AutoGrad模块的backward函数实现
  - 1）正向图解析
  - 2）构建反向计算图的节点
  - 3）进行反向梯度传播

PyTorch中的自动求导

AutoGrad模块的backward函数实现

1）正向图解析

```
// 创建根节点和梯度的列表，并预分配num_tensors大小的空间
std::vector<Edge> roots; // 反向传播根节点集合
variable_list grads; // 反向传播的梯度集合

for (int i = 0; i < num_tensors; i++) {
  // 获取正向图的输出张量
  at::Tensor tensor = py::handle(PyTuple_GET_ITEM(tensors, i)).cast<at::Tensor>();
  // 获取从梯度函数指向输出结果的边
  auto gradient_edge = torch::autograd::impl::gradient_edge(tensor);
  roots.push_back(std::move(gradient_edge)); //将获取的边加入到根节点集合中
  at::Tensor grad_tensor = py::handle(PyTuple_GET_ITEM(grad_tensors,
i)).cast<at::Tensor>();
  auto grad_var = torch::autograd::make_variable(grad_tensor);
  grads.push_back(grad_var); // 将梯度变量加入到梯度集合中
}
```

PyTorch中的自动求导

- AutoGrad模块的backward函数实现
  - 2）构建反向计算图的节点

```
std::vector<Edge> output_edges;//反向计算图中所有边的集合
if (inputs != nullptr) {
   for (int i = 0; i < num_inputs; ++i) { // 初始化列表
      ...
      const auto output_nr = tensor.output_nr();
      auto grad_fn = tensor.grad_fn();
      if (!grad_fn) { // 没梯度函数，则标记是叶子节点
      output_edges.emplace_back(std::make_shared<Identity>(), 0);
      } else { // 有梯度函数，创建梯度函数指向该节点的边（构造反向计算图）
         output_edges.emplace_back(grad_fn, output_nr);
      }
   }
}
```

JAX = 函数式自动微分 + XLA编译 + vmap向量化

PyTorch中的自动求导

- AutoGrad模块的backward函数实现
  - 3）进行反向梯度传播

```
// 反向计算图已经构建完成，可进行执行
// roots中包含了反向传播根节点
// grads中包含了反向传播产生的梯度，output_edges中是构建的反向计算图的边
variable_list outputs;
{
  pybind11::gil_scoped_release no_gil;
  auto& engine = python::PythonEngine::get_python_engine();
  // 进入引擎执行
  outputs = engine.execute(roots, grads, keep_graph, create_graph, accumulate_grad,
                           output_edges);
}
```

