CS336 Notes

1 Overview

> CS336 Spring 2025

---

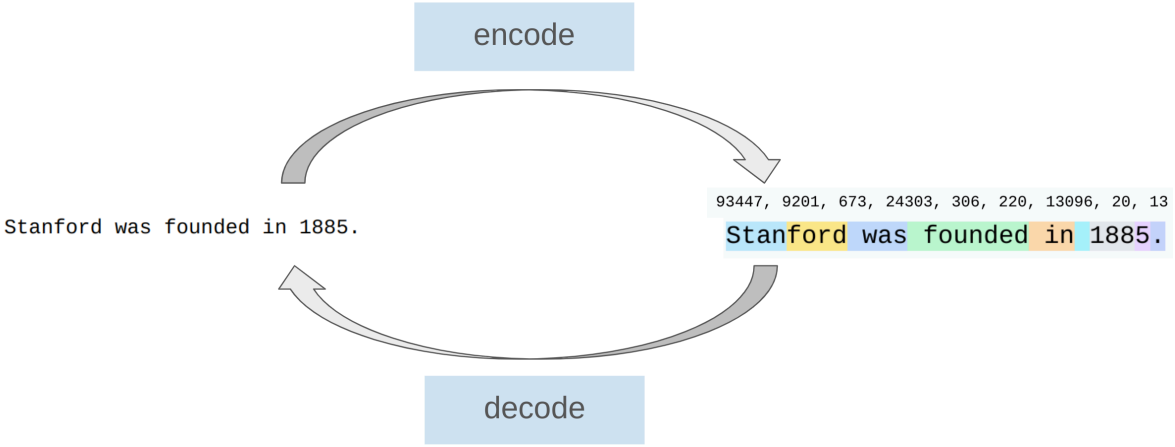
2 Tokenize

A language model places a probability distribution over sequences of tokens (usually integer indices)

> **Example**
>
> `hello, world! 你好!` => `[15496, 11, 995, 0]`

Tokenizer 就是完成将 Strings Encode（编码）到 Tokens 和将 Tokens Decode（解码） 回 Strings 的工具/Class。主要包括两个过程：分词+编码映射

其中字典大小（Vocabulary Size）就是可能出现的所有不同 Tokens 的数量。



> Tokenizer interactive site

2.1 Observations

- A word and its preceding space are part of the same token (e.g., " world").
- A word at the beginning and in the middle are represented differently (e.g., "hello hello").
- Numbers are tokenized into every few digits.

2.2 Character-Base Tokenizer

一个句子（Strings）必然由多个字母组成，对每个字母独立编码，就是 Character-based Tokenization。

Problems:

- 字典会非常大
- characters 的出现频率不同，效率差

2.3 Byte-based Tokenization

将所有字符看作字节（UTF-8）进行编码

- 字典很小（256）
- Problems：压缩太差，Token 序列会很长

2.4 Word-based Tokenization

将句子中每个单词独立映射成 Tokens。

- 需要很大的字典（number of words is huge）
- 很多单词出现频率很低，模型可能不认识他们
- 字典更新很麻烦（出现新的单词 word 不在训练中，模型容易困惑）

2.5 Byte Pair Encoding

在原始文本上训练 Tokenization，自由地处理句子切分
一开始将每个字节作为 Token，后续将经常出现的 Token 组合（pair）合并

---

3 PyTorch & Resource Accounting

Lecture Materials

How long would it take to train a 70B parameter model on 15T tokens on 1024 H100s?

$$\text{total} = 6 \times 70e9 \times 15e12$$
$$\text{flops\_per\_day} = 1979e12/2 \times \text{mfu} \times 1024 \times 60 \times 60 \times 24$$
$$\text{days} = \frac{\text{total}}{\text{flops\_per\_day}} = 143.927$$

What's the largest model that can you can train on 8 H100s using AdamW (naively)?

```
h100_bytes = 80e9
bytes_per_parameter = 4 + 4 + (4 + 4) # parameters, gradients, optimizer state @inspect bytes_per_parameter
num_parameters = (h100_bytes * 8) / bytes_per_parameter
```

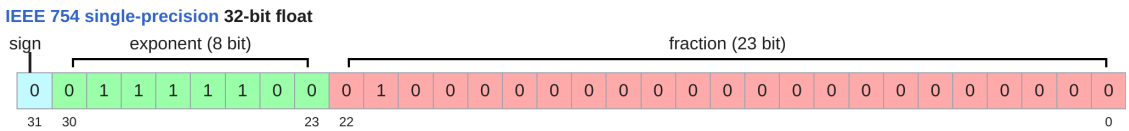
✓ **40, 000, 000, 000 Float32 Parameters**

P.S. Default data type is Float32. Maybe we can try mixed precision train like BF16, but we still need keep an extra float32 copy of the parameters. It has nothing to do with memory, it just speeds up.

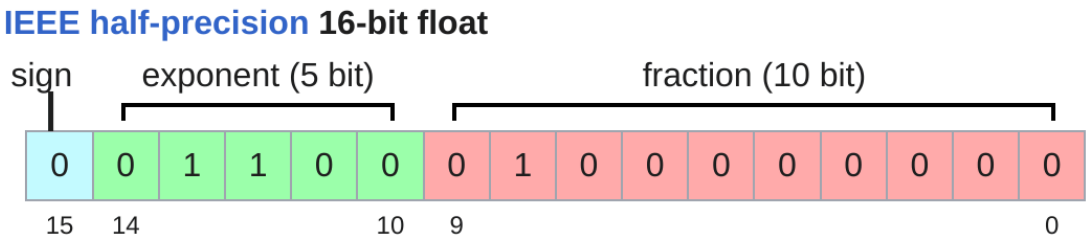
3.1 Floats

- **float32**: default. (fp32, single precision)

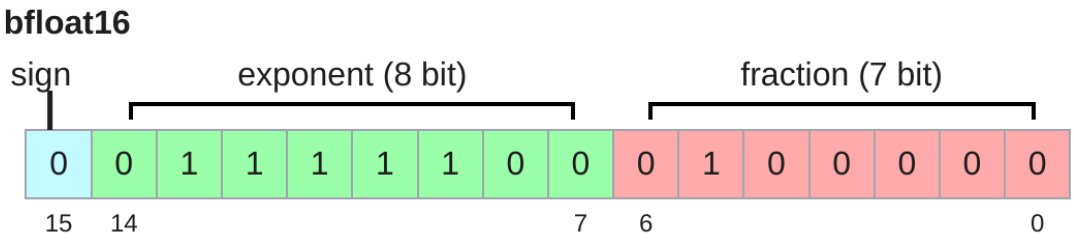
  - double precision (float64)



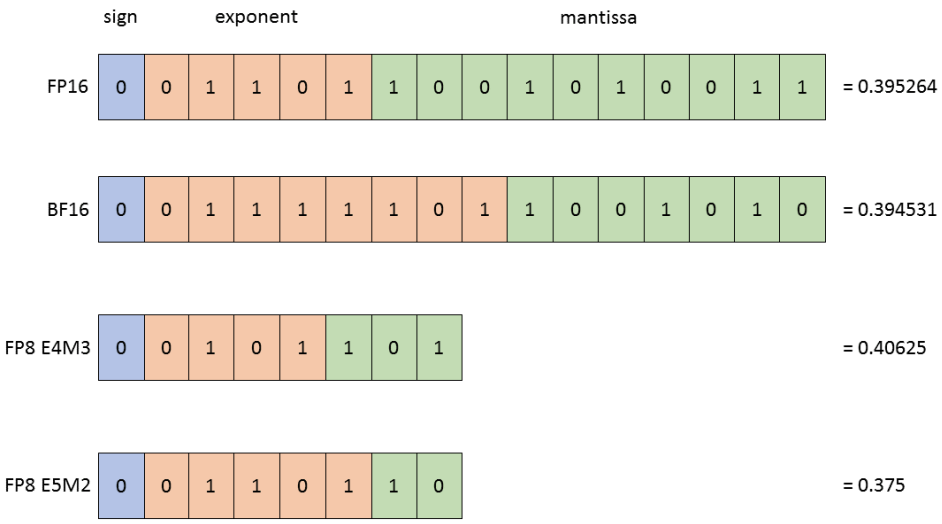
- **float16** (fp16, half precision)



