从零构建大模型 (Sebastian Raschka, 塞巴斯蒂安·拉施卡, 覃立波, 冯晓骋, .pdf

- **视频课**
  - **Lec 1: Overview And Tokenization**

    https://stanford-cs336.github.io/spring2025-lectures/?trace=var/traces/lecture_01.json
    - 基于字节的编码有什么问题：long sequences (压缩比为1)
    - 基于词元的分词
    - BPE
  - **Lec 2: Pytorch, Resource Accounting**

    https://stanford-cs336.github.io/spring2025-lectures/?trace=var/traces/lecture_02.json
    - 计算
      - float32
      - float16
      - bfloat16
      - fp8
    - 成本计算
      - FLOPs：指的是浮点运算的次数 ，是一个用于衡量算法、模型或者计算任务计算量大小的指标。
      - FLOPS：表示每秒能够执行的浮点运算次数，是一个衡量硬件（如 CPU、GPU 等计算设备）计算性能的指标。它反映了计算设备在单位时间内处理浮点运算的能力 。
      - 样例：

        输入张量 x 的形状是 [B, D]（可理解为 B 个样本，每个样本有 D 个特征）

        权重矩阵 w 的形状是 [D, K]（D 个输入特征 → K 个输出特征的映射关系）

        输出张量 y 的形状是 [B, K]（B 个样本，每个样本输出 K 个结果）

        假设 B=2（2 个样本）、D=3（3 个输入特征）、K=2（2 个输出特征）：

        x = [[x_{00}, x_{01}, x_{02}], # 第0个样本 [x_{10}, x_{11}, x_{12}]] # 第1个样本 w = [[w_{00}, w_{01}], # 第0列：映射到第0个输出 [w_{10}, w_{11}], # 第1列：映射到第1个输出 [w_{20}, w_{21}]] 则 y` 的计算为：

        y [0][0] = x_{00}×w_{00} + x_{01}×w_{10} + x_{02}×w_{20} # x 第 0 行 × w 第 0 列

        y [0][1] = x_{00}×w_{01} + x_{01}×w_{11} + x_{02}×w_{21} # x 第 0 行 × w 第 1 列

        y [1][0] = x_{10}×w_{00} + x_{11}×w_{10} + x_{12}×w_{20} # x 第 1 行 × w 第 0 列

        y [1][1] = x_{10}×w_{01} + x_{11}×w_{11} + x_{12}×w_{21} # x 第 1 行 × w 第 1 列

        单个元素 y[i][k] 的运算量：计算点积时需要 D 次乘法（如 x[i][0]×w[0][k]）和 D-1 次加法（累加结果），约等于 2D 次浮点运算（FLOPs）。

        所有元素的总运算量：y 共有 B×K 个元素（B 行 K 列）。每个元素需要 2D 次运算（近似）。总运算量 = B×K × 2D = 2×B×D×K。

近似D：我们更关心 运算量的数量级（比如是百万级、十亿级还是万亿级），而非精确到个位的差异。当 D=32768 时，D-1=32767，和 D=32768 的差异仅为 1，占比约 0.003%，对整体数量级分析几乎无影响。

Linear model

As motivation, suppose you have a linear model.

- We have n points
- Each point is d-dimsional
- The linear model maps each d-dimensional vector to a k outputs

```
if torch.cuda.is_available():
    B = 16384  # Number of points
    D = 32768  # Dimension
    K = 8192   # Number of outputs
else:
    B = 1024
    D = 256
    K = 64


device = get_device()
x = torch.ones(B, D, device=device)
w = torch.randn(D, K, device=device)
y = x @ w
```

We have one multiplication (x[i][j] * w[j][k]) and one addition per (i, j, k) triple.

```
actual_num_flops = 2 * B * D * K  # @inspect actual_num_flops
```