可微分编程（Differentiable Programming）

核心思想：程序即可微分函数

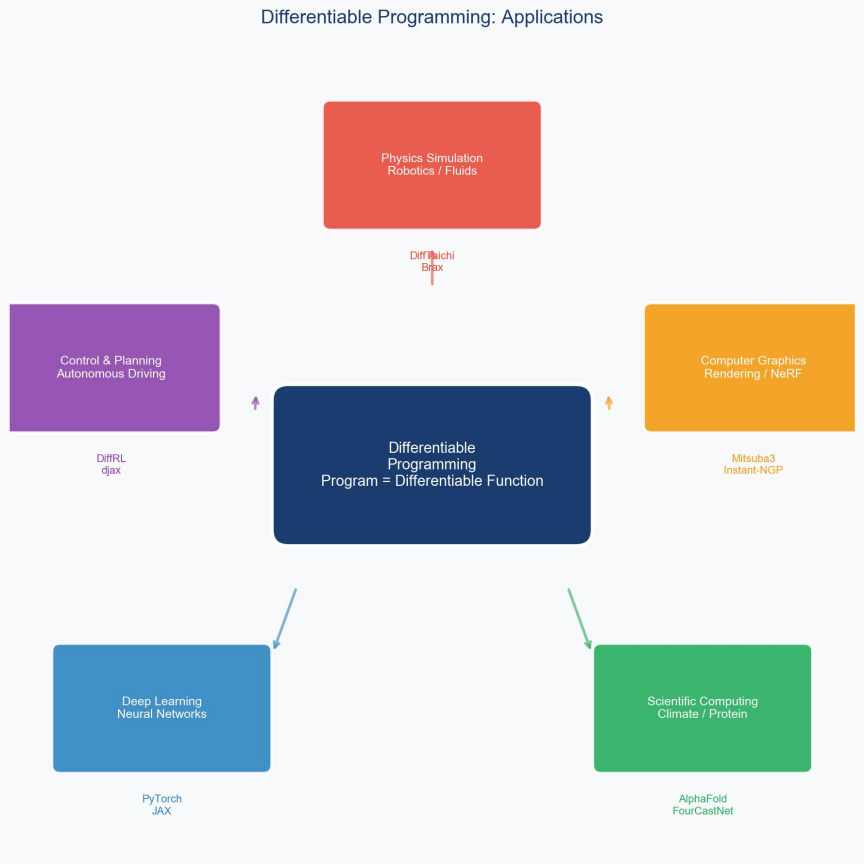
超越深度学习的自动微分应用

应用：物理仿真、渲染、机器人控制

DeepMind《可微分编程基础》450页

代表：JAX生态、DiffTaichi

端到端可微分流水线设计



计算图编译优化

torch.compile: 图捕获+编译优化

TorchDynamo: Python字节码层图捕获

TorchInductor: 算子融合+代码生成

XLA: HLO中间表示+多后端编译

Graph Break: 编译中断与性能影响

AOTAutograd: 提前编译反向图

本节课内容

- 机器学习
- 神经网络
- 神经网络训练方法
- 编程框架设计
- 计算图构建
- **计算图执行**
- 本章小结

计算图执行

- 将计算图中的**张量**和**操作**（本节又称算子）映射到给定**设备**上具体执行
  - 设备管理
  - 张量实现
  - 算子执行
    - 获取算子执行序列
    - 实现算子：前端定义、后端实现、前后端绑定
    - 查找并调用算子

1、在哪里执行？设备管理

- 设备是编程框架中计算图执行时的硬件实体，每个设备都具体负责计算子图中的张量存放和算子运算
- 常见设备包括通用处理器（如CPU）和领域专用处理器（如GPU和DLP等）
- 添加对领域专用处理器的设备管理支持（三个模块）
  - 设备操作
  - 执行流管理
  - 事件管理

PyTorch中的设备类型

- PyTorch中的设备被直接按照类型分类，例如CPU, CUDA, DLP等
- 通过索引表示特定设备，设备索引唯一，在有多个特定类型的设备时标识特定的计算设备

```
DeviceType parse_type(const std::string& device_string) {
  static const std::array<
      std::pair<const char*, DeviceType>,
      static_cast<size_t>(DeviceType::COMPILE_TIME_MAX_DEVICE_TYPES)>
      types = {{
          {"cpu", DeviceType::CPU},
          {"cuda", DeviceType::CUDA},
          {"ipu", DeviceType::IPU},
          {"xpu", DeviceType::XPU},
          {"mkldnn", DeviceType::MKLDNN},
          {"opengl", DeviceType::OPENGL},
          {"opencl", DeviceType::OPENCL},
          {"ideep", DeviceType::IDEEP},
          {"hip", DeviceType::HIP},
          {"ve", DeviceType::VE},
          {"fpga", DeviceType::FPGA},
          {"ort", DeviceType::ORT},
          {"xla", DeviceType::XLA},
          {"lazy", DeviceType::Lazy},
          {"vulkan", DeviceType::Vulkan},
          {"mps", DeviceType::MPS},
          {"meta", DeviceType::Meta},
          {"hpu", DeviceType::HPU},
          {"mtia", DeviceType::MTIA},
          {"privateuseone", DeviceType::PrivateUse1},
      }};
```

设备管理

- 在pytorch/c10/core/impl/DeviceGuardImplInterface.h中定义了抽象的设备管理类DeviceGuardImplInterface
- 设备操作
  - 初始化设备运行环境、获取设备句柄和关闭并释放设备等
- 执行流管理：设备上抽象出来的管理计算任务的软件概念
  - 在异构编程模型下，完成设备上任务执行的下发和同步操作
  - 执行流创建、执行流同步和执行流销毁等
- 事件管理
  - 表示设备上任务运行的状态和进展
  - 事件创建、事件记录和事件销毁等基本操作

在哪里执行？设备管理

```
struct C10_API DeviceGuardImplInterface {
  ...
// 设备相关函数 （设备是计算资源的抽象，可以是CPU、GPU、DLP等）
  virtual DeviceType type() const = 0; // 设备类型定义
  virtual Device exchangeDevice(Device) const = 0; // 将当前设备设置为指定设备，并返回之前的设备
  virtual Device getDevice() const = 0; // 获得当前设备标识符
  virtual void setDevice(Device) const = 0; // 设置当前上下文所使用的设备，所有被调用设备接口开始之前需要调用本接口
  virtual DeviceIndex deviceCount() const noexcept = 0; // 获取系统内的设备数量

// 执行流相关函数（执行流是设备用于执行任务的队列，用于设备上任务执行的下发和同步操作）
  virtual Stream getStream(Device) const noexcept = 0; // 得到当前设备的执行流，
  virtual Stream getDefaultStream(Device) const {...} // 得到当前设备的默认执行流
  virtual bool queryStream(const Stream& /*stream*/) const {...} // 执行流查询操作，如果异步执行流的任务全部执行完成则返回为真
  virtual void synchronizeStream(const Stream& /*stream*/) {...} // 执行流同步操作，以等到执行流中的计算任务全部结束

// 事件相关函数 （事件是设备在软件层面抽象出的概念，用来监控设备上任务运行的状态和进展）
  void record(void** /*event*/, const Stream& /*stream*/,  // 在给定设备上，把事件加入到执行流中
              const DeviceIndex /*device_index*/, const EventFlag /*flag*/) const override {...}
  void block(void* /*event*/, const Stream& /*stream*/) const override {...} // 事件阻塞操作，用于阻塞主机端线程直到事件被完成
  bool queryEvent(void* /*event*/) const override {...} // 事件查询操作，用于在执行流中查询事件是否已执行完成
  void destroyEvent(void* /*event*/, const DeviceIndex /*device_index*/) const noexcept override {} // 事件销毁操作，回收事件资源

  virtual ~DeviceGuardImplInterface() = default;
};
```

