北京大学
PEKING UNIVERSITY

基于AI的数字系统设计

第二讲：设计范式演变与AI算法基础

主讲：陶耀宇

2026年秋季

自我加速的飞轮：AI 既是设计对象，也是设计工具

飞轮转起来之后，速度取决于最慢的那一环 —— 今天通常是验证

接下来的五个案例，正是这个飞轮上不同位置的真实证据。

案例一 · AlphaChip：宏单元布局为什么难？

Google DeepMind | Nature 2021

- **任务。**在给定的芯片画布上，为数百至上千个宏单元（SRAM、寄存器堆等）选定位置与朝向，再把确定性算法摆放标准单元。
- **目标。**同时压低线长、布线拥塞与单元密度——三者互相冲突。
- **代价。**一个大模块的平面规划，资深工程师通常要迭代数周到数月。

组合爆炸有多大？

一块的状态空间约 10^{360}；而一块含 1000 个宏单元的芯片，其布局解空间常被估计在 10^{2500} 量级以上。穷举与人工试错都不成立。

真正的困难：评估一次太贵

给出一个布局方案

↓

跑布线与时序分析

↓

得到真实 PPA 与标

单次评估：数小时

- 于是必须学一个便宜的代理奖励，去近似昂贵的真实指标 —— 这是全文方法的出发点。

图示为依据论文思路重绘的示意图

AlphaChip 方法（一）：把布局建模为序贯决策问题

像下棋一样，一次放一个宏单元；棋盘是离散化的芯片画布

依据 Mirhoseini et al., Nature 2021 的方法描述重绘

- **为什么是序贯的。**先放的宏单元会改变后面所有位置的可行性，天然是决策序列而非一次性优化。
- **为什么是稀疏奖励。**只有全部放完才能算出线长与拥塞，中间步骤奖励为零。
- **动作掩码是关键工程。**把重叠、越界、违反约束的网格点直接屏蔽，极大缩小有效动作空间。

与传统方法的差别

- 模拟退火 / 力导向：每块新芯片都从零开始搜索，历史经验不留存。
- 强化学习：策略参数就是经验的载体，见过的芯片越多，起点越好。

AlphaChip 方法（二）：边特征图神经网络与代理奖励

让模型学到“电路连接关系”的通用表示，从而在新芯片上也能泛化

策略 / 价值网络结构（依据论文重绘示意）

- **关键创新在“边”。**把连线本身也作为可学习的对象，模型因此学到的是电路的连接模式，而不是某一块芯片的坐标。
- **于是可以泛化。**在 A 芯片上学到的经验，能迁移到从未见过的 B 芯片 —— 这是预训练能起作用的根本原因。
- **价值网络的作用。**在稀疏奖励下预测“这一步之后大概能得多少分”，为策略提供稳定反馈，否则训练几乎无法收敛。

代理奖励函数

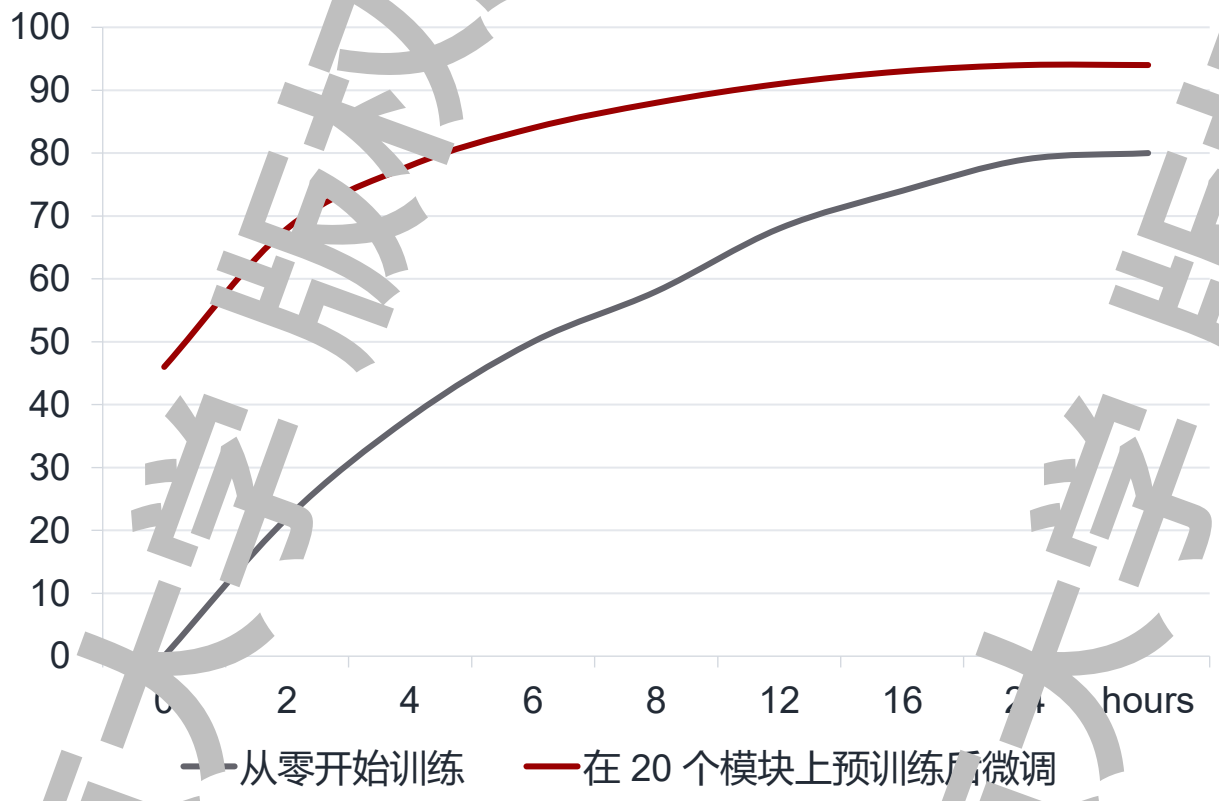
$$R = -(\text{线长} + \lambda \cdot \text{拥塞} + \gamma \cdot \text{密度})$$

- **线长。**用半周长线长（HPWL）近似，计算便宜。
- **拥塞。**用布线资源使用量的估计值代替真实布线。
- **密度。**约束单元过度堆叠，保证后续可布通。

代理奖励是整套方法的软肋，也是后来争议的焦点：它必须便宜到可以在训练中调用上百万次，又要与真实签核指标足够相关。一旦代理与真实指标脱钩，模型优化的就不再是我们真正想要的东西。

AlphaChip 方法（三）：预训练让它“越用越强

在 20 个历史 TPU 模块上预训练，再在新模块上微调 —— 这是与传统优化最根本的差别



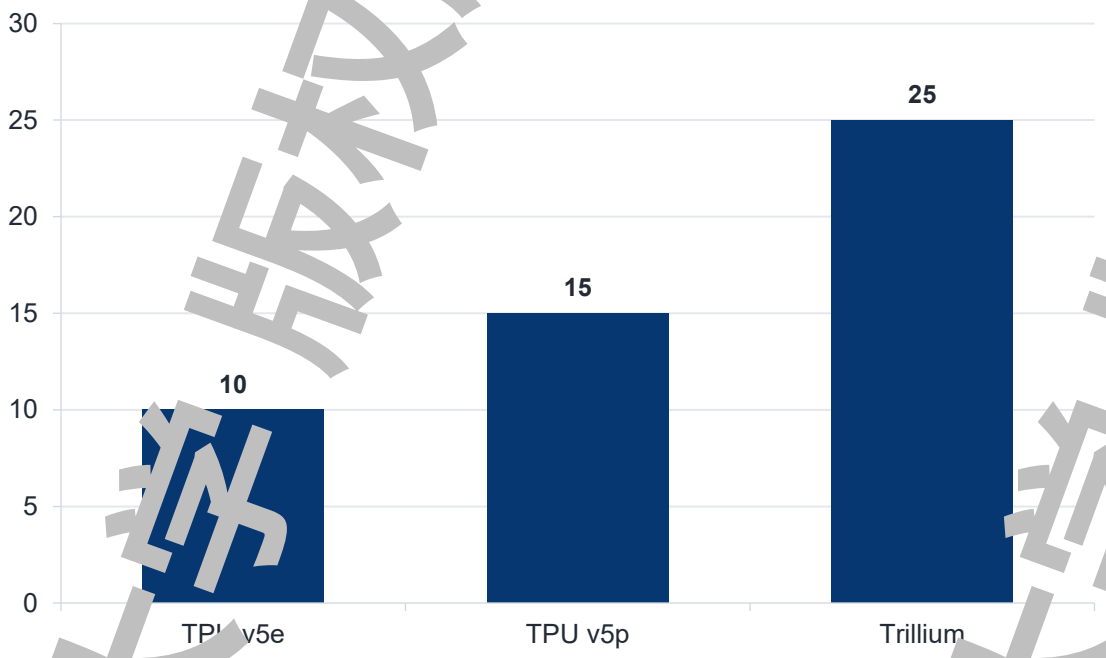
示意曲线：纵轴为布局质量（越高越好），横轴为微调时长；基于论文对预训练收益的描述重绘，非原始数据。

- **样本起点更高。**预训练模型在没见过的新模块上，第一次输出就已接近可用。
- **收敛更快。**达到同等质量所需的微调时间显著缩短。
- **经验会累积。**见过的芯片模块越多，模型越强 —— 这与人类专家的成长方式一致，而确定性算法做不到。
- **已公开预训练权重。**2022 年开源 Circuit Training 代码，2024 年随 Nature 附录发布在 20 个 TPU 模块上预训练的检查点。

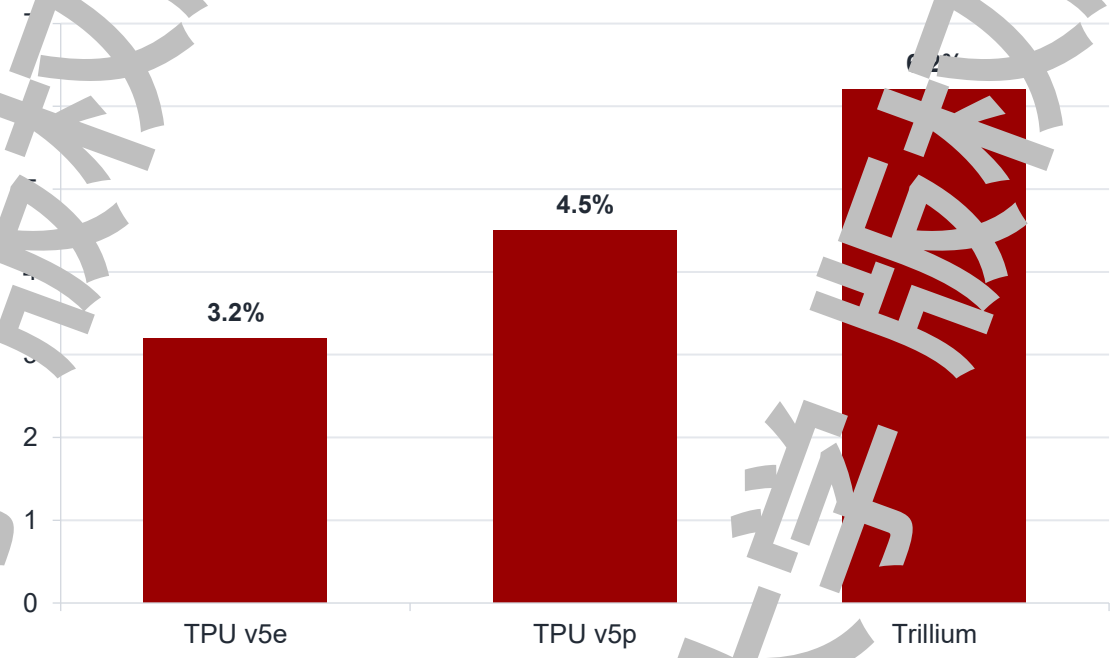
教学要点：“会积累经验”是 AI 方法相对传统 EDA 算法唯一的结构性优势，其他都是量的差别。

AlphaChip 结果：三代 TPU 上的量化收益

官方数据：AlphaChip 布局的模块数与相对人类专家的线长缩短，逐代提升



AlphaChip 布局的模块数量（个）



相对人类专家的平均线长缩短（%）

- **时间：**生成可用平面规划从数周至数月缩短到数小时。
- **扩展：**同一方法已用于 Google Axion（Arm 架构数据中心 CPU）等芯片；联发科亦在其先进制程芯片上做了扩展应用。

数据来源：Google DeepMind 2024 年 Nature 附录及官方博客公布的数值。

思想自由 兼容并包

AlphaChip 的争议

定义了“AI 设计结果如何计算可信”的行业标准

2020.04 — arXiv 预印本发布
《Chip Placement with Deep RL》

2021.06 — Nature 正式发表
同期配发 News & Views 评论

2022 — 开源 Circuit Training
代码公开，但预训练数据未公开

2023.02– — 两项质疑先后发布
Cheng 等（UCSD，Kahng 组）复现失败；
[illegible] 发表《The [illegible]》

2023.09 — Nature 加注编辑说明
称性能主张受到质疑，正在调查；
次日撤下配套评论文章

2024.09 — Nature 附录 + 命名 AlphaChip
公布三代 TPU 数据，公开预训练权重

2024.11 — 《[illegible] That Chip [illegible]ed》
Google 逐条回应质疑，
指出复现方未做预训练、算力不足

争议的三个技术焦点

- 对照基准是谁：与“某位未具名的人类专家”比，还是与模拟退火、商用工具在公开基准上比？
- 手动处理是否被计入：论文未明确说明布局前[illegible]是否用商用 EDA 工具做过准备；复现方未做预训练、算力也远低于原作
- ，双方对“是否复现了原方法”各执一词。

AI参与设计的评价标准

把它当成一份 AI 设计成果的验收清单

1 对照基准必须公开可比

不能只与一个“专家”比。要与模拟退火、商用工具、公开基准（如 ISPD、MacroPlacement）同台。

2 完整流程必须披露

预处理、后处理、使用了哪些商用工具，都会影响结果归属，必须写清楚。

3 算力与超参必须报告

同一算法在不同算力预算下差异巨大；不报告算力的对比没有意义。

4 随机性必须被固定

报告多次运行的均值与方差，公开随机种子，而不是只报最好的一次。

5 数据与权重尽量开放

能否复现，最终取决于别人能否拿到同样的起点。

6 结论要与代理指标脱钩

代理奖励好看不等于签核指标好看；最终结论必须落到真实 PPA 上。

案例二 · PRIME：用离线数据设计 AI 加速器架构

Google | ICLR 2022 | 从“用 AI 优化布局”上升到“用 AI 决定架构参数”

- **问题。**加速器架构有大量离散参数：PE 阵列规模、片上存储容量、带宽分配、数据流方式……组合空间巨大。
- **传统做法。**仿真驱动搜索：每评估一个配置就跑一次周期精确仿真，慢且每换一个应用就要重来。
- **PRIME 的做法。**只用历史积累的日志数据（含大量不可行配置），离线学一个保守的代理模型，再对代理模型做优化，全程不再调用仿真器。
- **“用不可行数据”是关键。**失败的设计同样携带信息，让代理模型知道哪里是悬崖。
- **代价大幅下降。**论文报告：相对最优仿真驱动方法，性能提升约 1.54×，同时总仿真时间减少 93%–99%。

