Transformer

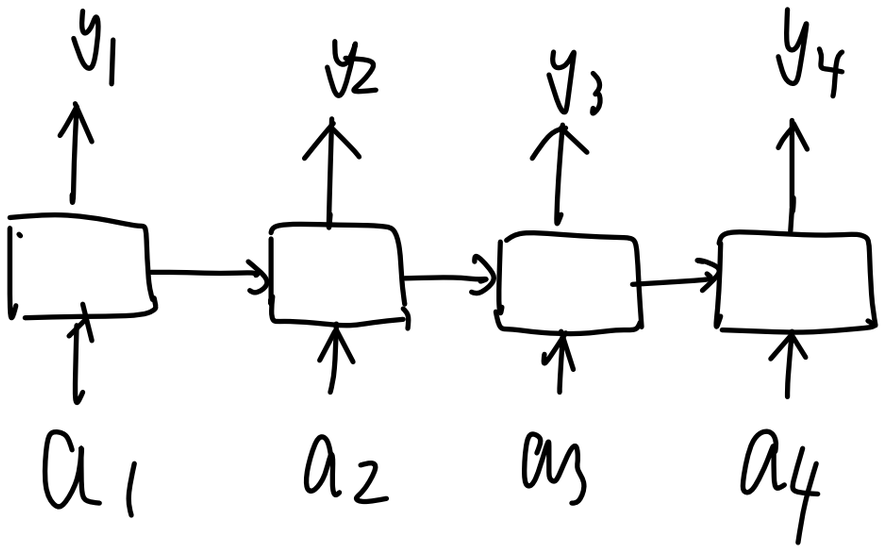
https://github.com/xiahouzuoxin

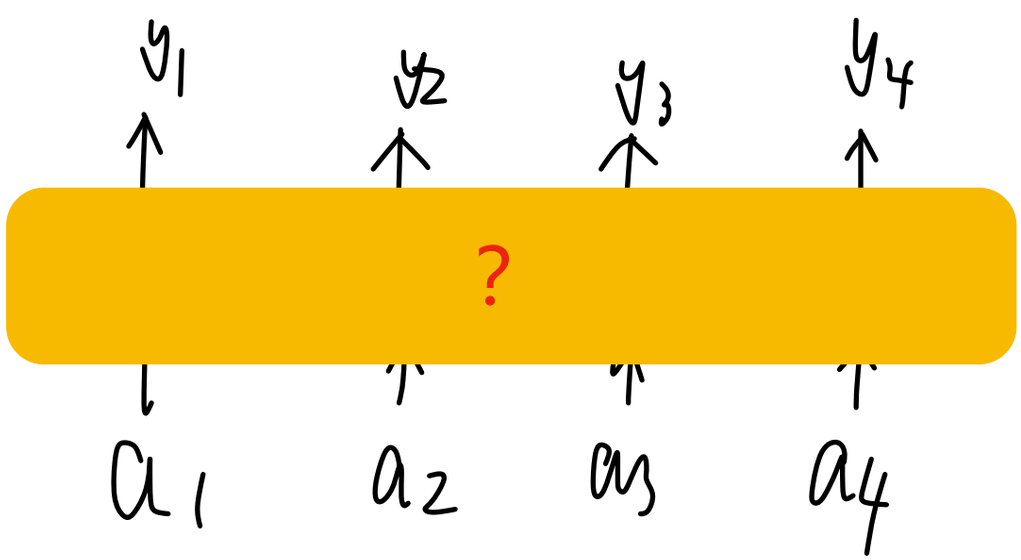
Contents

1. What is attention and why we need?
2. How to convert raw text to self-attention inputs?
3. How to build a model with self-attention layers?
4. LLM Optimizations – LoRA, KV Cache, Deepseek etc.

What is attention and why we need?

Why we need attention





Issues with RNNs/Seq2Seq:

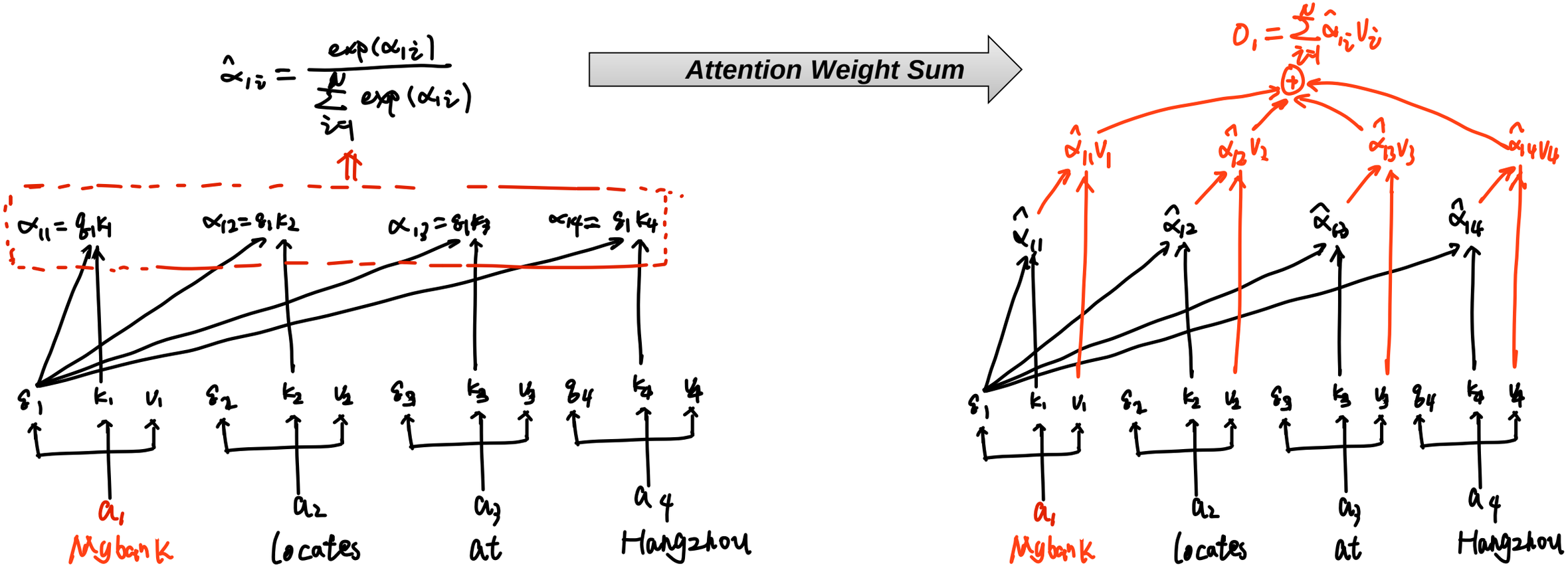
- **Temporal Dependency.**They cannot be parallelized, which is time-consuming.

- **Vanishing Gradient over time.**The influence of earlier time steps diminishes as the time gap increases.

CNN is all you need
https://arxiv.org/pdf/1712.09662.pdf

Attention is all you need
https://arxiv.org/pdf/1712.09662.pdf

Self Attention Layer

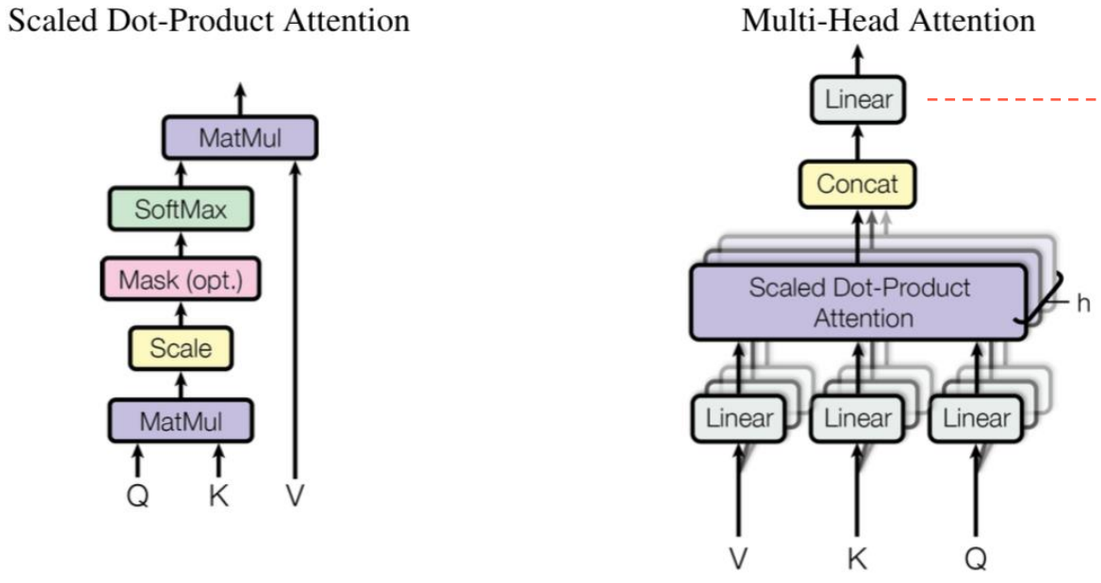


- ✓ By the same way as getting O1, we can get O2, O3, O4。

- ✓ Key Points：

1. All text computations can be performed ***independently and in parallel*** — the calculation of O_2 does not depend on O_1.
2. For each word, all other words contribute to its output — with the contribution strength determined by attention weights — there's ***no temporal forgetting issue***.

Self Attention Layer



Linear layer for mapping concat result to the input shape to avoid dimension increasing disaster.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

3.2.1 Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

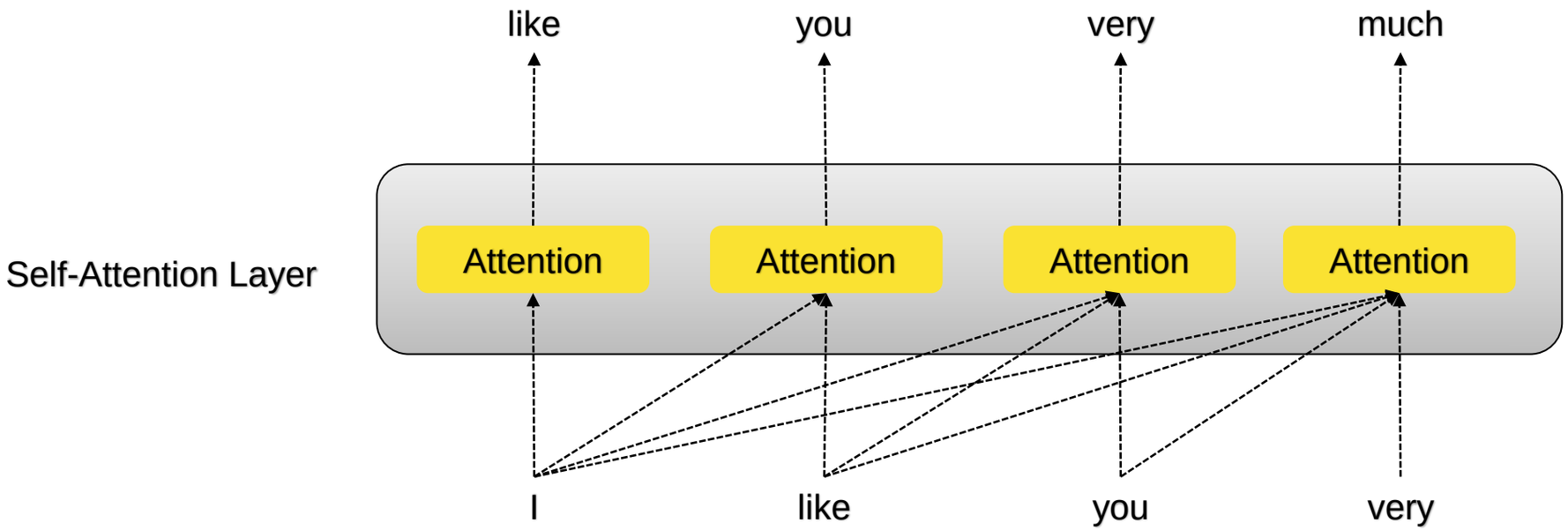
In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \quad (1)$$

d_k :
The QK^T product is scaled to avoid softmax saturation for improved training stability.

Causal Self-Attention

In tasks like autoregressive text generation, where words are produced from left to right, the generation of the current word must not depend on future words that have not yet been generated.
To enforce this, ***Causal Self-Attention*** is used, which restricts each position to attend only to previous positions in the sequence by using only the keys (K) and values (V) of the preceding tokens.