运行时三大任务：设备管理、内存分配、算子调度

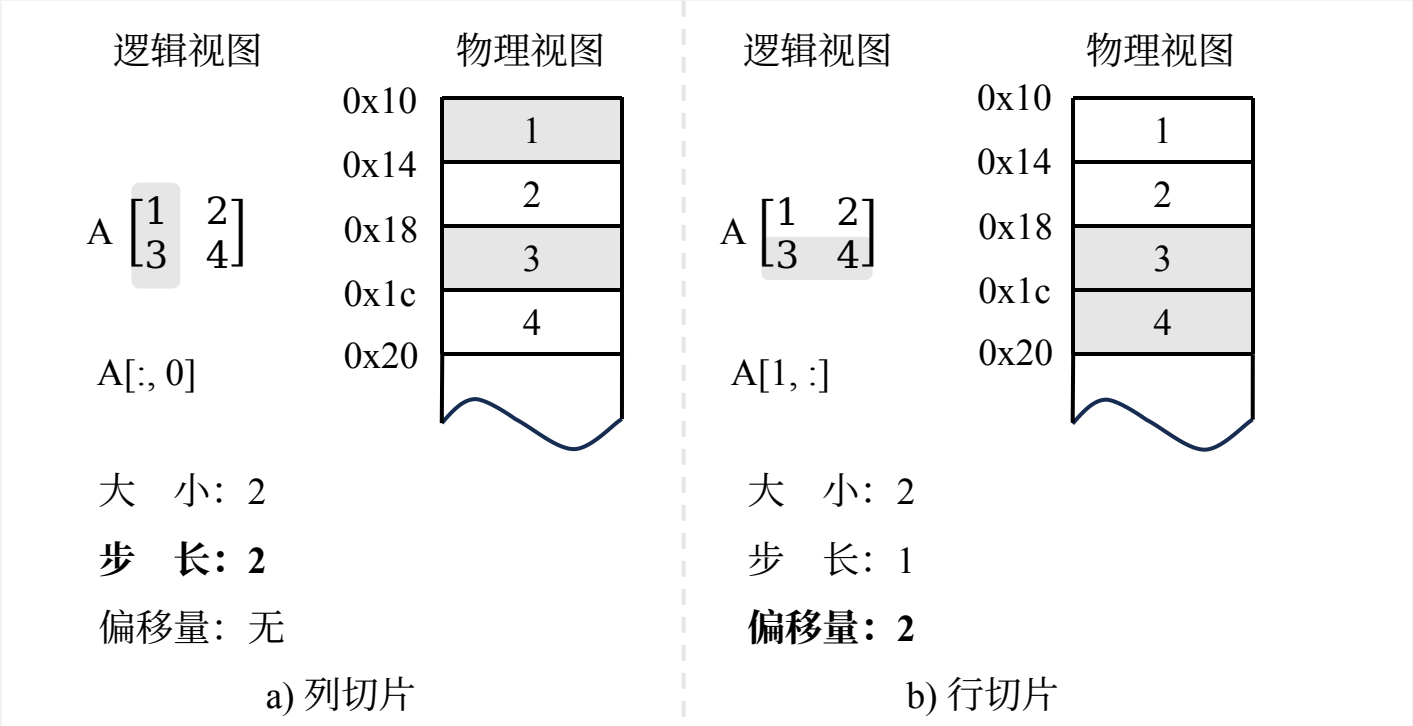
2、执行对象？张量实现

- 逻辑视图：形状、布局、步长、偏移量、数据类型和设备等。是框架使用者能直接控制和表达的基本属性
- 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

	属性名	示例	备注
逻辑视图	size	(D, H, W)	维度
	stride	$(1, D, D * H)$	步长
	offset	0	存储位置的偏移
	datatype	float	数据类型
	device	cpu	设备类型
	layout	”NHWC”	布局信息
物理视图	data_ptr	(cpu, 0x1234, ...)	存储在设备内存上的地址
	size	$D * H * W$	存储长度
	datatype	float	在设备上的实际存储类型

