

## 基础篇

- 1、为什么主流LLM都是Decoder-Only的? (Meituan)
- 2、GPT-2参数量怎么计算? (ByteDance-AML-1)
- 3、Lora原理? Lora的参数量计算? Lora参数是包含Attention还是MLP? Lora参数的初始化? 为什么这样初始化? (ByteDance-1) (Ali-Damo-1) (Ele-1)
- 4、LLM预训练参数的初始化?
- 5、Attention有哪几种? 位置编码有哪几种? (ByteDance-2)
- 6、Speculative Decoding 原文是怎么执行的? 正确性保证? (ByteDance-AML-3)
- 7、WordPiece、BPE、BBPE算法
- 8、扩充词表怎么做 (ByteDance-2)
- 9、ROPE、相对位置编码的好处 (ByteDance-1) (Gaode-1)
- 10、特殊token
- 11、Bert用作分类问题细节? Bert模型pretrain任务? Bert可以直接用作计算文本cosine相似度? 计算cosine相似度用的是什么embedding? (Meituan-1)
- 12、介绍 vLLM、FlashAttention (Damo-1)
- 13、Pre-train和SFT的区别 (Ele-3)
- 14、手写transformer (Kwai-1、NetEase-1)
- 15、prenorm / postnorm区别 (Ele-1)
- 16、量化(感知训练、后训练) / 稀疏化 (Ele-1)
- 17、外推能力? (Ele-1)
- 18、Pretrain方法 (Ele-1)
- 19、为什么训练时候Transformer的Decoder生成不是从bos开始的?
- 20、Batch Norm/Layer Norm (幻方-2)
- 21、为什么transformer成为主流方案? 是否有替代方案? (Damo-2)
- 22、transformer的K、Q、V可以使用同一个值吗?
- 23、手写attention (numpy的两种写法)
- 24、Attention为什么要做scale? (JD-1, NetEase-2)
- 25、Transformer的encoder和decoder有什么区别?
- 26、LLM中常用激活函数?
- 27、SFT报NaN要怎么排查?
- 28、DeepSpeed Zero三个阶段? (Ant-1)
- 29、LLM复读机问题产生原因、怎么排查、怎么解决?
- 30、怎么缓解特殊下游的SFT对模型通用能力造成损害? (蔚来-1)
- 31、RMSNorm (PDD-1)
- 32、优化器有哪些, 大模型训练时候到优化器用哪个, Adam和AdamW区别? (Damo-2)
- 33、Llama各代之间的差异? (ByteDance-3)

## 多模态篇

- 1、为什么MLLM普遍都是2阶段训练, 而不是1阶段? (ByteDance-1)
- 2、为什么不将ViT加入到MLLM训练过程中? ViT的参数量以及显存占用量? (ByteDance-1)
- 3、几种MLLM的架构极其特点? 优势? (ByteDance-1)
- 4、clip细节: 数据怎么构造、怎么训练、怎么设计loss (Gaode-1)
- 5、ViT的视觉表征是取哪些embedding? (ByteDance-2)

## Reference

## 基础篇

### 1、为什么主流LLM都是Decoder-Only的?[^1] (Meituan)

先横向比较下，再说为什么：

LLM架构主要分为三类：

Encoder-Only、Encoder-Decoder、Decoder-Only

首先纯Encoder+MLM的模型（bert）只适合做NLU，不适合做生成。

现在做Encoder-Decoder、Decoder-Only的比较：

- 1) decoder-Only的zero-shot效果更好，因为decoder-Only不会出现低秩问题，建模难度更大
- 2) in-context learning: Decoder-Only更好适应，因为其prompt会更直接作用于decoder参数

但是LLM架构问题和Language Modeling是紧密相关的，其实就是在于用什么样的mask和什么样的objective：

用non-casual modeling的模型在进行multi-task ft之后效果是最好的。

至于为什么现在decoder-only比较火，其实有一点历史原因。

## 2、GPT-2参数量怎么计算？（ByteDance-AML-1）

e.g. GPT2-small 参数量计算

```
{
  "activation_function": "gelu_new",
  "architectures": [
    "GPT2LMHeadModel"
  ],
  "attn_pdrop": 0.1,
  "bos_token_id": 50256,
  "embd_pdrop": 0.1,
  "eos_token_id": 50256,
  "initializer_range": 0.02,
  "layer_norm_epsilon": 1e-05,
  "model_type": "gpt2",
  "n_ctx": 1024,
  "n_embd": 768,
  "n_head": 12,
  "n_layer": 12,
  "n_positions": 1024,
  "resid_pdrop": 0.1,
  "summary_activation": null,
  "summary_first_dropout": 0.1,
  "summary_proj_to_labels": true,
  "summary_type": "cls_index",
  "summary_use_proj": true,
  "task_specific_params": {
    "text-generation": {
      "do_sample": true,
```

```

        "max_length": 50
    }
},
"vocab_size": 50257
}