- **bfloat16**: uses the same memory as float16 but has the same dynamic range as float32!
  - The only catch is that the resolution is worse, but this matters less for deep learning.（唯一的问题是精度/分辨率会差一点，但这在深度学习领域并不重要）



- **fp8**

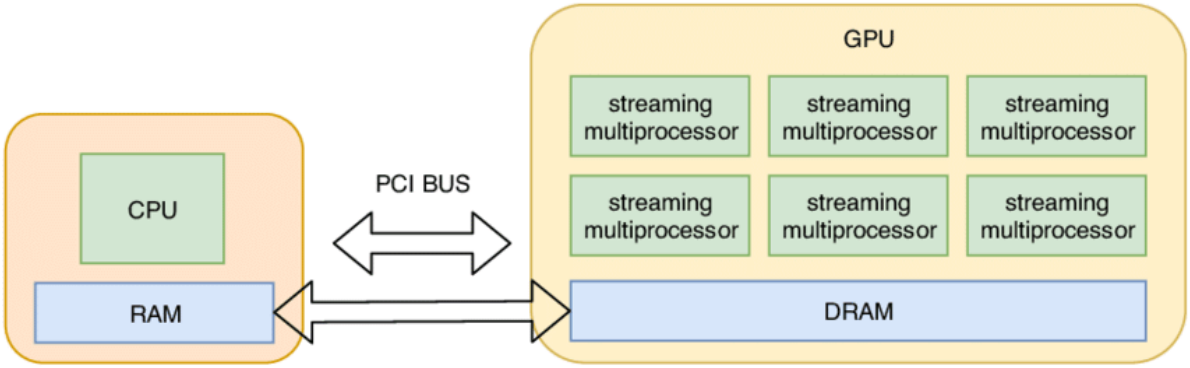


- Mixed Precision Training

3.2 PyTorch Skills

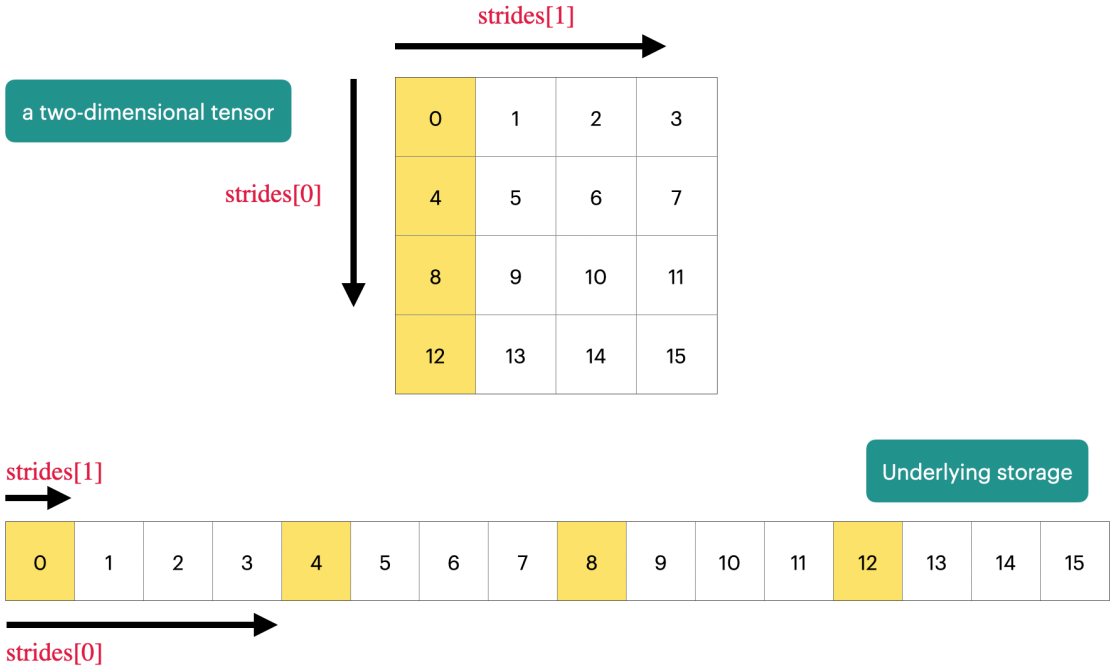
In order to take advantage of the massive parallelism of GPUs, we need to move them to GPU memory.

```
torch.cuda.is_available()
y = x.to("cuda:0")
```



3.2.1 PyTorch Tensors

PyTorch Tensors is pointer into allocated continuous block of memory (row major).

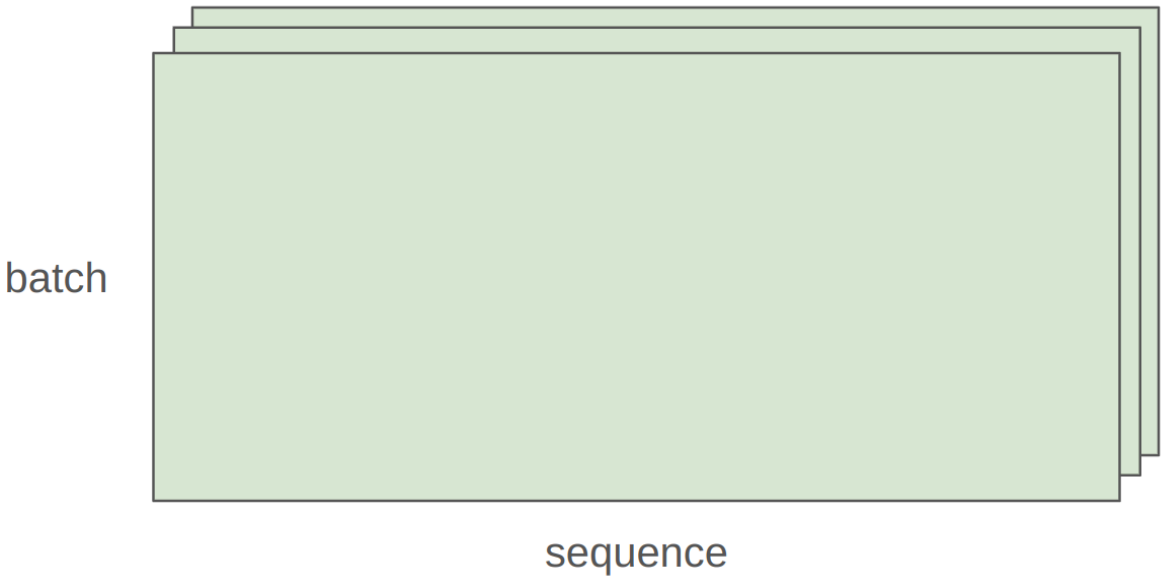


Many operations simply provide a different **view** of the tensor.
Note that some views are non-contiguous entries (**not stored in the logical order you see**), which means that further views aren't possible.

```
x = torch.tensor([[1., 2, 3], [4, 5, 6]]) # @inspect x
y = x.transpose(1, 0) # @inspect y
```

```
assert not y.is_contiguous()
try:
    y.view(2, 3)
    assert False
except RuntimeError as e:
    assert "view size is not compatible with input tensor's
size and stride" in str(e)
```

In general, we perform operations for every example in a batch and token in a sequence.




```
y = x @ w
```

3.2.2 PyTorch Einops

Einops is a library for manipulating tensors where dimensions are named.

```
x = torch.ones(2, 2, 3) # batch, sequence, hidden
y = torch.ones(2, 2, 3) # batch, sequence, hidden
z = x @ y.transpose(-2, -1) # batch, sequence, sequence
```

- jaxtyping

```
# old way
# x = torch.ones(2, 2, 1, 3)
```

```
x: Float[torch.Tensor, "batch seq heads hidden"] =
torch.ones(2, 2, 1, 3)
```

- einops einsum

```
# old way
# z = x @ y.transpose(-2, -1)
z = einsum(x, y, "batch seq1 hidden, batch seq2 hidden ->
batch seq1 seq2")
```

- einops reduce (e.g., sum, mean, max, min)

```
# old way
# y = x.mean(dim=-1)
y = reduce(x, "... hidden -> ...", "sum")
```

- einops rearrange

```
x = rearrange(x, "... (heads hidden1) -> ... heads
hidden1", heads=2)
x = einsum(x, w, "... hidden1, hidden1 hidden2 -> ...
hidden2")
```

3.2.3 Randomness

```
# Torch
seed = 0
torch.manual_seed(seed)

# NumPy
import numpy as np
np.random.seed(seed)

# Python
import random
random.seed(seed)
```

3.3 Computation

- FLOPs: floating-point operations.
- FLOP/s: floating-point operations per second (FLOPS).