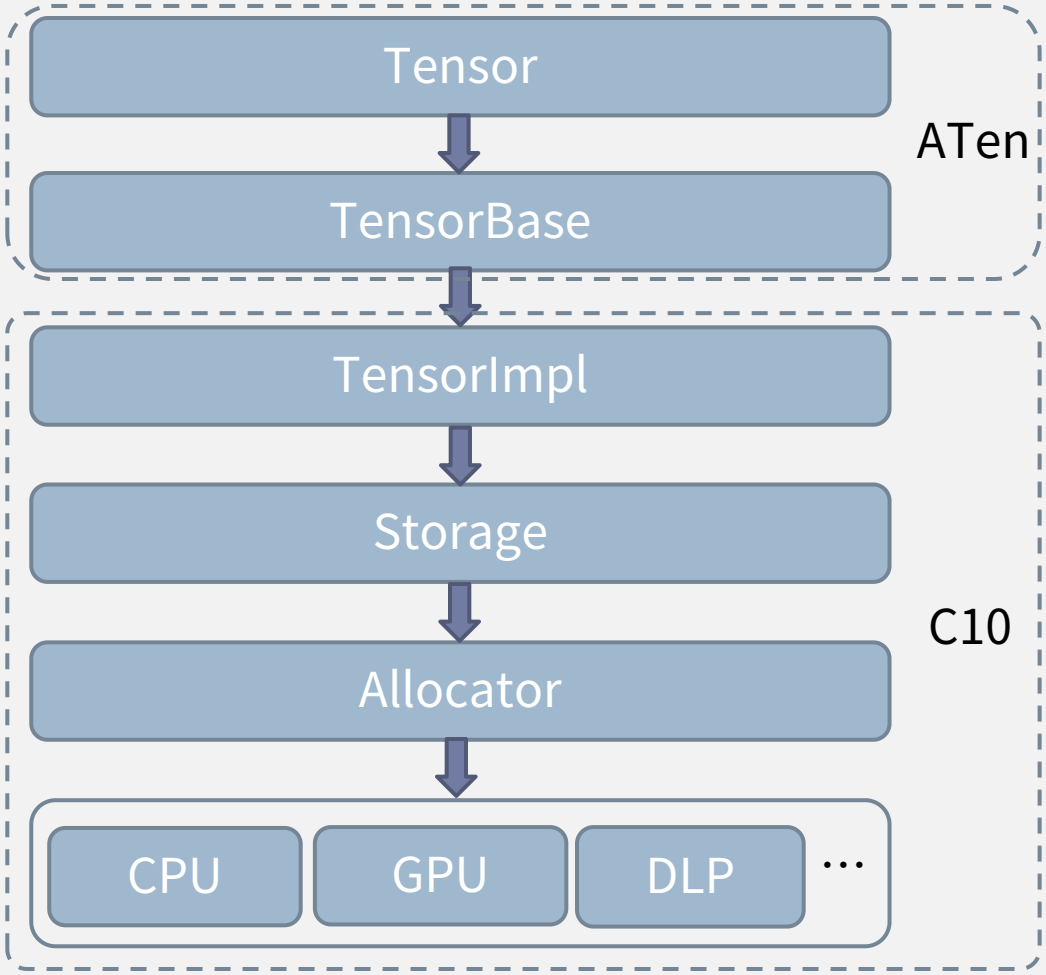
张量数据结构

- A的逻辑视图是一个形状为[2,2]的张量，物理视图是物理地址空间中从0x10位置开始连续存储的一块数据
- 逻辑视图通过偏移量和步长来确定物理视图中物理地址空间的寻址空间
- **一个物理视图可以对应多个逻辑视图**：切片的结果不是新的物理视图，而是原本物理视图下的一个新的逻辑视图



PyTorch中的张量抽象

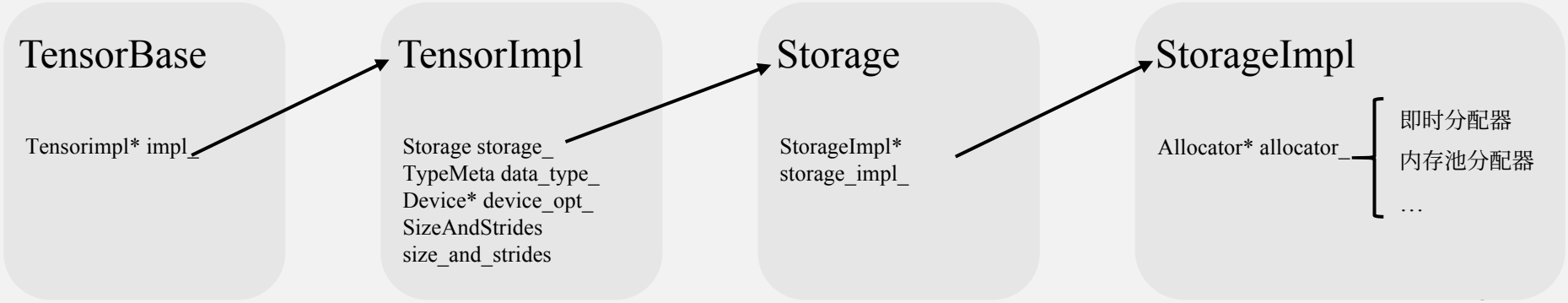
- PyTorch中存在与张量对应的类Tensor
- 持有一个指向底层TensorImpl对象的指针