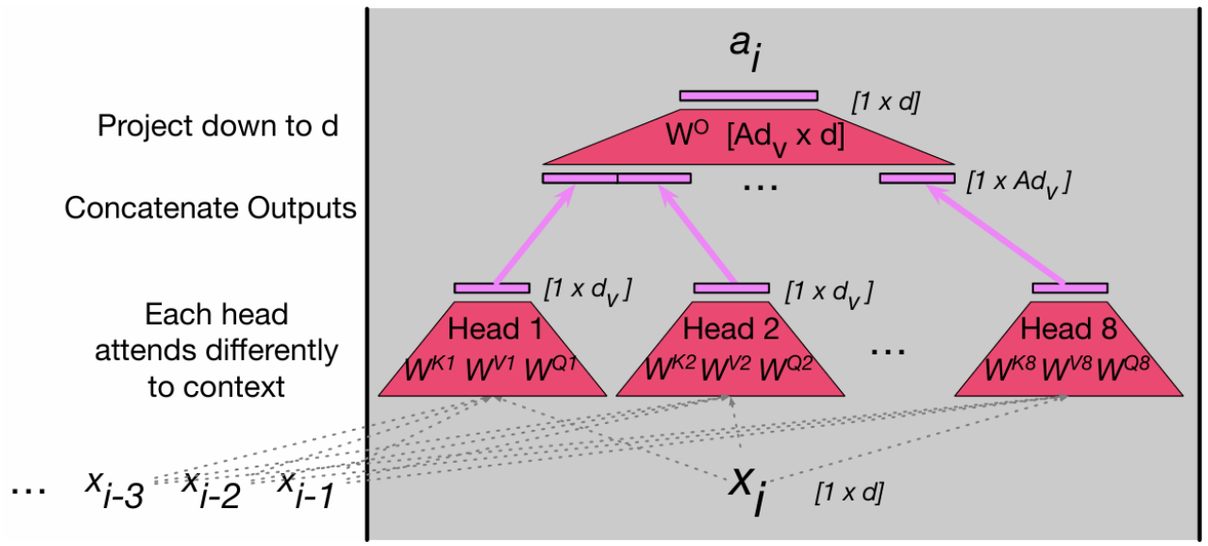


Implement in pytorch by ***Tensor.masked_fill_***

nanoGPT: https://github.com/karpathy/nanoGPT/blob/93a43d9a5c22450bbf06e78da2cb6eeef084b717/model.py#L68
Deepseek V3: https://github.com/deepseek-ai/DeepSeek-V3/blob/9b4e9788e4a3a731f7567338ed15d3ec549ce03b/inference/model.py#L592

Multi-Head Attention

https://web.stanford.edu/~jurafsky/slp3/8.pdf



原始的Transformer Paper使用了8个Head

实现上，为了利用提高并行计算效率：
1. *先按d维度计算QKV，然后拆成MultiHead；*
2. *甚至在nanoGPT中，仅用了一个nn.Linear，拆分得到QKV*

GPT中也有对Attention Weights使用了Dropout

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads=8, bias=False, dropout=0.1, qkv_proj=False):
        '''
        The original paper uses d_k = d_v = d_model / num_heads = 512 / 8 = 64
        '''
        super(MultiHeadAttention, self).__init__()
        self.d_k = d_model // num_heads
        assert d_model % num_heads == 0 and self.d_k > 0
        self.num_heads = num_heads

        self.qkv_proj = qkv_proj
        if qkv_proj:
            self.q_proj = nn.Linear(d_model, d_model, bias=bias)
            self.k_proj = nn.Linear(d_model, d_model, bias=bias)
            self.v_proj = nn.Linear(d_model, d_model, bias=bias)

        self.dropout = nn.Dropout(dropout)
        self.output_fc = nn.Linear(d_model, d_model, bias=bias)

    def forward(self, Q, K, V, mask=None):
        # input: (batch_size, seq_len, d_model)
        # softmax(QK^T / sqrt(d_k)) * V

        B, T, d_model = Q.size()

        if self.qkv_proj:
            Q = self.q_proj(Q)
            K = self.k_proj(K)
            V = self.v_proj(V)

        Q = Q.view(Q.size(0), Q.size(1), self.num_heads, self.d_k).permute(0, 2, 1, 3) # bs x num_heads x seq_len x d_k
        K = K.view(K.size(0), K.size(1), self.num_heads, self.d_k).permute(0, 2, 1, 3)
        V = V.view(V.size(0), V.size(1), self.num_heads, self.d_k).permute(0, 2, 1, 3) # bs x num_heads x seq_len x d_k
        attention = torch.matmul(Q, K.permute(0, 1, 3, 2)) / (self.d_k ** 0.5) # bs x num_heads x seq_len x seq_len
        if mask is not None:
            attention = attention.masked_fill(mask[:,:,:T,:T] == 0, float('-inf'))
        attention = F.softmax(attention, dim=-1) # bs x num_heads x seq_len x seq_len
        attention = self.dropout(attention)
        output = torch.matmul(attention, V) # bs x num_heads x seq_len x d_k
        output = output.permute(0, 2, 1, 3).contiguous().view(B, T, -1)
        output = self.output_fc(output)
        return output
```

https://github.com/xiahouzuoxin/zxlearn/blob/6589952c15ab6b8834d64b1e048373b573e193f5/transformer/transformer.py#L26

Self Attention In nanoGPT

https://github.com/karpathy/nanoGPT/blob/93a43d9a5c22450bbf06e78da2cb6eeef084b717/model.py#L29

```python
class CausalSelfAttention(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd, bias=config.bias)
        # output projection
        self.c_proj = nn.Linear(config.n_embd, config.n_embd, bias=config.bias)
        # regularization
        self.attn_dropout = nn.Dropout(config.dropout)
        self.resid_dropout = nn.Dropout(config.dropout)
        self.n_head = config.n_head
        self.n_embd = config.n_embd
        self.dropout = config.dropout
        # flash attention make GPU go brrrrr but support is only in PyTorch >= 2.0
        self.flash = hasattr(torch.nn.functional, 'scaled_dot_product_attention')
        if not self.flash:
            print("WARNING: using slow attention. Flash Attention requires PyTorch >= 2.0")
            # causal mask to ensure that attention is only applied to the left in the input sequence
            self.register_buffer("bias", torch.tril(torch.ones(config.block_size, config.block_size))
                                        .view(1, 1, config.block_size, config.block_size))

    def forward(self, x):
        B, T, C = x.size() # batch size, sequence length, embedding dimensionality (n_embd)

        # calculate query, key, values for all heads in batch and move head forward to be the batch dim
        q, k, v  = self.c_attn(x).split(self.n_embd, dim=2)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)

        # causal self-attention; Self-attend: (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh, T, T)
        if self.flash:
            # efficient attention using Flash Attention CUDA kernels
            y = torch.nn.functional.scaled_dot_product_attention(q, k, v, attn_mask=None, dropout_p=self.dropout if self.training else 0, is_causal=True)
        else:
            # manual implementation of attention
            att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
            att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
            att = F.softmax(att, dim=-1)
            att = self.attn_dropout(att)
            y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
        y = y.transpose(1, 2).contiguous().view(B, T, C) # re-assemble all head outputs side by side

        # output projection
        y = self.resid_dropout(self.c_proj(y))
        return y
```

For projecting to QKV

Output projection, exist in original Transformer for mapping multi-head attention concat result to the same shape as input

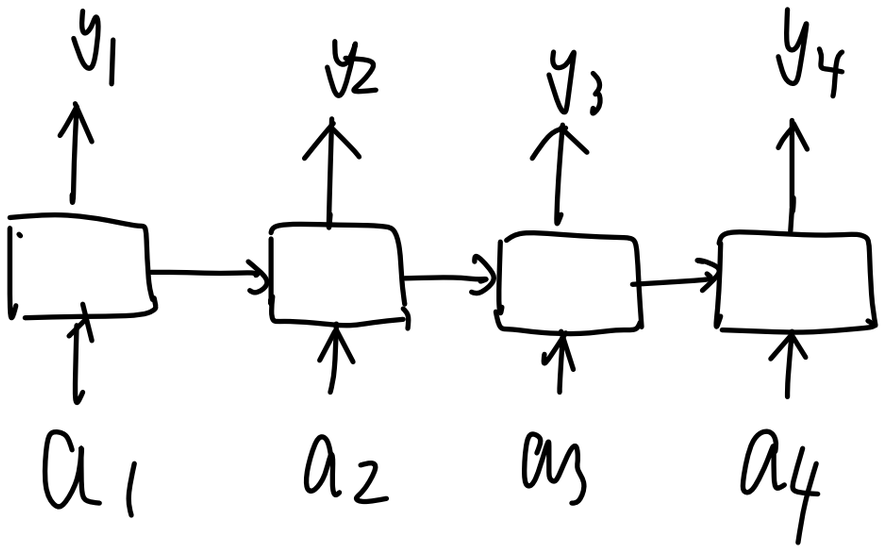
Multi-Head Attention

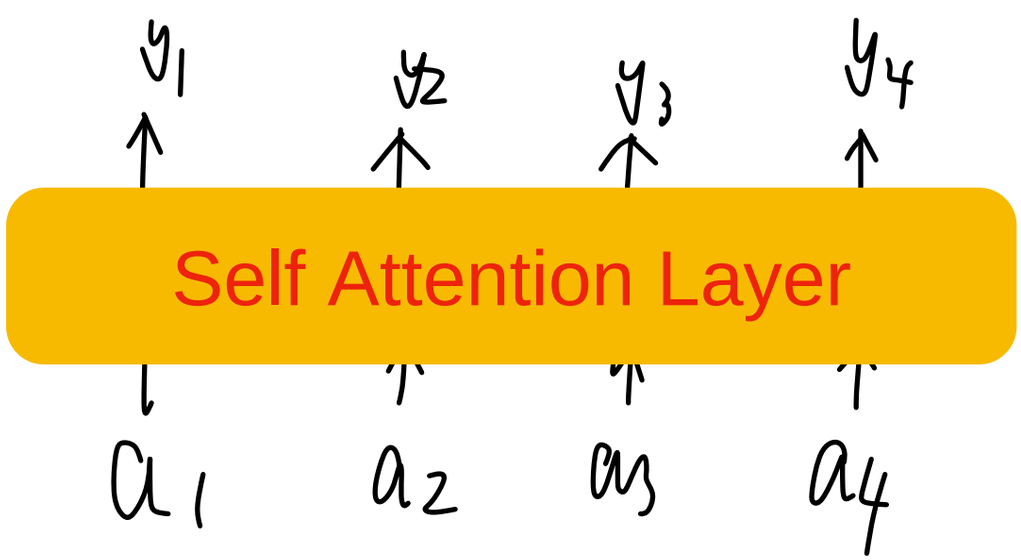
Speed up attention calculation by Flash Attention, we may talk about it later

Causal attention

Apply dropout on normalized attention weight

Revisit: why we need attention





Issues with RNNs/Seq2Seq:

- **Temporal Dependency.** They cannot be parallelized, which is time-consuming.
- **Vanishing Gradient over time.** The influence of earlier time steps diminishes as the time gap increases.

Self-Attention Layer:

- **No sequential dependency.** It is fully parallelizable and GPU-friendly.
- **No global vanishing issues.** Each query can interact with all keys across the entire sequence via dot products.
- **Drawback:** Lacks positional information.

Convert Raw Text to Self-Attention Inputs?

Tokenizer & Positional Encoding

Tokenizer

How to feed our raw texts to the self attention layer?

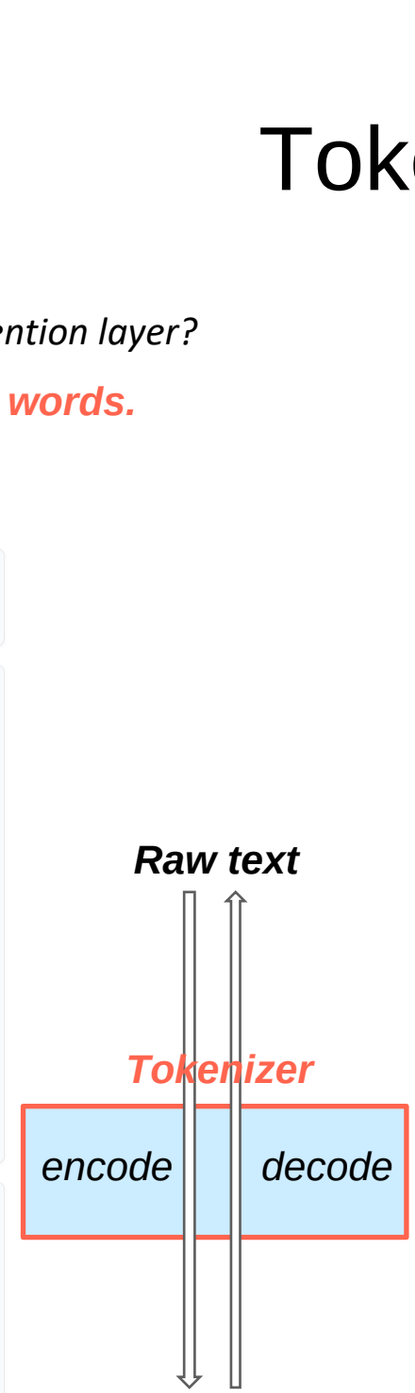
Language model reads tokens, not words.

Token count
142