依据 Kumar et al., ICLR 2022 论文流程重绘

自我加速的飞轮：AI既是设计对象，也是设计工具

- 设计AI芯片架构 -> 利用AI设计AI芯片架构

参数化硬件单元库　　针对某类任务的最优芯片设计

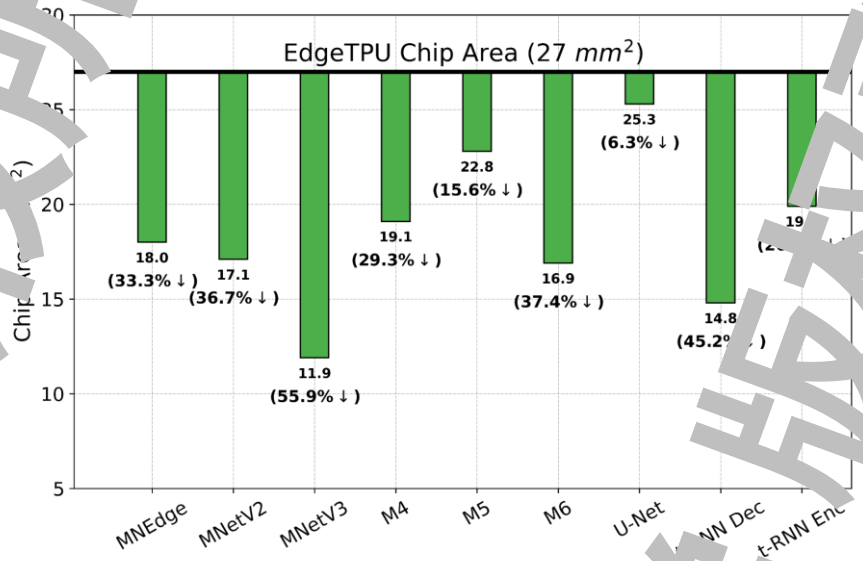
PRIME

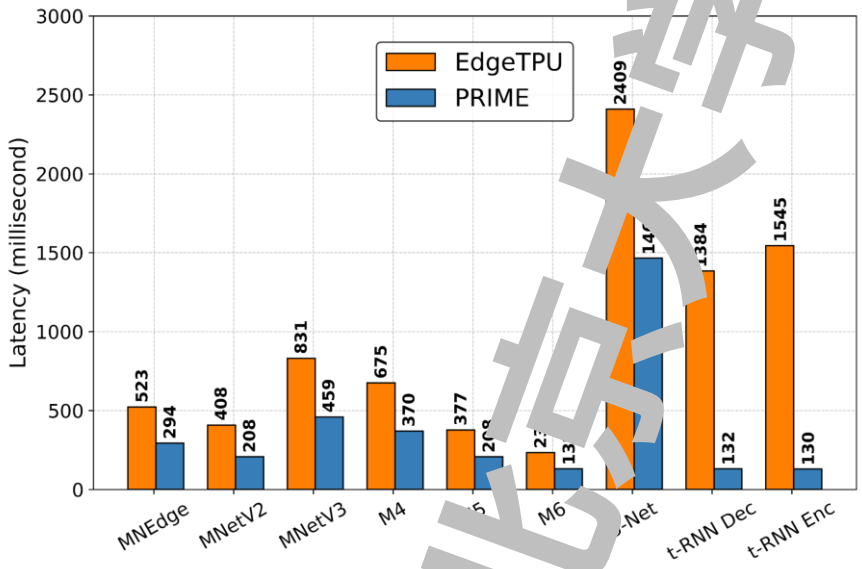
Optimizer

RL、大模型等方式

设计AI芯片的AI模型

AI芯片

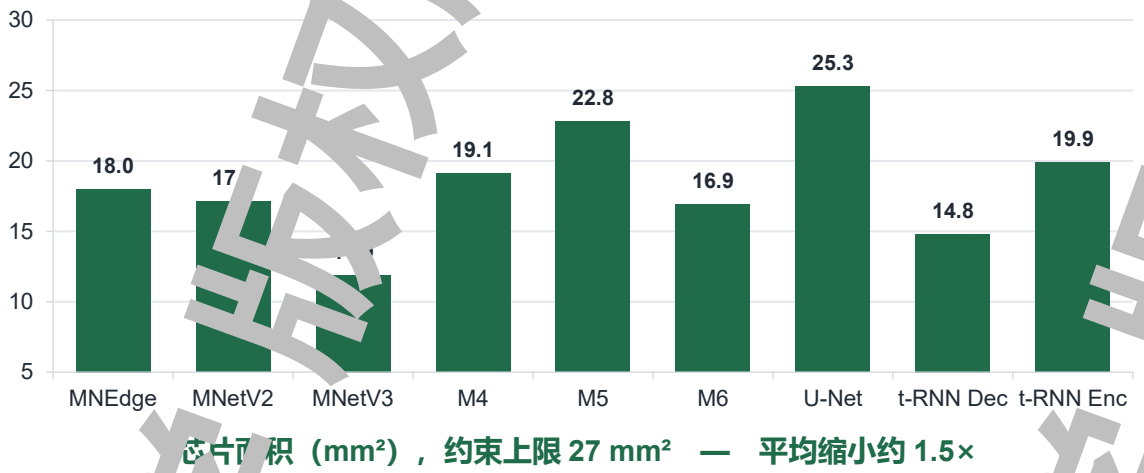




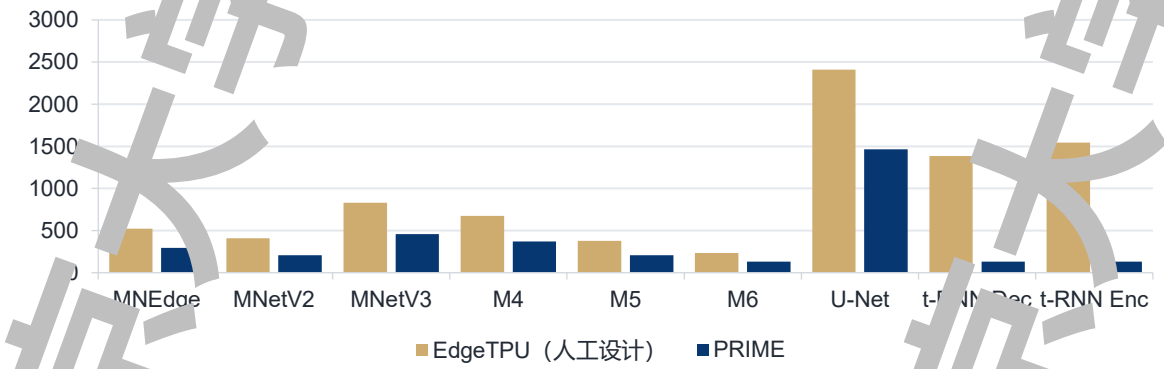
思想自由 兼容并包

PRIME 结果：比人工定制的 EdgeTPU 更小、更快

在 27 mm² 面积约束下，针对九个模型分别做专用化设计



芯片面积（mm²），约束上限 27 mm² — 平均缩小约 1.5×



延迟（毫秒） — 平均加速约 2.69×，最高 11.84×（t-RNN Enc）

两个值得注意的细节

- 面积不是优化目标，却顺带下降了 —— 说明代理模型学到了参数之间的真实权衡。
- t-RNN 上的巨大加速来自结构专用化，而不是简单地堆资源。

- **对照要看清楚。** PRIME 是为每个应用单独定制，EdgeTPU 是一颗通用芯片，这个对比展示的是“专用化的潜力”，不能读成“AI 全面超越人类设计”。
- **方法论价值。** 它证明了历史日志数据可以直接变成设计能力 —— 数据本身就是资产。

数据来源：PRIME 论文附录 Table 14（27 mm² 面积约束，25 GBps DRAM 带宽）

案例三 · ChipNeMo：芯片设计的领域大模型

NVIDIA | 2023 | 问题不是“大模型行不行”，而是“通用大模型为什么不够”

通用大模型在芯片设计场景的三个短板

- **专有语言与内部工具。**公司内部的脚本语言、工具命令、库函数在公开语料中几乎不存在，模型没见过。
- **术语被错误分词。**通用分词器会把电路术语、信号名切得支离破碎，浪费上下文且损失语义。
- **知识不在参数里。**设计规范、方法学文档是内部资产，模型无从知晓，只能靠检索补进来。
- **成本敏感。**内部部署要考虑推理成本，不可能对每个工程师都挂一个 70B 模型。

训练语料：全部来自内部资产

- 内部设计源码（RTL、C/C++、脚本）
- 验证代码与测试平台
- 架构文档与设计规范
- Bug 记录与工程师讨论
- 内部 Wiki、工具手册与 FAQ

按比例混入公开数据，抑制通用能力遗忘

语料类型依据 ChipNeMo 论文描述整理；具体规模以论文为准

ChipNeMo 方法：四步领域适配流水线

不是简单微调 —— 四项技术分别解决分词、知识、指令与检索四个问题

① 定制分词器

在通用分词器上扩充
芯片领域词表
复用原有嵌入初始化

② 领域自适应
继续预训练 DAPT

在 LLaMA2 7B/13B 上
用内部语料继续预训练

③ 领域指令微调
SFT

用领域指令数据 +
通用指令数据混合微调

④ 领域适配检索器
生成模型 RAG

微调检索器
显著提升命中率

解决什么	术语分词效率	领域知识注入	任务对齐	实时知识接入
代价	需重训嵌入层	算力开销最大	需人工构造指令	需构建知识库
可迁移性	高，任何团队可做	需十亿级 token 语料	中	高，最容易先落地

落地顺序建议：先做 ④ RAG（成本最低、见效最快），再考虑 ②DAPT（成本最高、收益也最大）。

ChipNeMo 发现（一）：小而专，胜过大而全

领域适配后的 13B 模型，在多数设计任务上追平甚至超过 70B 通用模型

最多 5×

在效果相当或更优的前提下，模型规模可以缩小的倍数

- **部署门槛。** 13B 模型无需量化即可装入单张 A100；70B 模型做不到，这直接决定了能不能给全公司工程师用。
- **成本结构。** 论文引用的估计是：同等延迟目标下，8B 模型的推理成本比 62B 低 8–12 倍。
- **适用边界。** 领域适配的收益，集中在公开语料稀缺的任务上，通用任务上通用大模型仍占优。

什么时候值得做领域适配？

语料规模

内部语料要达到十亿级 token 量级，否则 DAPT 收益有限

任务独特性

越是专有语言、内部库、私有方法学，收益越大

使用频次

高频、全员使用的场景才摊得平前期训练成本

评测能力

必须先有可信的领域评测集，否则无法判断是不是变好

四条都满足，才建议自建领域模型；否则先用 RAG。

ChipNeMo 发现（二）：检索器也需要领域适配

RAG 的瓶颈往往不在生成模型，而在“有没有把对的文档取回来”

工程师提问 → **检索器 召回相关文档** → 拼接上下文 → 模型生成回答

真正的瓶颈在这一步

- **通用检索器不懂芯路术语。**同义词、缩写、内部代号在通用嵌入空间里距离很远，召回率因此偏低。
- **用领域数据微调检索器。**论文报告：检索命中率显著提升，并直接带来生成质量的进一步改善。
- **检索质量决定回答上限。**取回错误文档时，模型会用一本正经流畅的语言给出错误答案 —— 这比不回答更危险。

课程实验里可以直接复用的做法

- 把设计规范、方法学文档、工具手册切块入库，先跑通一个最小 RAG。
- 构造 30–50 条带标准答案的问题，作为固定评测集。
- 对比“有无检索”“检索器是否微调”三种配置，量化每一步的增量。

结论：先把检索对，再考虑把模型做大。

ChipNeMo 三大应用与评测结果

三个场景覆盖了“问、写、读”——正是工程师日常最耗时的三件事

1 工程助手问答

回答设计规范、方法学与内部工具相关问题

评分 7.4 / 10

2 EDA 脚本生成

生成内部工具与专有语言的脚本，通用模型最吃力的场景

正确率 > 50%

3 Bug 摘要与分析

把冗长的缺陷记录与讨论压缩成可读摘要

评分 4.1 / 7

怎么读这三个数字？

- **7.4 / 10 是“可用但需核对”。**足以当查询入口，不足以当权威答案。
- **> 50% 正确率听起来不高，但它对应的是通用模型几乎做不了的任务 —— 参照系变了，结论就变了。**
- **论文自己也强调仍有改进空间。**把它读成“方向被验证”，而不是“问题已解决”。

案例四 · 商用 EDA 中的 AI：从论文走进产线

同样是强化学习，落点从“布局”扩展到“整条流程的参数与策略”

Synopsys DSO.ai

- 用强化学习在综合与布局布线的巨大参数空间中自主搜索 PPA 最优解
- 2021首个客户流片；2023公布“首个 100 例商业流片”里程碑
- 公开客户案例：生产率提升 3 倍以上、总功耗最多降低 25%、芯片面积可观缩小。

Cadence Cerebrus

- 机器学习驱动 RTL-to-signoff 全流程，官方宣称最多 3× 设计生产率与 20% PPA 改善。
- 客户案例：单名工程师 10 天内完成 3.5 GHz 移动 CPU 的实现收敛。
- 同类还有 Siemens 在物理验证与变差分析上的 AI 能力。

商用工具中通用 AI 优化循环（重绘示意）：本质是把工程师的调参经验变成可搜索的策略

读厂商数据时要小心：“最多”“up to”通常是最好情况；缺少统一基准与第三方复现，横向对比几乎不可能。把它当作“方向已被产业验证”的证据，而不是精确的性能承诺。

案例五 · 大模型与 RTL：从能生成到能评测

没有可信评测集，“生成能力”就无法被讨论 —— 这是这一方向的真正起点

VerilogEval（NVIDIA，ICCAD 2023）

- **题库。**取自 HDLBits 教学网站，人工描述版 156 题，机器生成描述版 143 题，覆盖组合逻辑到复杂状态机。
- **判定。**在沙箱中用 iVerilog 仿真，把生成代码的波形与标准答案比对，功能等价才算通过。
- **指标。**pass@k：每题采样 20 次，只要有一次通过即算通过。

圈里的其他基准与方法

- **RTLLM。**50 个更接近真实模块的设计题，考察规模更大的生成任务。
- **RTLCoder / VeriGen 等。**用合成数据做监督微调，让开源小模型在 Verilog 上追赶闭源大模型。
- **合成数据自举。**VerilogEval 论文本身也验证了：用模型生成的“题目—代码”对做微调，可显著提升生成能力。

真正有工程价值的形态：把生成放进闭环

自然语言规格 → LLM 生成 RTL → 自动跑仿真 / 形式检查 → 失败则回灌错误信息 → 重新生成直到通过

关键不在“一次写对”，而在“能不能自动判定对错并迭代”——判定器才是这条链路的核心资产。

案例六 · AI 用于验证：最大的工作量，最保守的采用

验证占整体工时六成以上，却是 AI 渗透最慢的环节 —— 因为责任无法转移

已经在做的

- 用覆盖率反馈引导激励生成，减少无效随机用例
- 从设计文档与代码中挖掘候选断言，交人工确认
- 回归失败的自动聚类与根因摘要，缩短调试时间
- 在大规模回归中预测哪些用例最可能失败，优先跑

仍然做不到的

- 证明“没有遗漏的场景”——覆盖率高不等于验证充分
- 替代形式验证给出的数学保证
- 承担签核结论的责任：出了问题，模型不会被追责
- 在没有黄金参考模型时判断“什么才是对的”

为什么验证是飞轮上最慢的一环？

- 生成任务允许出错——错了重来一次成本很低；验证任务不允许漏判——漏一次就是一次流片失败。
- 所以 AI 在“提出候选”上很有价值，在“下最终结论”上短期内不会被采用。

六个案例的横向对比

看清每个案例站在“流程”的哪个位置、用了哪一类方法

案例	介入环节	方法类型	层次	证据	最大的软肋
AlphaChip	物理设计 · 宏单元布局	强化学习 + 图神经网络	L2	三代 TPU 量产采用	对照基准与可复现性争议
PRIME	架构探索 · 参数选择	离线数据驱动优化	L2	面积与延迟同时改善	对照的是通用[illegible]芯片
ChipNeMo	问答 / 脚本 / 摘要	领域自适应大模型 + RAG	L1	13B 追平 70B	评分仍未达生产级权威
DSO.ai / Cerebrus	综合与 P&R 全流程	强化学习参数搜索	L2	百例级商业流片	厂商数据，缺少第三方复现
VerilogEval 生态	RTL 生成	大模型 + 自动判定闭环	L1→L3	有了可复现的公开基准	真实模块规模仍偏小
AI 辅助验证	功能验证	覆盖率引导 / 断言挖掘	L1	确实减少无效用例	无法[illegible]担签核责任