- MFU
- 梯度

  https://www.bilibili.com/video/BV18m411S79t?spmidfrom=333.788.videopod.sections&vd_source=2467b9eb914331d7f163eb52ed0d1c22
- 前向传播与反向传播FLOPs运算
  - 前向传播计算：

    前向传播的运算包括 矩阵乘法的“乘 + 加”，公式推导：

    h1 = x @ w1：矩阵乘法需 B×D×D 次乘法 + B×D×D 次加法 → 共 2×B×D×D FLOPs。

    h2 = h1 @ w2：矩阵乘法需 B×D×K 次乘法 + B×D×K 次加法 → 共 2×B×D×K FLOPs。

    总前向 FLOPs：numforwardflops = (2 * B * D * D) + (2 * B * D * K)

```
def gradients_flops():
```

Let us do count the FLOPs for computing gradients.

Revisit our linear model

```
if torch.cuda.is_available():
    B = 16384  # Number of points
    D = 32768  # Dimension
    K = 8192   # Number of outputs
else:
    B = 1024
    D = 256
    K = 64


device = get_device()
x = torch.ones(B, D, device=device)
w1 = torch.randn(D, D, device=device, requires_grad=True)
w2 = torch.randn(D, K, device=device, requires_grad=True)
```

Model: x --w1--> h1 --w2--> h2 -> loss

```
h1 = x @ w1
h2 = h1 @ w2
loss = h2.pow(2).mean()
```

Recall the number of forward FLOPs: tensor_operations_flops

- Multiply x[i][j] * w1[j][k]
- Add to h1[i][k]
- Multiply h1[i][j] * w2[j][k]
- Add to h2[i][k]

```
num_forward_flops = (2 * B * D * D) + (2 * B * D * K)  # @inspect num_forward_flops
```

- 反向传播计算：

  反向传播通过链式法则计算梯度（d(loss)/d(w1)、d(loss)/d(w2) 等），核心是对每个参数的梯度计算过程统计 FLOPs。

  示例中的模型是一个简单的两层线性网络

  x --> w1 --> h1 --> w2 --> h2 --> loss

  对应的计算步骤：

  第一层：h1 = x @ w1

  第二层：h2 = h1 @ w2

  损失：loss = h2.pow(2).mean()

反向传播的核心是链式法则：梯度需要从损失“反向传递”到所有参数和输入。对于每个前向矩阵乘法 C = A×B：要计算对权重 B 的梯度，要计算对输入 A 的梯度（用于传递给前一层）

w2.grad：h1^T @ ∂L/∂h2 → 2×B×D×K（与 h1@w2 的前向 FLOPs 相等）

h1.grad：∂L/∂h2 @ w2^T → 2×B×D×K（另一次矩阵乘法，与前向相等）

w1.grad：x^T @ ∂L/∂h1 → 2×B×D×D（与 x@w1 的前向 FLOPs 相等）

x.grad：∂L/∂h1 @ w1^T → 2×B×D×D（另一次矩阵乘法，与前向相等）

How many FLOPs is running the backward pass?

```
h1.retain_grad()  # For debugging
h2.retain_grad()  # For debugging
loss.backward()
```

Recall model: x --w1--> h1 --w2--> h2 -> loss

- h1.grad = d loss / d h1
- h2.grad = d loss / d h2
- w1.grad = d loss / d w1
- w2.grad = d loss / d w2

Focus on the parameter w2.
Invoke the chain rule.

```
num_backward_flops = 0  # @inspect num_backward_flops
```

w2.grad[j,k] = sum_i h1[i,j] * h2.grad[i,k]

```
assert w2.grad.size() == torch.Size([D, K])
assert h1.size() == torch.Size([B, D])
assert h2.grad.size() == torch.Size([B, K])
```

For each (i, j, k), multiply and add.

```
num_backward_flops += 2 * B * D * K  # @inspect num_backward_flops
```

h1.grad[i,j] = sum_k w2[j,k] * h2.grad[i,k]

```
assert h1.grad.size() == torch.Size([B, D])
assert w2.size() == torch.Size([D, K])
assert h2.grad.size() == torch.Size([B, K])
```