```

- input embedding

词表embedding  $50257 * 768 = 38597376$

(不计入, 不可学习) 位置编码embedding  $1024 * 768 = 786432$

- transformer block \* 12

attention模块权重矩阵加偏置:  $768 * 768 * 3 + 768 * 3 = 1771776$

attention结尾的线性变换:  $768 * 768 + 768 = 590592$

第一层全连接:  $768 * 768 * 4 + 768 * 4 = 2362368$

第二层全连接:  $768 * 768 * 4 + 768 = 2360064$

2个Layer Normalization (其中参数量只涉及  $\gamma$  和  $\beta$ ):  $2 * (768 + 768) = 3072$

- output embedding

一般和input embedding共享参数, 一般bert、GPT都是采用这样的策略

相加上述数据, 得到总的参数量为0.12365184B, 与官方公布的124M一致

```

hidden_dim = 768
vocab_size = 50257
n_layers = 12

hidden_dim_2 = 4*hidden_dim
res = 0
res += hidden_dim * vocab_size
for i in range(n_layers):
    res += 4*(hidden_dim * hidden_dim + hidden_dim)
    res += 2*(2*hidden_dim)
    res += hidden_dim*hidden_dim_2+hidden_dim_2
    res += hidden_dim_2*hidden_dim+hidden_dim
print(f"{res/10**9}B")

```

### 3、Lora原理? Lora的参数量计算? Lora参数是包含Attention还是MLP? Lora参数的初始化? 为什么这样初始化? (ByteDance-1) (Ali-Damo-1) (Ele-1)

参考 [https://zhuanlan.zhihu.com/p/665407489?utm\\_id=0](https://zhuanlan.zhihu.com/p/665407489?utm_id=0)

Lora为什么会起作用: 低秩分解 <sup>1</sup>

Armen Aghajanyan<sup>2</sup> 等人在探究微调内在原理中提出，越大的模型有越小的内在维度，微调这个内在维度和在全参数空间中微调能起到相同的效果。

Lora初始化：初始化的时候，用随机高斯分布初始化A，用零矩阵初始化B。然后模型训练变成了更新AB的参数。

微调那一部分：Wq、Wv效果是最好的<sup>3</sup>

Lora微调参数：r秩, alpha学习率放大, dropout

## 4、LLM预训练参数的初始化？

1) 随机初始化：均匀分布，高斯分布

2) 方差缩放（保证每个神经元的输入和输出方差一致，防止梯度消失或者爆炸）：Xavier初始化、Kaiming初始化

为什么不能全零初始化？或全相同值初始化？

对称失效：无论经过多少次网络训练，相同网络层内的参数值都是相同的，这会导致网络在学习时没有重点，对所有的特征处理相同，这很可能导致模型无法收敛训练失败。这种现象被称为**对称失效**。

## 5、Attention有哪几种？位置编码有哪几种？(ByteDance-2)

参考 [https://blog.csdn.net/weixin\\_41862755/article/details/134289843](https://blog.csdn.net/weixin_41862755/article/details/134289843)

注意llama的hf位置编码实现和公式的区别：

```
171
172
173 def rotate_half(x):
174     """Rotates half the hidden dims of the input."""
175     x1 = x[..., : x.shape[-1] // 2]
176     x2 = x[..., x.shape[-1] // 2 :]
177     return torch.cat((-x2, x1), dim=-1)
178
179
180 def apply_rotary_pos_emb(q, k, cos, sin, position_ids):
181     # The first two dimensions of cos and sin are always 1, so we can `squeeze` them.
182     cos = cos.squeeze(1).squeeze(0) # [seq_len, dim]
183     sin = sin.squeeze(1).squeeze(0) # [seq_len, dim]
184     cos = cos[position_ids].unsqueeze(1) # [bs, 1, seq_len, dim]
185     sin = sin[position_ids].unsqueeze(1) # [bs, 1, seq_len, dim]
186     q_embed = (q * cos) + (rotate_half(q) * sin)
187     k_embed = (k * cos) + (rotate_half(k) * sin)
188     return q_embed, k_embed
189
190
```

注意看这个代码实现和公式不对应，找到一个解释<sup>[^1]</sup>：



@naubull2 3 months ago

Thanks for a great explanation! By the way, I was curious when I understood from the initial explanation and the rotational equations, consecutive pairs of coordinates seem to be rotated, as in  $(x_1, x_2) / (x_3, x_4) \dots$  are each rotated. However from most of the implementations as suggested in the video, the codes pair up not by adjacent indices but with a window size of half the dimension, which would be  $(x_1, x_{d//2+1}) / (x_2, x_{d//2+2}) \dots$  since the code states that we split the hdim by half and swap their order.. did I understand correctly or am I missing something?