Forward & Backward & Update with Batch Size

- Forward Pass: Computes the model outputs. $\propto batch\_size$
- Backward Pass: Computes gradients. $\propto batch\_size$
- Parameter update = Weight update: Updates model parameters using gradients (Optimizer's job). **Stay same** while batch size scaling.

a linear layer (fully connected layer)

$$y = xW,\ x \in \mathbb{R}^{B\times D}, W \in \mathbb{R}^{D\times K}$$

3.3.1 Forward

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

```
# (B, D) @ (D, K)
for i in range(B):
    for j in range(D):
        for k in range(K):
            y[i][k] += x[i][j] * w[j][k] # two FLOPs
```

So in one matrix multiplication we need $2 \times B \times D \times K$ FLOPs.

- elementswise operation needs $O(mn)$ FLOPs.
    - Addition of two $m \times n$ matrices requires $m \times n$ FLOPs.
- FLOP/s depends on **hardware** and **data type** (e.g. BF16 vs Float32)
- Model FLOPs Utilization(MFU): (actual FLOP/s) / (promised FLOP/s)
    - ignore communication/overhead

- = 0.5 is very good.

No other operation that you'd encounter in deep learning is as expensive as matrix multiplication

3.3.2 Backward & Gradients Computation

> Backward Zhihu

In the backward pass, we compute the gradients of the loss L with respect to the weights W, the input x.

1. The first step is to compute the gradient of the loss function L with respect to each output y, i.e., $\frac{\partial L}{\partial y}$.

For example, assume we are using Mean Squared Error (MSE) as the loss function:

$$L = \frac{1}{2} \sum_{i=1}^{B} \|y_i - \hat{y}_i\|^2$$

$$\frac{\partial L}{\partial y_i} = y_i - \hat{y}_i$$

where $\hat{y}_i$ is the true label.

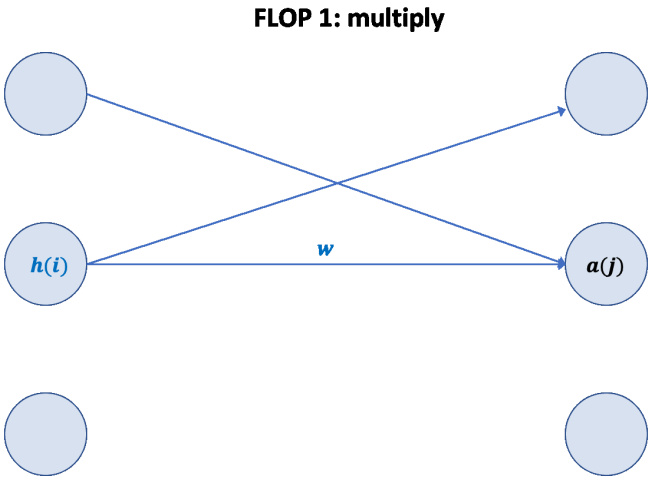
- $B \times K$ FLOPs ~~(ignore)~~

2. Gradient with Respect to the Weights and Inputs.

$$\frac{\partial L}{\partial W} = x^T \cdot \frac{\partial L}{\partial y}, \ x^T \in \mathbb{R}^{D \times B}, \frac{\partial L}{\partial y} \in \mathbb{R}^{B \times K}$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot W^T, \ W^T \in \mathbb{R}^{K \times D}$$

- $2 \times D \times B \times K + 2 \times B \times K \times D$ FLOPs



3. Gradient with Respect to the Bias.

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}$$

- $B \times K$ FLOPs (~~Reuse. ignore~~)

4. Optimizer (e.g. Adam): Formula Derivation

- $D \times K$ = (#parameters) FLOPs (~~ignore~~)

Summary:

- Forward pass: 2 (#data points) (#parameters) FLOPs
- Backward pass: 4 (#data points) (#parameters) FLOPs
- Total: 6 (#data points) (#parameters) FLOPs

3.4 Parameter Initialization & Data Loading

3.4.1 nn.Parameter()

```
w = nn.Parameter(torch.randn(input_dim, output_dim) /
np.sqrt(input_dim))
```

Large values can cause gradients to blow up and cause training to be unstable. So, we simply rescale by $1/\sqrt{\text{input\_dim}}$.

3.4.2 Data Loader

Don't want to load the entire data into memory at once.
so use `memmap` to lazily load only the accessed parts into memory.

```
data = np.memmap("data.npy", dtype=np.int32)
```

A **data loader** generates a **batch** of sequences for training.

```
start_indices = torch.randint(len(data) - sequence_length,
(batch_size,))

x = torch.tensor([data[start:start + sequence_length] for
start in start_indices])
```

3.4.3 Pin Memory

By default, PyTorch CPU tensors are stored in **paged memory**. Because of the OS's paging mechanism, this memory may be swapped out to disk.
However, that also means the GPU **cannot use direct memory access (DMA)** efficiently to read from it, which slows down (CPU => GPU) data transfers.
So `pin_memory()` moves the tensor `x` from normal paged memory to **page-locked (pinned) memory**, which allows us to copy `x` from CPU into GPU **asynchronously**.

```
if torch.cuda.is_available():
    x = x.pin_memory()

x = x.to(device, non_blocking=True)
```

This allows us to do two things in parallel (not done here):

- Fetch the next batch of data into CPU
- Process `x` on the GPU.

3.5 Optimizer

- momentum = SGD + exponential averaging of grad
- AdaGrad = SGD + averaging by grad^2
- RMSProp = AdaGrad + exponentially averaging of grad^2
- Adam = RMSProp + momentum

```
optimizer = AdaGrad(model.parameters(), lr=0.01)
optimizer.step()
optimizer.zero_grad(set_to_none=True) # free up memory
```

3.6 Memory

```
# B points x D dimension
# Parameters
num_parameters = (D * D * num_layers) + D
assert num_parameters == get_num_parameters(model)

# Activations: element wise
num_activations = B * D * num_layers

# Gradients
num_gradients = num_parameters

# Optimizer states
num_optimizer_states = num_parameters

# Putting it all together, assuming float32
total_memory = 4 * (num_parameters + num_activations +
num_gradients + num_optimizer_states)
```

3.7 Checkpoint

During training, it is useful to periodically save your model and optimizer state to disk.

```
checkpoint = {
    "model": model.state_dict(),
    "optimizer": optimizer.state_dict(),
}
torch.save(checkpoint, "model_checkpoint.pt")

# Load the checkpoint:
loaded_checkpoint = torch.load("model_checkpoint.pt")
```

3.8 Mixed Precision Training

> https://arxiv.org/pdf/1710.03740.pdf

Choice of data type (float32, bfloat16, fp8) have tradeoffs.

- Higher precision: more accurate/stable, more memory, more compute
- Lower precision: less accurate/stable, less memory, less compute

Solution: use float32 by default, but use {bfloat16, fp8} when possible.
A concrete plan:

- Use {bfloat16, fp8} for the forward pass (activations).
- Use float32 for the rest (parameters, gradients).