共同规律：AI 目前最擅长“在明确目标下搜索”和“在海量文本中检索归纳”，最不擅长“对最终正确性负责”。

北京大学
PEKING UNIVERSITY

PART 04 | Tool Stack & Boundaries

工具栈与能力边界

AI 在设计全流程上的落地地图

越靠近“生成”越成熟，越靠近“签核”越保守 —— 因为责任无法转移给模型

架构探索 中

设计空间搜索、性能建模与快速估算

代表案例：PRIME

RTL 设计 高

代码补全、模块生成、跨语言转换

代表案例：Verilog 开源生态

功能验证 中

激励生成、断言挖掘、失败聚类

代表案例：AI 辅助验证

逻辑综合 中

约束推荐、PPA 多目标参数寻优

代表案例：DSO.ai / Cerebrus

物理设计 中

布局规划、拥塞预测、时序收敛辅助

代表案例：AlphaChip

签核与 DFT 低

报告解读、缺陷定位、根因摘要

代表案例：尚无标志性案例

右上角为成熟度判断（高 / 中 / 低），仅代表 2026 年中的工程可用程度。

能力边界：AI 今天能做什么，不能做什么

生成不是瓶颈，验证才是

已经足够好用

- 写模板化、高重复的 RTL 与 testbench 草稿
- 生成与解释 EDA 脚本、约束文件、构建流程
- 把长报告、日志、缺陷记录压缩成可读摘要
- 在明确目标函数下做参数寻优与方案排序
- 作为查询接口，快速定位方法学与规范条款

必须由人把关

- 功能正确性判定 —— 模型会给出看起来对的错误代码
- 架构级权衡与需求取舍，这是价值判断而非搜索问题
- 签核结论与流片决策，责任无法转移给模型
- 专有 IP 与设计数据的合规使用与外泄风险控制
- 结果可复现性：随机性与版本漂移必须被显式固定

工具栈明细：今天都能用到什么

课程实验从“开源流程 + 通用大模型”起步 —— 可复现、零成本，且足以走通完整闭环

类别	代表工具 / 项目	典型用途	接入门槛
商用 EDA 的 AI 能力	Synopsys DSO.ai、Cadence Cerebrus、Siemens	在既有流程内做 PPA 自动寻优与设计空间探索	高（商用授权）
开源设计流程	OpenROAD、OpenLane、Yosys、iEDA	端到端 RTL-to-GDS 实验平台，适合教学与研究	低（免费可跑）
RL 布局开源实现	Circuit Training（AlphaChip）、MacroPlacement 基准	复现与对比宏单元布局方法	中
通用大模型	AI 、GPT、Gemini 等及其编码接口	RTL 草稿、脚本、文档解读、报告摘要	低
领域模型与评测集	ChipNeMo、VerilogEval、RTLLM、RTLCoder	专有语言与方法学任务；客观评测模型好坏	中
智能体与编排	工具调用框架、MCP 类协议、CI 流水线集成	把“生成—仿真—修复”串成可重复闭环	中
高层次设计语言	Chisel、SpinalHDL、HLS（C/C++ 综合）	抬升描述抽象层次，与生成式方法天然互补	中

第 3 周起，我们统一使用通用大模型接口完成全部实验，同学们可以自行定义Skill.md或者其他scripts、references。

北京大学
PEKING UNIVERSITY

基于AI的HDL提示工程简述

1 为什么 HDL/Verilog 需要专门的提示工程

- LLM 默认按「软件思维」生成代码，而硬件描述的是并行电路——[illegible]高频错误来自这个错位

1 隐含 Latch

组合逻辑分支缺[illegible]值 / 缺 else，综合出锁存器

```
if (se[illegible] y = a;   // 无 else → latch
```

2 阻塞 / 非阻塞混用

时序块里用 =，仿真通过、综合结果与预期不一致

```
always @(posedge clk) q = d;
```

3 位宽隐式截断

[illegible]乘法、加法进位在自决定上下文里被丢弃

```
um = a + b;       // 进位丢失
```

4 时钟 / 复位语义不清

用组合逻辑或计数器输出直接当时钟，[illegible]同步

```
assign clk2 = cnt[0]; // gate[illegible]
```

2 LLM 与 SystemVerilog 的交互逻辑

- **模型逐 token 顺序生成，电路并行运行——把「并行语义」通过提示文本装进上下文，是提示有效的前提**

1 系统层 AI IDE / 系统提示

编码规范、命名约定、禁用写法、目标工具链 (Vivado / Quartus / Verilator)

2 任务层 单次 Prompt

接口契约、时序 / 复位约定、参数化范围、验证要求、输出格式

3 反馈层 编译 / 仿真 / Lint 回灌

iverilog / verilator --lint-only / 断言失败信息作为下一轮输入

模型如何「读」HDL

- 关键字是最强的语义锚点：always_ff / always_comb 比注释更能约束输出结构
- 接口先行：先给端口表，模型会围绕端口推导内部逻辑
- 数字要显式：位宽、深度、延迟拍数写成 parameter，避免模型自行假设
- 负向约束有效：「不要用 initial 赋初值」「不要生成门控时钟」
- 反馈闭环：错误信息比重写提示更能收敛

3 高效 HDL 提示的五要素

- **缺任何一项，模型都会用「最常见的默认值」补全——而默认值往往不是你要的**

1

接口契约

模块名、每个端口的方向 / 位宽 / 含义、握手协议 (valid/ready)

2

时序与复位

同步 / 异步复位、极性、单时钟还是多时钟、输出延迟拍数

3

参数化

parameter 列表与合法范围，$clog2 派生量，是否要求 2 的幂

4

编码规范

SV-2012、always_ff / always_comb、logic 类型、禁用写法

5

验证要求

附带 testbench / 断言、期望的仿真器、给出预期波形可用例

经验法则：一个好的 HDL 提示读起来应该像一份精简的设计规格书 (mini-spec)，而不是一句愿望。

每加一条像综合工具能检查的东西，输出越可靠。

4 Spec-first 提示模板

- **把五要素固化成[illegible]模板，每次只替换内容**

TEMPLATE

```
【模块】<module name>, SystemVerilog-2012, 可综合
【功能】<一句话说明> + 关键行为>
【接口】
  clk, rst_n(异步低有效)
  <port> : <dir> [<W>-1:0]  // <含义>
【参数】<NAME> = <default>, 范围 <..>
【时序】输入到输出 <N> 拍; <握手协议>
【规范】
  - always_ff / always_comb, 禁止 always @*
  - 全部使用 logic; 非阻塞赋值仅用于时序块
  - [illegible]组合逻辑给默认值; 显式位宽, 禁止隐式截断
  - 不生成门控时钟 / 不用 initial 初始化
【输出】RTL + 自检 testbench (iverilog -g2012 可运行)
【验证】<用例 / 边界条件>
```

✕ 模糊提示

"帮我写一个 FIFO。"

模型必须猜：深度、位宽、同步/异步、复位极性、是[illegible]FWFT、满/空判定、要不要 count ...每一项猜错都要返工一[illegible]

✓ 规格化提示

"sync_fifo，DW=8 / DEPTH=16 (2 的幂)，单时钟，异步低有效复位，FWFT 读，full/empty/count 输出，always_ff 风格，附 testbench。"

一次生成即可编译仿真 → 见第[illegible]8 页

5 SystemVerilog 规范即约束

- 在提示中点名 SV 构造，等于给模型 (和综合工具) 一条可检查的规则

SV 构造	对 LLM 输出的约束效果	提示中的写法示例
always_ff / always_comb	声明设计意图；always_comb 缺省值缺失、always_ff 出现阻塞赋值时工具直接报错	禁止 always @* 与 always @(posedge clk) 混用
logic	统一 reg / wire，消除模型在两者间摇摆产生的多驱动错误	全部端口与内部信号使用 logic
parameter int + $clog2	位宽与地址宽度由参数派生，模型不再手算常量	localparam int AW = $clog2(DEPTH)
'0 / '1 / W(…) 显式转换	复位值与位宽扩展写法唯一，避免 32 位类型隐式规则	sum <= {(H+1)'(a) + b + c[0]}
typedef enum logic [..]	状态机状态命名可读，模型输出与规格一一对应	typedef enum logic [1:0] {IDLE, RUN, DONE} st_e;
unique / priority case	把优先级语义写进代码，同时让仿真器检查 case 完整性	unique case (sel) … endcase
assert property	让模型自带可执行的检查，验证层直接利用断言失败	assert property (@(posedge clk) full |-> !wr_en);

6 例 1 · 可编程 FIFO — 提示与规格

- **参数化深度 / 位宽，满空判定用「多一位指针」，首字直通读**

PROMPT

用 SystemVerilog 2012 写可综合模块 sync_fifo：
• 参数 DW=8, DEPTH=16（DEPTH 为 2 的幂），localparam AW = $clog2(DEPTH)
• 端口：clk, rst_n（异步低有效）；wr_en/wdata；rd_en/rdata；full, empty, count[AW:0]
• 满/空判定使用 AW+1 位读写指针（MSB 区分绕回），不要用额外的 flag 寄存器
• rdata 为首字直通 (FWFT)：empty=0 时 rdata 即队首数据
• 写满时忽略写，读空时忽略读
• 存储用 logic [DW-1:0] mem [DEPTH]，无复位；指针用 always_ff 异步复位
• 全部 logic，非阻塞赋值仅在 always_ff；不要 initial
• 附 iverilog -g2012 可运行的自检 testbench：写满 16 个再读空，检查 full/empty/数据顺序

为什么这样写提示

- 「DEPTH 为 2 的幂」+「AW+1 位指针」把满空判定方案钉死——否则模型可能生成 count 比较或 flag 方案，三种写法各有边界 bug
- 「mem 无复位」防止模型对存储阵列写 rst_n 循环复位，导致无法推断 BRAM
- 「FWFT」一个词决定 rdata 是组合读还是寄存读，避免多一拍延迟的返工
- 「写满忽略 / 读空忽略」定义溢出行为，模型不再自行加 error 端口
- 要求 testbench 与自检语句，生成后可直接进入反馈层

6 例 1 · 可编程 FIFO — AI Code 输出

- 已通过verilog编译 16 写 / 16 读自检仿真

AI code › rtl/sync_fifo.sv

```
module sync_fifo #(
  parameter  int DW = 8,                  // DEPTH 须为 2 的幂
  localparam int AW = $clog2(DEPTH)
)(
  input  logic          clk, rst_n,
  input  logic          wr_en,
  input  logic [DW-1:0] wdata,
  input  logic          rd_en,
  output logic [DW-1:0] rdata,
  output logic          full, empty,
  output logic [AW:0]   count
);
  logic [DW-1:0] mem [DEPTH];
  logic [AW:0]   wptr, rptr;              // 多 1 位区分满/空

  assign empty = (wptr == rptr);
  assign full  = (wptr[AW] != rptr[AW]) && (wptr[AW-1:0] == rptr[AW-1:0]);
  assign count = wptr - rptr;
  assign rdata = mem[rptr[AW-1:0]];       // 首字直通 (FWFT)

  always @(posedge clk)
    if (wr_en && !full) mem[wptr[AW-1:0]] <= wdata;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin wptr <= '0; rptr <= '0; end
    else begin
      if (wr_en && !full)  wptr <= wptr + 1'b1;
      if (rd_en && !empty) rptr <= rptr + 1'b1;
    end
endmodule
```

要点

- 指针 [AW:0]，full 与 MSB 相异、低位相同——与提示一致
- count = wptr − rptr，AW+1 位自然溢出
- mem 写入单独 always_ff，无复位，可映射 BRAM
- rdata 为 assign 组合读 (FWFT)
- 写满 / 读空多 1 位内联在使能中，无额外队列

仿真：err=0（full/empty/顺序均正确）

7 例 2 · 可编程的分频

- **提示要点：分频比范围、占空比要求、输出是否作为真正的时钟**

PROMPT

写 clk_div：参数 W=，输入 div[W-1:0]（偶数、≥2），输出 clk_out 为 50% 占空比、周期 = div × T_clk，异步低有效复位。用计数器在半周期翻转 clk_out，不要用 div 计数后再比较两次。仅一个 always_ff。

提示里必须写清的边界

- 奇数分频 / 50% 占空比需双沿或额外相位，属另一个规格
- clk_out 若驱动其他触发器，在 FPGA 中应改为「时钟使能脉冲」或经 BUFG——把这句写进 AI .md
- div 动态改变时是否允许毛刺：不说明，模型默认不处理

AI code › rtl/clk_div.sv

```
module clk_div #(parameter int W = 8)(
  input  logic          clk, rst_n,
  input  logic [W-1:0] div,      // 分频比，偶数且 >= 2
  output logic          clk_out  // 50% 占空比
);
  logic [W-1:0] cnt;
  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      cnt <= '0; clk_out <= 1'b0;
    end else if (cnt == (div >> 1) - 1'b1) begin
      cnt <= '0; clk_out <= ~clk_out;   // 半周期翻转
    end else
      cnt <= cnt + 1'b1;
endmodule
```

8 例 3 · 时钟升频 —— 让模型知道逻辑的边界

- 纯 RTL 无法「倍频」，正确的提示是要求实例化 PLL / MMCM 原语，并给出频率与器件

PROMPT

目标 Xilinx 7 系列，输入 clk_100 (100 MHz)，输出 clk_200 (200 MHz)。不要用组合逻辑或 XOR 倍频；实例化 MMCME2_BASE，计算 CLKFBOUT_MULT_F / CLKOUT0_DIVIDE_F 使 VCO 在 600–1200 MHz，输出经 BUFG，导出 locked。

模型常见的错误回答

- assign clk2x = clk ^ clk_delayed —— 只能在仿真里「看起来」倍频
- always #(T/4) clk2x = ~clk2x —— 仅限 testbench，不可综合
- 解决：提示中明确器件 + 原语名 + VCO 范围，模型会正确算参数

AI code › pll/clk_x2.sv （器件原语）

```
// 100 MHz × 10 / 5 = 200 MHz, VCO = 1000 MHz (600-1200 OK)
logic clk_200_unbuf;
MMCME2_BASE #(
  .CLKIN1_PERIOD    (10.0),   // 100 MHz
  .DIVCLK_DIVIDE    (1),
  .CLKFBOUT_MULT_F  (10.0),   // VCO = 100 × 10
  .CLKOUT0_DIVIDE_F (5.0)     // 1000 / 5 = 200 MHz
) u_mmcm (
  .CLKIN1   (clk_100),
  .CLKFBIN  (clkfb),  .CLKFBOUT (clkfb),
  .RST      (rst),    .PWRDWN   (1'b0),
  .CLKOUT0  (clk_200_unbuf),
  .LOCKED   (locked)          // 其余 CLKOUTn 悬空
);
BUFG u_bufg (.I(clk_200_unbuf), .O(clk_200));
```

9 例 4 · 简单组合逻辑 — 优先编码器

- 组合逻辑的提示[illegible]默认值、优先级方向、always_comb

PROMPT

写参数化优先[illegible]器 prio_enc：N=8，输入 req[N-1:0]，输出 idx[$clog2(N)-1:0] 与 valid。低位优先（bit[illegible]级最高）。用 always_comb + for 循环，先给默认值，不允许 latch，不用 casez 展开。

AI code › [illegible]/prio_enc.sv

```
module prio_enc #(parameter int N = 8)(
  input  logic [N-1:0]          req,
  output logic [$clog2(N)-1:0] idx,   // 低位优先
  output logic                  valid
);
  always_comb begin
    idx = '0; valid = 1'b0;          // 默认赋值：无 latch
    for (int i = N-1; i >= 0; i--)
      if (req[i]) begin
        idx = i[$clog2(N)-1:0]; valid = 1'b1;
      end
  end
endmodule
```

组合逻辑[illegible]提示[illegible]要

- 默认值：每个输出在块首赋值，是消除 latch 的唯一可[illegible]
- 优先级方向：「低位优先」决定 for 循环方向，模型只有被告知[illegible]才不会猜
- [illegible]选择：for / casez / 三元链综合结果不同，提示[illegible]指定

同类可复用提示： N:1 参数化 mux（unique case）、二进制[illegible]独热译码、奇偶 / 汉明校验位、多数表决——只需替换【功能】和【接口】两行

