SigLIP

👫 **论文：Sigmoid Loss for Language Image Pre-Training**

论文链接：https://arxiv.org/pdf/2303.15343

可以参考的博客：https://zhuanlan.zhihu.com/p/709572492，https://blog.csdn.net/weixin_42357472/article/details/136638067，https://zhuanlan.zhihu.com/p/714731384，https://ahmdtaha.medium.com/sigmoid-loss-for-language-image-pre-training-2dd5e7d1af84

可以参考的视频：https://www.bilibili.com/video/BV1i6kBYEELj/?spm_id_from=333.337.search-card.all.click，https://www.bilibili.com/video/BV1HTzEYbEv3/?spm_id_from=333.337.search-card.all.click，https://www.bilibili.com/video/BV14NZBYTEU8/?spm_id_from=333.337.search-card.all.click

1. SigLIP 概述

1.1 SigLIP 的背景与动机

🚵 视觉-语言对比预训练（如 CLIP、ALIGN）通过对齐图像-文本嵌入成为主流，**但传统 softmax 损失存在缺陷：需全局归一化，计算复杂，内存成本高**（依赖 $|B| \times |B|$ 相似度矩阵）

SigLIP 是一种用于视觉-语言预训练的新方案，核心是在 CLIP 架构中**引入 pairwise sigmoid 损失，替代传统的 softmax contrastive loss**，以提升训练效率和性能表现。与传统 softmax 对比损失不同，它**无需全局归一化，仅基于图像-文本对操作，提升了内存效率和分布式实现的简便性**

该损失在小 `batch_size`（<16k）时表现优于 softmax，且支持更大 `batch_size`（高达 100w, 32k 已足够达到接近最优性能）。结合 Locked-image Tuning，仅用 4 个 TPUv4 芯片训练两天的 SigLiT 模型实现了 84.5% 的 ImageNet 零样本准确率。此外，多语言版本 mSigLIP 在 36 种语言的跨模态检索任务中表现优异，且模型对数据噪声的鲁棒性更强

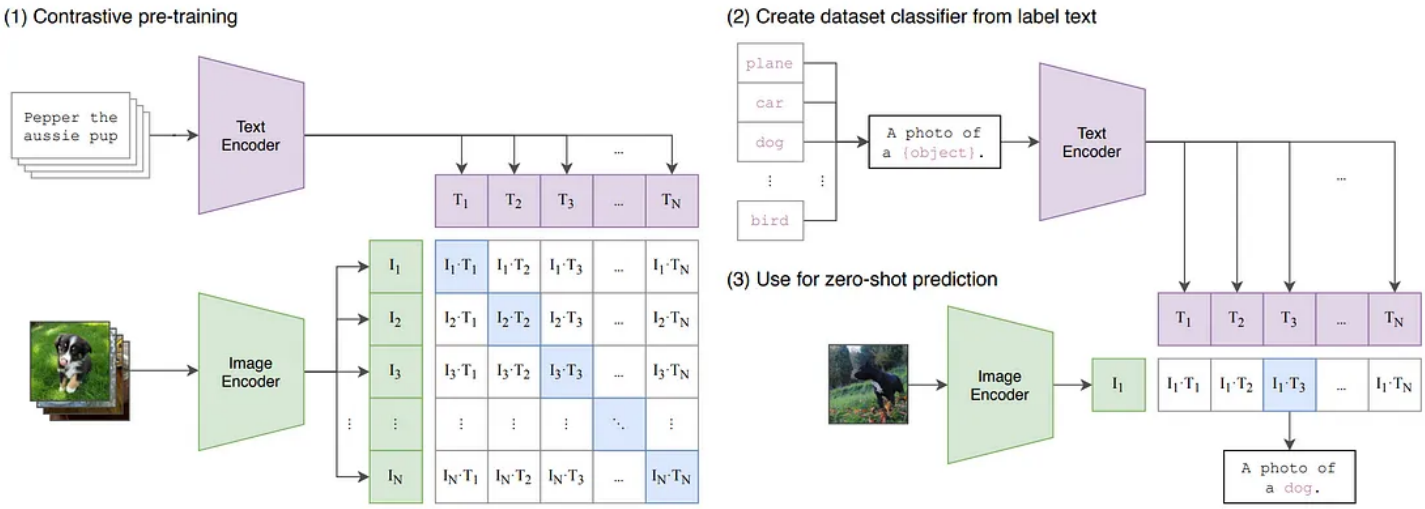
1.2 CLIP 的局限性

参考： CLIP

🤔 CLIP 使用图像-文本对通过对比损失预训练网络。这种方法具有多个优势：

1. 通过爬取互联网数据，收集图像-文本对数据集的成本相对较低
2. 支持零样本迁移到下游任务（如图像分类/检索）
3. 性能随模型和数据集的规模扩展，即更大的网络和数据集带来更好的性能