PyTorch has an automatic mixed precision (AMP) library.
https://pytorch.org/docs/stable/amp.html
https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/

NVIDIA's Transformer Engine supports FP8 for linear layers.

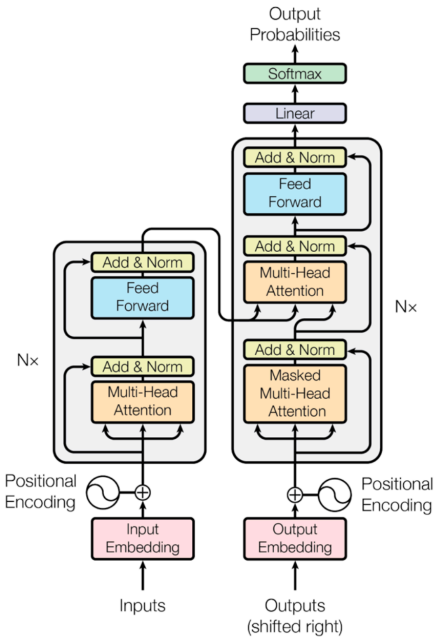
---

4 Architectures

- Pre vs Post Norm
- Layer Norm vs RMS Norm
    - RMS Faster and cheaper in memory: FLOPs is not Runtime
    - drop bias

- Activation: ReLU, SwiGLU, GeGLU
- Serial vs Parallel Layers
- Position Embeddings: Rope ???
- hyperparameters:
    - d_{ff}, d_{model}
    - head
    - vocabulary size
    - dropout and regularization: weight decay interact with learning rate.
- softmax stability: z-loss, attention QK softmax, logit soft-capping
- attention heads: GQA/MQA, Sparse/sliding window attention

4.1 Original Transformers



Review: choices in the standard transformer

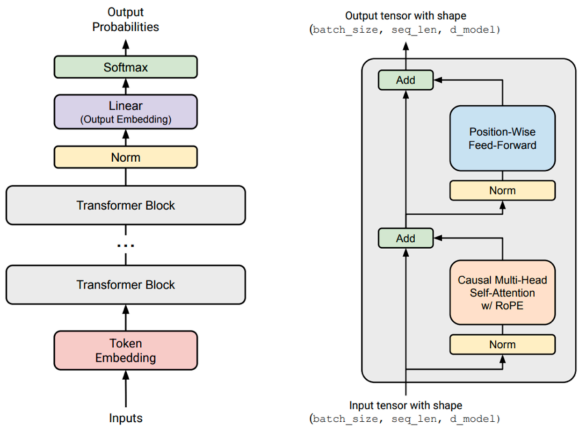
Position embedding: sines and cosines

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

FFN: ReLU

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Norm type: post-norm, LayerNorm



Differences:

- **LayerNorm** is in front of the block
- **Rotary position embeddings (RoPE)**
- FF layers use **SwiGLU**, not ReLU
- Linear layers (and layernorm) have **no bias** (constant) terms

Why did we pick these?
What should you pick?

- Pre-Norm
- **RoPE**
- SwiGLU
- ...

We will talk through many major architecture and hyperparameter variants.

4.2 Pre vs Post Norm

The one thing *everyone* agrees on (in 2024)

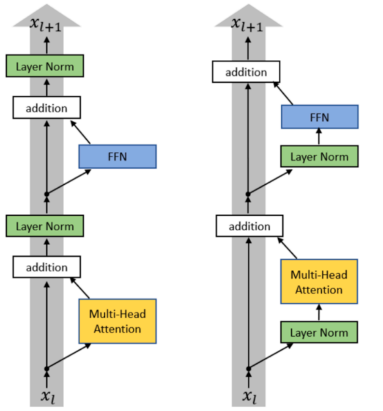


Figure from Xiong 2020

Post-LN Transformer	Pre-LN Transformer
$x_{l,i}^{post,1} = \text{MultiHeadAtt}(x_{l,i}^{post}, [x_{l,1}^{post}, \cdots, x_{l,n}^{post}])$	$x_{l,i}^{pre,1} = \text{LayerNorm}(x_{l,i}^{pre})$
$x_{l,i}^{post,2} = x_{l,i}^{post} + x_{l,i}^{post,1}$	$x_{l,i}^{pre,2} = \text{MultiHeadAtt}(x_{l,i}^{pre,1}, [x_{l,1}^{pre,1}, \cdots, x_{l,n}^{pre,1}])$
$x_{l,i}^{post,3} = \text{LayerNorm}(x_{l,i}^{post,2})$	$x_{l,i}^{pre,3} = x_{l,i}^{pre} + x_{l,i}^{pre,2}$
$x_{l,i}^{post,4} = \text{ReLU}(x_{l,i}^{post,3} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l}$	$x_{l,i}^{pre,4} = \text{LayerNorm}(x_{l,i}^{pre,3})$
$x_{l,i}^{post,5} = x_{l,i}^{post,3} + x_{l,i}^{post,4}$	$x_{l,i}^{pre,5} = \text{ReLU}(x_{l,i}^{pre,4} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l}$
$x_{l+1,i}^{post} = \text{LayerNorm}(x_{l,i}^{post,5})$	$x_{l+1,i}^{pre} = x_{l,i}^{pre,5} + x_{l,i}^{pre,3}$
	Final LayerNorm: $x_{Final,i}^{pre} \leftarrow \text{LayerNorm}(x_{L+1,i}^{pre})$

Set up LayerNorm so that it doesn't affect the main residual signal path (on the left)

Almost all modern LMs use pre-norm (but BERT was post-norm)

Almost all modern LMs use Pre-Norm

why?

1. Intuition – keep the good parts of residual connections
2. Observations – nicer gradient propagation, fewer spike
3. stability and larger LRs for large networks

4.2.1 RMSNorm

- LayerNorm

$$BatchNorm(\boldsymbol{x}) = \gamma \odot \frac{\boldsymbol{x} - \hat{\boldsymbol{\mu}}_B}{\hat{\boldsymbol{\sigma}}_B} + \beta$$

$$\hat{\boldsymbol{\mu}}_B = \frac{1}{|B|} \sum_{x \in B} x$$

$$\hat{\boldsymbol{\sigma}}_B = \frac{1}{|B|} \sum_{x \in B} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_B)^2 + \epsilon$$

- Root Mean Square Normalization (RMSNorm)

$$RMSNorm(\boldsymbol{x}) = \frac{\boldsymbol{x}}{\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2 + \epsilon}} \odot \boldsymbol{\gamma}$$

or

$$RMSNorm(\boldsymbol{x}) = \frac{x}{\sqrt{\|x\|_2^2 + \epsilon}} \odot \gamma$$

> $\|x\|_2^2 = \sum_{i=1}^{D} x_i^2$ is Squared L2 norm

Why?

1. it's faster: Fewer operations (no mean calculation), Fewer parameters (no bias term to store)
2. In practice, works as well as LayerNorm

4.2.2 Drop bias terms

Original Transformers:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Modern Transformer:

$$FFN(x) = \sigma(xW_1)W_2$$

Why?

Memory (similar to RMSnorm) and optimization stability.
People more generally drop bias terms since the compute/param tradeoffs are not great.

4.3 Activations

4.3.1 ReLU

$$FF(x) = \max(0, xW_1)W_2$$

- Notable Models: Original transformer, T5, Gopher, Chinchilla, OPT

4.3.2 GeLU

$$FF(x) = GeLU(xW_1)W_2$$
$$GeLU(x) = x\Phi(x),\ \Phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-t^2/2}dt$$

- Notable Models: GPT1/2/3, GPTJ, GPT-Neox, BLOOM

4.3.3 Gated activations (*GLU)

Instead of a linear + ReLU, augment the above with an (entrywise) linear term. （用一个逐元素的线性项来加强）

$$\max(0, xW_1) \to \max(0, xW_1) \otimes (xV)$$

This gives the gated variant (ReGLU) – note that we have an extra parameter (V).

$$\text{FF}_{\text{ReGLU}}(x) = (\max(0, xW_1) \otimes xV)W_2$$

> **The GLU (Gated Linear Unit) introduces a "Gating" mechanism.**
>
> It splits the input signal into two paths (or adds a parallel path), allowing the network to learn a "switch" that controls the flow of information.
> Here, $\max(0, xW_1)$ acts as a **Filter (Gate)**, determining which information in xV is allowed to pass through and which is suppressed.