10 例 5 · 两级流水线加法器

- 算术类提示的关键点：位宽（含进位）、延迟拍数、valid 随路

PROMPT

写 pipe_adder，W=32，a/b 相加，输出 sum[W:0]（含进位）。两级流水：第 1 拍算低 W/2 位并寄存高半部，第 2 拍算高半部 + 进位。in_valid 与 out_valid 延迟 2 拍。异步复位。显式位宽转换，拼接中不得丢进位。

第一版常见缺陷 → 反馈修复

- {a_hi + b_hi + c, lo} 在拼接里自决定为 H 位，进位丢失；仿真 0xFFFF_FFFF+1 ≠ 2^32 立刻暴露
- 反馈一句「拼接内用 (H+1)'( ) 显式扩展」即收敛
- 给定验证：sum = 1_0000_0000，2 拍后 out_valid

AI code › rtl/pipe_adder.sv

```
module pipe_adder #(parameter int W = 32)(
  input  logic          clk, rst_n, in_valid,
  input  logic [W-1:0] a, b,
  output logic          out_valid,
  output logic [W:0]    sum
);
  localparam int H = W/2;
  logic [H:0]   lo_s;
  logic [H-1:0] a_hi_q, b_hi_q;
  logic         v_q;
  // Stage 1: 低半部相加，高半部寄存
  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) begin lo_s <= '0; a_hi_q <= '0; b_hi_q <= '0; v_q <= 1'b0; end
    else begin
      lo_s   <= a[H-1:0] + b[H-1:0];
      a_hi_q <= a[W-1:H]; b_hi_q <= b[W-1:H]; v_q <= in_valid;
    end
  // Stage 2: 高半部 + 进位，显式位宽避免截断
  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) begin sum <= '0; out_valid <= 1'b0; end
    else begin
      sum <= {(H+1)'(a_hi_q) + b_hi_q + lo_s[H], lo_s[H-1:0]};
      out_valid <= v_q;
    end
endmodule
```

11 例 6 · 有符号流水线乘法器

- 让综合工具推断 DSP：用 * 加寄存器链，而不是让模型手写 Booth / 移位相加

PROMPT

写 mul_signed：W=16，a/b 为 signed，输出 p[2W-1:0] signed。参数 STAGES=2 的寄存器链（让 Vivado 推断为 DSP48 流水）。用 * 运算，不要手写部分积。in_valid 随路延迟 STAGES 拍。复位清零所有级。

提示中的取舍

- signed 要写在端口声明与提示里——否则模型很可能生成无符号乘再截断
- 「不要手写部分积」：手写 shift-add 面积 / 时序都更差，除非目标是无 DSP 的 ASIC 教学
- 优美：(-300)×123 = -36900 ✓

AI code › mul_signed.sv

```
module mul_signed #(parameter int W = 16, STAGES = 2)(
  input  logic                    clk, rst_n, in_valid,
  input  logic signed [W-1:0]     a, b,
  output logic                    out_valid,
  output logic signed [2*W-1:0] p
);
  logic signed [2*W-1:0] pipe [STAGES];   // 寄存器链，综合可映射为 DSP
  logic [STAGES-1:0]     v;
  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      v <= '0;
      for (int i = 0; i < STAGES; i++) pipe[i] <= '0;
    end else begin
      pipe[0] <= a * b;  v[0] <= in_valid;        // 有符号乘，L 决定位宽
      for (int i = 1; i < STAGES; i++) begin
        pipe[i] <= pipe[i-1]; v[i] <= v[i-1];
      end
    end
  assign p = pipe[STAGES-1];
  assign out_valid = v[STAGES-1];
endmodule
```

12 AI Code 工作流：规范固化 + 仿真闭环

- **把系统层写进.md，把反馈层交给命令行工具，模型自己跑循环**

```
AI  code ›  AI .md

# AI .md  (项目根目录)
## HDL 规范
- SystemVerilog-2012, 可综合, 目标 Vivado 2024 / Verilator 5
- always_ff / always_comb; 禁止 always @*, 禁止 initial
- 全部 logic; 复位统一 rst_n 异步低有效
- 组合块先给默认值, 避免锁存, 不生成门控时钟
- 每个模块附 tb_<name>.sv, 含 $error 自检
## 验证命令
- lint: verilator --lint-only -Wall rtl/<m>.sv
- sim:  iverilog -g2012 -o build/<m> tb/tb_<m>.sv rtl/<m>.sv && vvp build/<m>
## 交付
- 先输出端口表, 再写 RTL; 仿真通过后再回复
```

1. **Prompt**
   用 Spec-first 模板描述模块
2. **Generate**
   AI Code 写 rtl/ + tb/
3. **Lint & Sim**
   自动运行 verilator / iverilog
4. **Feedback**
   错误与断言信息回灌，迭代直到 errs=0

本次 6 个案例：首轮 5/6 直接通过，加注释 1 轮反馈收敛

13 总结：HDL 提示检查清单

- 提交提示前逐项对照——每一项都对应一类可预测的模型错误

- ✓ **接口表完整** 方向 / 位宽 / 含义 / 握手
- ✓ **复位与时钟已声明** 同步或异步、极性、时钟数
- ✓ **参数与范围** parameter int、$clog2、2 的幂约束
- ✓ **延迟拍数写明** in_valid → out_valid 多少拍
- ✓ **[illegible]点名** always_ff / always_comb / logic
- ✓ **负向约束** 无 initial、无门控时钟、无隐式[illegible]
- ✓ **[illegible]边界** 倍频用 PLL；跨时钟域用同步器
- ✓ **可执行验证** testbench + 自检 + 运行[illegible]

好的 HDL 提示 = 一份综合工具[illegible]的 mini-spec + 一条模型能自己跑通的[illegible]命令

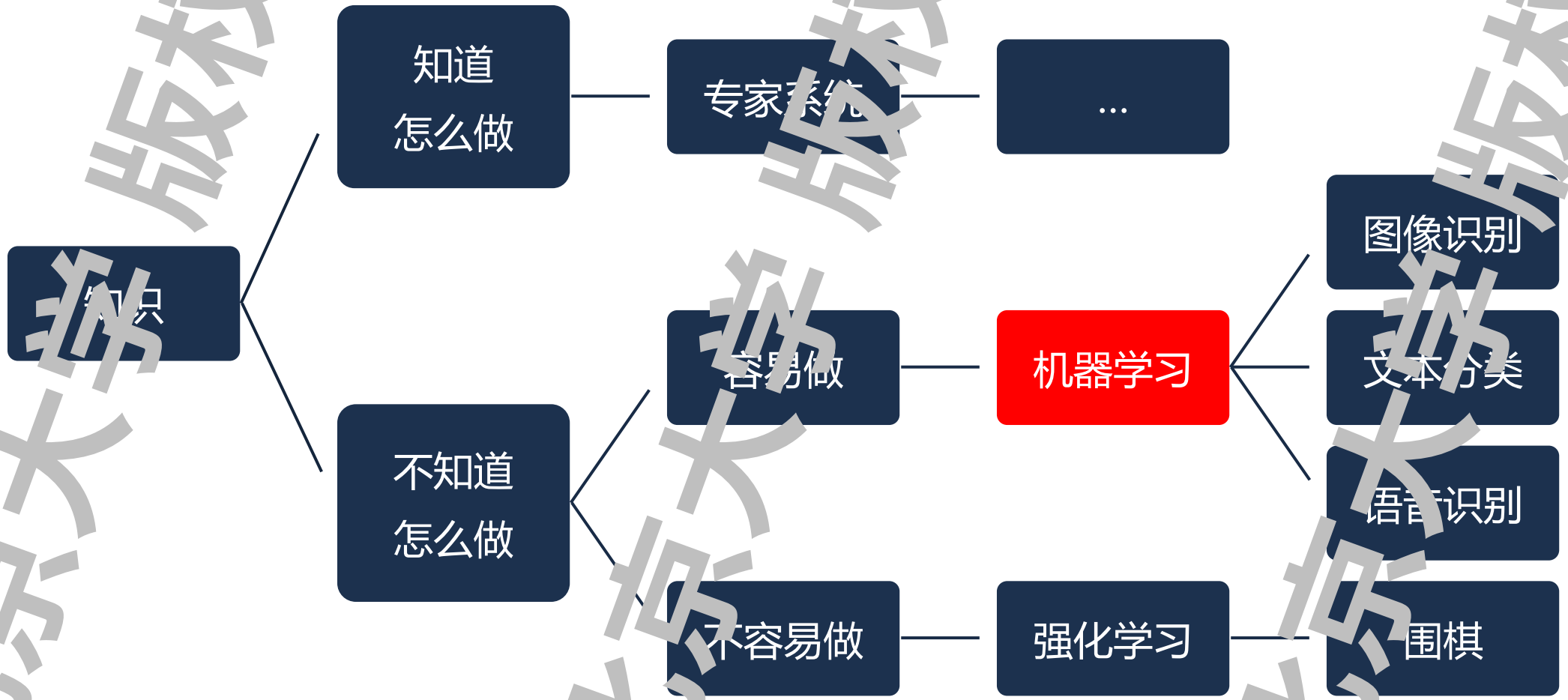
北京大学
PEKING UNIVERSITY

PART 05 | Algorithms

面向电路架构优化的AI算法

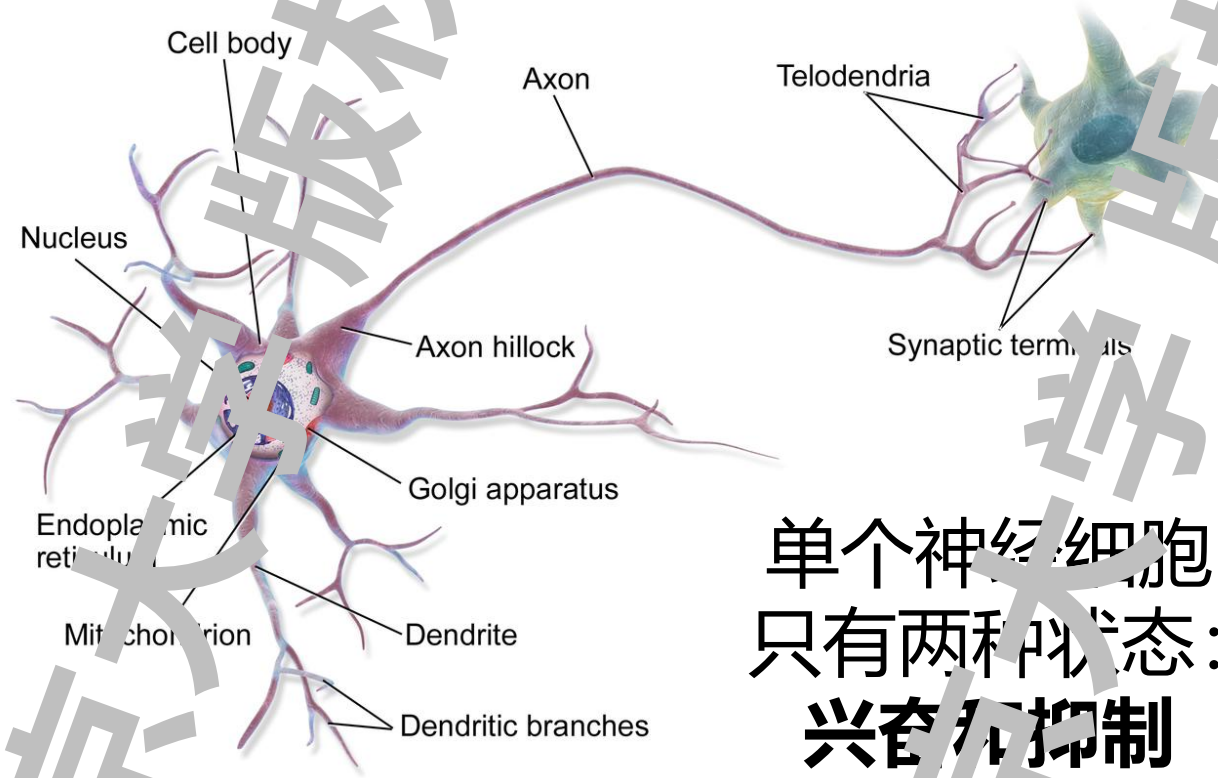
如何开发一个人工智能算法?

- 三大根本性问题：做什么、怎么做、做的好?