👍 1 🗨️ Reply

▲ • 1 reply



@EfficientNLP 3 months ago

You are correct. In many implementations, rather than rotating each pair of adjacent dimensions, they choose to split the entire vector in half and rotate the two halves. Ultimately, this does not matter because the dimensions of vectors are interchangeable and do not affect vector addition and multiplication. This is likely to be more efficient from an implementation standpoint and is equivalent to the original formula.

👍 3 🗨️ Reply

## 6、Speculative Decoding 原文是怎么执行的？正确性保证？(ByteDance-AML-3)

参考<sup>[8]</sup>

为什么从  $(q(x)-p(x))_+$  重采样<sup>4</sup>？

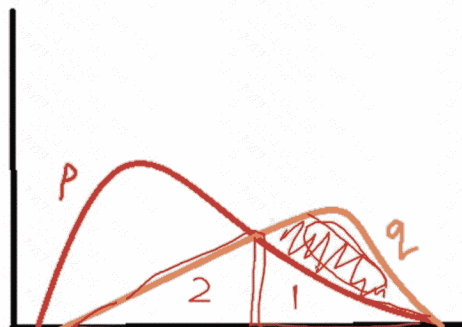
Why sample from  $(q(x) - p(x))_+$ ?

**Goal:** Sample a token from  $q(x)$

**Case 1:** If  $q(x) \geq p(x)$ , then accept

**Case 2:** If  $q(x) < p(x)$ , then accept with probability  $\frac{q(x)}{p(x)}$

Only remaining case:  $q(x) < p(x)$  and rejected, so sample from  $(q(x) - p(x))_+$



始终记得，我们的目标就是要补全q分布。这个概率分布图中，可以看到2区域被reject之后，要补全全部q分布，需要的那一部分就是 $(q(x)-p(x))_+$

## 7、WordPiece、BPE、BBPE算法

中文BPE算法

<https://0809zheng.github.io/2022/07/01/posencode.html>

<https://zhuanlan.zhihu.com/p/652520262>

## 8、扩充词表怎么做 (ByteDance-2)

参考<sup>5</sup>

优点：提高编解码效率、提高上下文长度（因为中文的平均token表示中文字符数提高了）、提高中文能力（存疑）

步骤：

1) 扩充词表: 直接sentencePiece加上就可以（要做去重）

2) Embedding初始化：

- 随机初始化
- 用原词表的均值扩充

## 9、ROPE、相对位置编码的好处（ByteDance-1）（Gaode-1）

参考 <https://0809zheng.github.io/2022/07/01/posencode.html>

## 10、特殊token

BERT、DistilBERT架构

1. "[PAD]": Padding Token。用于将所有句子或输入序列填充到相同的长度，以便模型能够批量处理它们。在处理不同长度的输入时，较短的输入会被这个token填充以达到批中最长序列的长度。
2. "[UNK]": Unknown Token。用于表示未知的单词或无法识别的字符。在预处理数据时，如果遇到词汇表(vocabulary)之外的单词，会用这个token替换。
3. "[CLS]": Classifier Token。在BERT及其变种中，这个token被添加到每个输入序列的开头。在句子分类或其他需要整体序列表示的任务中，"[CLS]" token的最终隐藏状态会被用作整个输入序列的聚合表示。
4. "[SEP]": Separator Token。用作分隔符，它可以分隔两个句子或标记序列的结束。在处理句子对的任务中，比如问答或句子关系模型中，"[SEP]"被用来分隔句子对。
5. "[MASK]": Masking Token。在进行遮蔽语言模型（Masked Language Model, MLM）训练时使用。某些单词会被这个token替换，模型的任务是预测原始的单词。这是BERT训练过程中的一个关键步骤，使得模型能够学习到丰富的语言表示。