Some variants:

1. **GeGLU**

$$\text{FF}_{\text{GeGLU}}(x, W_1, V, W_2) = (GeLU(0, xW_1) \otimes xV)W_2$$

- Notable Models: T5 v1.1, mT5, LaMDA, Phi3, Gemma 2, Gemma 3

1. **SwiGLU**

$$\text{FF}_{\text{SwiGLU}}(x, W_1, V, W_2) = (\text{Swish}(0, xW_1) \otimes xV)W_2$$
$$\text{Swish}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

- Notable Models: LLaMa 1/2/3, PaLM, Mistral, OlMo, most models post 2023

Gated models use smaller dimensions for the d_{ff} by 2/3 ›

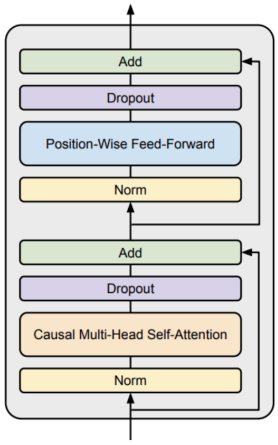
Gated Linear Units (GLUs) require reducing the Feed-Forward Network (FFN) hidden dimension (d_{ff}) to approximately 2/3 of the standard size to maintain parameter parity with non-gated FFNs, due to the introduction of V.

Why *GLU? ›

GLUs seem generally better, though differences are small...

4.4 Serial vs Parallel layers

Normal transformer blocks are *serial* – they compute attention, then the MLP



Parallel Layers:

- Standard Formulation:

$$y = x + \text{MLP}(\text{LayerNorm}(x + \text{Attention}(\text{LayerNorm}(x))))$$

- Parallel: 15% faster. recent models (Cohere Command A, Falcon 2 11B, Command R+)

$$y = x + \text{MLP}(\text{LayerNorm}(x) + \text{Attention}(\text{LayerNorm}(x)))$$

Why? ›

compute win!

4.5 Position Embeddings

4.5.1 Sine embeddings

add sines and cosines that enable localization.

$$
\begin{aligned}
Embed(x, i) &= v_x + PE_{pos} \\
PE(pos, 2i) &= \sin(pos/10000^{2i/d_{model}}) \\
PE(pos, 2i+1) &= \cos(pos/10000^{2i/d_{model}})
\end{aligned}
$$

- Notable Models: Original Transformer.

4.5.2 Absolute embeddings

add a position vector to the embedding.

$$
Embed(x, i) = v_x + u_i
$$

- Notable models: GPT1/2/3, OPT.

4.5.3 Relative embeddings

add a vector to the attention computation.

$$
e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}}
$$

- Notable models: T5, Gopher, Chinchilla.

4.5.4 Rotary Position Embeddings

> Su, J..., "RoFormer: Enhanced Transformer with Rotary Position Embedding"

https://arxiv.org/pdf/2104.09864

RoFormer: https://github.com/ZhuiyiTechnology/roformer

科学空间：https://kexue.fm/archives/8265
补充理解：RoPE

1. **Core Objective** The primary goal of RoPE is to design a position encoding function $f(x, i)$ such that the interaction (inner product) between a query at position i and a key at position j depends **only** on their content and relative distance $(i - j)$, rather than their absolute positions.

$$\langle f(x, i), f(y, j) \rangle = g(x, y, i - j)$$

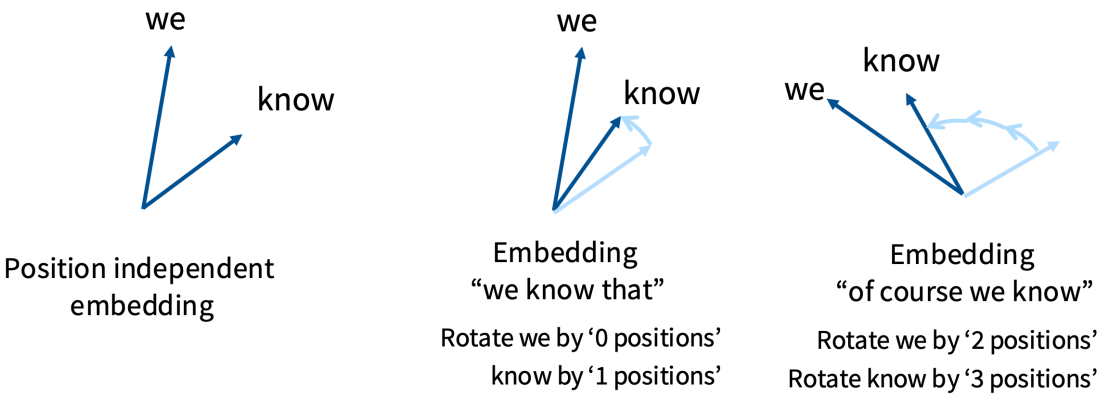
2. **Limitations of Previous Approaches**
   - Sinusoidal Embeddings: The expansion of the inner product results in various **cross-terms** (e.g., $\langle \dots \rangle = \langle v_x, v_y \rangle + \langle PE_i, v_y \rangle + \dots$).
   - Absolute Embeddings: the model learns **specific positions rather than relative** relationships.
   - Additive Relative Embeddings: While they capture relative distance, they do so by **adding a bias term** to the attention score (e.g., $q^T k + bias$). This alters the fundamental structure of the attention mechanism, meaning it is **no longer a pure inner product** of transformed vectors.
3. **The RoPE Solution** Instead of adding a position vector, RoPE multiplies the context vector by a rotation matrix determined by the position index.

$$\begin{aligned} \boldsymbol{q}_m &= R_{\Theta,m}\boldsymbol{q} \\ \boldsymbol{k}_n &= R_{\Theta,n}\boldsymbol{k} \\ f_{\{q,k\}}(\boldsymbol{x}_m, m) &= \boldsymbol{R}^d_{\Theta,m}\boldsymbol{W}_{\{q,k\}}\boldsymbol{x}_m \end{aligned}$$

RoPE: rotary position embeddings

How can we solve this problem?

- We want our embeddings to be invariant to absolute position
- We know that inner products are invariant to arbitrary rotation.



Position independent embedding

Embedding "we know that"

Rotate we by '0 positions'
know by '1 positions'

Embedding "of course we know"

Rotate we by '2 positions'
Rotate know by '3 positions'

Due to the geometric properties of rotation, the inner product of two rotated vectors depends **only** on the difference in their angles (positions), satisfying the relative distance goal without altering the standard attention formula.

$$\langle \boldsymbol{q}_m, \boldsymbol{k}_n \rangle = \langle R_{\Theta,m}\boldsymbol{q}, R_{\Theta,n}\boldsymbol{k} \rangle = \langle \boldsymbol{q}, R^{-1}_{\Theta,n-m} R_{\Theta,m}\boldsymbol{k} \rangle = \langle \boldsymbol{q}, R_{\Theta,n-m}\boldsymbol{k} \rangle$$

$$\boldsymbol{R}^{d}_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \dots & 0 & \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \dots & 0 & \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \dots & 0 & \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \dots & 0 & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & 0 & 0 & \dots & \cos m\theta_{d/2} & -\sin \\ 0 & 0 & 0 & 0 & \dots & \sin m\theta_{d/2} & \cos \end{pmatrix}$$

In code: instead of matrix multiplication, use efficient **element-wise complex multiplication**.