在训练过程中，CLIP 联合训练一个图像编码器和一个文本编码器，以预测一批（图像，文本）训练样本的正确配对。在测试过程中，学习到的文本编码器通过嵌入目标数据集类别的名称或描述，生成一个零样本线性分类器



🤦 然而，CLIP 存在两个技术挑战：

1. **它需要大批量训练。**例如，CLIP 使用了 32K 的批量大小，这需要大量的 GPU
2. **它需要这些 GPU 之间进行大量通信。**具体来说，图像和文本特征都需要在所有 GPU 之间进行全收集操作（all-gather）。考虑到所需的大批量，这种通信量非常大

这些主要是因为**其使用的基于 softmax 损失函数本身具有的问题**，给定正样本对（图像-文本）的损失依赖于对 batch 中的所有负样本对进行 normalize，如下图所示：

$$\mathcal{L} = -\frac{1}{2N}\sum_{i=1}^{N}\left(\overbrace{\log\frac{e^{t\mathbf{x}_i\cdot\mathbf{y}_i}}{\sum_{j=1}^{N}e^{t\mathbf{x}_i\cdot\mathbf{y}_j}}}^{\text{image}\to\text{text softmax}} + \overbrace{\log\frac{e^{t\mathbf{x}_i\cdot\mathbf{y}_i}}{\sum_{j=1}^{N}e^{t\mathbf{x}_j\cdot\mathbf{y}_i}}}^{\text{text}\to\text{image softmax}}\right)$$

Every positive pair is normalized by **all** negative pairs

CLIP 使用 softmax 操作，**每个正样本对的相似度通过所有负样本对进行归一化**。因此，**每个 GPU 需要维护一个 $N \times N$ 的矩阵来存储所有成对相似度**，这为 CLIP 带来了二次复杂度

其中，N 表示批量大小（正样本对的数量），x 表示图像特征，y 表示文本特征，t 是一个标量温度超参数，用于控制 softmax 输出的锐度/平滑度

上述公式中有两个关键细节：

1. CLIP（softmax）**损失是非对称的**，第一项为给定查询图像找到最佳匹配文本，而第二项为给定查询文本找到最佳匹配图像
2. CLIP（softmax）损失需要**全局归一化因子**（高亮的分母），这引入了二次内存复杂度，是一个 NxN 的成对相似度矩阵

总结：通过图像-文本双向 softmax 归一化，**需两次全局归一化，依赖全批量信息，内存效率低**

2. SigLIP 方法

2.1 SigLIP 的loss

SigLIP 减少了 CLIP 对大批量的需求，SigLIP 的核心思想是使用 sigmoid 操作替代 softmax 操作

与 CLIP 相比，SigLIP **既不是非对称的，也不需要全局归一化因子。因此，每个样本对（正样本或负样本）的损失独立于小批量中的其他样本对**，如下图所示：

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{N}\log\frac{1}{1+e^{z_{ij}(t\mathbf{x}_i\cdot\mathbf{y}_j+b)}} \quad \text{s.t.} \quad z_{ij} = \begin{cases} 1, & \text{for positive pairs.} \\ -1, & \text{for negative pairs.} \end{cases}$$

Every pair (positive/negative) is independent of other pairs

Sigmoid 损失：将**任务转化为二元分类，对正样本（匹配对）和负样本（非匹配对）分别计算损失**，其中 $z_{ij} = 1$（正样本，即图文匹配）或 -1（负样本，图文不匹配），引入可学习温度 t 和偏置 b 避免初始优化波动

- 每个 image-text pair 的正 / 负关系直接计算，**不局限于 batch 全局 normalization**
- **对数 sigmoid 简化运算**，显著节省内存和通信时间

Algorithm 1 Sigmoid loss pseudo-implementation.

```
# img_emb        : image model embedding [n, dim]
# txt_emb        : text model embedding [n, dim]
# t_prime, b     : learnable temperature and bias
# n              : mini-batch size

t = exp(t_prime)
zimg = l2_normalize(img_emb)
ztxt = l2_normalize(txt_emb)
logits = dot(zimg, ztxt.T) * t + b
labels = 2 * eye(n) - ones(n) # -1 with diagonal 1
l = -sum(log_sigmoid(labels * logits)) / n
```