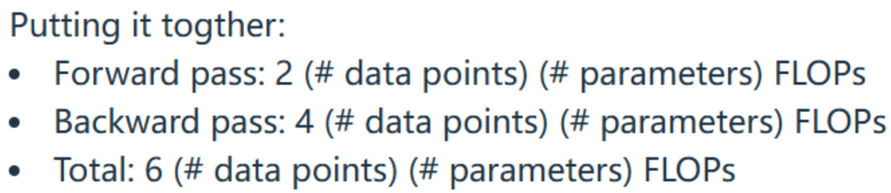
For each (i, j, k), multiply and add.

```
num_backward_flops += 2 * B * D * K  # @inspect num_backward_flops
```

This was for just w2 (D*K parameters).
Can do it for w1 (D*D parameters) as well (though don't need x.grad).

```
num_backward_flops += (2 + 2) * B * D * D  # @inspect num_backward_flops
```

- 结论：反向传播的 FLOPs 是前向的 2 倍



- 模型构建
  - 参数初始化
- **Lec 3: Architectures, Hyperparameters**
  - 架构变体
    - 前置归一化和后置归一化
      - 前置归一化
        - 梯度大小保持恒定
      - 后置归一化
        - 梯度爆发增长
      - 为什么残差中的层归一化不好
        - 添加层归一化破坏了梯度传播过程中的恒等连接
        - 采用预归一化，至少将层归一化放到残差流之外，使梯度传播更顺畅，训练过程更稳定
      - 层归一化和均方根归一化
      - 是否存在更具未来适应性的方案
        - ①保持直接恒等映射的残差连接
        - ②防止激活值漂移和规模失控
        - ③考虑模型架构对系统组件的影响
    - 激活函数
      - ReLU
      - GeLU
      - 门控线性单元（*GLU）
    - 串行还是并行
      - 串行：先进行注意力计算再执行MLP
      - 并行：同时计算注意力和MLP，然后将他们相加到一起进入残差流中
      - 串行计算是否比并行计算更加高效
        - 并行比串行更加高效，并行的优势在于，如果编写了正确类型的融合内核，可以并行完成很多操作（计算可以在不同的并行部分之间共享）
    - 位置嵌入的变体

- RoPE旋转位置嵌入
- 角度旋转会不会给训练带来困难
  - 旋转本身不会造成任何问题，旋转的角度θ是固定的，绝对位置矩阵是固定的，实际上知识一个固定矩阵与向量相乘（将旋转理解为是一个矩阵乘法）

- 超参数选择
  - FFN
    - 未使用GLU的情况下d ff通常为4倍的d model，使用GLU的情况下d ff通常为2.66倍左右的d model

Exception #1 – GLU variants

Remember that GLU variants scale down by 2/3rd. This means most GLU variants have $d_{ff} = \frac{8}{3} d_{model}$. This is mostly what happens. Some notable such examples.

Model	d_{ff}/d_{model}
PaLM	4
Mistral 7B	3.5
LLaMA-2 70B	3.5
LLaMA 70B	2.68
Qwen 14B	2.67
DeepSeek 67B	2.68
Yi 34B	2.85
T5 v1.1	2.5

Models are roughly in this range, though PaLM, LLaMA2 and Mistral are slightly larger

- 这样的比例与模型整体影响之间存在什么关系
  - 比例实际上控制着模型特征拓展的“宽度”，即中间层（隐藏层）的维度

  - ‘模型维度’与‘头维度乘以头数’之间的比例
    - Multi-Head Attention 设计的核心约束（大部分模型严格遵循比例 = 1）

How many heads, whats the model dim?

Some examples of this hyperparameter

	Num heads	Head dim	Model dim	Ratio
GPT3	96	128	12288	1
T5	128	128	1024	16
T5 v1.1	64	64	4096	1
LaMDA	128	128	8192	2
PaLM	48	258	18432	1.48
LLaMA2	64	128	8192	1

Most models have ratios around 1 – notable exceptions by some google models.