- CUDA核心 vs Tensor Core：通用计算 vs 矩阵专用加速

PyTorch中的张量抽象

- 通过张量（Tensor）抽象类和存储（Storage）抽象类来分别表示张量数据结构中的逻辑视图和物理视图
  - TensorImpl类：张量抽象的实现，包含了维度信息，步长信息，数据类型，设备，布局等逻辑视角的张量信息
  - StorageImpl类：张量的存储实现，包含了内存指针、数据总数等物理视角的张量信息，调用结构体Allocator进行张量数据空间的分配



张量内存分配

- 从逻辑视图到物理视图的转换需要完成对张量的内存分配，即对张量进行内存管理
- 根据设备的类型不同，张量管理的方式不同
  - 即时分配---CPU
  - 内存池分配---GPU

- 张量 = 多维数组 + 元信息（dtype/shape/stride/device）

张量内存分配--即时分配

- 每当需要分配张量的内存时，就立即从系统中申请一块合适大小的内存空间
- 代码核心部分：malloc()和free()函数

```
struct C10_API DefaultCPUAllocator final : at::Allocator {
  DefaultCPUAllocator() = default;

  // 覆盖基类的分配函数，用于分配n字节的内存空间
  at::DataPtr allocate(size_t nbytes) const override {
    void* data = nullptr;
    ...
    data = malloc(nbytes); // 封装的alloc_cpu()函数进行内存申请，我们在这里进行了简化
    ...
    return {data, data, &ReportAndDelete, at::Device(at::DeviceType::CPU)};
  }

  // 内部实现的上报信息及释放内存函数
  static void ReportAndDelete(void* ptr) {
    ...
    free(ptr); // 释放此前分配的内存
  }
  ...
```

张量内存分配--内存池分配

- 预先分配一块固定大小的内存池，然后在需要时从内存池中分配内存
- 自我维护：内存块的拆分和合并
- 优点：节约设备内存使用，减少设备内存碎片化

```
class NativeCachingAllocator : public CUDAAllocator {
 private:
  // 互斥锁，用于多线程时锁定哈希表防止竞争
  std::mutex mutex;

  // 重要：哈希表，用于记录分配的指针
  ska::flat_hash_map<void*, Block*> allocated_blocks;

  // 添加分配的内存块，添加时需要上锁
  void add_allocated_block(Block* block) {
    std::lock_guard<std::mutex> lock(mutex);
    allocated_blocks[block->ptr] = block;
  }

 public:
  // 获取一个已经分配的内存块的指针，可附带执行删除操作
  Block* get_allocated_block(void* ptr, bool remove = false) {
    std::lock_guard<std::mutex> lock(mutex);
    auto it = allocated_blocks.find(ptr);
    ...
    Block* block = it->second;
    if (remove) {
      allocated_blocks.erase(it);
    }
    return block;
  }
  ...
};
```

张量初始化

- 在CPU上创建一个空张量
  - 选择分配器　创建StorageImpl类　创建TensoImpl类

```
TensorBase empty_cpu(IntArrayRef size, ScalarType dtype, bool pin_memory,c10::optional<c10::MemoryFormat> memory_format_opt) {
  auto allocator = GetCPUAllocatorMaybePinned(pin_memory);
  constexpr c10::DispatchKeySet cpu_ks(c10::DispatchKey::CPU);
  return empty_generic(size, allocator, cpu_ks, dtype, memory_format_opt);
}

TensorBase empty_generic(IntArrayRef size, c10::Allocator* allocator, c10::DispatchKeySet ks, ScalarType scalar_type,
                         c10::optional<c10::MemoryFormat> memory_format_opt) {
  return _empty_generic(size, allocator, ks, scalar_type, memory_format_opt);
}

template <typename T>
// 通用分配函数，通过传入的分配器和分派键集合为不同平台分配
TensorBase _empty_generic(ArrayRef<T> size, c10::Allocator* allocator, c10::DispatchKeySet ks, ScalarType scalar_type,
                          c10::optional<c10::MemoryFormat> memory_format_opt) {
  ...
  // 创建StorageImpl实例
  auto storage_impl = c10::make_intrusive<StorageImpl>(c10::StorageImpl::use_byte_size_t(), size_bytes,
                                                       allocator, /*resizeable=*/true);
  // 将StorageImpl的所有权交给新创建的TensorImpl
  auto tensor = detail::make_tensor_base<TensorImpl>(std::move(storage_impl), ks, dtype);
  ... // 设置相关信息
  return tensor;
}
```