```
# quick start
As the simplest example, we can reproduce the Wikipedia art
icle on BPE as follows:

from minbpe import BasicTokenizer
tokenizer = BasicTokenizer()
text = "aaabdaaabac"
tokenizer.train(text, 256 + 3) # 256 are the byte tokens, t
hen do 3 merges
print(tokenizer.encode(text))
# [258, 100, 258, 97, 99]
print(tokenizer.decode([258, 100, 258, 97, 99]))
# aaabdaaabac
tokenizer.save("toy")
# writes two files: toy.model (for loading) and toy.vocab
(for viewing)
```

```
2, 4853, 1604, 198, 2305, 290, 63122, 4994, 11, 581, 665, 4
5472, 290, 26487, 5787, 402, 418, 3111, 472, 18183, 1402, 2
845, 1349, 65, 424, 1588, 21976, 53560, 198, 10346, 4492, 3
14, 21976, 53560, 1234, 919, 314, 392, 3545, 378, 90964, 37
8, 359, 1092, 10346, 4492, 45043, 13414, 11, 220, 5780, 65
9, 220, 18, 8, 1069, 220, 5780, 553, 290, 9239, 20290, 11,
1815, 621, 220, 18, 176901, 198, 1598, 23460, 4492, 31812,
13414, 2210, 2, 723, 28320, 11, 220, 1353, 11, 220, 28320,
11, 220, 5170, 11, 220, 2058, 1592, 1598, 23460, 4492, 3061
9, 5194, 28320, 11, 220, 1353, 11, 220, 28320, 11, 220, 517
0, 11, 220, 2058, 24801, 2, 40260, 378, 90964, 378, 359, 19
8, 10346, 4492, 10017, 568, 189102, 1896, 2, 26475, 1920, 6
291, 25, 30427, 5212, 350, 1938, 14742, 8, 326, 30427, 552
0, 57008, 350, 1938, 26379, 8
```

https://tiktokenizer.vercel.app/?model=gpt-4o



*One of the most common **tokenizer** is **BPE (Byte Pair Encoding):***

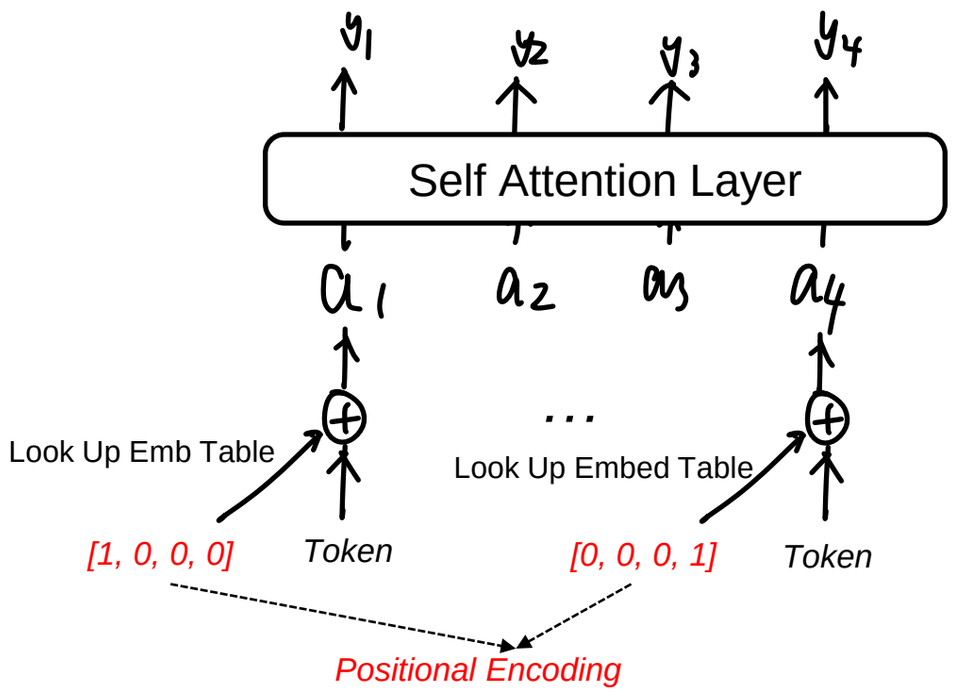
- *BPE iteratively merges the most frequent pairs of symbols in a corpus to form subword units that balance vocabulary size and text coverage.*
- *Ref code: https://github.com/karpathy/minbpe/blob/master/minbpe/regex.py*
- *A small BPE merge operator from https://arxiv.org/pdf/1508.07909*

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

Iter count the pair frequencies and merge

Positional Encoding

There's different meaning of "love you" and "you love", though it just swapped the position of the text.
Therefore*, the order of the text is still important but not considered in self attention layer*. How to solve this?




```
Text Embedding：  wte = nn.Embedding(config.vocab_size, config.n_embd)
Pos  Embedding：  wpe = nn.Embedding(config.block_size, config.n_embd)
# vocab_size token词表大小
# block_size 模型最大处理序列长度

Add Embeddings：  a_i = wte(TextToken) + wpe(PositionalEncoding)

# Some models (nanoGPT/GPT2), they apply dropout after the embeddings like in DNN
a_i = F.dropout(a_i)
```

Why adding the positional embedding, not concat?

Reason 1: Adding in embedding equal to the concat on one-hot encoding, as below equation where W is the embedding transform matrix

$$W \cdot x_p^i = [W^I, W^P] \cdot [[x^i]^T, [x^p]^T]^T = W^I \cdot x^i + W^P \cdot x^p = embed^i + pos^i$$

https://www.zhihu.com/question/485476372

Reason 2: There are also some models adopted concat, but it doubled the dimension of self-attention layer but little improvement

Sinusoidal Positional Encoding

Positional encoding by one-hot has 2 issues:
- One-hot encoding cannot scalable when the inference sequence length > training length
- One-hot encoding is orthogonal (正交的), model not able to learn the relative distances of token pairs

⇩

Sinusoidal Positional Encoding

Each position pos and dimension i is encoded as:

$$PE_{\mathrm{pos},2i} = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right)$$

$$PE_{\mathrm{pos},2i+1} = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d_{\mathrm{model}}}}\right)$$

PE_{pos+k} can be represented as a linear function of PE_pos, which gives the possibility the model can learnt from it

The encoding creates smooth, continuous patterns that let the Transformer model infer **relative and absolute positions** in a sequence without recurrence.

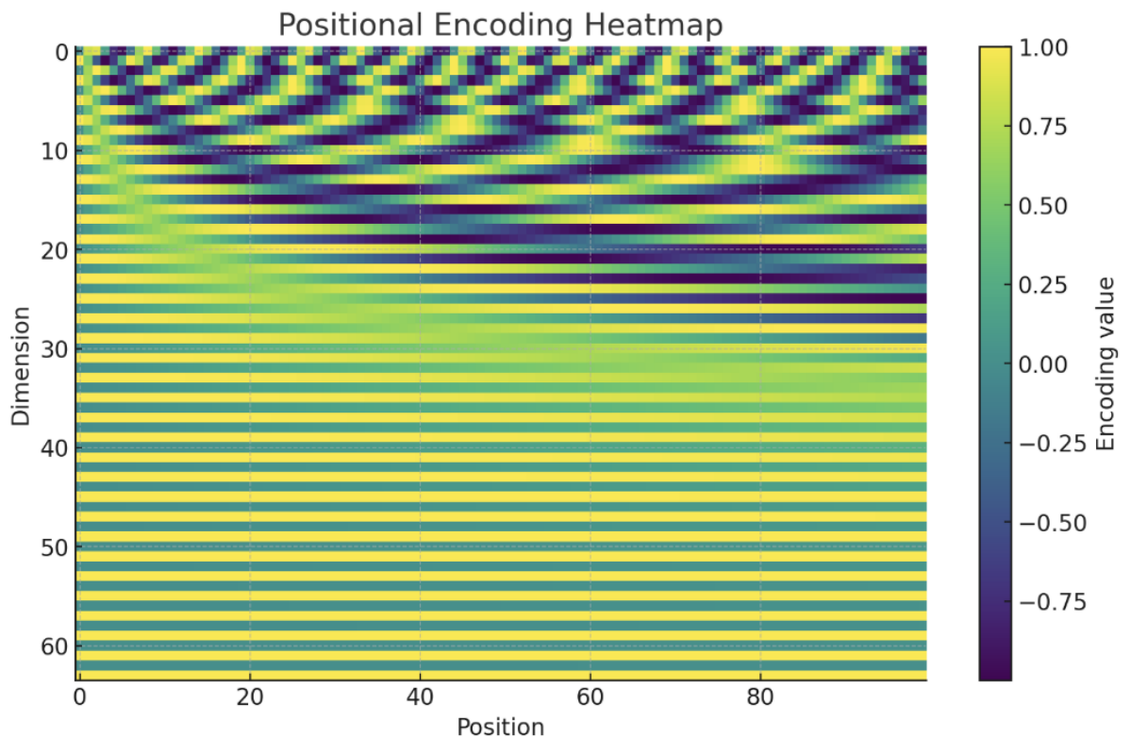


Figure is generated by ChatGPT

RoPE - Rotary Position Embedding

在普通的 **Sinusoidal Positional Encoding (PE)** 里，我们是：

$$x_{\text{input}} = x_{\text{token}} + PE_{\text{pos}}$$

即直接相加。

而在 **RoPE** 中，我们不加，而是对每个 token 的 embedding（通常是 query 和 key 向量）做一个 **二维平面旋转**：

$$\text{RoPE}(x, \text{pos}) = \begin{bmatrix} x_{2i} \\ x_{2i+1} \end{bmatrix} \text{ 旋转一个角度 } \theta_{pos,i}$$

其中：

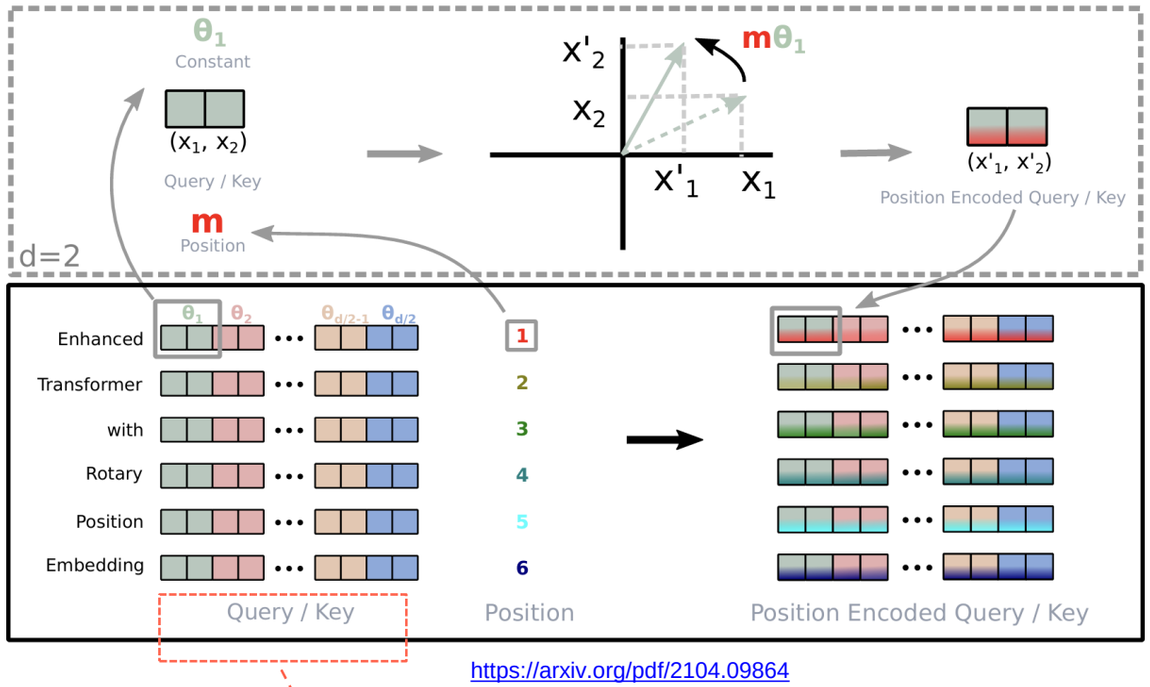
$$\theta_{pos,i} = \text{pos} \times \frac{1}{10000^{2i/d}}$$

这个旋转等价于复数乘法：

$$x' = x \cdot e^{j\theta_{pos}}$$

于是每个维度对都围绕原点旋转不同的角度，**把位置信息编码进方向（phase）**而不是幅度。

RoPE不再把“位置”直接加到 embedding 上，而是通过一个旋转变换（rotation），把位置编码“嵌入”到 attention 的计算中。



https://arxiv.org/pdf/2104.09864

```python
def apply_rotary_emb(x: torch.Tensor, freqs_cis: torch.Tensor) -> torch.Tensor:
    """
    Applies rotary positional embeddings to the input tensor.