- 长宽比 aspect ratios
  - 通常每层设置约128个隐藏维度是比较理想的选择

Aspect ratios

Should my model be deep or wide? *How* deep and how wide?

Most models are surprisingly consistent on this one too!

	Model	d_{model}/n_{layer}
	BLOOM	205
	T5 v1.1	171
Sweet spot?	PaLM (540B)	156
Sweet spot?	GPT3/OPT/Mistral/Qwen	128
Sweet spot?	LLaMA / LLaMA2 / Chinchila	102
	T5 (11B)	43
	GPT2	33

  - 为什么考量长宽比
    - 长宽比控制着并行度的规模
- 词汇量规模选择
  - 早期单语种模型词汇量范围3-5万，多语言模型趋向采用的词汇量规模在10-25万，多数研究表明词汇量规模约10-20万较为合适

What are typical vocabulary sizes?

Monolingual models – 30-50k vocab

Model	Token count
Original transformer	37000
GPT	40257
GPT2/3	50257
T5/T5v1.1	32128
LLaMA	32000

Multilingual / production systems 100-250k

Model	Token count
mT5	250000
PaLM	256000
GPT4	100276
Command A	255000
DeepSeek	100000
Qwen 15B	152064
Yi	64000

Monolingual vocabs don't need to be huge, but multilingual ones do

- 多语言词汇表是否有助于提升单一语言的表现性能
  - 多语言词汇表有助于提升单一语言表现，核心逻辑是跨语言知识迁移与语义泛化
- dropout和其他类型的正则化
  - 早期模型较小时使用dropout较多，新型的模型规模较大，因此大都使用权重衰减

Dropout and weight decay in practice

Model	Dropout*	Weight decay
Original transformer	0.1	0
GPT2	0.1	0.1
T5	0.1	0
GPT3	0.1	0.1
T5 v1.1	0	0
PaLM	0	(variable)
OPT	0.1	0.1
LLaMA	0	0.1
Qwen 14B	0.1	0.1

Many older models used dropout during pretraining

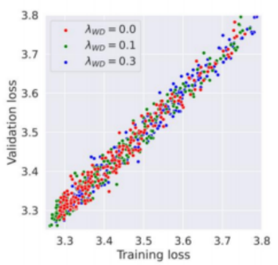
Newer models (except Qwen) rely only on weight decay

* Most of the times papers just don't discuss dropout. On open models, this closely matches not doing dropout. This may not be true of closed models.

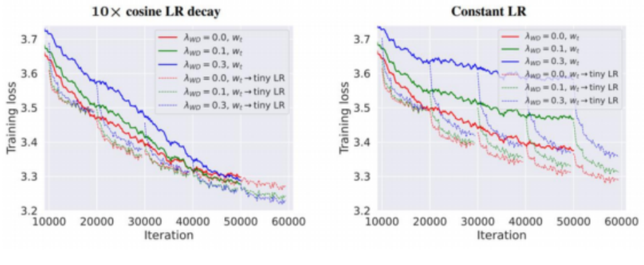
  - 进行权重衰减并不是为了正则化模型，实则是为了获得更好的训练损失

Why weight decay LLMs?

[Andriushchenko et al 2023] has interesting observations about LLM weight decay



It's not to control overfitting



Weight decay interacts with learning rates (cosine schedule)