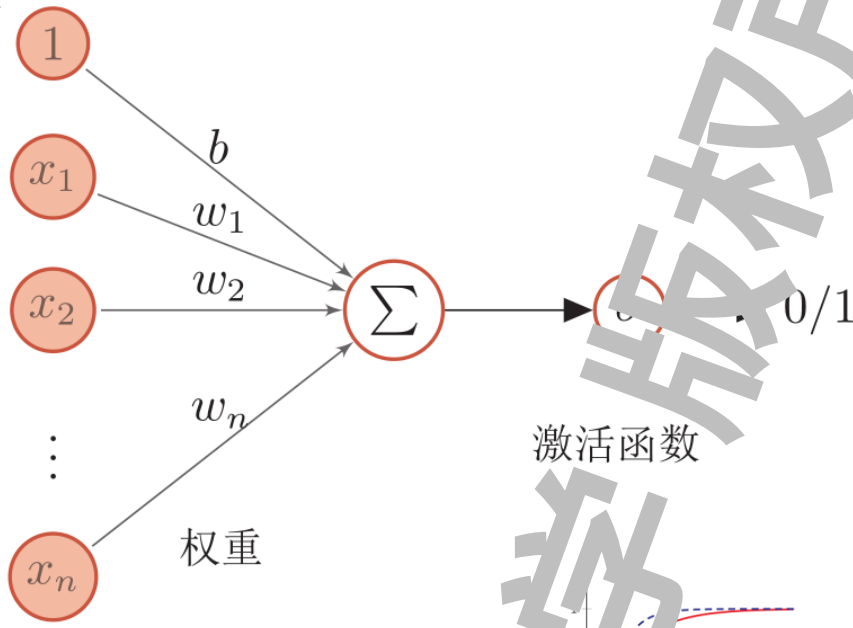


神经网络在最近十余年成为主流

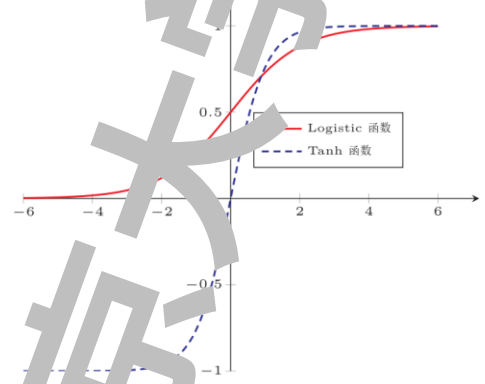
- 受到生物神经元启发



单个神经细胞只有两种状态：**兴奋和抑制**

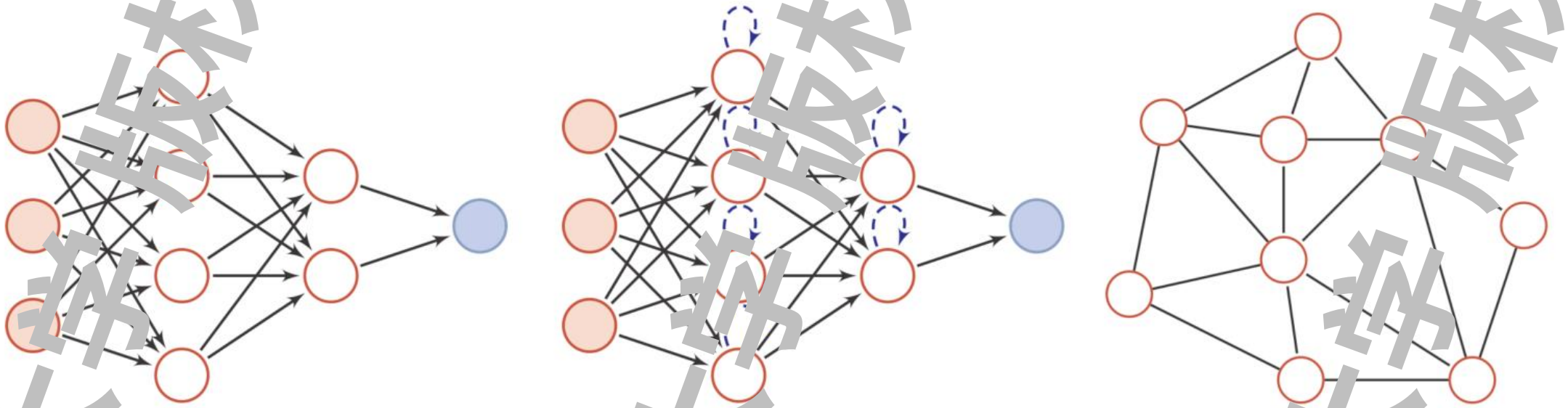


人工神经元



神经网络在最近十余年成为主流

- 人工神经网络由神经元模型构成，这种由许多神经元组成的信息处理网络具有并行分布结构。

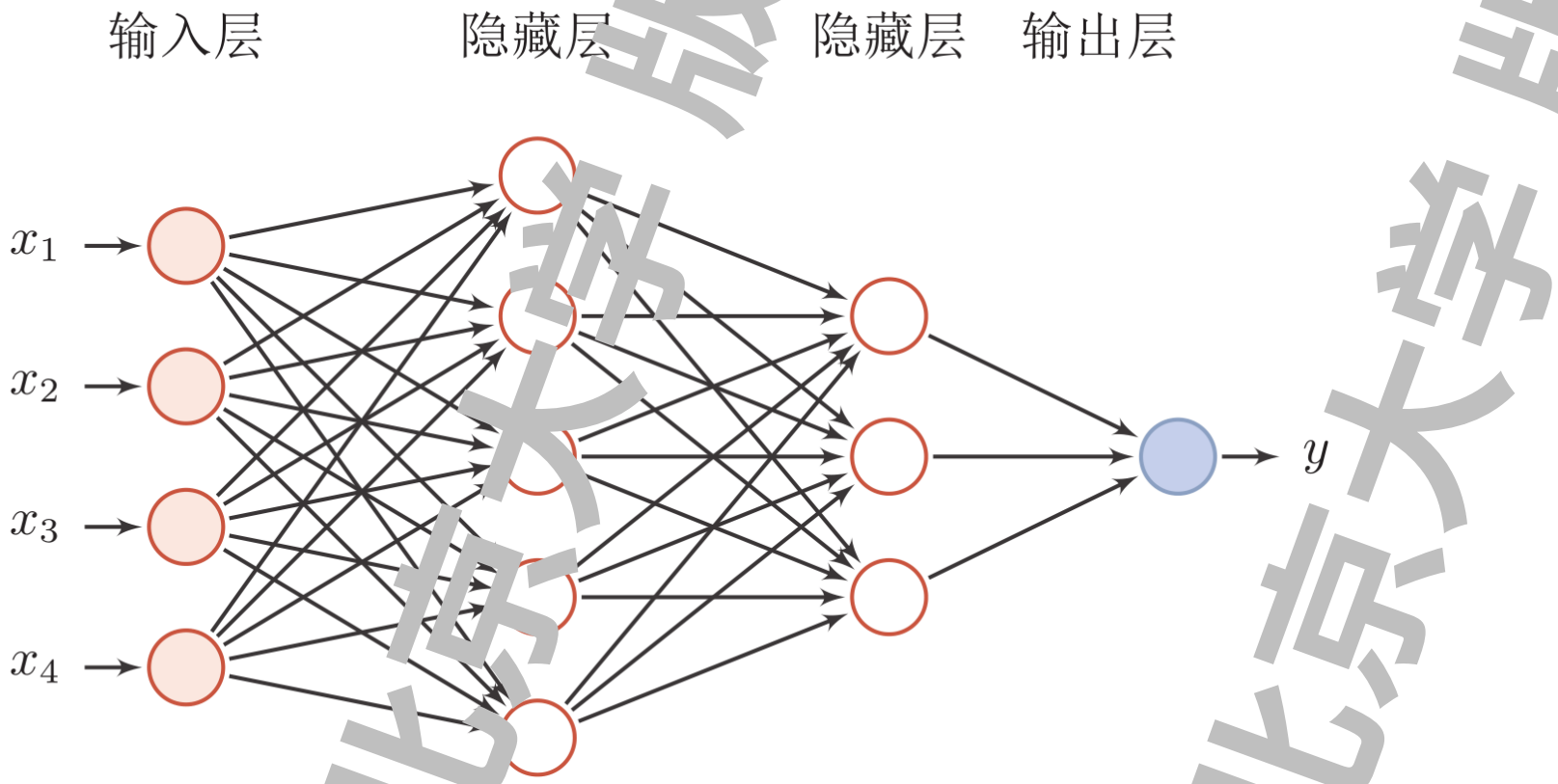


(a) 前馈网络　(b) 反馈网络　(c) 图网络

前馈神经网络

- **网络结构**

- 在前馈神经网络中，**各神经元分别属于不同的层**
- 整个网络中无反馈，信号从输入层向输出层单向传播，**可用一个有向无环图表示**



前馈神经网络

- 通用近似定理

- 只要其隐藏层神经元的数量足够，它可以以任意精度来近似任何从一个定义在实数空间中的有界闭集函数。

- 模型
  - $y = f^5(f^4(f^3(f^2(f^1(x)))))$

- 学习准则
  - $\mathrm{L}(y, y^*)$

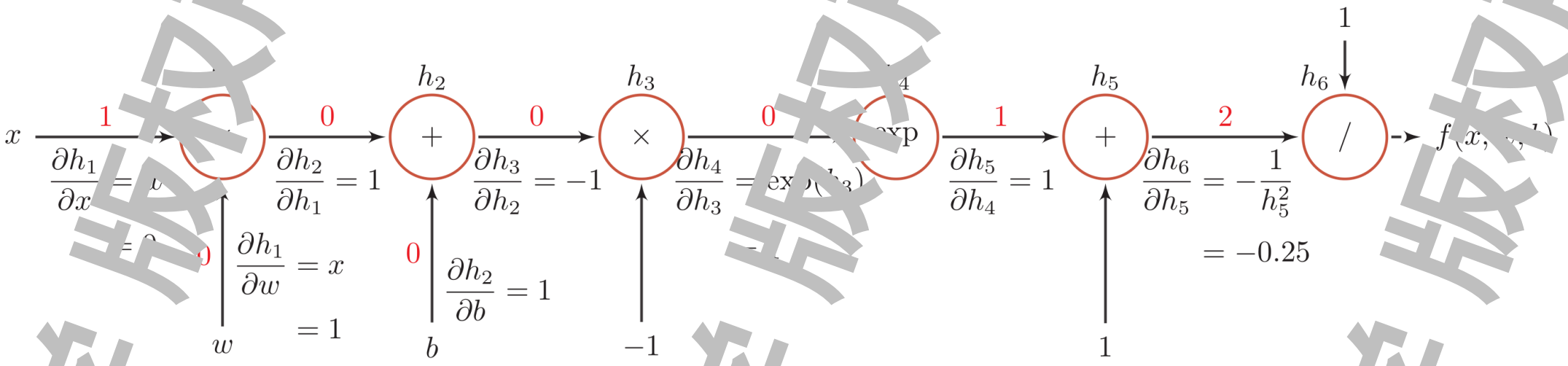
- 优化
  - 梯度下降

链式法则，可以自动计算！

$$\frac{\partial \mathrm{L}(y, y^*)}{\partial f^1} = \frac{\partial f^2}{\partial f^1} \times \frac{\partial f^3}{\partial f^2} \times \frac{\partial f^4}{\partial f^3} \times \frac{\partial f^5}{\partial f^4} \times \frac{\partial \mathrm{L}(y, y^*)}{\partial f^5}$$

计算图与自动微分

- 复合函数f(x; w, b) = σ(wx + b)的计算图



链式法则

$$\frac{\partial f(x;w,b)}{\partial w} = \frac{\partial f(x;w,b)}{\partial h_6}\frac{\partial h_6}{\partial h_5}\frac{\partial h_5}{\partial h_4}\frac{\partial h_4}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial w}$$

反向传播算法只是自动微分的一种特殊形式

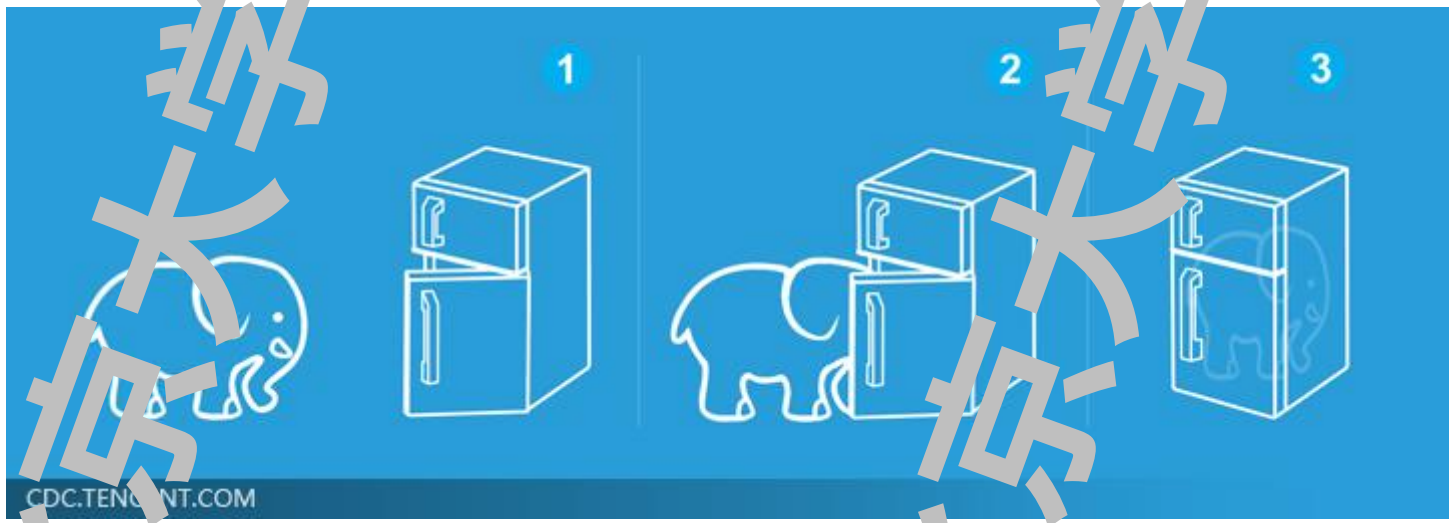
深度学习的三个步骤

1 定义网络

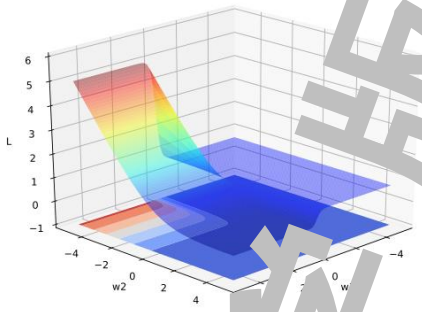
2 损失函数

3 优化

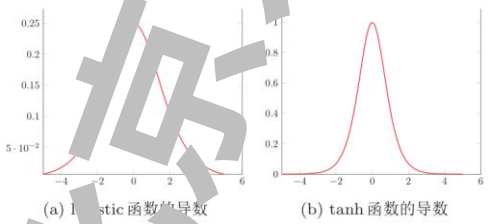
Deep Learning is so simple ……



- 非凸优化问题
  - y = σ(w 2 σ(w 1 x)) 的损失函数



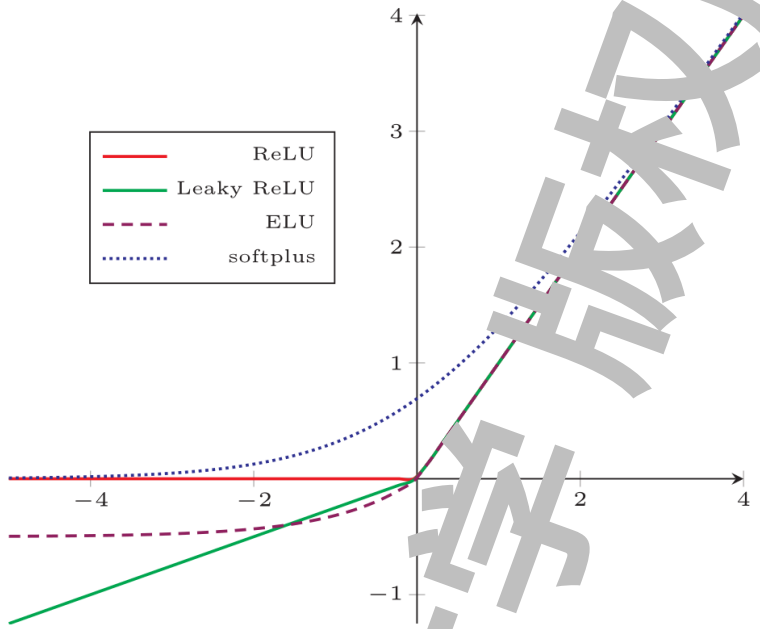
- 梯度消失问题
  - 在每一层都要乘以该层的激活函数的导数



激活函数

有效减轻梯度消失问题！

激活函数	函数	导数
Logisti 函数	$f(x)=\frac{1}{1+e^{-x}}$	$f'(x)=f(x)($ $f(x))$
Tann 函数	$f(x)=\frac{2}{1+e^{-2x}}-1$	$f'(x)=1-$ $(x)^2$
ReL	$f(x)=\max(0,x)$	$f'(x)=I(x>0)$
EL	$f(x)=\max(0,x)+\min(0,\gamma(\exp(x)-1))$	$f'(x)=(x>0)+I(x\leq 0)\cdot\gamma\exp(x)$
o Plus 函数	$f(x)=\ln(1+e^x)$	$f'(x)=\frac{1}{1+e^{-x}}$

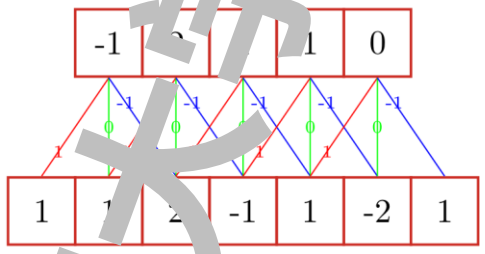


典型神经网络模型：卷积神经网络

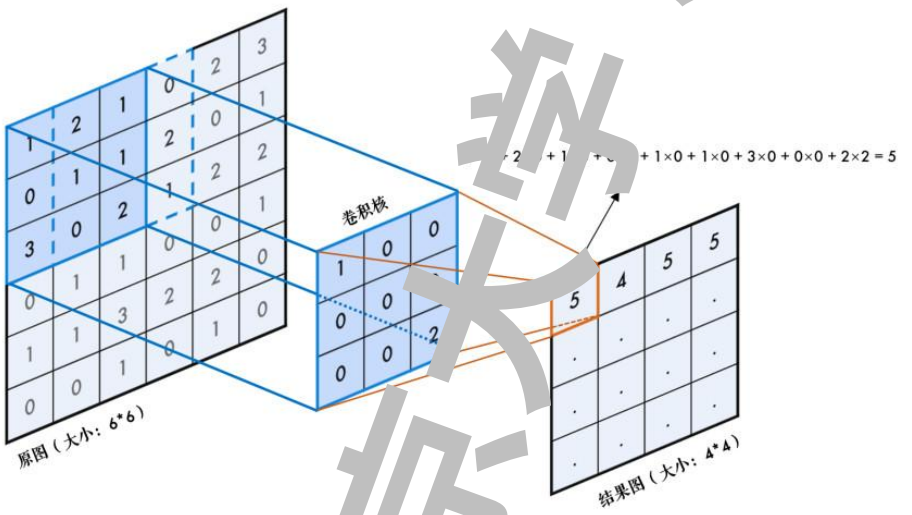
- 卷积经常出现在信号处理中，用于计算信号的延迟累积。
- 假设一个信号发生器每个时刻t产生一个信号x_t，其信息的衰减率为w_k，即在k−1个时间步长后，信息为原来的w_k 倍
  - 假设$w_1 = 1, w_2 = 1/2, w_3 = 1/4$
- 时刻t收到的信号y_t 为当前时刻产生的信息和以前时刻延迟信息的叠加

$$y_t = 1 \times x_t + 1/2 \times x_{t-1} + 1/4 \times x_{t-2}$$
$$= w_1 \times x_t + w_2 \times x_{t-1} + w_3 \times x_{t-2}$$
$$= \sum_{k=1}^{3} w_k \cdot x_{t-k+1}.$$