    Args:
        x (torch.Tensor): Input tensor with positional embeddings to be applied.
        freqs_cis (torch.Tensor): Precomputed complex exponential values for positional embeddings.

    Returns:
        torch.Tensor: Tensor with rotary embeddings applied.
    """
    dtype = x.dtype
    x = torch.view_as_complex(x.float().view(*x.shape[:-1], -1, 2))
    freqs_cis = freqs_cis.view(1, x.size(1), 1, x.size(-1))
    y = torch.view_as_real(x * freqs_cis).flatten(3)
    return y.to(dtype)
```

旋转参数Theta可以预计算好

复数乘法

https://github.com/deepseek-ai/DeepSeek-V3/blob/9b4e9788e4a3a731f7567338ed15d3ec549ce03b/inference/model.py#L378

注意，RoPE只需要对Q和K进行旋转Encoding：

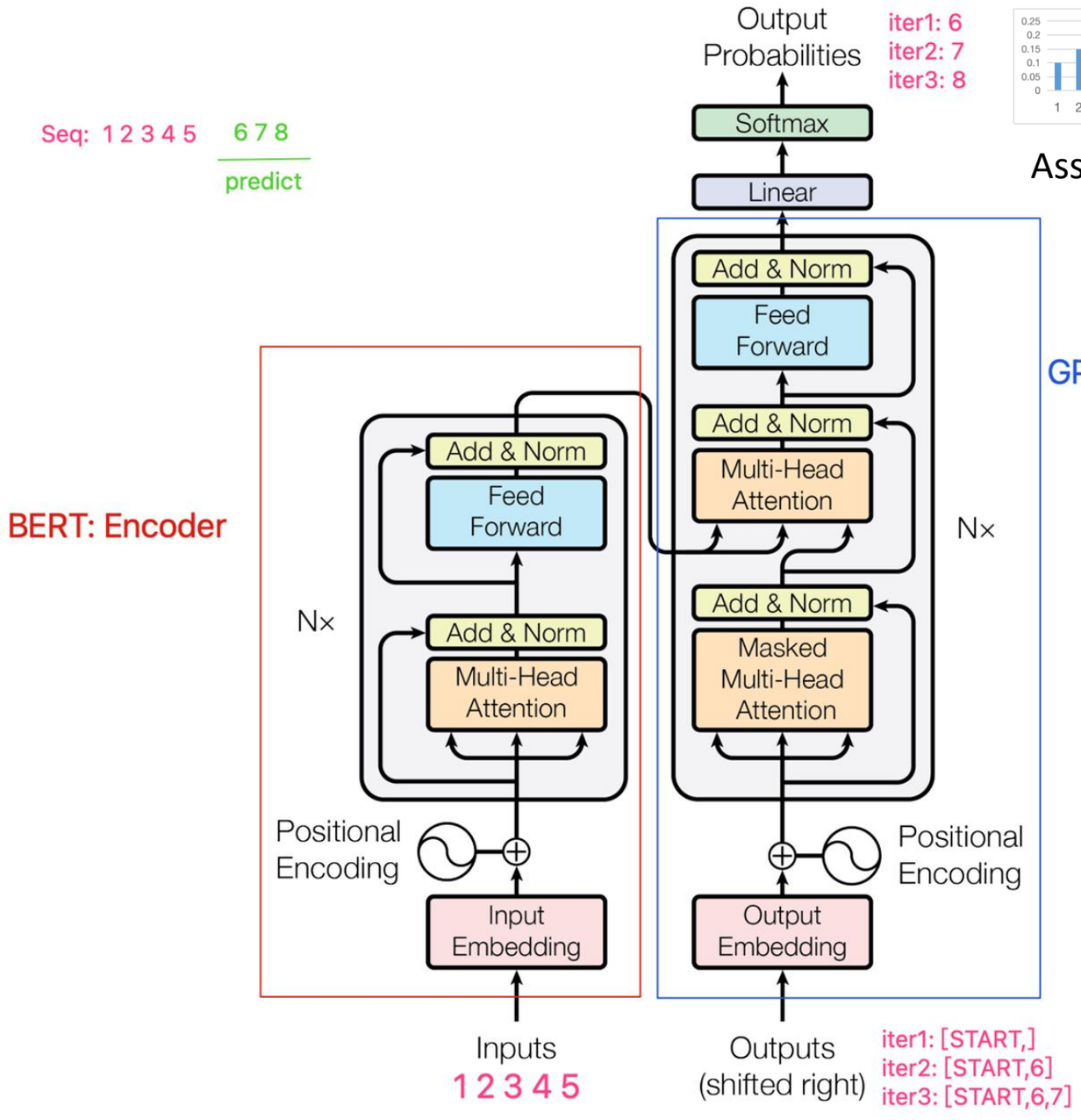
1. 因为RoPE 通过对 Q 和 K 的旋转，让注意力分数变成位置差的函数
2. 但是V是内容本身信息，不需要被扭曲

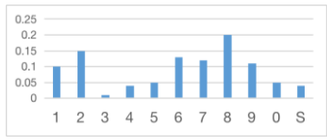
How to build a model with self-attention layers?

That is **Transformer**

Transformer Overview

Now, we have the Self Attention Layer, and we have our input texts are tokenized and processed as embeddings. How to build a scalable model?

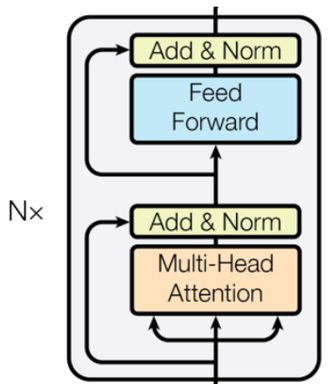




Assume DICT={0,1,2,3,4,5,6,7,8,9,0,S}, where S means SPACE

- ***GPT*** using the decoder structure, ***BERT*** using the encoder structure
- Multi-head attention can be parallel execution, thus can be optimized when resource enough, while the decoder cannot be parallel execution because it relies on the previous output words. It means that the run time of GPT mainly limited to the output length when compute resource is enough
- Output *Linear (LMHead)* may shared the parameters with the *Embedding*

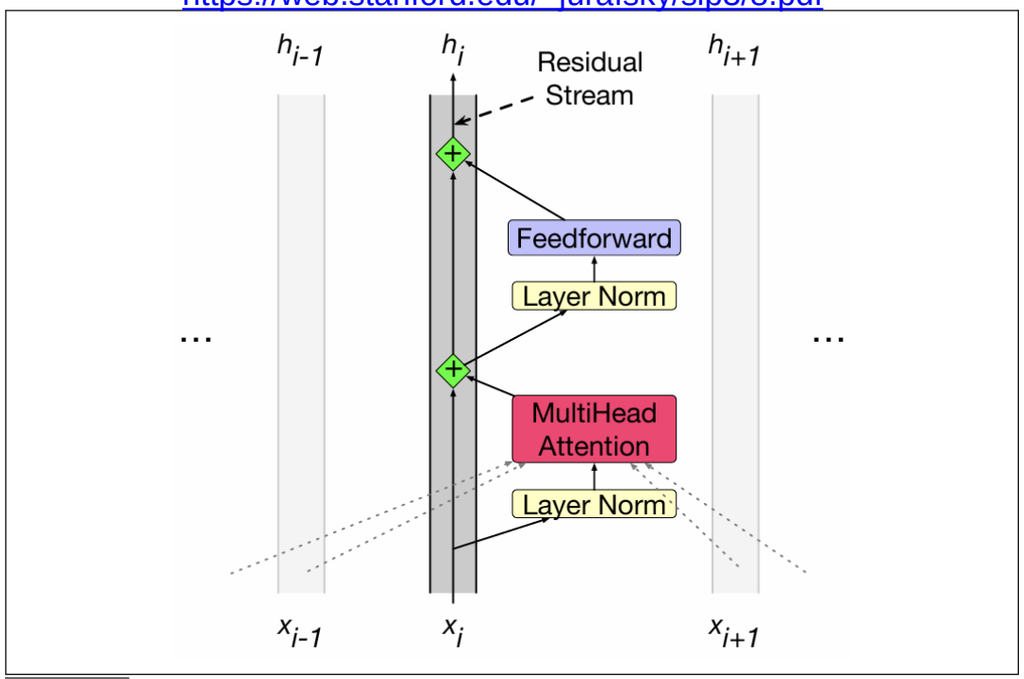
Transformer - Resnet View



Resnet View

Pre-norm is more stable and used on most of the latest LLMs

https://web.stanford.edu/~jurafsky/slp3/8.pdf



Vanilla Transformer Block

$$x_{l+1} = \text{LayerNorm}(x_l + \text{Sublayer}(x_l))$$

Transformer Block in Latest GPTs

$$x_{l+1} = x_l + \text{Sublayer}(\text{LayerNorm}(x_l))$$

```python
self.transformer = nn.ModuleDict(dict(
    wte = nn.Embedding(config.vocab_size, config.n_embd),
    wpe = nn.Embedding(config.block_size, config.n_embd),
    drop = nn.Dropout(config.dropout),
    h = nn.ModuleList([Block(config) for _ in range(config.n_layer)]),
    ln_f = LayerNorm(config.n_embd, bias=config.bias),
))
self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
# with weight tying when using torch.compile() some warnings get generated:
# "UserWarning: functional_call was passed multiple values for tied weights.
# This behavior is deprecated and will be an error in future versions"
# not 100% sure what this is, so far seems to be harmless. TODO investigate
self.transformer.wte.weight = self.lm_head.weight # https://paperswithcode.com/method
```

Pre-norm require a final norm for output

```python
class Block(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.ln_1 = LayerNorm(config.n_embd, bias=config.bias)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = LayerNorm(config.n_embd, bias=config.bias)
        self.mlp = MLP(config)