$$\text{RoPE}(x) = \begin{pmatrix} x_1 \\ x_2 \\ \dots \end{pmatrix} \odot \begin{pmatrix} \cos\phi \\ \cos\phi \\ \dots \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ \dots \end{pmatrix} \odot \begin{pmatrix} \sin\phi \\ \sin\phi \\ \dots \end{pmatrix}$$

4.6 Hyperparameters

4.6.1 Feedforward Dim and Model Dim

- Feedforward Dim (d_{ff}, the dimensions of the intermediate layer in a two-layer FFN)
  and
- Model Dim(d_{model}, the size of the hidden representation for each token, i.e., the number of features after the model's embedding.)

$$d_{ff} = 4d_{model}$$

Because GLU variants scale down by 2/3, so,

$$d_{ff} = \frac{8}{3}d_{model}$$

4.6.2 Head-dim * Num-heads to Model-dim Ratio

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

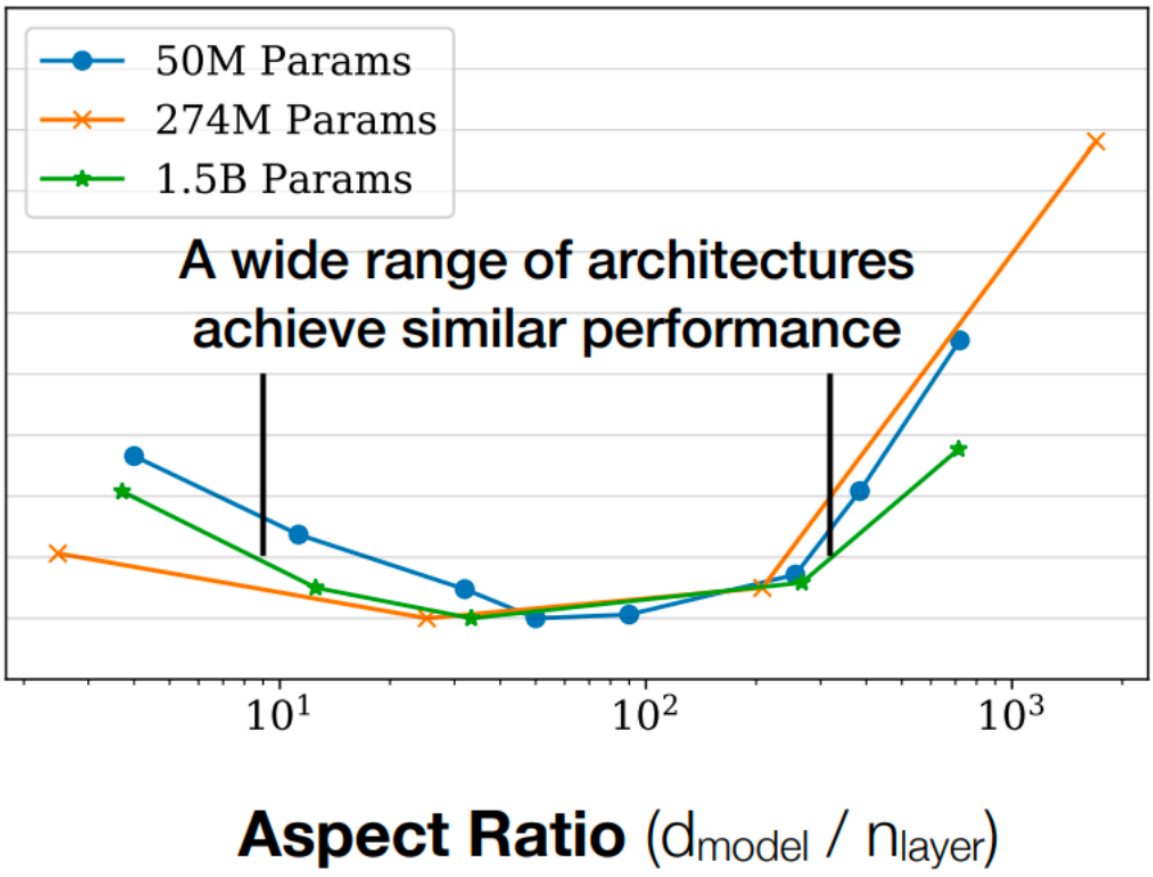
4.6.3 Aspect Ratios

$$d_{model}/n_{layers}$$

Extremely deep models are harder to parallelize and have higher latency. (Pipeline Parallel)

Sweet spot?

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



4.6.4 Vocabulary Sizes

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

4.6.5 Dropout and Weight Decay

- Many older models used dropout during pretraining.
- Newer models (except Qwen) rely only on weight decay.
  - Why? Weight decay interacts with learning rates (cosine schedule).

Weght Decay:

The original empirical loss function ($L_{Empirical}$) is augmented with an L2 penalty term (the square of the Euclidean norm of the weights):

$$\mathcal{L}_{\text{Augmented}}(\boldsymbol{W}) = \mathcal{L}_{\text{Empirical}}(\boldsymbol{W}) + \lambda \cdot ||\boldsymbol{W}||_2^2$$

- λ: The **Weight Decay Coefficient** (a hyperparameter)

Gradient Descent step:

$$\frac{\partial \mathcal{L}_{\text{Augmented}}}{\partial W_i} = \frac{\partial \mathcal{L}_{\text{Empirical}}}{\partial W_i} + 2\lambda W_i$$

The standard update rule for Stochastic Gradient Descent (SGD) is:

$$W_i^{\text{new}} = W_i^{\text{old}} - \eta \cdot \frac{\partial \mathcal{L}_{\text{Augmented}}}{\partial W_i}$$

Substituting the gradient term:

$$W_i^{\text{new}} = W_i^{\text{old}} - \eta \cdot \left(\frac{\partial \mathcal{L}_{\text{Empirical}}}{\partial W_i} + 2\lambda W_i^{\text{old}} \right)$$

Rearranging the terms highlights the **decay** component:

$$W_i^{\text{new}} = \underbrace{W_i^{\text{old}}(1 - 2\eta\lambda)}_{\text{Weight Decay / Shrinkage}} \underbrace{- \eta \cdot \frac{\partial \mathcal{L}_{\text{Empirical}}}{\partial W_i}}_{\text{Standard Update}}$$

Regardless of the empirical gradient, the weight W_i is **multiplicatively shrunk** by a factor of $(1 - 2\eta\lambda)$ in every step. This mechanism forces the model to maintain **small**, distributed weights.

4.7 Stability Tricks

4.7.1 Softmaxes

Recall the softmax calculation:

$$\log(P(x)) = U_r(x) - \log(Z(x)) = \log(\frac{e^{U_r(x)}}{Z(x)})$$

$$Z(x) = \sum_{r'=1}^{|V|} e^{U_{r'}(x)}$$

- $U_r(x)$ is the Logit value (the un-normalized score) that the model assigns to the vocabulary token r, given the sequence input x.
- $|V|$ denotes the size of the vocabulary.

In large-scale training with massive vocabularies, the Logits ($U_r(x)$) can become unstable, causing the Normalizer Term $Z(x)$ to overflow or underflow during computation.

"z-loss":

$$L = \sum_{i}^{Batch} \left[\log(P(x_i)) - \alpha \log^2 Z(x_i) \right]$$

- α: The scaling coefficient, set to 10^{-4} in PaLM.

Why "z-loss"?

Encouraging $\log Z \approx 0$, Stabilizing Softmax.