滤波器（filter）或卷积核（convolution kernel）



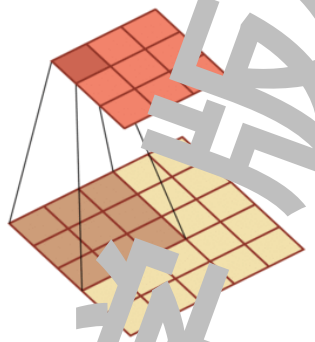
滤波器：[1, 0, -1]



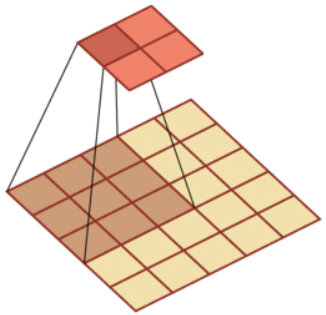
二维卷积

典型神经网络模型：卷积神经网络

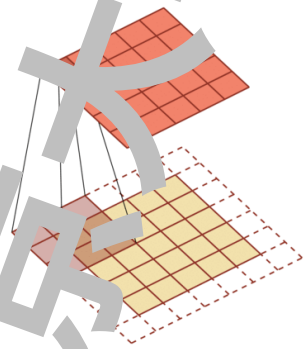
- **二维卷积：在数据的两端填充零的技术，它的主要目的是使卷积后的图像大小不变，从而方便后续的操作。当卷积核的大小大于输入图像的大小时，零填充就变得特别重要，因为它能够避免图像的尺寸减小**



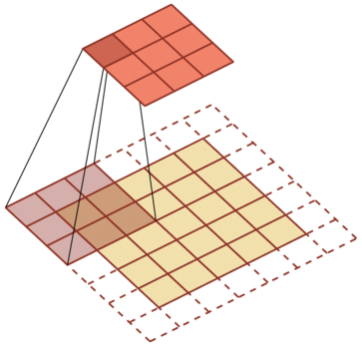
步长1，零填充0



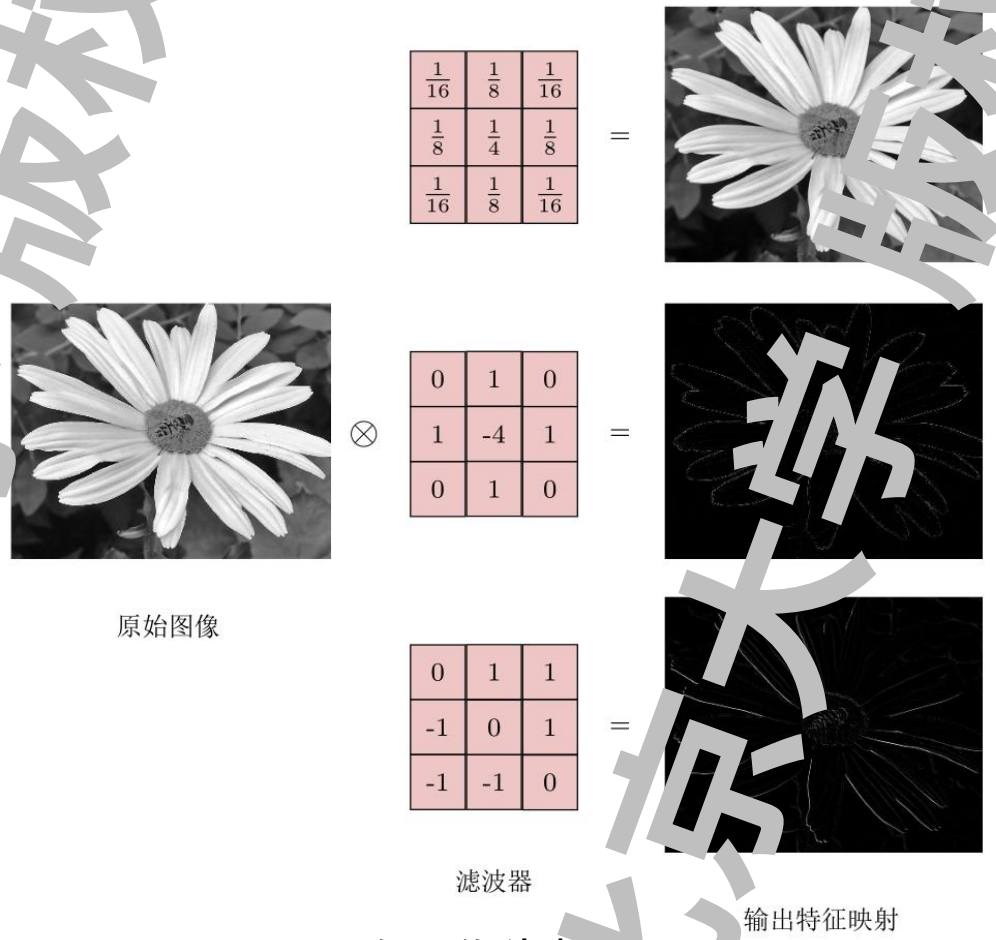
步长2，零填充0



步长1，零填充1



步长2，零填充1



原始图像

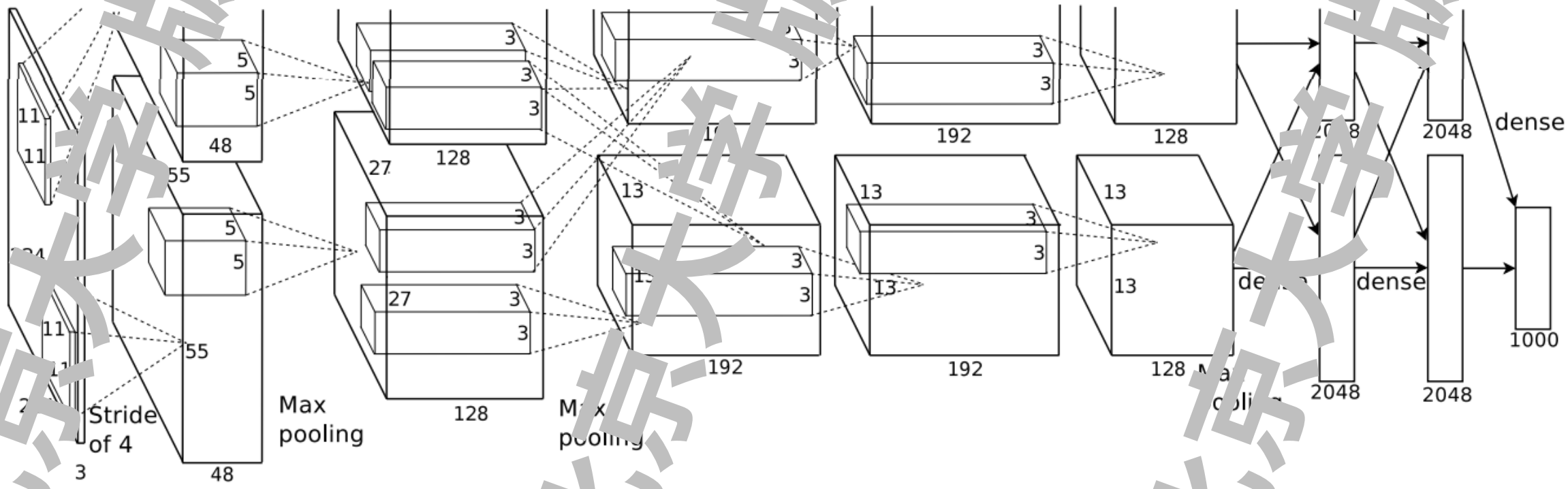
滤波器

输出特征映射

卷积作为特征提取器

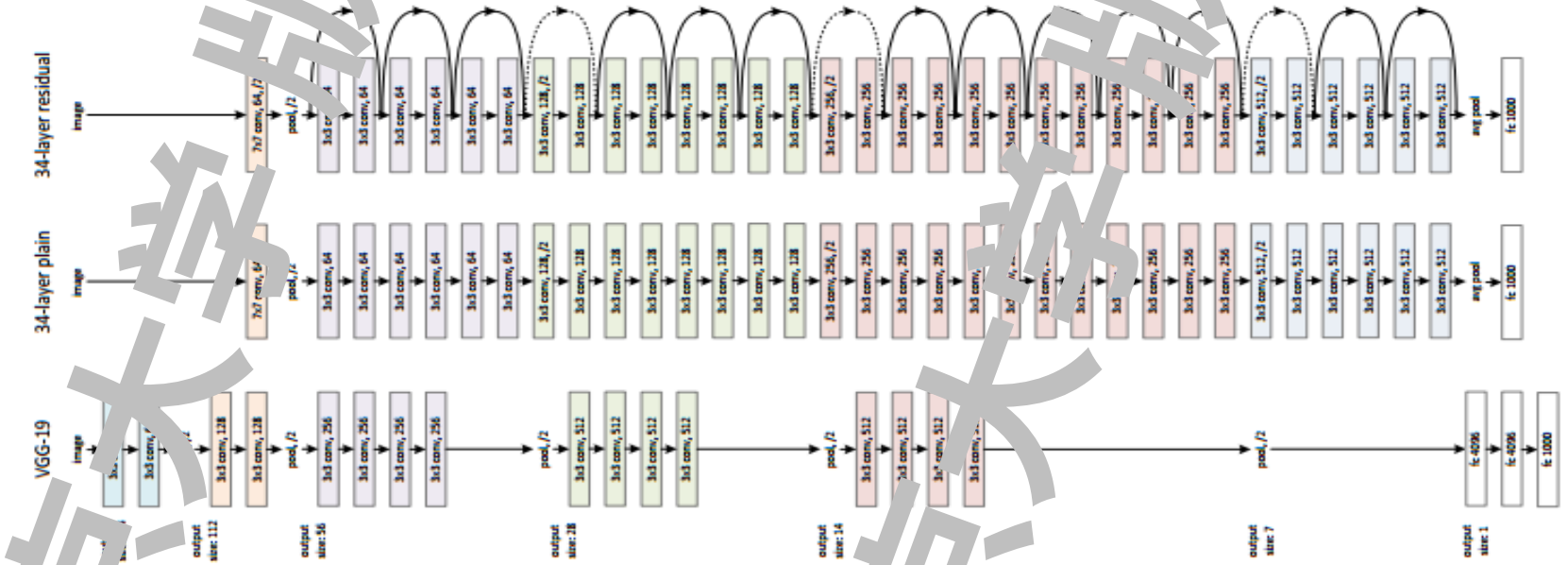
典型神经网络模型：卷积神经网络

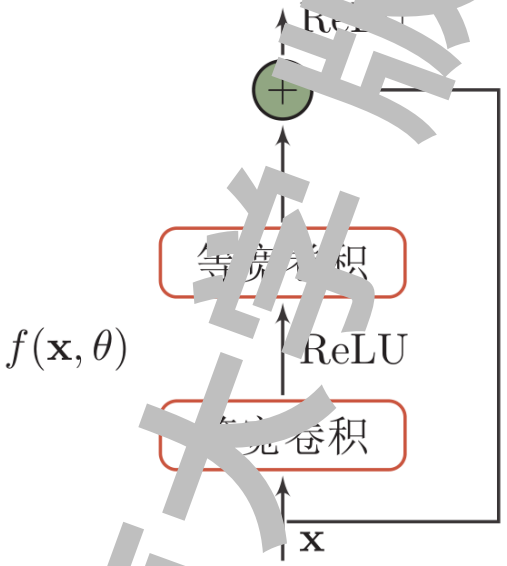
- AlexNet、2012 ILSVRC winner
- (top 5 error of 16\% compared to runner-up with 26\% error)
- 共有8层，其中前5层卷积层，后边3层全连接层



典型神经网络模型：卷积神经网络

- ResNet、2015 ILSVRC winner （152层）
- 错误率：3.57%



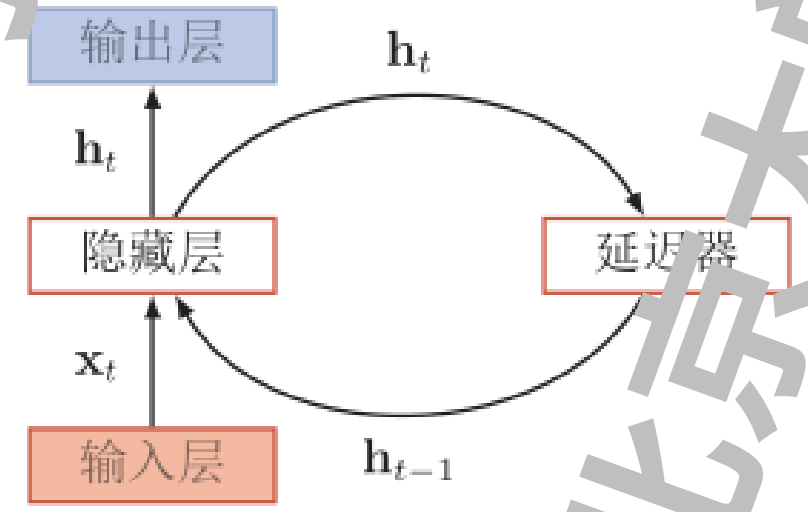


典型神经网络模型：循环神经网络

- 循环神经网络通过使用带自反馈的神经元，能够处理任意长度的序列。
- 循环神经网络比前馈神经网络更加符合生物神经网络的结构。
- 循环神经网络已经被广泛应用在语音识别、语言模型以及自然语言生成等任务上。

对于一个输入序列$\mathbf{x}_{1:T} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t, \ldots, \mathbf{x}_T)$，循环神经网络通过下面公式更新带反馈边的隐藏层的活性值$\mathbf{h}_t$：

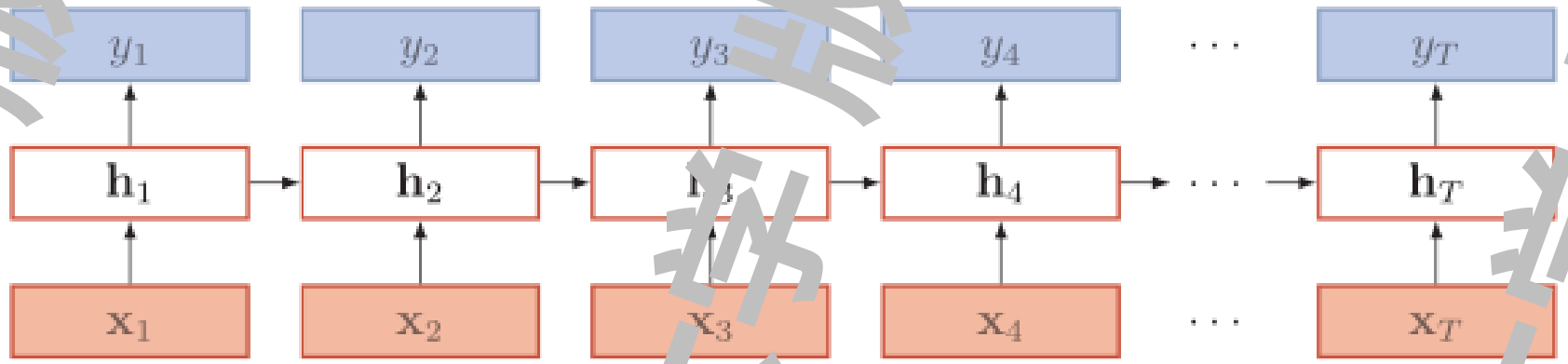
$$\mathbf{h}_t = \begin{cases} 0 & t = 0 \\ f(\mathbf{h}_{t-1}, \mathbf{x}_t) & \text{otherwise} \end{cases}$$