RoBERTa、GPT、GPT2、BLOOM、Llama、Falcon

1. "[PAD]" (Padding Token)：与之前解释的相同，用于将序列填充到相同的长度，以便模型能够以批处理的方式处理它们。
2. "[UNK]" (Unknown Token)：也与之前相同，用于表示未知的单词或无法识别的字符，即不在模型的词汇表中的词。
3. "[BOS]" (Beginning Of Sentence Token)：表示句子开始的token。它在模型处理序列数据时用来标记句子的开始位置。在某些模型中，"[BOS]"被用来提示模型开始生成文本或处理一个新的输入序列。
4. "[EOS]" (End Of Sentence Token)：表示句子结束的token。它用于标记序列的结束，让模型知道一个句子或输入序列在哪里结束。在文本生成任务中，当模型生成了"[EOS]" token时，通常意味着完成了生成任务。

## 11、Bert用作分类问题细节？ Bert模型pretrain任务？ Bert可以直接用作计算文本cosine相似度？ 计算cosine相似度用的是什麼embedding？（Meituan-1）

参考<sup>6</sup> [https://github.com/hichenway/CodeShare/blob/master/bert\\_pytorch\\_source\\_code/modeling.py](https://github.com/hichenway/CodeShare/blob/master/bert_pytorch_source_code/modeling.py)

怎么提取sentence-level的特征用于下游任务？一种直观的想法是使用bert提取句子级别的embedding，那么我们有几种选择：

注意，以下几种方案的优劣仅仅是在MLM+NSP任务下pre-train bert的基础上讨论

- CLS Token Embedding

Sentence-BERT<sup>[2]</sup> 说CLS不好，原因就是bert原生的CLS是用于NSP分类任务的，对句子表征效果不好。

- Pooling Context Embedding

Sentence-BERT<sup>[2]</sup> 认为这个特征比CLS好，并且在此基础上可以做fine-tune达到更好的表征效果

但是存在各向异性（Anisotropic）的问题，导致词余弦相似与pretrain词频相关

Anisotropic最早在ICLR<sup>[3]</sup>里面描述，在bert-flow里面定义：“Anisotropic” means word embeddings occupy a narrow cone in the vector space.

除此之外，Sentence-BERT<sup>[2]</sup> 为了能到达用cosine-similarity衡量词距离的效果，在三个obj下进行训练：

**Classification Objective Function.** We concatenate the sentence embeddings  $u$  and  $v$  with the element-wise difference  $|u - v|$  and multiply it with the trainable weight  $W_t \in \mathbb{R}^{3n \times k}$ :

$$o = \text{softmax}(W_t(u, v, |u - v|))$$

where  $n$  is the dimension of the sentence embeddings and  $k$  the number of labels. We optimize cross-entropy loss. This structure is depicted in Figure 1.

**Regression Objective Function.** The cosine-similarity between the two sentence embeddings  $u$  and  $v$  is computed (Figure 2). We use mean-squared-error loss as the objective function.

**Triplet Objective Function.** Given an anchor sentence  $a$ , a positive sentence  $p$ , and a negative sentence  $n$ , triplet loss tunes the network such that the distance between  $a$  and  $p$  is smaller than the distance between  $a$  and  $n$ . Mathematically, we minimize the following loss function:

$$\max(|s_a - s_p| - |s_a - s_n| + \epsilon, 0)$$

with  $s_x$  the sentence embedding for  $a/n/p$ ,  $\|\cdot\|$  a distance metric and margin  $\epsilon$ . Margin  $\epsilon$  ensures that  $s_p$  is at least  $\epsilon$  closer to  $s_a$  than  $s_n$ . As metric we use Euclidean distance and we set  $\epsilon = 1$  in our experiments.

克服Anisotropic的方法<sup>7</sup>：

- 1) 映射
- 2) 主成分消除
- 3) 正则化

- EOS Token Embedding

这是CLIP方法中文本编码器的特征提取方式，其实类似于Pooling Context Embedding，但是注意clip的文本编码器是从随机初始化开始训练的



## 12、介绍 vLLM、FlashAttention (Damo-1)

## 13、Pre-train和SFT的区别 (Ele-3)

参考 <sup>8</sup>

数据格式：

1、 `preprocess_pretrain_dataset` 处理PreTraining阶段的数据

- 数据组成形式：
  - 输入input: X1 X2 X3
  - 标签labels: X1 X2 X3
- 典型的Decoder架构的数据训练方式；

2、 `preprocess_supervised_dataset` 处理SFT阶段的数据

- 数据组成形式：
  - 输入input: prompt response
  - 标签labels: **-100 ... -100** response
- 这里面labels的重点在于prompt部分的被**-100**所填充，主要作用在下面会介绍到。

Loss：

都是算cross entropy

但是SFT只计算response部分的loss

## 14、手写transformer (Kwai-1、NetEase-1)

主要几个部分：MHA、FFN、Positional Embedding、Encoder&Decoder

## 15、prenorm / postnorm区别 (Ele-1)

前比较明确的结论是：同一设置之下，Pre Norm结构往往更容易训练，但最终效果通常不如Post Norm <sup>9</sup>。(迁移性能，就是sft之后在下游的性能)