    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x
```

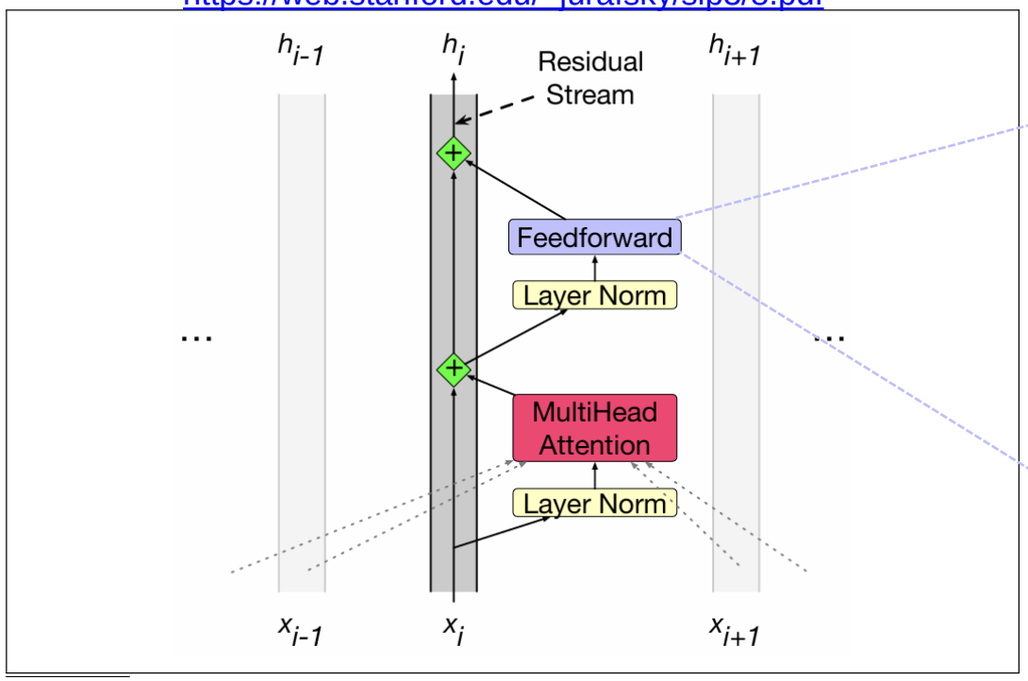
layer norm is not applied to an entire transformer layer, but just to the embedding vector of each token independent.

Pre-norm and residual implementation

https://github.com/karpathy/nanoGPT/blob/93a43d9a5c22450bbf06e78da2cb6eeef084b717/model.py#L126

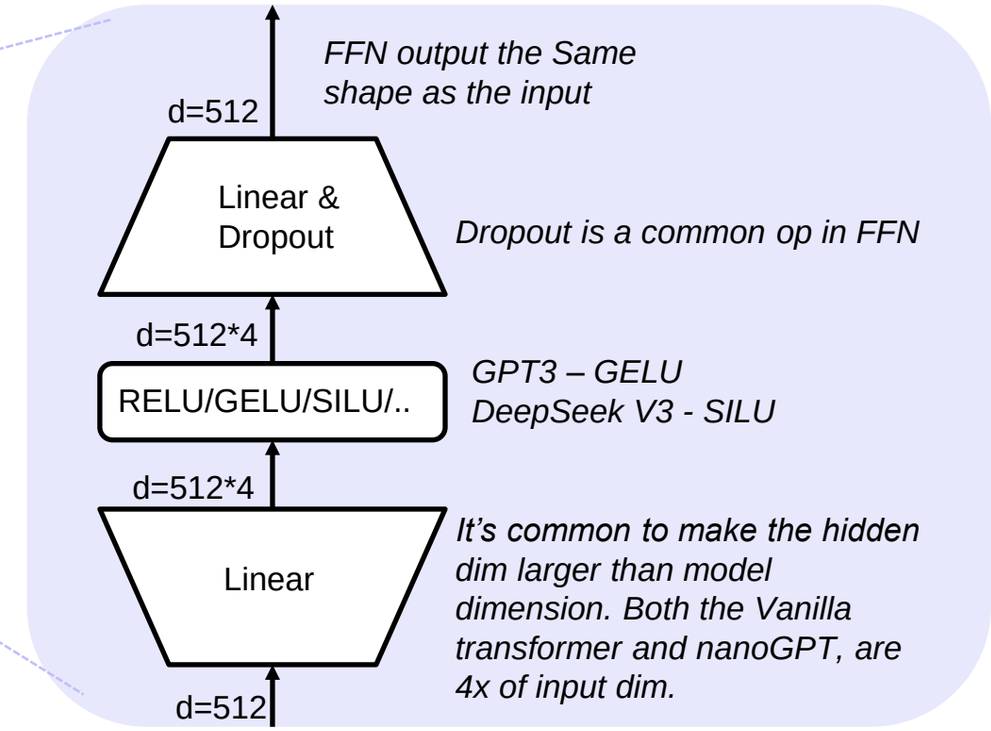
Transformer – Feed Forward

https://web.stanford.edu/~jurafsky/slp3/8.pdf



Transformer Block in Latest GPTs

$$x_{l+1} = x_l + \text{Sublayer}(\text{LayerNorm}(x_l))$$



Why not all attention layers, why we need FFN?

- **There are no non-linear transformer in attention layers**. You may say, softmax not non-linear op? But softmax in attention is just for weight normalization, not applied on Values directly.
- **FFN has the memory function**. If we count the parameters, FFN account for around 2/3 of the total parameters of Transformer. For example, if model d = 512, then parameters of FFN=512*(512*4)*2, while parameters of attention layer in Original GPT = 512*512*3

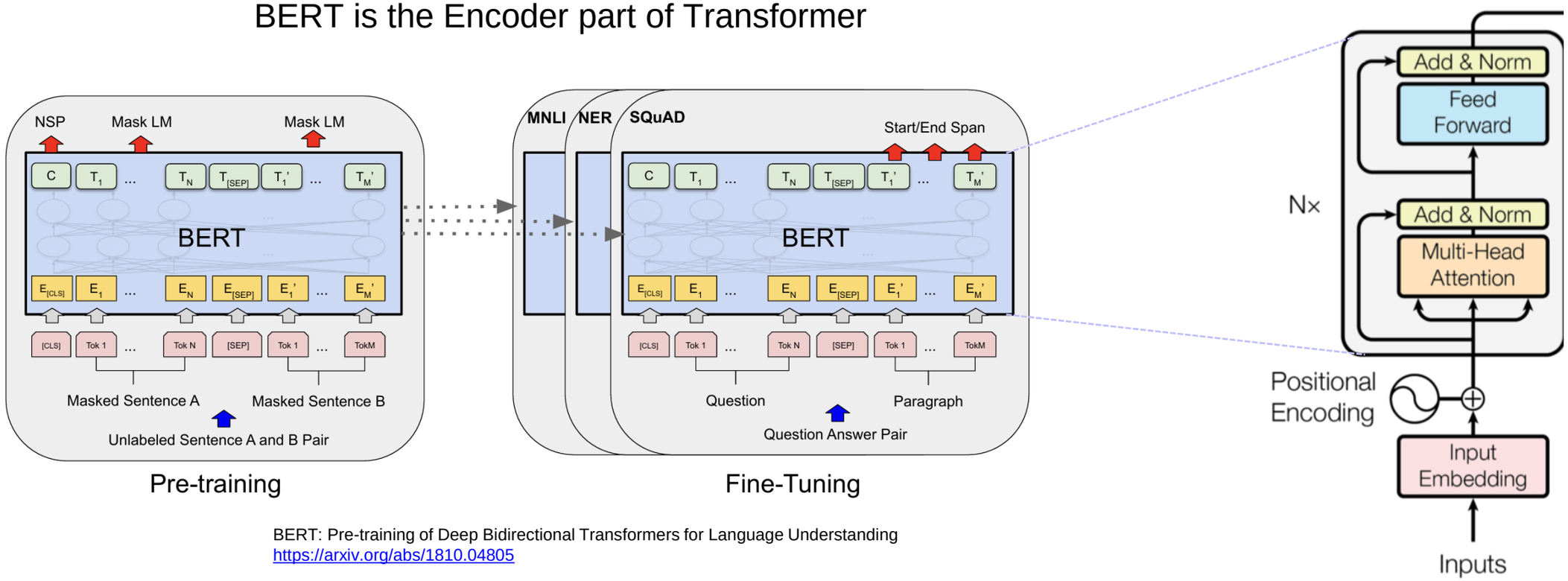
Like LayerNorm, the FFN also applied on each token independent while shared parameters, not related to sequence length,

```
class MLP(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.c_fc    = nn.Linear(config.n_embd, 4 * config.n_embd, bias=config.bias)
        self.gelu    = nn.GELU()
        self.c_proj  = nn.Linear(4 * config.n_embd, config.n_embd, bias=config.bias)
        self.dropout = nn.Dropout(config.dropout)