典型神经网络模型：循环神经网络

- **状态更新：**

$$\mathbf{h}_t = f(U\mathbf{h}_{t-1} + W\mathbf{x}_t + \mathbf{b}),$$

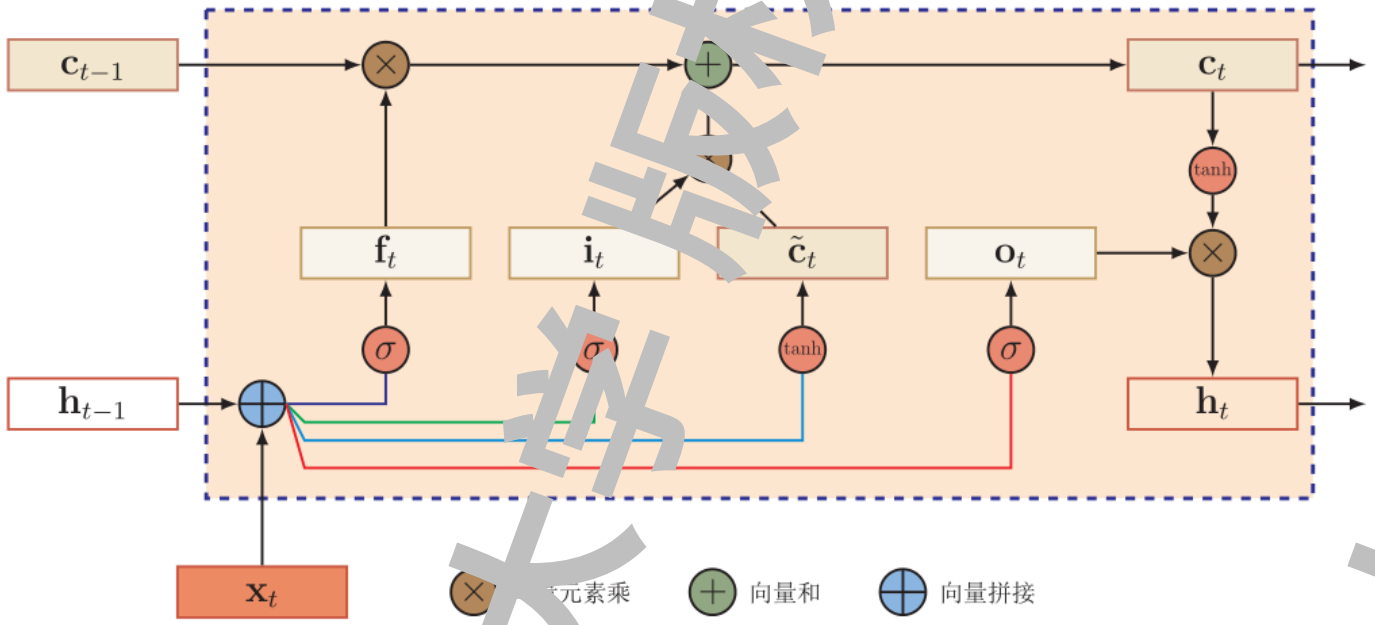


RNN是图灵完全等价的 (Siegelmann and Sontag, 1995)

FNN：模拟任何函数
RNN：模拟任何程序（计算过程）。

典型神经网络模型：长短时记忆神经网络LSTM

- 循环神经网络在时间维度上非常深！
- 梯度消失或梯度爆炸



$$\mathbf{i}_t = \sigma(W_i\mathbf{x}_t + U_i\mathbf{h}_{t-1} + \mathbf{b}_i),$$

$$\mathbf{f}_t = \sigma(W_f\mathbf{x}_t + U_f\mathbf{h}_{t-1} + \mathbf{b}_f),$$

$$\mathbf{o}_t = \sigma(W_o\mathbf{x}_t + U_o\mathbf{h}_{t-1} + \mathbf{b}_o),$$

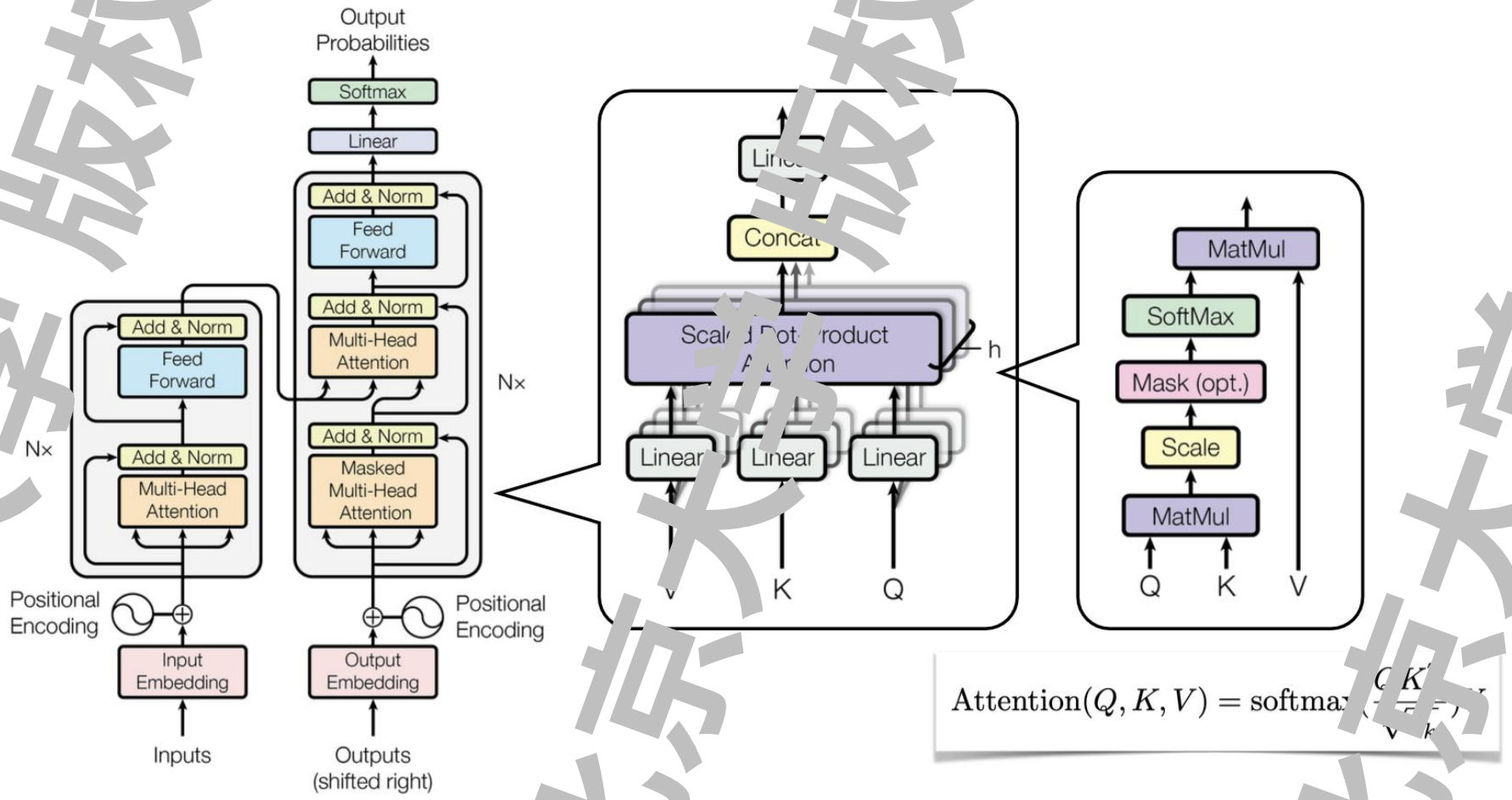
$$\tilde{\mathbf{c}}_t = \tanh(W_c\mathbf{x}_t + U_c\mathbf{h}_{t-1} + \mathbf{b}_c)$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t,$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t),$$

典型神经网络模型：自注意力机制神经网络Transformer

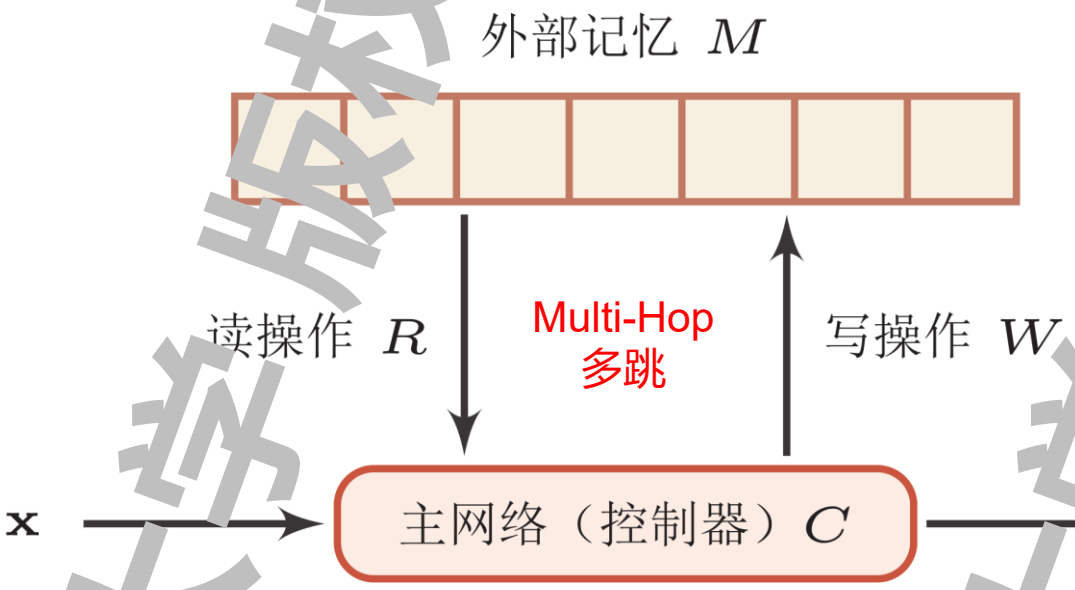
- RNN的问题：处理长序列时，信息会丢失（多个输入序列xi ->单个向量c）
- Transformer：基于注意力机制，抛弃卷积，模仿人类视觉处理信息时的选择性关注视点