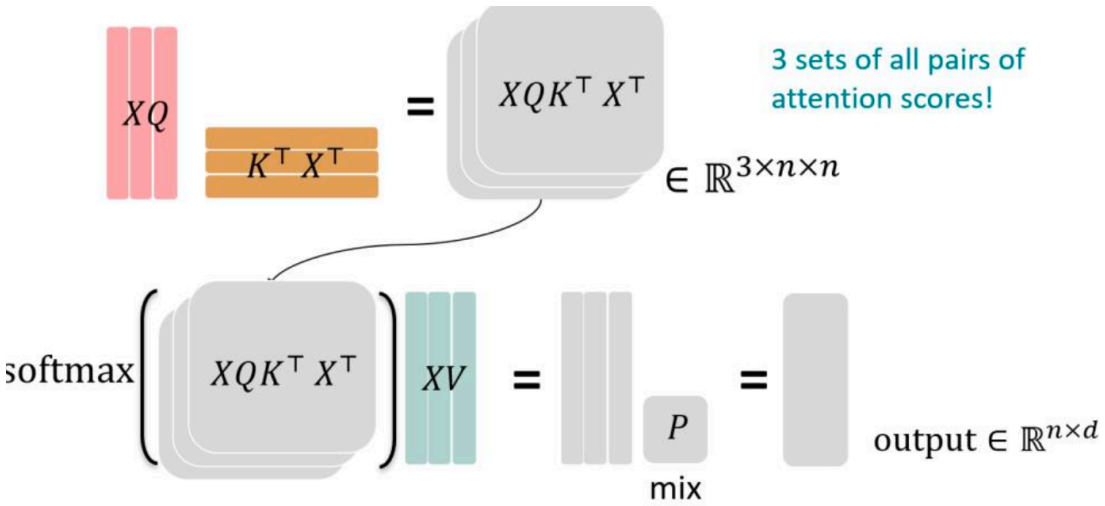
4.7.2 Logit Soft-capping

$$\text{logits} \leftarrow \text{soft\_cap} \cdot \tanh\left(\frac{\text{logits}}{\text{soft\_cap}}\right)$$

Soft Constraint: For values within the stable range, the output is largely unchanged ($\tanh\left(\frac{\text{logits}}{\text{soft\_cap}}\right) \approx \frac{\text{logits}}{\text{soft\_cap}}$). For extremely large values, the function "softly caps" the output near the maximum limit ($\pm\text{soft\_cap}$).

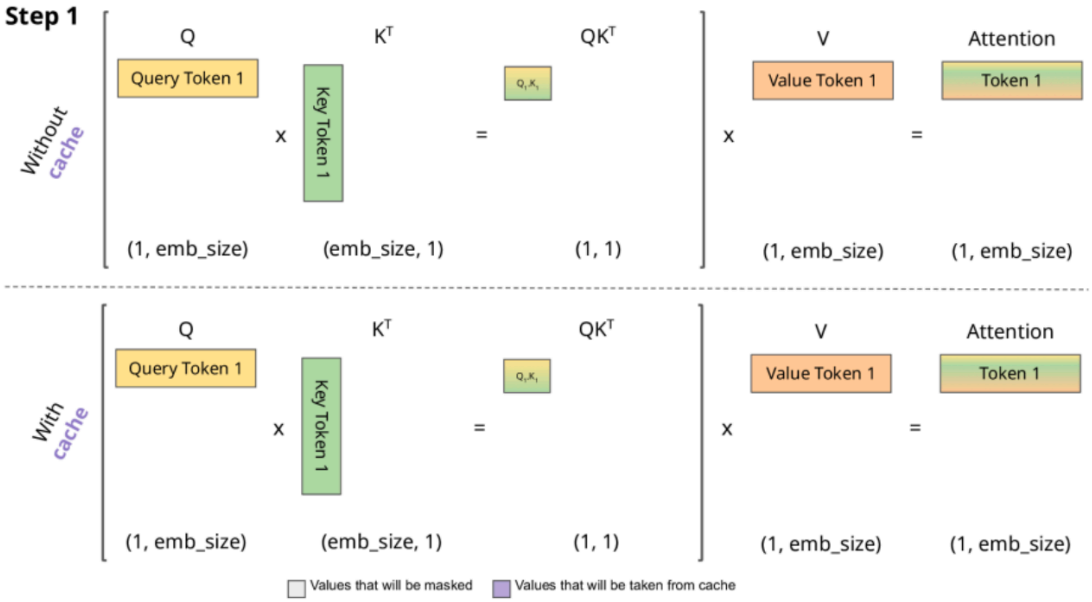
4.8 Attention Heads

1. **Parallel Phase (Training/Prefill):**
   - Operations are parallelizable across sequence length.
   - Total arithmetic operations (bnd^2), total memory accesses ($\underbrace{bnd}_{x} + \underbrace{bhn^2}_{softmax/Attention} + \underbrace{d^2}_{Projection/Weight}$)
   - **High Arithmetic Intensity:** $\approx O\left(\left(\frac{1}{k} + \frac{1}{bn}\right)^{-1}\right)$
   - GPU utilization is optimal (Compute-bound).



2. **Incremental Phase (Generation):**
   - Generation is sequential; requires KV Cache to avoid re-computation. For each new token, the entire KV Cache (history) **must be loaded from memory.**
   - total arithmetic operations (bnd^2), total memory accesses ($\underbrace{bn^2d}_{KVCache} + \underbrace{nd^2}_{Weights}$)

- **Low Arithmetic Intensity:** $\approx O\left(\left(\frac{n}{d}+\frac{1}{b}\right)^{-1}\right)$ (memory bandwidth)

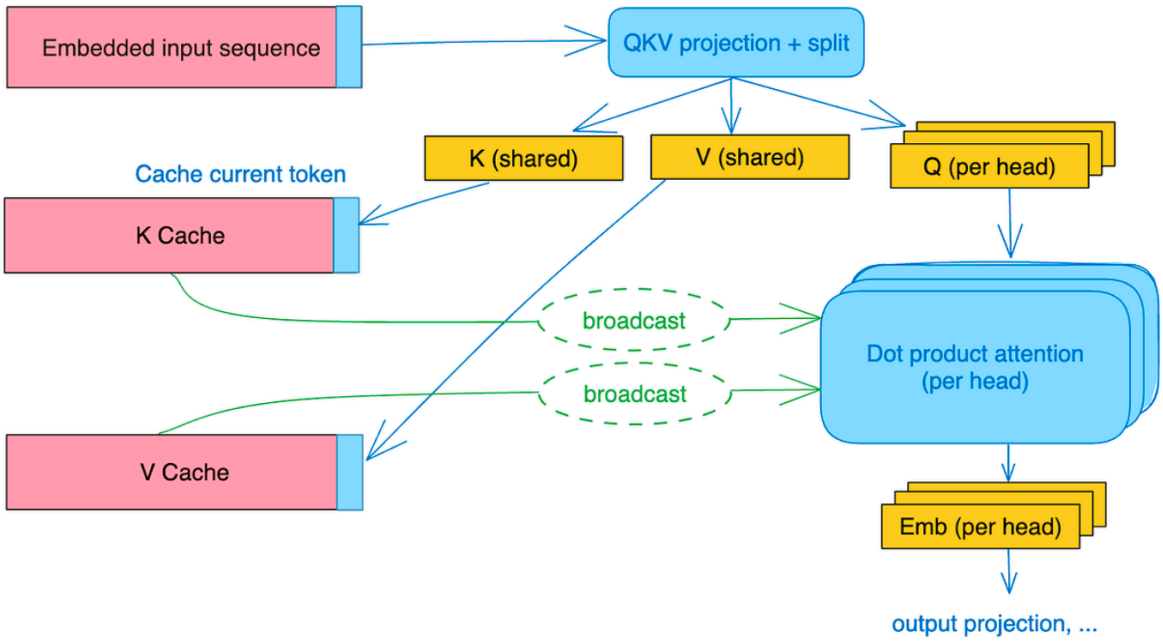


3. **Role of GQA/MQA:**
   - Standard Multi-Head Attention (MHA) has a large KV cache size (dmodel×n).
   - **MQA/GQA** are designed to reduces the number of KV heads.

> n is sequence length, d is model dimension, b is batch size, h is the number of attention heads, and k is head dimension (d_k).

4.8.1 GQA / MQA -- Reducing attention head cost

1. **MQA:**



- Key idea: have multiple queries (per head), but just one dimension for keys and values (just one for all heads).