Pairwise Sigmoid Loss 伪代码

```
logits = dot(z_img, z_txt.T)*t + b
labels = 2*I - 1  # 正负标签
loss = -sum(sigmoid(labels * logits)) / n
```

SigLIP 使用 sigmoid 操作，**每个图像-文本对（正样本或负样本）独立评估。不需要维护全局的 NxN 归一化矩阵。因此，SigLIP 的损失可以逐步计算，适合大批量训练**

值得注意的是，CLIP 和 SigLIP 都计算小批量中每个样本对（正样本/负样本）的相似度。然而，两者的内存需求存在细微差异。对于 CLIP，每个 GPU 需要维护一个 $N \times N$ 的矩阵来存储所有成对相似度，以便对正样本对进行归一化。而对于 SigLIP，由于每个正/负样本对是独立的，因此不需要维护 NxN 矩阵

另一种理解 CLIP 和 SigLIP 之间差异的方法是检查它们的问题表述:

- 给定查询图像 I ，**CLIP 解决一个多分类问题**，将图像 I 分配给其对应的正文本 T ，而忽略 batch 中的所有其他负文本
- 相反，**SigLIP 解决一个二分类问题**，为正样本对 (I, T) 分配正标签，为所有其他对分配负标签。因此，CLIP 计算全局归一化因子，而 SigLIP 不需要

2.2 SigLIP的高效实现

- 一般进行对比学习训练的时候会使用数据并行，CLIP 的基于 softmax 的损失函数需要**在所有 GPU 之间传递图像和文本特征以计算 $N \times N$ 归一化矩阵**，这**需要两次 all-gathers 操作, GPU 之间的通信需求很高**

- 相比之下，**SigLIP 只需要在所有 GPU 之间传递文本特征以计算所有成对相似度。这只需要一次 all-gathers 操作**

然而，all-gathers 操作仍然很昂贵，因为所有 GPU 在接收所有特征之前都会保持空闲状态以计算损失，因此，SigLIP 提出了一种高效的分块实现

Sigmoid 损失特别适合一种内存高效、速度快且数值稳定的实现方式，这种方式能够同时缓解上述两个问题。高效实现的目标是逐步进行损失计算和特征通信，简单来说就是将 batch 拆分到多设备，通过设备间交换负样本分块计算损失，避免全局矩阵存储：

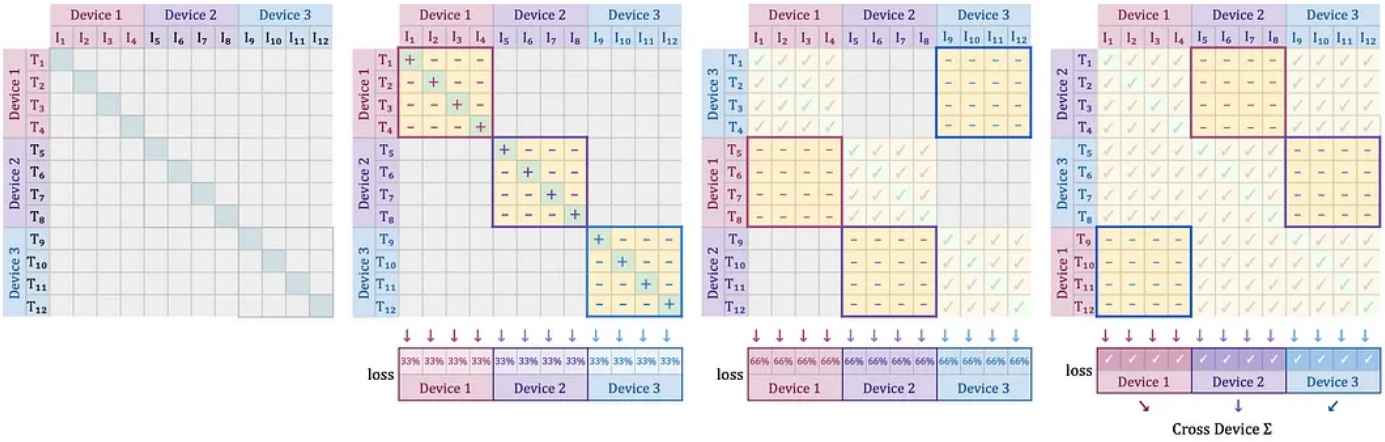
将每个设备上的批量大小记为 $b = \frac{|B|}{D}$，则该损失可重新表述为下图的公式