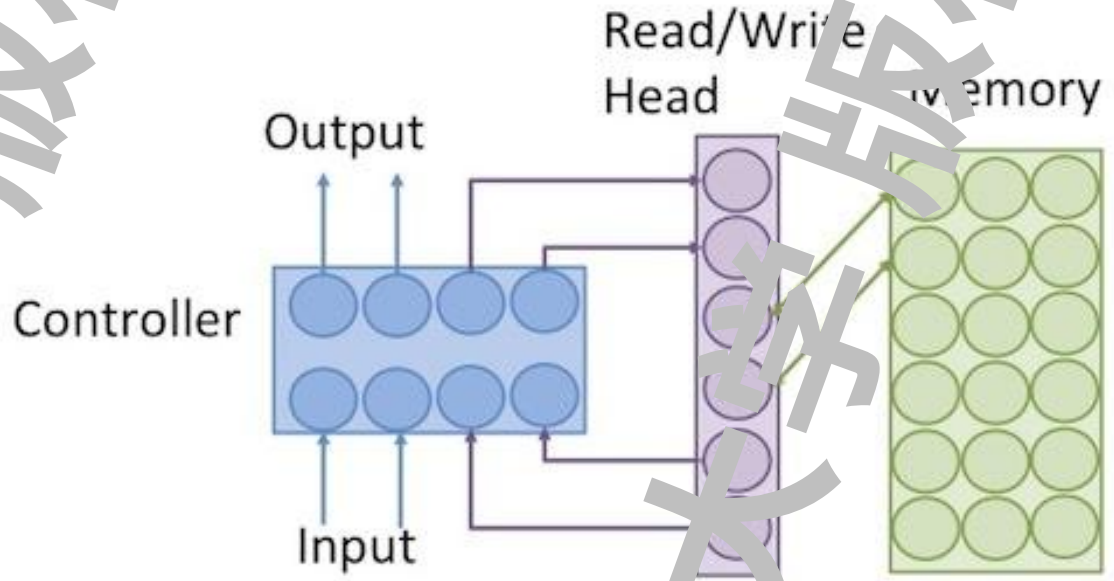


$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

典型神经网络模型：记忆增强神经网络NTM、DNC

- 神经图灵机、微分神经计算机等

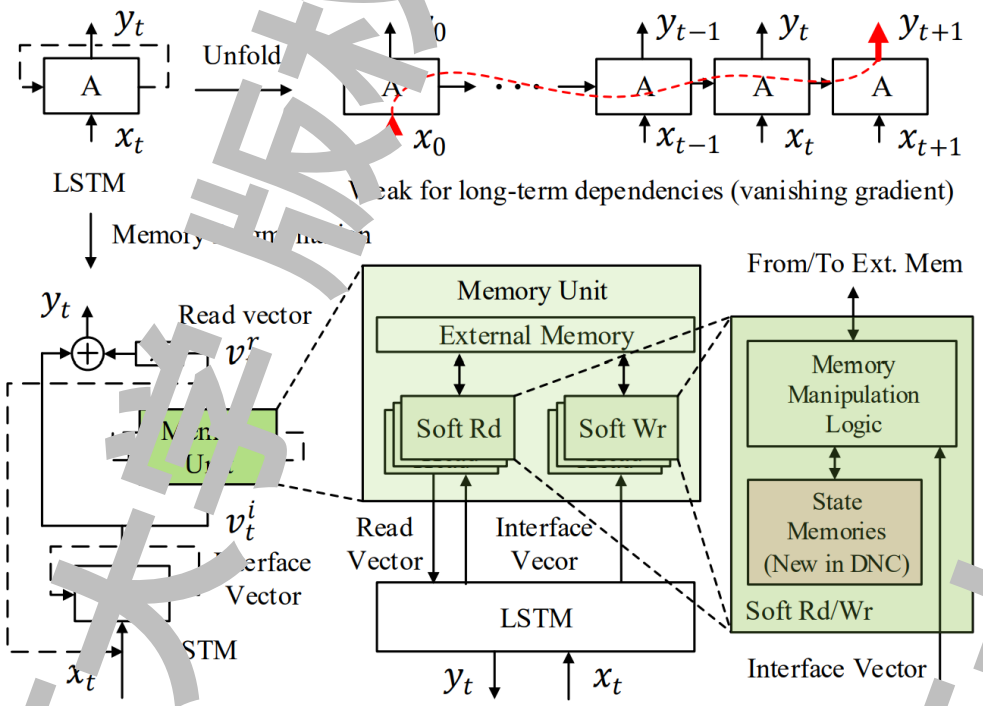


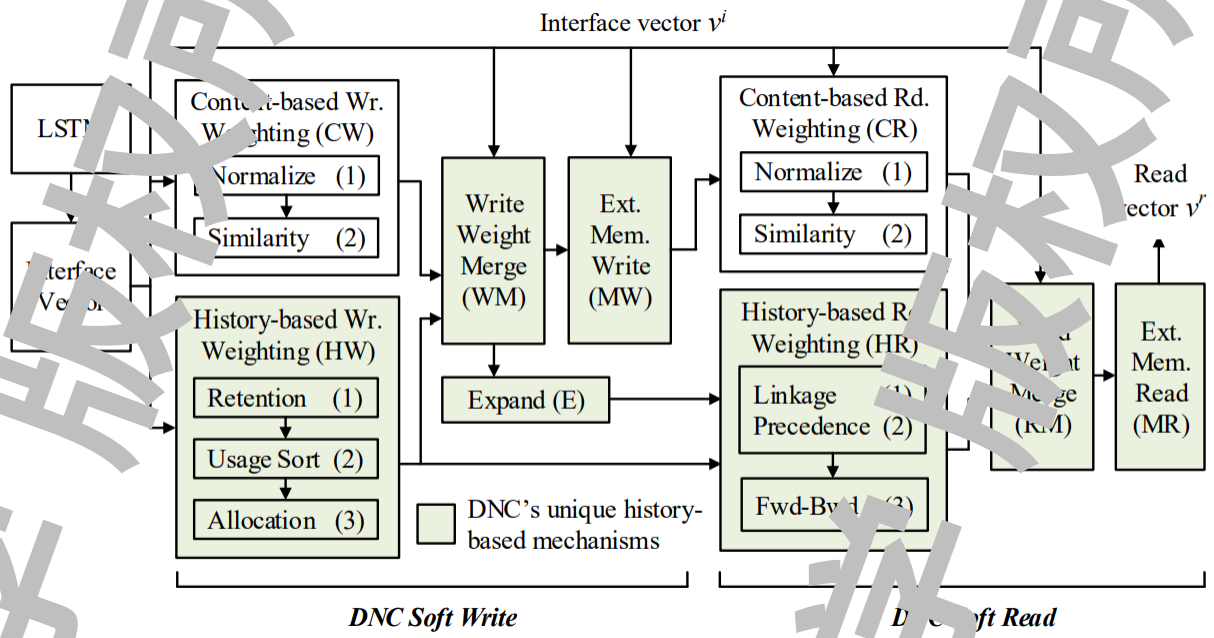


按内容寻址：通常利用注意力机制来完成。

典型神经网络模型：记忆增强神经网络NTM、DNC

- 神经图灵机、微分神经计算机等





DNC Soft Write

CW.(1): $\|M[i,\cdot]\| \leftarrow M[i,\cdot]\ \|k^w\| \leftarrow k^w$

CW.(2): $w^u \leftarrow \text{softmax}(s^w \|M[i,\cdot]\| \cdot \|k^w\|)$

HW.(1): $\psi[i] = \prod_{r=1}^{R}(1 - g^f[r] w^r[i,r])$

HW.(2): $I_s \leftarrow \text{sort}\ (u + w^w - u \circ w^w) \circ \psi$

HW.(3): $w^a[I_s^i] = (1 - u[I_s^i]) \prod u[I_s^{1 \to i}]$

WM: $w^w = g^w(g^a w^a + (1 - g^a w^u))$

E: $w[\cdot, i] = w^w$

MW: $M \leftarrow M \circ (E - w^w (v^e)^T) + w^w (v^w)^T$

DNC Soft Read

CR.(1): $\|M[i,\cdot]\| \cdot \|k^r\| \leftarrow k^r$

CR.(2): $r^u \leftarrow \text{softmax}(\cdot \|M[i,\cdot]\| \cdot \|k^r\|)$

HR.(1): $\{(E - w \cdot w^T) \circ L \cdot p^T\} \circ (E - I)$

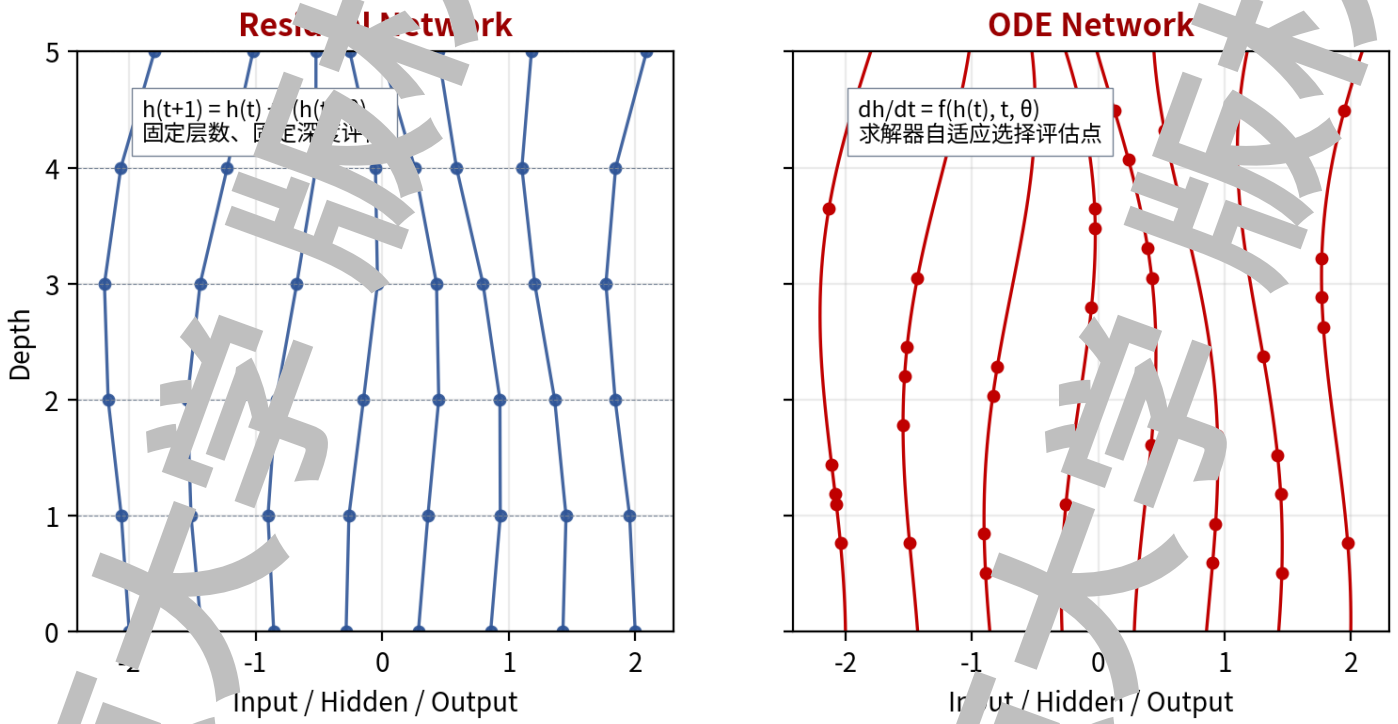
HR.(2): $p = \sum_{i=1}^{N} w^w[i]) p + w^w$

HR.: $b^r = L^T w^r$

MR: $v^r = M^T$

典型神经网络模型：Neural Ordinary Differential Equations

- Chen, Rubanova, Bettencourt & Duvenaud, NeurIPS 2018 最佳论文 — 把网络深度变成连续时间



核心思想

- ResNet 的 $h_{t+1} = h_t + f(h_t, \theta)$ 是欧拉法的一步；令步长 → 0 得 **dh/dt = f(h(t), t, θ)**，输出 h(T) 由 ODE 求解器给出
- 参数量与内存不随“深度”增长；精度-算力可在推理时通过求解器容忍度调节
- 衍生：连续归一化流 (CNF)、不规则采样时间序列建模

北京大学
PEKING UNIVERSITY

基于AI的PPA优化方案与框架

AI 在延迟 / 功耗 / 面积优化中的应用

- 无论生成还是优化，都需要“快速、准确、可迁移”的 PPA 评估器——下面按目标逐一展开

AI 模型在延迟 / 功耗 / 面积优化中的典型应用（按目标分类）

延迟 (Delay / Timing)	功耗 (Power)	面积 (Area)
布线前时序预测 (GNN)	逐周期功耗 (PRIMAL / GRANNITE)	RTL 面积预估
关键路径 / WNS 预估 (MasterRTL)	RTL 功耗预估 (含翻转率)	RL 逻辑综合 (面积奖励)
HLS 周期数代价模型	IR-drop 热图预测 (CNN)	布局密度 / 拥塞 (AlphaChip)
RL 逻辑综合 (延迟约束)	DVFS 策略 (RL)	架构参数优化 (PRIME)

延迟

- GNN 布线前时序预测 (TimingGCN)
- RTL 级 WNS/TNS 预估 (MasterRTL)

功耗

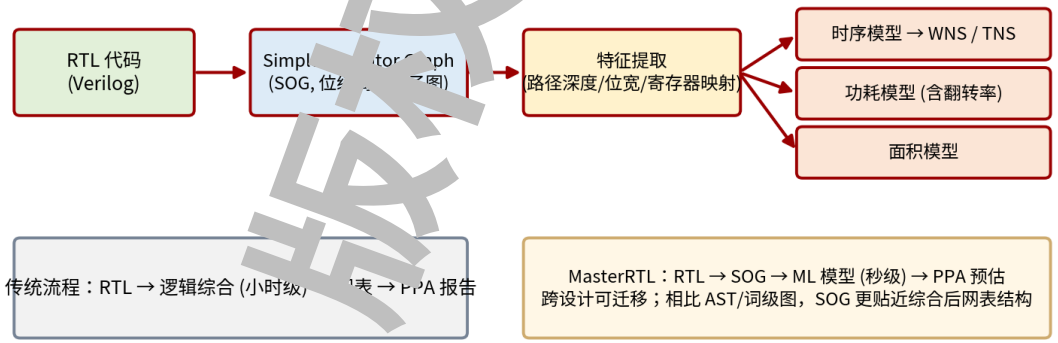
- 逐周期功耗 (PRIMAL / GRANNITE)
- RTL 功耗预估、IR-drop、DVFS

面积

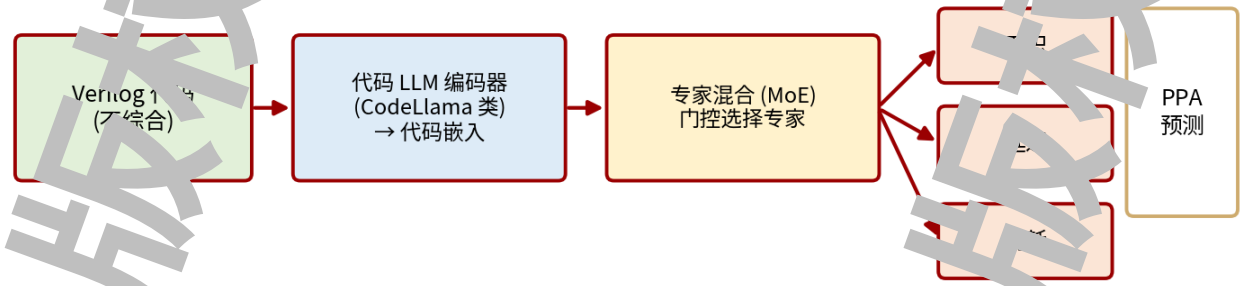
- RTL 面积预估、RL 逻辑综合 (DRiLLS)
- 布局密度 / 拥塞 (AlphaChip)、架构参数 (PRIME)

论文：MasterRTL 与 RocketPPA——RTL / 代码级 PPA 预估

- Fang et al., ICCAD 2023 / TCAD 2025；RocketPPA 2025 – 预测时序、功耗、面积，跨设计可迁移



→ 支撑早期架构探索、LLM 生成 RTL 的快速评估与优化闭环



无需 SOG/网表转换，直接从代码文本预测；训练于 LLM 生成 + 真实综合的数据集，据报道优于先前 RTL 级方法

MasterRTL

- SOG：HDL → 单比特与/或/非/M… 寄存器图，结构贴近综合后网表
- 时序导式提取寄存器对关键路径；功耗融合翻转率；WNS/TNS/功耗/面积相关系数普遍 > 0.9

RocketPPA

- 不做图转换，用代码 LLM 嵌入 Verilog 文本，MoE 头分别回归面积/延迟/功耗
- 训练集由 LLM 生成 RTL + 真实综合标签组成，据报道优于先前 RTL 级方法

硬件视角

- 使“写一版 RTL → 立即看 PPA”成为可能，是 LLM 生成 RTL 闭环的**评价函数**
- 数据稀缺是共同瓶颈：图生成、LLM 生成 RTL 做数据增强

延迟与功耗：GNN 时序预测与 ML 功耗估计

- TimingGCN (DAC 2022)；PRIMAL (DAC 2019)；GRANNITE (DAC 2020)

延迟：布线前时序预测

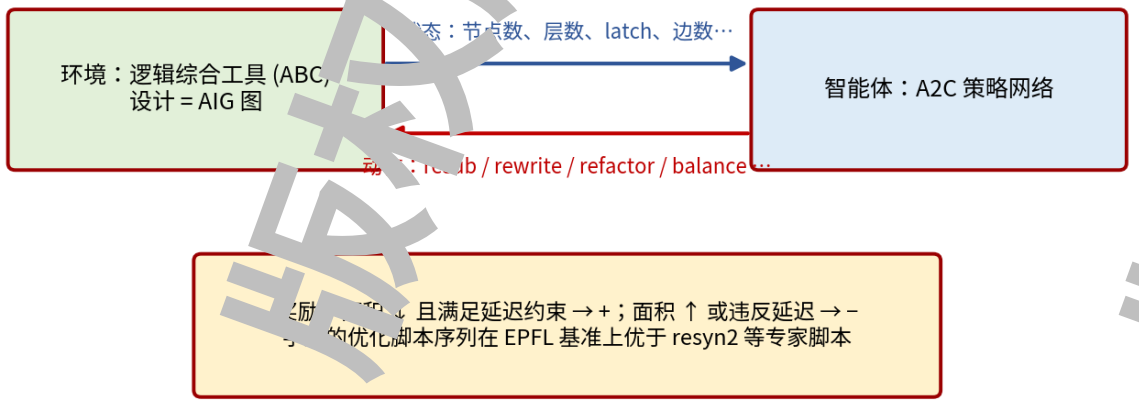
- 网表建成有向图，GNN 沿信号方向传播并预测每条线网延迟、到达时间与松弛；TimingGCN 在布线前即达接近 STA 的精度，速度快数十倍
- 价值：布局阶段的时序驱动优化、早期发现关键路径

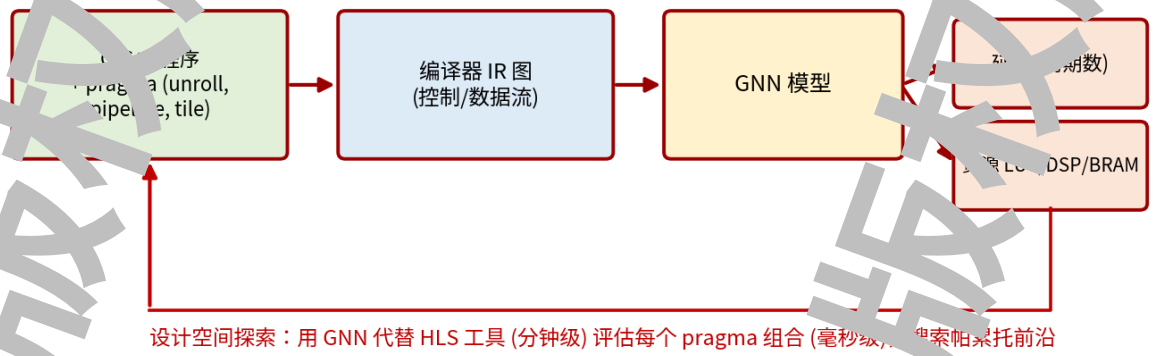
功耗：逐周期估计

- PRIMAL 把寄存器翻转映射成图像用 CNN 回归功耗；GRANNITE 用 GNN 在门级网表上推断翻转率，跨设计泛化
- 加速 50×以上、误差 < 5–10%，支撑 DVFS、热管理与 IR-drop 分析

面积与延迟优化：RL 逻辑综合与 HLS 设计空间探索

- DRiLLS (ASP-D[illegible]2020)；GNN-DSE (DAC 2022) — 学习“[illegible]化”而不只是“预测多少”





同类：Ansor/AutoTVM 的代价模型用于算子编译与调度；PRIME 用离线数据集直接优化加速器参数

DRiLLS：RL 搜索综合脚本

- 状[illegible] = AIG 统计量，动作 = ABC 变换 (re[illegible]te/resub/balance…)，奖励 = 面积下降且满[illegible]迟约束

  在 EPFL 基准上学到的序列优于 resyn2 等专家[illegible]本；同类：ABC-RL、Google 的 RL 综合

GNN-DSE：HLS pragma 探索

- 用 GNN 代价模型 (毫秒级) 代替 HLS 工具 (分钟级) 评估 unroll/pipeline/tile 组合，搜[illegible]迟-资源帕累托前沿
- 同一思想在编译器 (AutoTVM/Ansor 代价模型) 与物理设计 (AlphaChip 用 RL [illegible]元布局，Nature 2021) 中反复出现