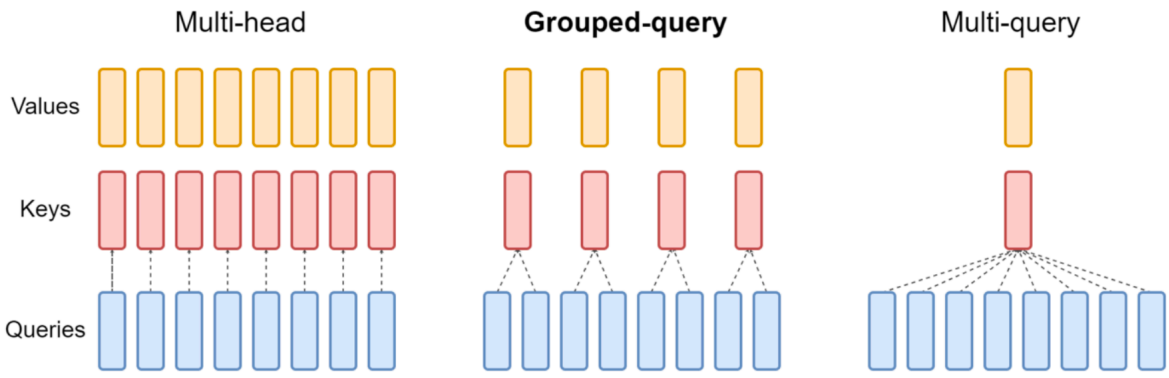
Total memory access ($\underbrace{bnd}_{Q} + \underbrace{bn^2k}_{KVCache} + \underbrace{nd^2}_{Weights}$), Arithmetic intensity $\approx O\left(\left(\frac{1}{d} + \frac{n}{dh} + \frac{1}{b}\right)^{-1}\right)$

2. **GQA:** (A Generalization of MQA.)

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

Mechanism:

- Divide the total H query heads into G groups.
- Each group shares a **single** Key (K) and Value (V) head.



So MQA can be viewed as a special case of GQA where the number of groups is one.

3. **Does MQA hurts?**

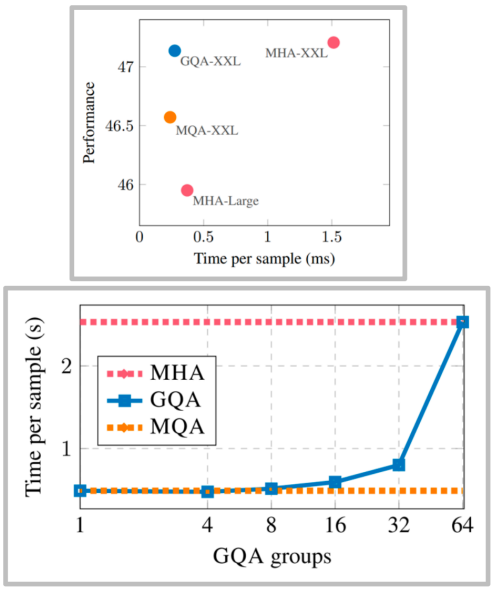
Does MQA hurt? Sometimes..

Small PPL hit w/ MQA [Shazeer 2019]

Table 3: Billion-Word LM Benchmark Results.

Attention	h	d_k, d_v	d_{ff}	dev-PPL
multi-head	8	128	8192	**29.9**
multi-query	8	128	9088	30.2
multi-head	1	128	9984	31.2
multi-head	2	64	9984	31.1
multi-head	4	32	9984	31.0
multi-head	8	16	9984	30.9

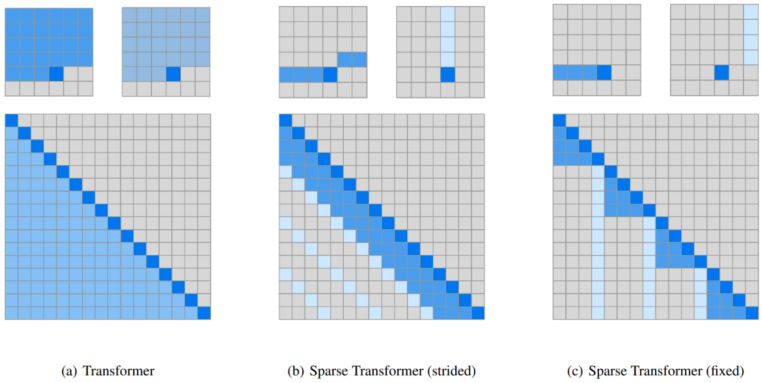
Low/no hit w/ GQA [Ainslie 2023]



4.8.2 Sparse / Sliding Window Attention

Standard Full Attention has quadratic complexity $O(n^2)$, which is expensive for long contexts.

1. **Sparse Attention (Early):** Used fixed patterns or strided steps (Child et al., 2019). Difficult to optimize on hardware due to irregular memory access.
   Build sparse / structured attention that trades off expressiveness vs runtime (GPT3)

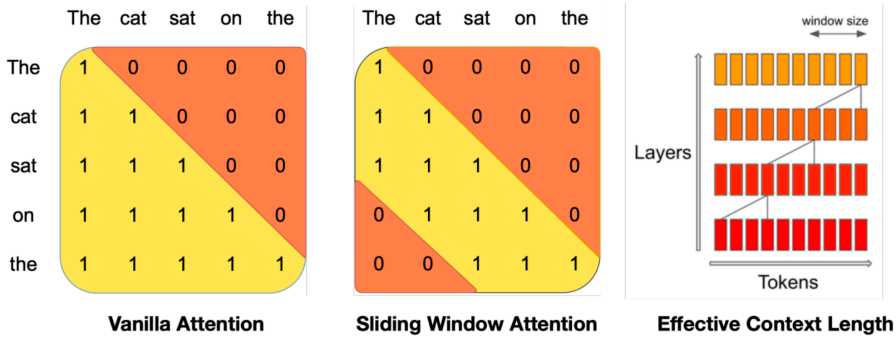


[Child et al 2019]

2. **Sliding Window Attention (SWA):** Tokens only attend to the previous W tokens.
   - Complexity: $O(n \times W)$ (Linear).

- Although the immediate window is limited, the effective receptive field grows linearly with network depth (stacking layers).

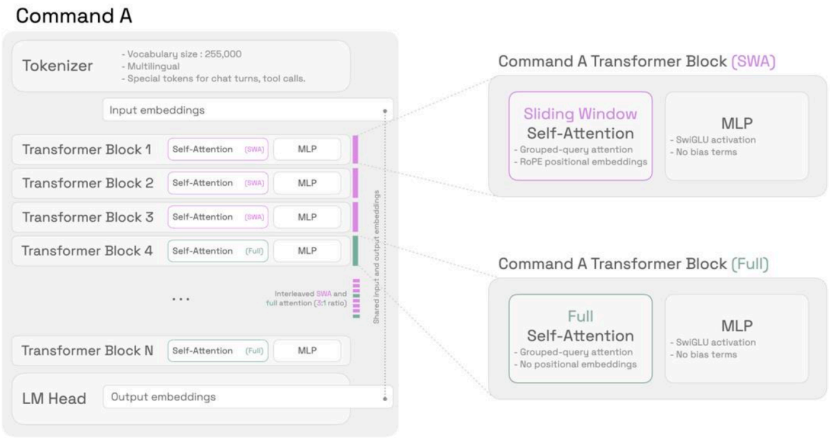
Another variation on this idea – sliding window attention



Just use the main part of the strided pattern – let depth extend effective context (Mistral)

3. **Current SOTA Strategy (The "Hybrid" Approach):** Pure SWA can lose global context. Modern models (e.g., Mistral, Cohere Command R) use an interleaved strategy.
   - Interleaving: Most layers use SWA for efficiency.
   - Global Anchors: Every k-th layer (e.g., every 4th layer) uses Full Attention.

   From Cohere Command A – Every 4th layer is a full attention



Long-range info via NoPE, short-range info via RoPE + SWA.

---

5 Mixtures of Experts