$$-\frac{1}{|\mathcal{B}|}\underbrace{\sum_{d_i=1}^{D}}_{\textbf{A: } \forall \text{ device } d_i}\ \underbrace{\sum_{d_j=1}^{D}}_{\textbf{B: swap negs across devices}}\ \overbrace{\underbrace{\sum_{i=bd_i}^{b(d_i+1)}}_{\text{all local positives}}\ \underbrace{\sum_{j=bd_j}^{b(d_j+1)}}_{\text{negs from next device}}\mathcal{L}_{ij}}^{\textbf{C: per device loss}}$$

🕓 方法的具体流程如下：

1. 首先，计算与正样本对以及 $b-1$ 个负样本对 对应的损失分量
2. 然后，在设备间对 embedding 进行置换，使每个设备从相邻设备获取负样本（即求和项 B 的下一轮迭代）
3. 接着，针对这部分样本块计算损失（求和项 C）。上述过程在每个设备上独立进行，因此每个设备仅需基于其本地批量 b 计算损失
4. 之后，只需在所有设备上对损失进行求和即可（求和项 A）

单个集合置换操作（用于求和项 B）速度很快（实际上，D 次集合置换通常比 D 个设备间的两次 all-gathers 操作更快），且**任一时刻的内存成本都从 $|B|^2$ 降至 b^2**（针对求和项 C）。由于原始损失计算与批量大小呈二次关系，其会迅速成为缩放的瓶颈。而这种分块方法能够**在相对较少的设备上支持超过 100 万的批量大小进行训练**



(a) Initially each device holds 4 image and 4 text representations. Each device needs to see the representations from other devices to calculate the full loss.

(b) They each compute the component of the loss (highlighted) for their representations, which includes the positives.

(c) Texts are swapped across the devices, so device 1 now has $I_{1:4}$ and $T_{5:8}$ etc. The new loss is computed and accumulated with the previous.

(d) This repeats till every image & text pair have interacted, e.g. device 1 has the loss of $I_{1:4}$ and $T_{1:12}$. A final cross-device sum brings everything together.

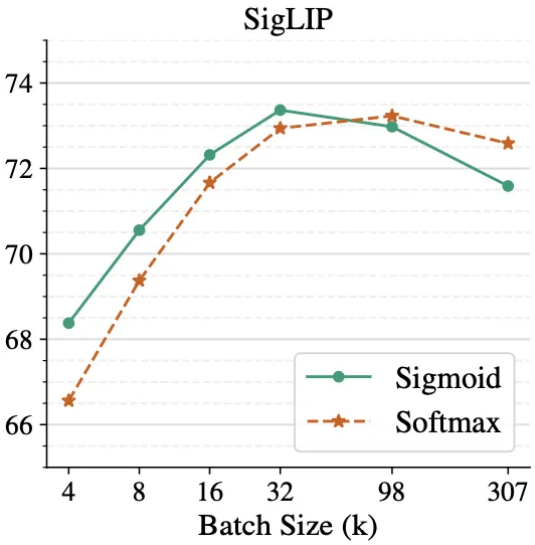
在 3 个 GPU 和全局 batch size 大小为 12 的玩具设置中（上图）演示的高效 SigLIP。没有全收集操作，任何时候只有亮黄色方块（大小为 4×4）被实例化在内存中，分块实现可以总结为每个 GPU（设备）的以下步骤：

1. 计算其本地文本和图像特征的损失
2. 从单个相邻 GPU 接收文本特征
3. 使用其本地图像特征和相邻 GPU 的文本特征计算新的损失
4. 将新计算的损失累加到总损失中
5. 重复步骤 2，直到所有相邻 GPU 传递完其文本特征

3. SigLIP 实验

在实验部分，SigLIP（sigmoid）与 CLIP（softmax）进行了对比评估。在预训练模型后，报告了 ImageNet 上的零样本性能，比较了 SigLIP 和 CLIP 在不同预训练批量大小下的 ImageNet 零样本性能。主要发现如下：

1. 在小批量（例如 4-8K）情况下，SigLIP 的性能优于 CLIP；这一点很重要，因为许多研究人员缺乏大批量训练的计算资源（GPU）
2. 尽管相关文献声称大批量可以提高性能，但本文表明，SigLIP 和 CLIP 在 32K 批量大小时达到饱和
3. 随着批量大小的增加，SigLIP 和 CLIP 之间的性能差距逐渐缩小