Pre Norm:  $\mathbf{x}_{t+1} = \mathbf{x}_t + F_t(\text{Norm}(\mathbf{x}_t))$

Post Norm:  $\mathbf{x}_{t+1} = \text{Norm}(\mathbf{x}_t + F_t(\mathbf{x}_t))$

简单来说，就是prenorm有退化，在transformer layers比较多的情况下，其等效于宽网络，造成退化。postnorm削弱了恒等分支（残差连接），因为postnorm是在residual之后norm的，所以训练任务更难，但是也更不容易收敛。



## 16、量化（感知训练、后训练）/ 稀疏化（Ele-1）

## 17、外推能力？（Ele-1）

## 18、Pretrain方法（Ele-1）

## 19、为什么训练时候Transformer的Decoder生成不是从bos开始的？

## 20、Batch Norm/Layer Norm（幻方-2）

参考 10 11

原文极好 12

手撕layernorm（幻方）

```
import torch
import torch.nn as nn

class LayerNorm(nn.Module):
    """Layer Normalization class"""
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.gamma = nn.Parameter(torch.ones(features))
        self.beta = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True, unbiased=False)
        return self.gamma * (x - mean) / (std + self.eps) + self.beta

# Example usage
features = 768 # Typical BERT hidden size
layer_norm = LayerNorm(features)

# Dummy input tensor (batch_size, seq_length, features)
x = torch.randn(10, 20, features)
normalized_x = layer_norm(x)
print(normalized_x)
```

## 21、为什么transformer成为主流方案？是否有替代方案？（Damo-2）

- 从AI上来说，行之有效的架构：

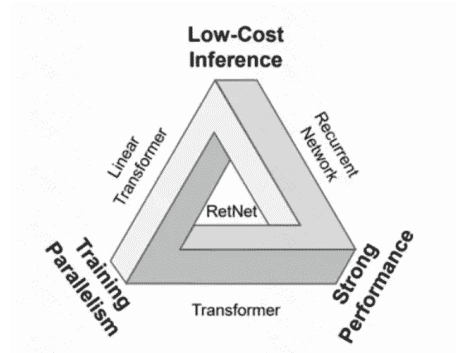
attention + mlp (memory network);

一些研究文章认为它优化上等同SVM，原理上等同GNN，并行度上优于RNN

- 从system上来说，架构适用于GPU计算，并行训练是十分高效的
- 训练范式上是自监督，无需数据标注，大规模预训练与scaling law

## 替代方案 <sup>13</sup>

Transformer 的不可能三角



RWKV：节约内存和计算

RetNet：并行计算循环推理

Mamba

## 22、transformer的K、Q、V可以使用同一个值吗？

### 参考 <sup>14</sup>

K和Q的点乘是为了得到一个attention score 矩阵，用来对V进行提纯。K和Q使用了不同的 $W_k$ ,  $W_q$ 来计算，可以理解为是在不同空间上的投影。正因为有了这种不同空间的投影，增加了表达能力，这样计算得到的attention score矩阵的泛化能力更高。这里解释下我理解的泛化能力，因为K和Q使用了不同的 $W_k$ ,  $W_q$ 来计算，得到的也是两个完全不同的矩阵，所以表达能力更强。

但是如果不用Q，直接拿K和K点乘的话，你会发现 attention score 矩阵是一个**对称矩阵**。因为是同样一个矩阵，都投影到了同样一个空间，所以泛化能力很差。

**attention score不需要是对称的原因**：词和词的修饰不是对称的，比如“我是一个男孩”这句话，男孩对修饰我的重要性应该要高于我修饰男孩的重要性。

## 23、手写attention (numpy的两种写法)

- np.matmul

```
def softmax(x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return e_x / e_x.sum(axis=-1, keepdims=True)
```

```

def user_multi_head_attention_forward(query, key, value, embd_dim, num_heads,
in_proj_weight, out_proj_weight):
    w_q, w_k, w_v =
in_proj_weight[:embd_dim,:], in_proj_weight[embd_dim:2*embd_dim,:], in_proj_weight[2*embd_d
im:,:]
    q = np.matmul(query, w_q)
    k = np.matmul(key, w_k)
    v = np.matmul(value, w_v)

    batch_size, seq_length, embed_dim = q.shape
    head_dim = embed_dim // num_heads
    q = q.reshape(batch_size, seq_length, num_heads, head_dim)
    k = k.reshape(batch_size, seq_length, num_heads, head_dim)
    v = v.reshape(batch_size, seq_length, num_heads, head_dim)

    # Using np.einsum to compute the scores
    scores = np.einsum('bqhd,bkhd->bhqk', q, k) / np.sqrt(head_dim)
    weights = softmax(scores)

    # Using np.einsum to compute the weighted sum of values
    att_output = np.einsum('bhqk,bkhd->bqhd', weights, v).reshape(batch_size, seq_length,
embed_dim)

    output = np.matmul(att_output, out_proj_weight)
    return output