    def forward(self, x):
        x = self.c_fc(x)
        x = self.gelu(x)
        x = self.c_proj(x)
        x = self.dropout(x)
        return x
```

https://github.com/karpathy/nanoGPT/blob/93a43d9a5c22450bbf06e78da2cb6eeef084b717/model.py#L78

BERT

BERT is the Encoder part of Transformer



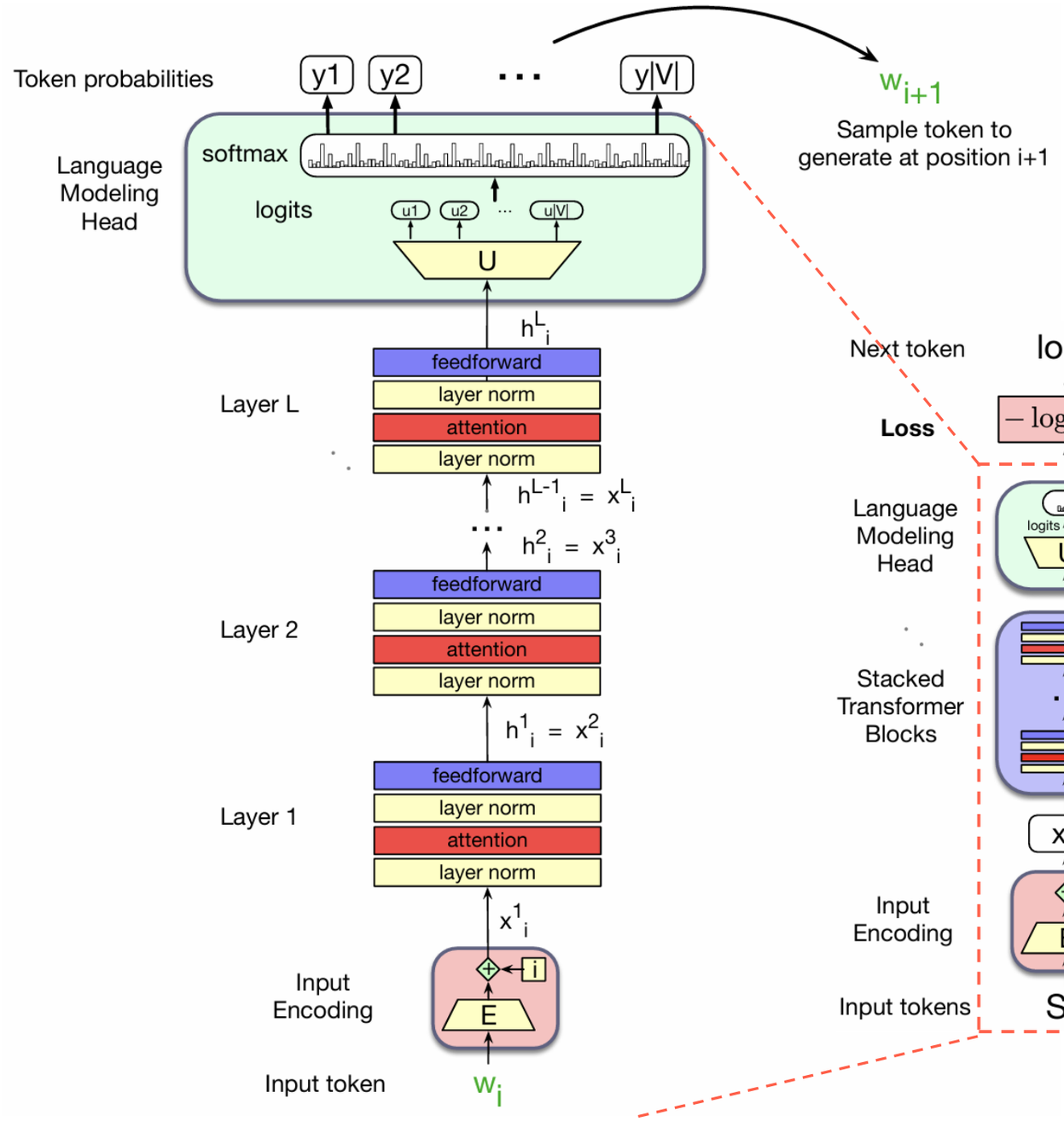
BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
https://arxiv.org/abs/1810.04805

Key points of BERT：

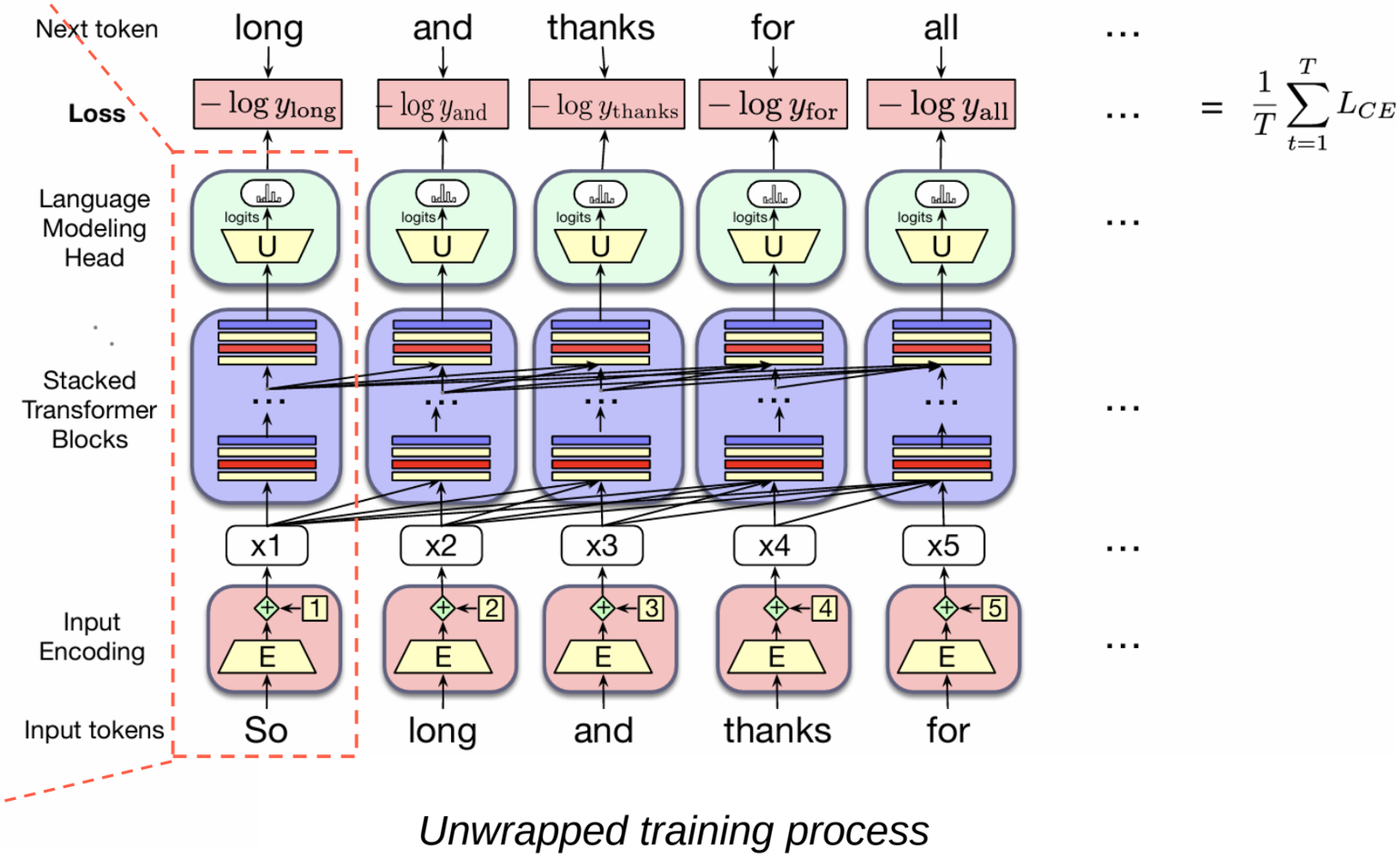
1. Keep the same structure as Transformer’s Encoder
2. Large text data size
3. Unsupervised pre-training and fine-tuning on down-steam tasks, unsupervised tasks such as
   - MLM：完型填空
   - NSP：预测下一句

GPT

GPT is the Decoder part of Transformer



More about sampling please ref:
https://xiahouzuoxin.github.io/posts/控制模型创造力的艺术llm输出策略解析/



Unwrapped training process

https://web.stanford.edu/~jurafsky/slp3/8.pdf

Calculate GPT Model Size

GPT Block	Parameter Volume (ignored bias,norm etc)
Multi-Head Attention	QKV together 3 * d^2
Output Projection After Every Attention Layer	d^2
FFN	The most common case, assume expanding input dim by 4x in FFN, 2 Linear layers together 2 * (1*d) * (4*d) = 8 * d^2
Embeddings	Text token embedding d * V, pos embedding d * L
Output LM Head	Shared with embedding
Total	12 * d^2 * N + d * (V+L)

d – model dimension (embedding dim)
V – vocabulary size
L – block size (sequence length)
N – Number of transformer block layers

Example:

GPT-3, d=12288, N=96, V=50K, L=1024

Model Size = 12 * 12288^2 * 96 + 12288 * (50K+1024) = 174573158400 ~ *175 Billion*

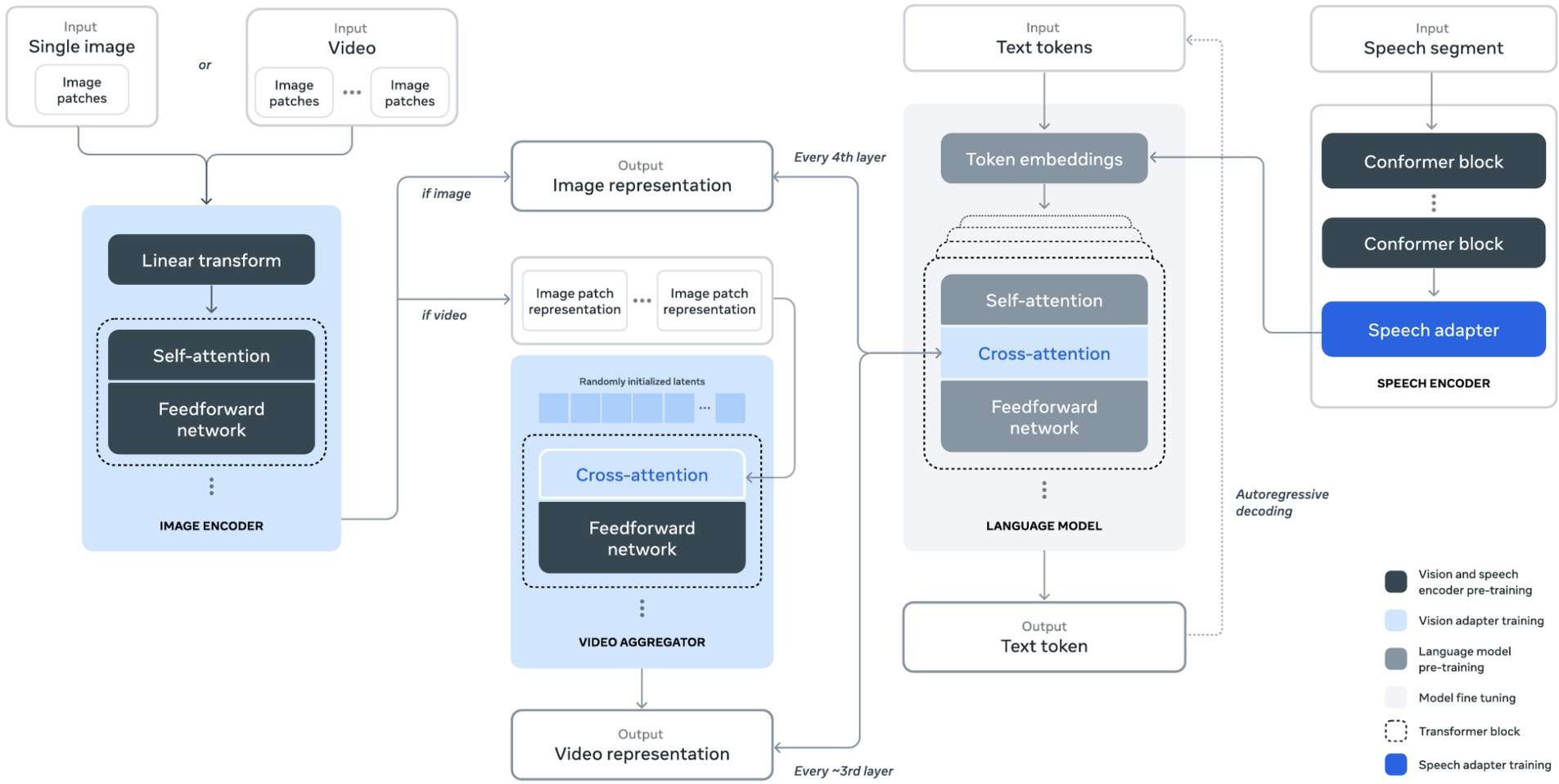
Multimodal

How to learn when there're image/video/speech inputs?

⇩

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

Where
Q: text embedding
K,V: multimodal embedding



The Llama 3 Herd of Models: https://arxiv.org/pdf/2407.21783

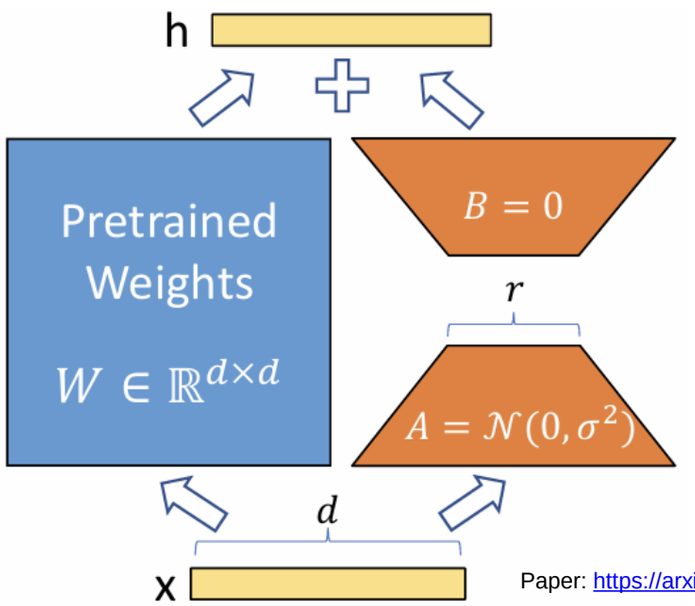
Figure 28 Illustration of the compositional approach to adding multimodal capabilities to Llama 3 that we study in this paper. This approach leads to a multimodal model that is trained in five stages: **(1)** language model pre-training, **(2)** multi-modal encoder pre-training, **(3)** vision adapter training, **(4)** model finetuning, and **(5)** speech adapter training.

LLM Optimizations

LoRA

Easy way to fine-tune an LLM?

- Fine-tune on raw LLM parameters for downstream tasks is hard and cost inefficient
- Easy to over-fitting with small task-specific data set



Low-Rank Adaptation, or LoRA

- Freezes the pre trained model weights and injects trainable rank decomposition matrices into each layer of the Transformer architecture
- Greatly reducing the number of trainable parameters for downstream tasks

Paper: https://arxiv.org/pdf/2106.09685

Update the model by equation $h = W_0 x + \Delta W x = W_0 x + BAx$ *Where W0 is frozen when training*

Only B and A is trained with a very small dimension r << d.
Computing complexity from O(d^2) to O(2dr), it reduce a lot when r << d.