Method	Image Encoder		ImageNet-1k				COCO R@1	
	ViT size	# Patches	Validation	v2	ReaL	ObjectNet	I → T	T → I
CLIP	B	196	68.3	61.9	-	55.3	52.4	33.1
OpenCLIP	B	196	70.2	62.3	-	56.0	59.4	42.3
EVA-CLIP	B	196	74.7	67.0	-	62.3	58.7	42.2
SigLIP	B	196	**76.2**	**69.6**	82.8	**70.7**	**64.4**	**47.2**
SigLIP	B	256	76.7	70.0	83.1	71.3	65.1	47.4
SigLIP	B	576	78.6	72.1	84.5	73.8	67.5	49.7
SigLIP	B	1024	**79.2**	**73.0**	**84.9**	**74.7**	**67.6**	**50.4**
CLIP	L	256	75.5	69.0	-	69.9	56.3	36.5
OpenCLIP	L	256	74.0	61.1	-	66.4	62.1	46.1
CLIPA-v2	L	256	79.7	72.8	-	71.1	64.1	46.3
EVA-CLIP	L	256	79.8	72.9	-	75.3	63.7	47.5
SigLIP	L	256	**80.5**	**74.2**	**85.9**	**77.9**	**69.5**	**51.1**
CLIP	L	576	76.6	72.0	-	70.9	57.9	37.1
CLIPA-v2	L	576	80.3	73.5	-	73.1	65.5	47.2
EVA-CLIP	L	576	80.4	73.8	-	78.4	64.1	47.9
SigLIP	L	576	**82.1**	**75.9**	**87.0**	**81.0**	**70.6**	**52.7**
OpenCLIP	G (2B)	256	80.1	73.6	-	73.0	67.3	51.4
CLIPA-v2	H (630M)	576	81.8	75.6	-	77.4	67.2	49.2
EVA-CLIP	E (5B)	256	82.0	75.7	-	79.6	68.8	51.1
SigLIP	SO (400M)	729	**83.2**	**77.2**	**87.5**	**82.9**	**70.2**	**52.0**

4. SigLIP 代码

官方实现：
https://github.com/huggingface/transformers/blob/main/src/transformers/models/siglip/modeling_siglip.py

代码块

```
class SigLipLoss(nn.Module):
    def __init__(
            self,
            cache_labels: bool = False,
            rank: int = 0,
            world_size: int = 1,
            dist_impl: Optional[str] = None,
    ):
        super().__init__()
        self.cache_labels = cache_labels
        self.rank = rank
        self.world_size = world_size
        self.dist_impl = dist_impl or 'bidir'
        assert self.dist_impl in ('bidir', 'shift', 'reduce', 'gather')
        self.prev_num_logits = 0
        self.labels = {}

    def get_ground_truth(self, device, dtype, num_logits, negative_only=False)
-> torch.Tensor:
        """
        构造 ground truth 标签矩阵:
        -1 表示负样本
        如果 negative_only=False，则对角线元素设置为 +1 表示正样本
        返回 size 为 (num_logits, num_logits) 的 tensor
        """
```

```
        labels = -torch.ones((num_logits, num_logits), device=device, dtype=dtype)
        if not negative_only:
            labels = 2 * torch.eye(num_logits, device=device, dtype=dtype) + labels
        return labels

    def get_logits(self, image_features, text_features, logit_scale, logit_bias=None):
        """
        根据 image 和 text 特征计算 logits:
        logits = scale * (image @ text.T) + bias (如果有)
        """
        logits = logit_scale * image_features @ text_features.T
        if logit_bias is not None:
            logits += logit_bias
        return logits

    def _loss(self, image_features, text_features, logit_scale, logit_bias=None, negative_only=False):
        """
        内部 loss 计算流程:
        1. 计算 logits;
        2. 生成 ground truth 标签;
        3. 通过 -logsigmoid(label * logit) 计算 pairwise sigmoid loss;
        4. 对 batch 求平均返回 loss 值
        """
        logits = self.get_logits(image_features, text_features, logit_scale, logit_bias)
        labels = self.get_ground_truth(
            image_features.device,
            image_features.dtype,
            image_features.shape[0],
            negative_only=negative_only,
        )
        loss = -F.logsigmoid(labels * logits).sum() / image_features.shape[0]
        return loss

    def forward(self, image_features, text_features, logit_scale, logit_bias, output_dict=False):
        """
        前向计算 contrastive loss，支持多进程分布式计算。
        - 首先计算该进程本地的 image-text loss
        - 若 world_size>1，根据 dist_impl 不同方式和其他进程交换 text 特征，累加负样本 loss
        - 支持返回 dict 格式或 tensor
        """
```

```
        # 本地对齐 loss（包括正负对）
        loss = self._loss(image_features, text_features, logit_scale,
logit_bias)

        if self.world_size > 1:
            # 不同通信策略下，将其他进程的 text 特征与本地 image 特征计算负样本 loss
            if self.dist_impl == 'bidir':
                # 双向邻居交换
                right_rank = (self.rank + 1) % self.world_size
                left_rank = (self.rank - 1 + self.world_size) % self.world_size
                text_features_to_right = text_features_to_left = text_features
                num_bidir, remainder = divmod(self.world_size - 1, 2)
                for i in range(num_bidir):
                    # 交换两侧邻居的特征，同时保留梯度传递
                    text_features_recv = neighbour_exchange_bidir_with_grad(
                        left_rank,
                        right_rank,
                        text_features_to_left,
                        text_features_to_right,
                    )
                    # 对每一组接收的特征计算负样本 loss
                    for f in text_features_recv:
                        loss += self._loss(
                            image_features,
                            f,
                            logit_scale,
                            logit_bias,
                            negative_only=True,
                        )
                    text_features_to_left, text_features_to_right =
text_features_recv

                # 若进程数为奇数，还需再与一个方向交换一次
                if remainder:
                    text_features_recv = neighbour_exchange_with_grad(
                        left_rank,
                        right_rank,
                        text_features_to_right
                    )
                    loss += self._loss(
                        image_features,
                        text_features_recv,
                        logit_scale,
                        logit_bias,
                        negative_only=True,
                    )