```

- np.einsum

```

def softmax(x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return e_x / e_x.sum(axis=-1, keepdims=True)

def user_multi_head_attention_forward(query, key, value, embd_dim, num_heads,
in_proj_weight, out_proj_weight):
    w_q, w_k, w_v =
in_proj_weight[:embd_dim,:], in_proj_weight[embd_dim:2*embd_dim,:], in_proj_weight[2*embd_d
im:,:]
    q = np.matmul(query, w_q)
    k = np.matmul(key, w_k)
    v = np.matmul(value, w_v)

    batch_size, seq_length, embed_dim = q.shape
    head_dim = embed_dim // num_heads
    q = q.reshape(batch_size, seq_length, num_heads, head_dim).transpose(0, 2, 1, 3)
    k = k.reshape(batch_size, seq_length, num_heads, head_dim).transpose(0, 2, 3, 1)
    v = v.reshape(batch_size, seq_length, num_heads, head_dim).transpose(0, 2, 1, 3)

    scores = np.matmul(q, k) / np.sqrt(head_dim)
    weights = softmax(scores)

```

```

    att_output = np.matmul(weights, v).transpose(0, 2, 1, 3).reshape(batch_size,
seq_length, embed_dim)
    output = np.matmul(att_output, out_proj_weight)
    return output

```

- 测试脚本

```

import numpy as np
from typing import Callable
import torch

batch_size, seq_len, embd_dim, num_heads = [int(_) for _ in input().split()]

np.random.seed(batch_size + seq_len + embd_dim + num_heads)
query = np.random.randn(batch_size, seq_len, embd_dim).astype(np.float32)
key = np.random.randn(batch_size, seq_len, embd_dim).astype(np.float32)
value = np.random.randn(batch_size, seq_len, embd_dim).astype(np.float32)

def softmax(x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return e_x / e_x.sum(axis=-1, keepdims=True)

def user_multi_head_attention_forward(query, key, value, embd_dim, num_heads,
in_proj_weight, out_proj_weight):
    w_q, w_k, w_v =
in_proj_weight[:embd_dim,:], in_proj_weight[embd_dim:2*embd_dim,:], in_proj_weight[2*embd_d
im:,:]

    q = np.matmul(query, w_q)
    k = np.matmul(key, w_k)
    v = np.matmul(value, w_v)

    batch_size, seq_length, embed_dim = q.shape
    head_dim = embed_dim // num_heads
    q = q.reshape(batch_size, seq_length, num_heads, head_dim)
    k = k.reshape(batch_size, seq_length, num_heads, head_dim)
    v = v.reshape(batch_size, seq_length, num_heads, head_dim)

    # Using np.einsum to compute the scores
    scores = np.einsum('bqhd,bkhd->bhqk', q, k) / np.sqrt(head_dim)
    weights = softmax(scores)

    # Using np.einsum to compute the weighted sum of values
    att_output = np.einsum('bhqk,bkhd->bqhd', weights, v).reshape(batch_size, seq_length,
embed_dim)

    output = np.matmul(att_output, out_proj_weight)
    return output

# 以下为测试代码
def xavier_uniform_(tensor, gain=1.): # 方阵的 xavier 初始化
    a = gain * np.sqrt(3 / tensor.shape[0])