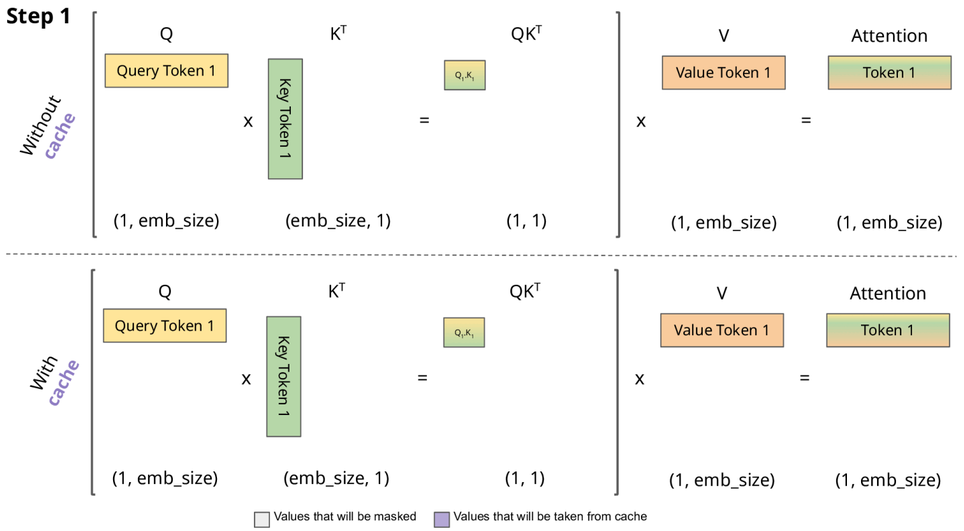
- 传统认知下，Weight Decay（权重衰减，类似 L2 正则化）的核心作用是 正则化：通过对权重施加惩罚（让权重尽可能小），抑制模型过拟合（尤其是小模型 + 小数据时，防止参数记住噪声）。此时，Weight Decay 的价值是“牺牲一点训练损失，换取更好的泛化（验证 / 测试损失）”。
- 如图左一，横轴是训练损失，纵轴是验证损失。不同 Weight Decay 系数下，验证损失与训练损失的趋势高度一致（而非传统认知中“Weight Decay 降低验证损失但可能增加训练损失”）。
- 如图 中，开启 Weight Decay）的模型，训练损失下降更快、最终更低。在大模型中，Weight Decay 与学习率调度（如 Cosine 退火）产生协同作用，让模型参数更新更“平滑”，避免学习率震荡导致的训练损失波动。
- 如图右一，即使学习率固定（不调度），开启 Weight Decay 的模型（绿色 / 蓝色曲线）训练损失仍比不开启的（红色曲线）更低、更稳定。Weight Decay 本质是对权重更新的“梯度修正”，这种修正让大模型的参数更新更“有方向感”，加速训练损失下降。

- 当人们转向多模态架构时，在超参数的选择上会有架构差异吗
  - 一般采用浅层(（如仅用少量层处理跨模态信息）)、后期模态融合(如特征拼接后用注意力融合)、早期模态融合（如输入层拼接）方式，具体实现方式是 eg:将视觉模态附加到现有语言模型上，这类情况下超参数和架构的选择都是固定的
- 稳定性技巧
  - Z-loss：促使输出层的softmax归一化因子保持良好行为
  - QK norm：控制注意力层的softmax的输入大小保持在一定的范围内
    - 在训练过程中应用了层归一化，在推理时候还会使用层归一化吗
      - 会，因为层归一化已经学习到了相关参数，如果去掉，模型将完全不知道如何进行处理
  - 进行软性限制 ： 软性上限处理
- 注意力头的变体
  - GQA和MQA

- KV cache ： 在自回归生成过程中，缓存已生成 token 对应的 Key（K）和 Value（V），避免重复计算；每次生成新 token 时，仅需计算当前 token 的 Query（Q），并与缓存的历史 K、V 交互，从而优化效率。

  动画中显示的内容是：首先处理当前步骤的 Query（Q，对应即将生成的新 token 的查询向量），以此为条件，从过往缓存的 K 和 V 中检索信息来匹配这个 Q。生成的 token 会从 1 逐渐递增到 N，因为每次只能生成一个新的 token。通过积累所有过往 token 的 K 来构建键缓存，过往 token 的 K 和 V 不会发生改变（因为它们仅依赖于先前的信息），因此在生成 token 的过程中会逐步积累这些历史的键值对，每次只需计算 Q 的一个新元素。因此这个注意力矩阵会是一个下三角矩阵，每次计算一行，该行正是生成下一个 token 所需要的全部注意力结果。