FlashAttention: 算子融合的经典案例

标准Attention的内存瓶颈分析

Tiling: 分块计算避免显存爆炸

IO-Awareness: 减少HBM访问次数

FlashAttention-2: 并行化优化

FlashAttention-3: Hopper异步流水线

从O(N^2)显存到O(N)的飞跃

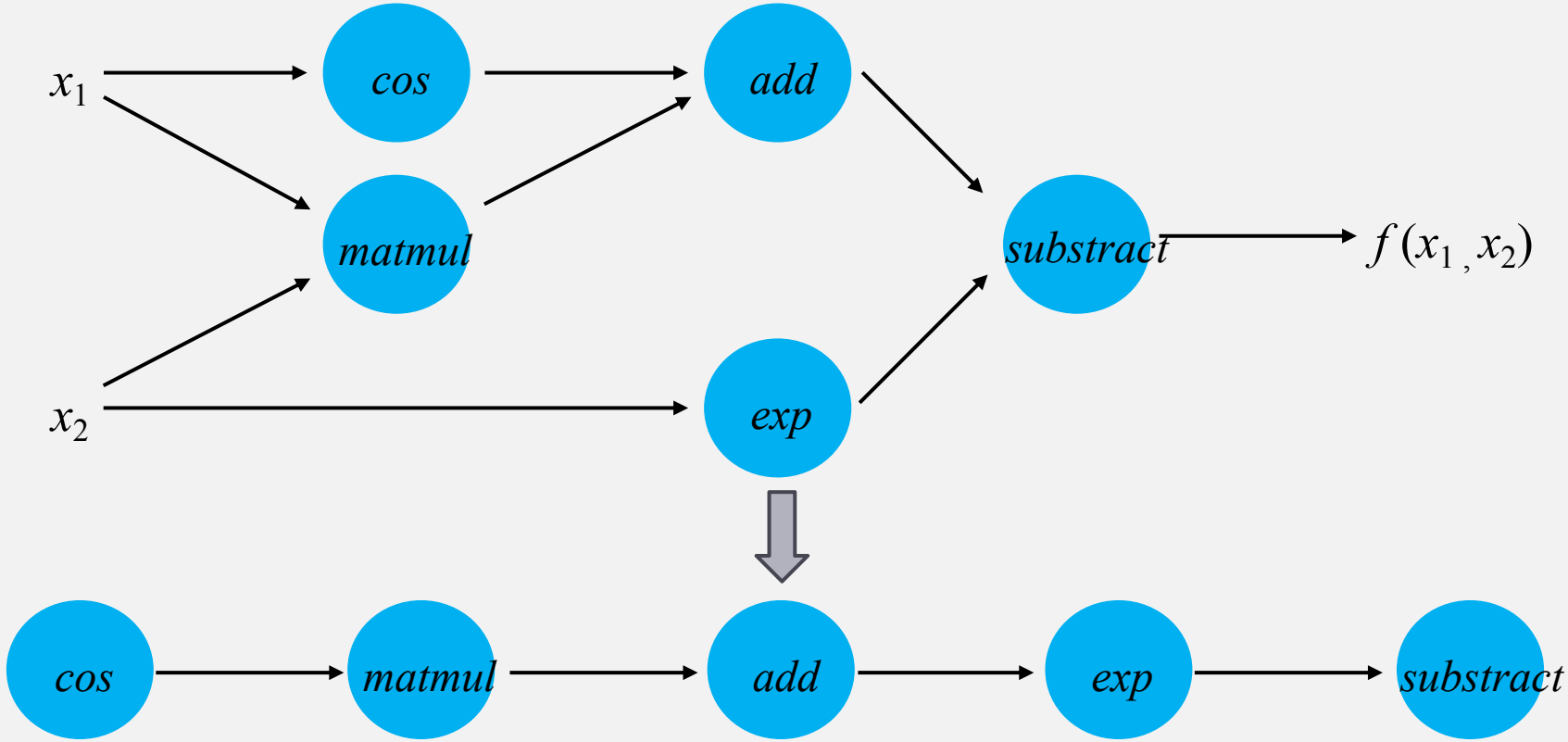
算子分类：逐元素/归约/矩阵运算/数据重排

3、算子执行

- 计算图的执行过程 = 每个算子独立执行的过程
- 计算图　　执行序列（确保正确的数据流和依赖关系）
- 针对每个算子进行算子实现：前端定义、后端实现和前后端绑定
- 分派执行：查找适合给定输入的算子实现，并调用相应的实现来执行具体的计算任务