```

```

res = np.random.uniform(low=-a, high=a, size=tensor.shape)
return (res + res.T) / 2 # 把矩阵变成对称的, 不用考虑实现时要不要转置

def test_multi_head_attention_forward(mha_impl_func: Callable, in_proj_weight:
np.ndarray, out_proj_weight: np.ndarray):
    # Run mha
    user_impl_out = mha_impl_func(query, key, value, embd_dim, num_heads,
                                   in_proj_weight, out_proj_weight)

    import torch
    import torch.nn as nn
    # 可以和 torch 的实现做对比
    mha = nn.MultiheadAttention(embd_dim, num_heads, bias=False, dropout=0)
    with torch.no_grad():
        mha.in_proj_weight.copy_(torch.from_numpy(in_proj_weight))
        mha.out_proj.weight.copy_(torch.from_numpy(out_proj_weight))
        mha_out, _ = mha(torch.from_numpy(query).transpose(0, 1),
torch.from_numpy(key).transpose(0, 1),
                        torch.from_numpy(value).transpose(0, 1))
        mha_out = mha_out.transpose(0, 1).numpy()

    assert np.allclose(user_impl_out, mha_out, atol=1e-6), "Outputs do not match!"

weights = [xavier_uniform_(np.random.randn(embd_dim, embd_dim)).astype(np.float32) for _
in range(4)]
in_proj_weight = np.concatenate(weights[:3], axis=0)
out_proj_weight = weights[-1]

# 为方便调试, 你可以用下面这个测试函数在本地验证自己实现的 user_multi_head_attention_forward() 的正
确性。
# 但在提交代码到系统时务必注释掉这行代码, 否则会因为找不到 torch 模块而报错。
# test_multi_head_attention_forward(user_multi_head_attention_forward, in_proj_weight,
out_proj_weight)

result = user_multi_head_attention_forward(query, key, value, embd_dim, num_heads,
in_proj_weight, out_proj_weight)

print(f'{int(np.linalg.norm(result.ravel()) * 100_000)}') # 把结果摊平成一维数组, 求范数然后保
留 5 位小数

```

## 24、Attention为什么要做scale? (JD-1, NetEase-2)

<https://blog.csdn.net/jokerxxy/article/details/116299343>

1. 归一化到均值为0, 方差为1
2. 如果不对softmax的输入做缩放, 那么万一输入的数量级很大, softmax的梯度就会趋向于0, 导致梯度消失。

## 25、Transformer的encoder和decoder有什么区别？

参考 <https://arxiv.org/abs/1706.03762>

## 26、LLM中常用激活函数？

GELU、Swish、SiLU

## 27、SFT报NaN要怎么排查？

## 28、DeepSpeed Zero三个阶段？（Ant-1）

### DeepSpeed Basic

- ZeRO优化器：ZeRO（Zero Redundancy Optimizer）是DeepSpeed中的关键组件之一，它通过优化模型状态的存储和通信来大幅减少所需的内存占用，使得可以在有限的资源下训练更大的模型。
- 分片参数：ZeRO通过对参数、梯度和优化器状态进行分片，将它们平均分配到所有的GPU中，这样每个GPU只存储一部分数据，从而减少了单个设备的内存需求。

### ZeRO-1、ZeRO-2和ZeRO-3的区别

ZeRO分为三个优化级别：ZeRO-1、ZeRO-2和ZeRO-3，每个级别都在前一个级别的基础上进一步减少内存占用。

## 29、LLM复读机问题产生原因、怎么排查、怎么解决？

<https://blog.csdn.net/aigchouse/article/details/139510878>

<https://www.cnblogs.com/mengrennwpu/p/17901318.html>

[https://blog.csdn.net/weixin\\_44826203/article/details/132837387](https://blog.csdn.net/weixin_44826203/article/details/132837387)

## 30、怎么缓解特殊下游的SFT对模型通用能力造成损害？（蔚来-1）

KL散度约束、混入pretrain数据

## 31、RMSNorm（PDD-1）

<https://arxiv.org/pdf/1910.07467>

$$\text{RMSNorm}(x) = \gamma \odot \left( \frac{x}{\sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2}} \right) \quad (1)$$

解释：

- 均方根（RMS）：计算输入向量中各元素平方的平均值的平方根。
- 归一化：将输入向量除以其均方根，达到标准化的效果。
- 缩放参数：使用可学习的参数  $\gamma$  对归一化后的向量进行缩放，增强模型的表达能力。

### 32、优化器有哪些，大模型训练时候到优化器用哪个，Adam和AdamW区别？（Damo-2）

理论上更优的原生Adam算法，有时表现并不如SGD momentum好，尤其是在模型泛化性上。

因此加上L2正则项，即Adam with L2 regularization:

$$x_t \leftarrow x_{t-1} - \alpha \frac{\beta_1 m_{t-1} + (1 - \beta_1)(\nabla f_t + wx_{t-1})}{\sqrt{v_t} + \epsilon}$$