```
def forward(self, x: torch.Tensor):
    def T(w):
        return w.transpose(0, 1) if self.fan_in_fan_out else w
    if self.r > 0 and not self.merged:
        result = F.linear(x, T(self.weight), bias=self.bias)
        result += (self.lora_dropout(x) @ self.lora_A.transpose(0, 1) @ self.lora_B.transpose(0, 1)) * self.scaling
        return result
    else:
        return F.linear(x, T(self.weight), bias=self.bias)
```

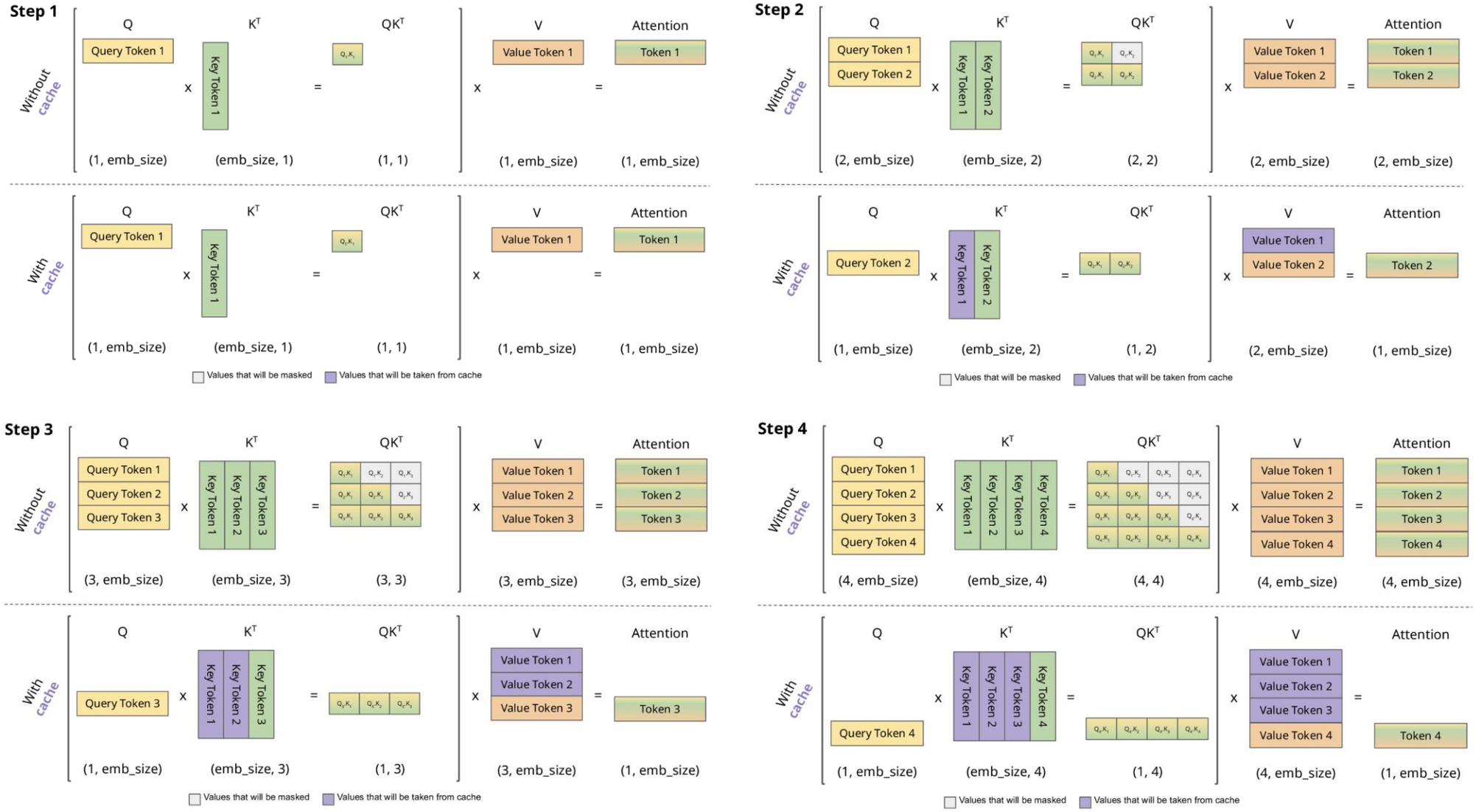
https://github.com/microsoft/LoRA/blob/c4593f060e6a368d7bb5af5273b8e42810cdef90/loralib/layers.py#L149

KV Cache

- There are ***many redundant calculations for Keys and Values during GPT inference***. ***The computed K/V pairs can be cached for generating the next token***. This is essentially a ***space–time tradeoff***, as it reduces computation at the cost of higher memory usage. KV Cache mainly used in decoder, as the encoder is computed in parallel.

Peak cached memory =
BatchSize * (InputLen+OutputLen) * EmbDim * Layers * 2 * sizeof(datatype)

LARGE



https://cloudthrill.ca/kv_cache-explained

KV Cache Optimize – Compression

Issue: KV Cache eat lots of memories, how to reduce the KV cache memories while the LLM performance not drop a lot in inference?

https://github.com/InternLM/lmdeploy/blob/main/docs/en/quantization/kv_quant.md shows that even keep only 50% memory by INT8 KV ***quantization***, LLM can still keep 98%+ performance

-	-	-	llama2-7b-chat	-	-	internlm2-chat-7b	-	-	internlm2.5-chat-7b	-	-	qwe 7b-
dataset	version	metric	kv fp16	kv int8	kv int4	kv fp16	kv int8	kv int4	kv fp16	kv int8	kv int4	fp16
ceval	-	naive_average	28.42	27.96	27.58	60.45	60.88	60.28	78.06	77.87	77.05	70.5
mmlu	-	naive_average	35.64	35.58	34.79	63.91	64	62.36	72.30	72.27	71.17	61.4
triviaqa	2121ce	score	56.09	56.13	53.71	58.73	58.7	58.18	65.09	64.87	63.28	44.6
gsm8k	1d7fe4	accuracy	28.2	28.05	27.37	70.13	69.75	66.87	85.67	85.44	83.78	54.9
race-middle	9a54b6	accuracy	41.57	41.78	41.23	88.93	88.93	88.93	92.76	92.83	92.55	87.3
race-high	9a54b6	accuracy	39.65	39.77	40.77	85.33	85.31	84.62	90.51	90.42	90.42	82.5

H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models reduced the KV cache memory by 90% with little accuracy drop, by using static sparsity + H2 eviction algorithm

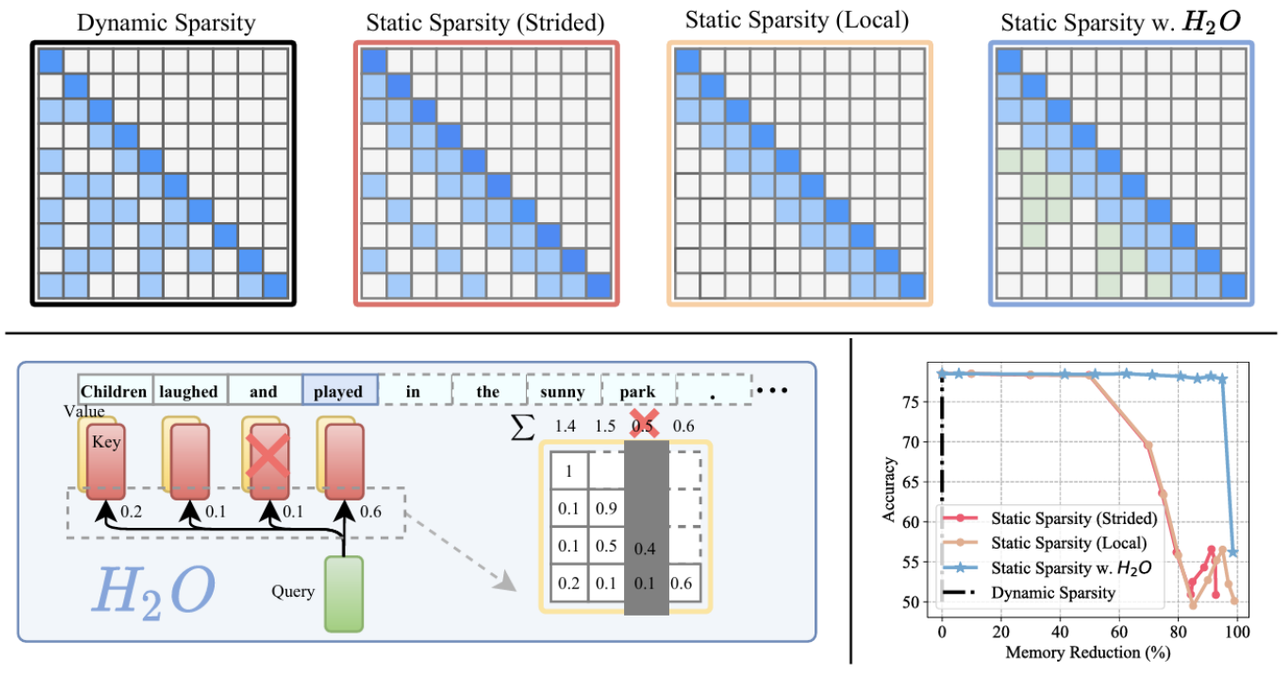


Figure 1: Upper plots illustrate symbolic plots of an attention map deploying different KV cache policies in LLM generation. Lower right: contrasts their accuracy-memory trade-off. Left: the overview of H_2O framework.

KV Cache Optimize - MHA/MQA/GQA

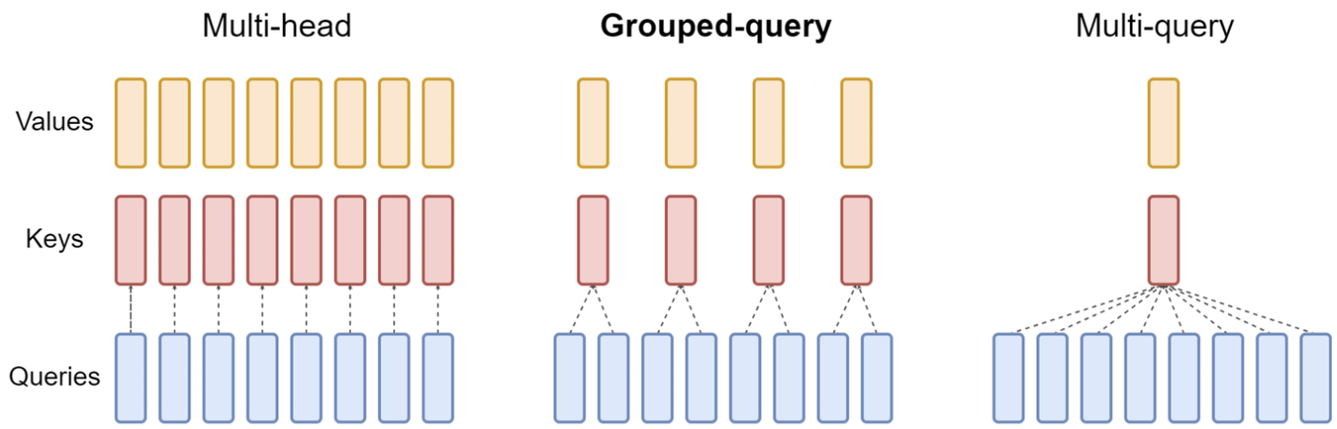


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

GQA:Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. https://arxiv.org/pdf/2305.13245

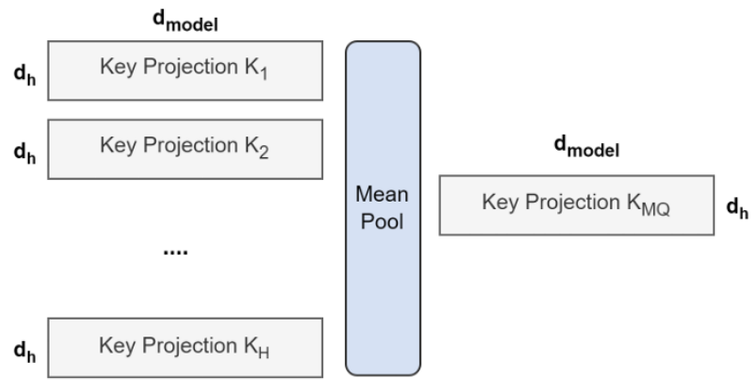
Issue: KV Cache costs more memories when in **multi-head attention (MHA)** situation

Multi-query Attention (MQA)