执行序列

- 分析计算图节点之间的依赖关系　　执行序列
- 拓扑排序算法（可有多种可行的结果）



算子实现

- 正向传播实现和反向传播实现分离
- 用户接口（前端）和具体实现（后端）分离
- 算子实现流程
  - **前端定义**：在编程框架中配置算子信息，包含算子的输入、输出以及相关的接口定义，最后生成前端接口（如Python API）
  - **后端实现**：使用C++或其他高级的编程语言，编写算子的底层实现代码，完成算子的计算逻辑部分实现
  - **前后端绑定**：编程框架将前端定义的算子与后端的具体实现进行绑定

native_function模式

- PyTorch用于管理整个算子实现模块
- 在使用该模式进行算子实现时，需要修改配置文件native_functions.yaml以添加算子配置信息
- native函数格式（位于native_functions.yaml）
  - func字段：定义了算子名称和输入输出的参数类型
  - variants字段：表示需要自动生成的高级方法
  - dispatch字段：表示该算子所支持的后端类型和对应的实现函数

```
- func: func_name(ArgType arg0[=default], ArgType arg1[=default], ...) -> Return
  variants: function, method
  dispatch:
    CPU: func_cpu
    CUDA: func_cuda
```

PReLU算子的native函数

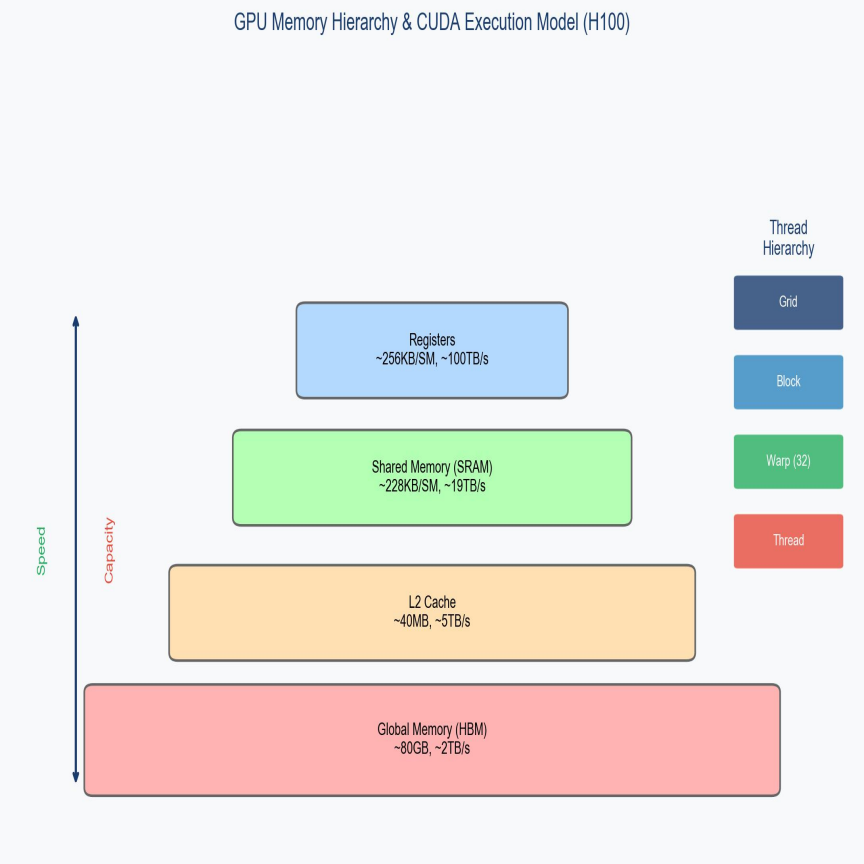
- PReLU算子实现、PReLU正向传播函数实现和PReLU反向传播函数实现

```
- func: prelu(Tensor self, Tensor weight) -> Tensor
  variants: function, method
  autogen: prelu.out

- func: _prelu_kernel(Tensor self, Tensor weight) -> Tensor
  dispatch:
    CPU, CUDA: _prelu_kernel
    QuantizedCPU: _prelu_kernel_quantized_cpu
    MkldnnCPU: mkldnn_prelu
    MPS: prelu_mps

- func: _prelu_kernel_backward(Tensor grad_output, Tensor self, Tensor weight) ->
(Tensor, Tensor)
  dispatch:
    CPU, CUDA: _prelu_kernel_backward
    MkldnnCPU: mkldnn_prelu_backward
    MPS: prelu_backward_mps
```

CUDA编程与GPU执行模型



模型部署：训练完 → 图优化 → 量化 → 推理引擎(TensorRT/ONNX)