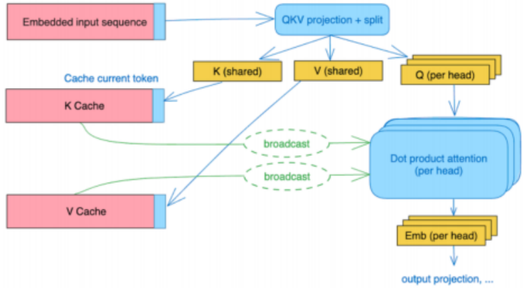
  https://medium.com/@joaolages/kv-caching-explained-276520203249



- 当我们生成文本时，必须先生成一个token，然后转换器才能读取该token，接着处理这些信息，此时可以获取下一个词元的概率分布，然后我们以自回归的方式逐个词元进行处理。因此无法进行并行运算，生成新词元时，只能逐步计算注意力。
- MQA：频繁的内存访问严重制约了系统的吞吐量表现 使KV cache面临推理成本的权衡，催生出MQA，这种结构设置了多个查询头，但键和值仅保留单个头。大幅减少了 KV cache 的内存占用和需要传输的数据量，从而降低内存访问频率。

MQA – just have fewer key dimensions.

Key idea – have multiple queries, but just one dimension for keys and values



We have much fewer items to move in and out of memory (KV Cache)

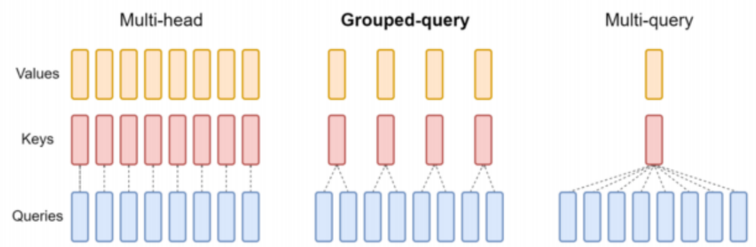
Total memory access $(bnd + bn^2k + nd^2)$, **Arithmetic intensity** $O\left(\left(\frac{1}{d} + \frac{n}{dh} + \frac{1}{b}\right)^{-1}\right)$

[figure from https://blog.fireworks.ai/multi-query-attention-is-all-you-need-db072e758055]

- GQA ： 不再采用多查询头与单一键值头的模式，而是让多个查询头共享一组键值头（即键值头数量少于查询头但多于 1），通过选择合适的键值头数量，实现推理效率与模型表达能力之间的权衡。

Recent extension – GQA

Don't go all the way to one dimension of KV – have fewer dims



Simple knob to control expressiveness (key-query ratio) and inference efficiency

- 滑动窗口注意力
  - 在每层中，每个 token 仅能关注自身周围固定窗口大小（局部感受野，如窗口大小为 k）的 token；随着网络层数加深，上层 token 的感受野会通过下层的窗口关联逐步扩大，有效感受野约为局部感受野乘以层数（更精确地说，是每层窗口范围的叠加，例如窗口大小 k、层数 L 时，有效感受野约为 k×L）。
- Lec 4: Mixtrue Of Experts

- Lec 5: Gpus
- Lec 6: Kernels, Triton
- Lec 7: Parallelism 1
- Lec 8: Parallelism 2
- Lec 9: Scaling Laws 1
- Lec 10: Inference
- Lec 11: Scaling Laws 2
- Lec 12: Evaluation
- Lec 13: Data 1
- Lec 14: Data 2
- Lec 15: Alignment - Sft/Rlhf
- Lec 16: Alignment - Rl 1
- Lec 17: Alignment - Rl 2

课本章节