AdamW对这个问题的改进就是将权重衰减和Adam算法解耦，让权重衰减的梯度单独去更新参数，改动点如下图所示：

| Algorithm 2 Adam with L <sub>2</sub> regularization and Adam with decoupled weight decay (AdamW)  |  |
|---|--|
| 1: <b>given</b> $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$  |  |
| 2: <b>initialize</b> time step $t \leftarrow 0$ , parameter vector $\theta_{t=0} \in \mathbb{R}^n$ , first moment vector $m_{t=0} \leftarrow \mathbf{0}$ , second moment vector $v_{t=0} \leftarrow \mathbf{0}$ , schedule multiplier $\eta_{t=0} \in \mathbb{R}$ |  |
| 3: <b>repeat</b>  |  |
| 4: $t \leftarrow t + 1$   |  |
| 5: $\nabla f_t(\theta_{t-1}) \leftarrow \text{SelectBatch}(\theta_{t-1})$   | ▷ select batch and return the corresponding gradient     |
| 6: $g_t \leftarrow \nabla f_t(\theta_{t-1}) + \lambda \theta_{t-1}$   |  |
| 7: $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$   | ▷ here and below all operations are element-wise         |
| 8: $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$   |  |
| 9: $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$   | ▷ $\beta_1$ is taken to the power of $t$                 |
| 10: $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  | ▷ $\beta_2$ is taken to the power of $t$                 |
| 11: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$   | ▷ can be fixed, decay, or also be used for warm restarts |
| 12: $\theta_t \leftarrow \theta_{t-1} - \eta_t \left( \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) + \lambda \theta_{t-1} \right)$  |  |
| 13: <b>until</b> stopping criterion is met  |  |
| 14: <b>return</b> optimized parameters $\theta_t$   |  |

绿色部分是AdamW改动的地方，红色的部分是Adam原有的运算过程。

AdamW算法是目前最热门的优化器，包括LLama2等大模型训练都采用的是AdamW。

### 33、Llama各代之间的差异？（ByteDance-3）



## 多模态篇

### 1、为什么MLLM普遍都是2阶段训练，而不是1阶段？（ByteDance-1）

主要是从数据上来考虑，pretrain阶段的数据是多样的，而instructionTuning阶段的数据只是特定任务的（i.e., 评论生成）

### 2、为什么不将ViT加入到MLLM训练过程中？ViT的参数数量以及显存占用量？（ByteDance-1）

ViT是通过CLIP进行对比学习得到的，有跨模态理解能力和zero-shot的能力。

参数量计算、vit实现参考<sup>[^2]</sup>

另外一版vit实现：<https://zhuanlan.zhihu.com/p/361686988>

### 3、几种MLLM的架构极其特点？优势？（ByteDance-1）

参考<sup>[^3]</sup>

### 4、clip细节：数据怎么构造、怎么训练、怎么设计loss（Gaode-1）

参考<sup>[^23]</sup>

### 5、ViT的视觉表征是取哪些embedding？（ByteDance-2）

## Reference










---

1. 高效参数微调PEFT——LORA. [https://zhuanlan.zhihu.com/p/665407489?utm\\_id=0](https://zhuanlan.zhihu.com/p/665407489?utm_id=0)

2. Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning <https://arxiv.org/abs/2012.13255>

3. LoRA: Low-Rank Adaptation of Large Language Models <https://arxiv.org/abs/2106.09685>

4. 使用 BPE 原理进行汉语字词切分. <https://www.less-bug.com/posts/using-bpe-principle-for-chinese-word-segmentation-plate/>

5. LLM大模型之基于SentencePiece扩充LLaMa中文词表实践. <https://zhuanlan.zhihu.com/p/655281268> 
6. pytorch bert源码. [https://github.com/hichenway/CodeShare/blob/master/bert\\_pytorch\\_source\\_code/modeling.py](https://github.com/hichenway/CodeShare/blob/master/bert_pytorch_source_code/modeling.py) 
7. LLM大语言模型之Tokenization分词方法(WordPiece, Byte-Pair Encoding (BPE), Byte-level BPE(BBPE)原理及其代码实现). <https://zhuanlan.zhihu.com/p/652520262> 
8. 剖析大模型Pretrain和SFT阶段的Loss差异. <https://zhuanlan.zhihu.com/p/652657011> 
9. 为什么Pre Norm的效果不如Post Norm? <https://kexue.fm/archives/9009> 
10. BERT用的LayerNorm可能不是你认为的那个Layer Norm? <https://cloud.tencent.com/developer/article/2159390> 
11. <https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html#torch.nn.LayerNorm> 
12. <https://stackoverflow.com/questions/70065235/understanding-torch-nn-layernorm-in-nlp> 
13. LLM (廿四) : Transformer 的结构改进与替代方案. <https://zhuanlan.zhihu.com/p/673781418> 
14. Transformer中K、Q、V的设置以及为什么不能使用同一个值. <https://www.cnblogs.com/jins-note/p/14508523.html> 