Performance worse

Grouped-query Attention (GQA)

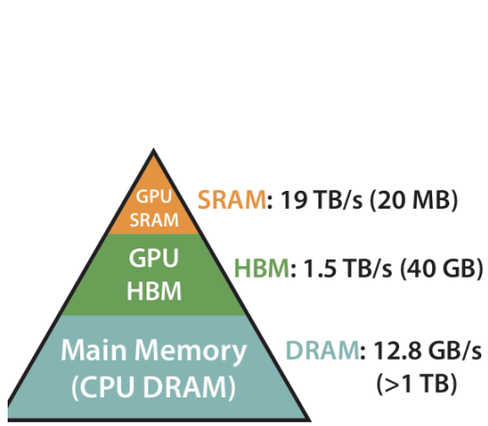
Multiple heads share the single Key and Value. Given a pre-trained MHA model, we can continue a small steps pre-training under the ***mean pooling of all K/Vs structure***. Then just need to cached the pooled K/Vs when inference.



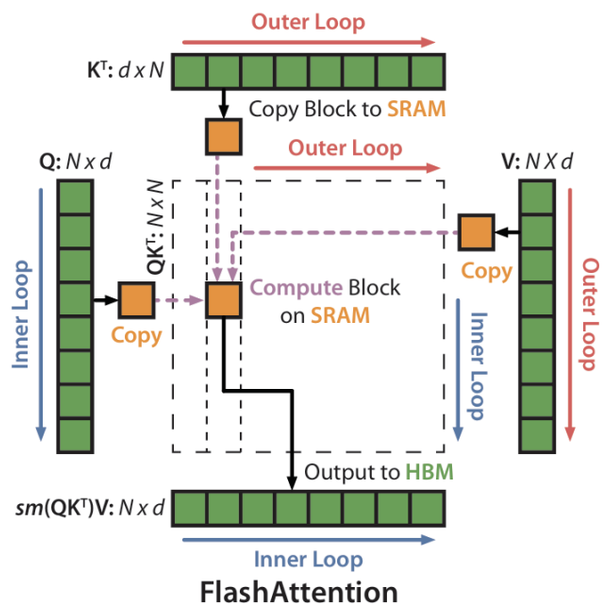
Split multi-heads to multiple groups, and implement MAQ in each group

Flash Attention

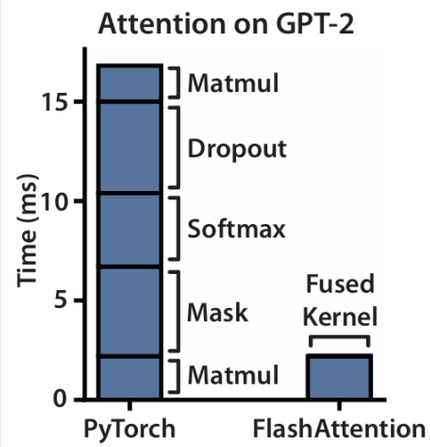
- Standard attention stores, reads, and writes keys, queries, and values in High Bandwidth Memory (HBM), ***but HBM access is relatively slow***.
- FlashAttention significantly accelerates attention computation by ***reorganizing the operations to minimize HBM reads and writes***.
- FlashAttention can be applied in both training and inference stages. And it's applied on almost all the latest LLMs



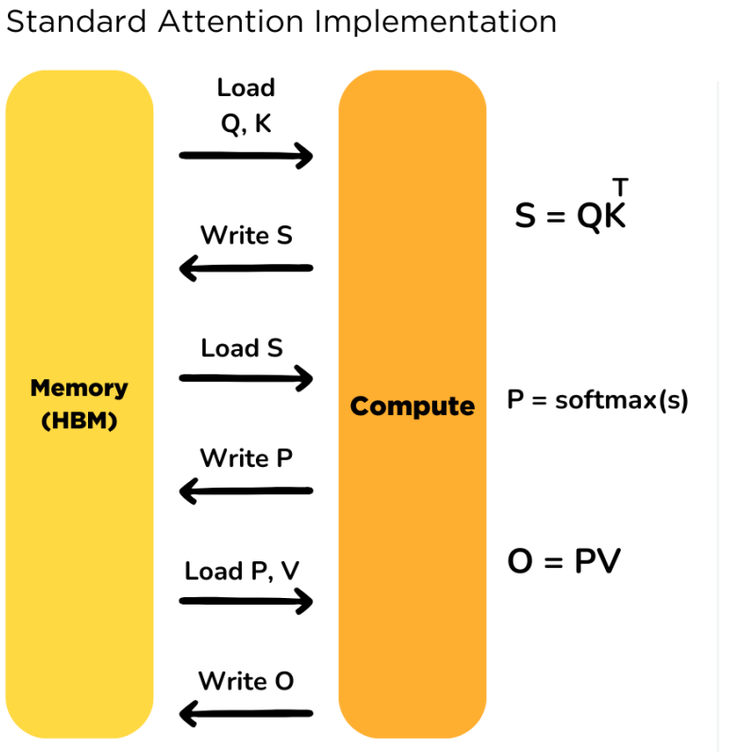
Memory Hierarchy with Bandwidth & Memory Size

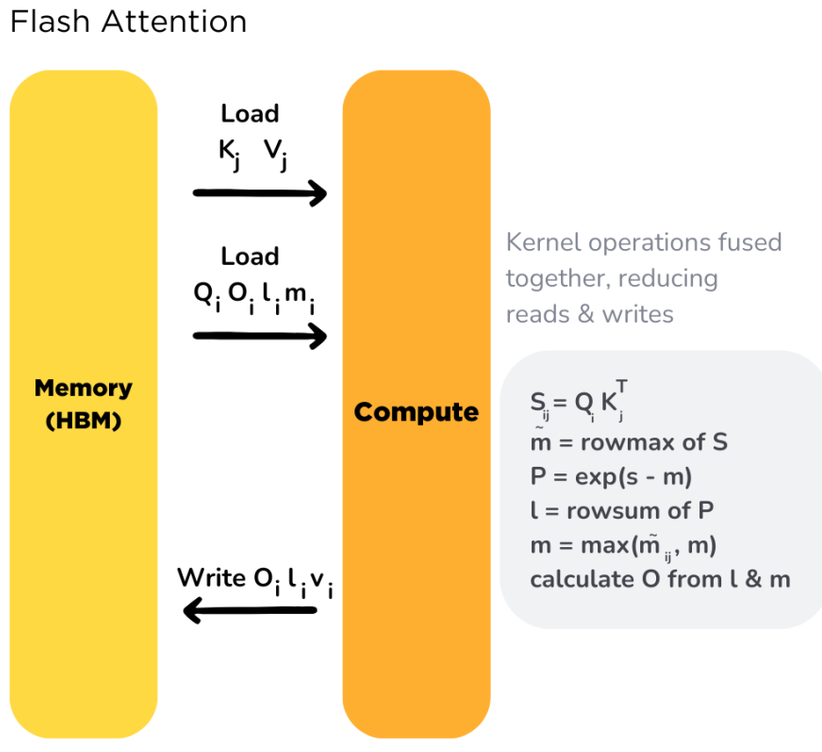


FlashAttention



[FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness]





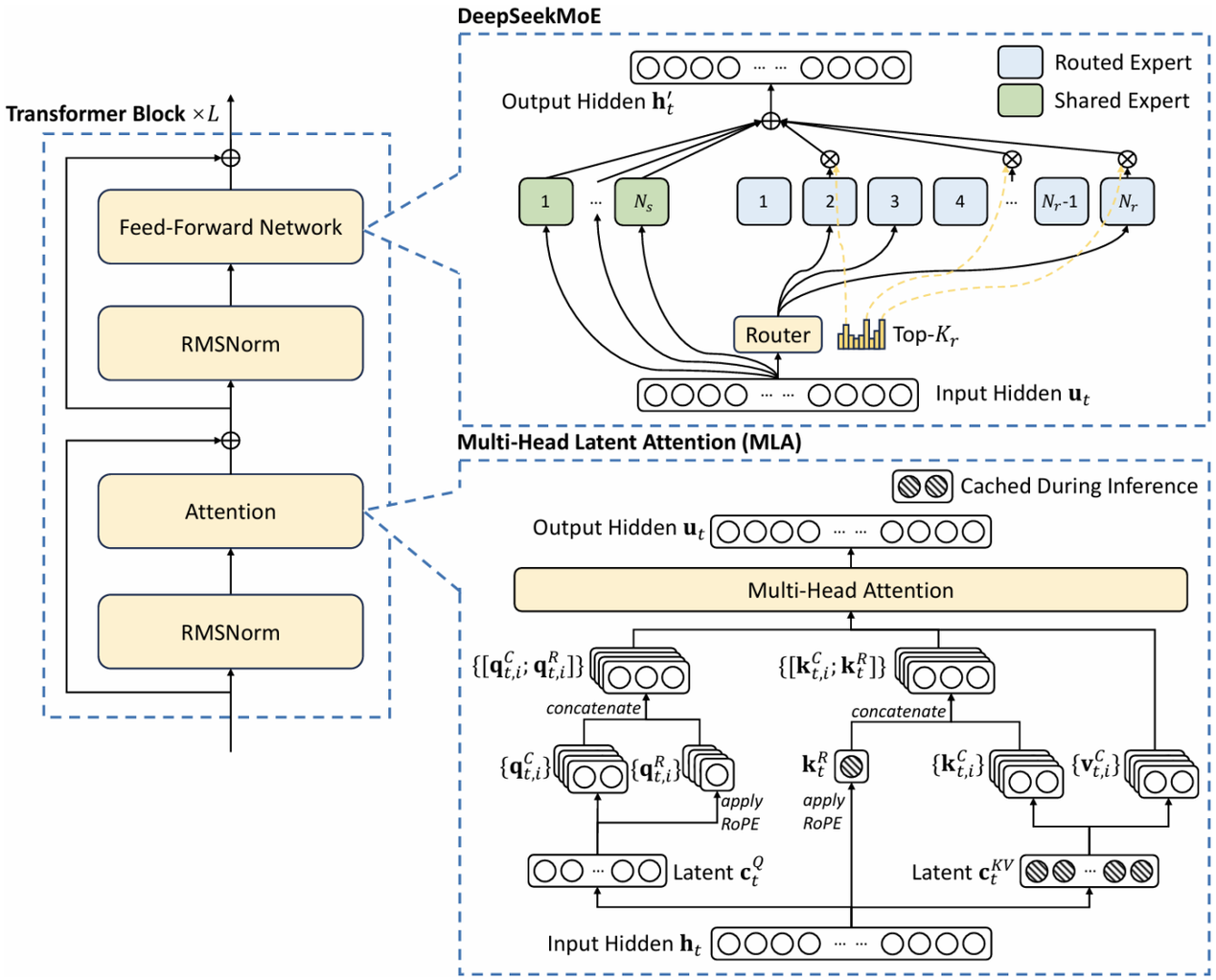
Initialize O, l and m matrices with zeroes. m and l are used to calculate cumulative softmax. Divide Q, K, V into blocks (due to SRAM's memory limits) and iterate over them, for i is row & j is column.

[https://huggingface.co/docs/text-generation-inference/en/conceptual/flash_attention]

Paged Attention

Efficient Memory Management for Large Language Model Serving with PagedAttention
https://github.com/vllm-project/vllm 基于Paged Attention实现的LLM推理引擎

Deepseek – MoE & MLA



Deepseek V3: https://arxiv.org/abs/2412.19437

MLA

作为对MQA/GQA的进一步改进，MLA也为了优化KV Cache设计的，通过low rank的latent vector，经过一个Projection得到K/V。*最终部署的时候只要Cache latent vector就行（如下蓝色的参数），大大减小了KV Cache的内存开销*

$$\boxed{\mathbf{c}^{KV}_t} = W^{DKV}\mathbf{h}_t,$$
$$[\mathbf{k}^C_{t,1};\mathbf{k}^C_{t,2};...;\mathbf{k}^C_{t,n_h}] = \mathbf{k}^C_t = W^{UK}\mathbf{c}^{KV}_t,$$
$$\boxed{\mathbf{k}^R_t} = \mathrm{RoPE}(W^{KR}\mathbf{h}_t),$$
$$\mathbf{k}_{t,i} = [\mathbf{k}^C_{t,i};\mathbf{k}^R_t],$$
$$[\mathbf{v}^C_{t,1};\mathbf{v}^C_{t,2};...;\mathbf{v}^C_{t,n_h}] = \mathbf{v}^C_t = W^{UV}\mathbf{c}^{KV}_t,$$

RMSNorm

不同于标准的Transformer采用LayerNorm，Deepseek为了追求计算上更轻，采用的RMSNorm归一化

给定输入向量 $x \in \mathbb{R}^d$，RMSNorm 的定义是：

$$\mathrm{RMSNorm}(x) = \frac{x}{\mathrm{RMS}(x)} \cdot g$$

其中

$$\mathrm{RMS}(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}$$

g 是可学习的缩放参数（通常为与 x 同维度的向量），

ϵ 是防止除零的常数。

> ✅ 与 LayerNorm 不同的是，RMSNorm **不减均值**，只根据平方和（均方根）进行归一化。

MoE

作为对原FFN的改进，Deepseek采用了MOE，也是基于MOE推理小考虑的。MoE能在训练的时候加大模型参数容量，提升LLM效果，但是在推理的时候却只需要激活部分Experts的参数，而不损失推理速度。

为什么MoE能提升效果?

1. 相同推理计算的情况下，训练参数的容量更大（LLM中Scaling Raw决定模型大小与效果成正比）
2. MoE本身也有类似Model Ensemble的效果（很早其实在推荐多目标场景中就有被使用）

MoE的Experts选择是每个token不一样吗?

1. 是的。MoE包括Router/Gating（决定选哪些Experts）和 Experts加权计算，Router的计算跟token有关