```

```
            elif self.dist_impl == "shift":
                # 单方向循环交换（shift）
                right_rank = (self.rank + 1) % self.world_size
                left_rank = (self.rank - 1 + self.world_size) % self.world_size
                text_features_to_right = text_features
                for i in range(self.world_size - 1):
                    text_features_from_left = neighbour_exchange_with_grad(
                        left_rank,
                        right_rank,
                        text_features_to_right,
                    )
                    loss += self._loss(
                        image_features,
                        text_features_from_left,
                        logit_scale,
                        logit_bias,
                        negative_only=True,
                    )
                    text_features_to_right = text_features_from_left

            elif self.dist_impl == "reduce":
                # 使用 all-reduce 聚合
                for i in range(self.world_size):
                    text_from_other = torch.distributed.nn.all_reduce(
                        text_features * (self.rank == i),
                        torch.distributed.ReduceOp.SUM,
                    )
                    # 仅 add 其他 rank 的负样本 loss
                    loss += float(i != self.rank) * self._loss(
                        image_features,
                        text_from_other,
                        logit_scale,
                        logit_bias,
                        negative_only=True,
                    )

            elif self.dist_impl == "gather":
                # 收集所有进程的 text_features
                all_text = torch.distributed.nn.all_gather(text_features)
                for i in range(self.world_size):
                    loss += float(i != self.rank) * self._loss(
                        image_features,
                        all_text[i],
                        logit_scale,
                        logit_bias,
                        negative_only=True,
                    )
```

```
157                 else:
158                     assert False
159 
160             # 输出格式
161             return {"contrastive_loss": loss} if output_dict else loss
162 
```

5. SigLIP 的关键问题

1. Sigmoid 损失**相比传统 softmax 损失的核心优势是什么？**

   Sigmoid 损失无需全局批量归一化，仅基于图像 - 文本对独立计算，简化了分布式实现；内存效率更高；在小批量（<16k）时性能更优，且支持更大批量（如 100 万）；对数据噪声的鲁棒性更强

2. 语言-图像预训练中，批量大小对模型性能的具体影响是什么？

   32k 批量性能接近最优，小批量（<16k）时 Sigmoid 损失显著优于 softmax；超大规模批量（如 100 万）增益微弱，甚至因优化不稳定导致性能下降；多语言场景中，32k 批量同样是最优选择

3. SigLiT 与 SigLIP 的核心区别是什么，各自适用于什么场景？

   SigLiT 冻结预训练视觉模型，仅训练文本模型，资源需求低（如 4 TPUv4 即可），适用于有限资源场景；SigLIP 从零训练视觉和文本模型，性能更高但资源需求大（如 32 TPUv4），适用于追求高精度的场景