前端定义

前端实现代码

```
class PReLU(Module):
    __constants__ = ['num_parameters']
    num_parameters: int

    def __init__(self, num_parameters: int = 1, init: float = 0.25, device=None, dtype=None) -> None:
        factory_kwargs = {'device': device, 'dtype': dtype}
        self.num_parameters = num_parameters
        super().__init__()
        self.weight = Parameter(torch.empty(num_parameters,  **factory_kwargs).fill_(init))

    def forward(self, input: Tensor) -> Tensor:
        return F.prelu(input, self.weight)

    def extra_repr(self) -> str:
        return 'num_parameters={}'.format(self.num_parameters)
```

关系

- name: _prelu_kernel (Tensor self, Tensor weight) -> Tensor
  self, weight: "grad.defined() ? _prelu_kernel_backward (grad, self, weight) : std::tuple<Tensor, Tensor>()"
  result: at::where(self_p >= 0, self_t, weight_p * self_t + weight_t * self_p)

后端实现

- **表层实现**：不同设备之间的抽象函数接口
  - _prelu_kernel()和_prelu_kernel_backward()
  - iter提供了统一的计算抽象，其封装了前向计算的输入input、权重weight，以及反向计算的梯度grad
  - 调用stub函数进行具体的实现

```
Tensor prelu(const Tensor& self, const Tensor& weight_) {
  ...
}

Tensor _prelu_kernel(const Tensor& self, const Tensor& weight) {
  // weight 在 self 上进行广播，并且它们具有相同的数据类型
  auto result = at::empty_like(self);
  auto iter = TensorIteratorConfig().add_output(result).add_input(self).add_input(weight).build();
  prelu_stub(iter.device_type(), iter);
  return result;
}

std::tuple<Tensor, Tensor> _prelu_kernel_backward(const Tensor& grad_out, const Tensor& self, const Tensor& weight) {
  Tensor grad_self = at::empty({0}, self.options());
  Tensor grad_weight = at::empty({0}, weight.options());
  auto iter = TensorIteratorConfig().add_output(grad_self).add_output(grad_weight).add_input(self).add_input(weight).add_input(grad_out).build();
  prelu_backward_stub(iter.device_type(), iter);
  return {grad_self, grad_weight};
}
```

后端实现

- **底层实现**：具体到某个设备上的实际代码实现
  - 正向prelu_kernel() 反向prelu_backward_kernel()
  - 这两个函数都利用了SIMD指令实现向量优化
  - 底层实现中的prelu_kernel和表层实现中的prelu_stub会在前后端绑定中完成对应

```
void prelu_kernel(TensorIterator& iter) {
  AT_DISPATCH_FLOATING_TYPES_AND2(kBFloat16, kHalf, iter.dtype(), "prelu_cpu", [&]() {
    using Vec = Vectorized<scalar_t>;
    cpu_kernel_vec(iter,
    [](scalar_t input, scalar_t weight) {return (input > scalar_t(0)) ? input : weight * input;},
    [](Vec input, Vec weight) {return Vec::blendv(weight * input, input, input > Vec(0));});
  });
}

void prelu_backward_kernel(TensorIterator& iter) {
  AT_DISPATCH_FLOATING_TYPES_AND2(kBFloat16, kHalf, iter.dtype(),  "prelu_backward_cpu", [&]() {
    cpu_kernel_multiple_outputs(iter, [](scalar_t input, scalar_t weight, scalar_t grad) -> std::tuple<scalar_t, scalar_t> {
        auto mask = input > scalar_t{0};
        auto grad_input = mask ? grad : weight * grad;
        auto grad_weight = mask ? scalar_t{0} : input * grad;
        return {grad_input, grad_weight};
      });
  });
}
```

前后端绑定

- 同一个算子可能会有多个后端实现的代码
  - 多种后端 & 多种输入，根据不同情况调用相应的后端实现
  - PyTorch使用**分派机制**来管理前后端对应关系，由Dispatcher管理分派表
  - 分派表的表项记录着算子到具体的后端实现对应关系，纵轴表示PyTorch所支持的算子，横轴表示支持的分派键（与后端相关的标识符）

```
TORCH_LIBRARY_IMPL(aten, CPU, m)
{
  m.impl( “prelu” , cpu_prelu);
}
```

	CPU	GPU	DLP	…
add				
mul				
prelu	cpu_prelu			
…				

分派执行

- 获得算子执行序列　实现对应算子　对算子分派执行
- 分派执行：在运行时根据输入张量的类型和设备类型查找并调用合适的算子实现方法
- Dispatcher计算分派键，并由此找到对应的内核函数
  - 算子：Dispatcher的调度对象，代表了具体的计算任务
  - 分派键：根据输入张量和其他信息计算，可简单地理解为与硬件平台相关联的标识符
  - 内核函数：特定硬件平台上实现算子功能的具体代码

谢谢大家！