華中科技大學

视觉认知工程

基于 ViT-B/16 实现 CIFAR100 分类

院　　系　人工智能与自动化学院

班　　级　人工智能本硕博 2101 班

姓　　名　张伟业

学　　号　U202115203

指导教师　曹治国　肖阳　陆昊

日　　期　2024 年 6 月 27 日

目录

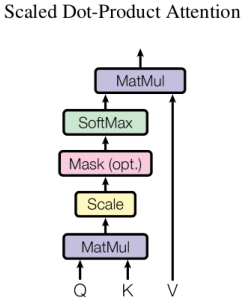
摘要

本实验以 Vision Transformer 模型为基础, 复现了 ViT-B/16 模型, 并实现了在 CIFAR100 数据集上的分类任务, 最终测试准确率达到 89.57%。报告首先介绍实验原理和模型结构。之后展示了所编写的代码结构, 并展示了部署环境, 列出了实验的核心模块。接着展示实验结果, 包含测试准确率、测试损失、训练损失曲线, 数据集展示以及注意力图的绘制。最后总结实验特点以及问题, 并撰写了实验心得。实验代码可以从https://github.com/KingDomDom/下载。

1 实验原理

1.1 Attention 机制

1.1.1 Scaled Dot-Product Attention



1. 对输入序列 X 进行线性变换，乘以不同的权重矩阵得到 Query、Key 和 Value：

$$Q = W_q X, \quad K = W_k X, \quad V = W_v X \tag{1}$$

其中，W_q, W_k, W_v 是可学习的权重矩阵，X 是输入矩阵。

2. 通过计算 Q 和 K 的点积，并除以 K 向量维度 d 的平方根，然后应用 softmax 函数，得到注意力权重：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2}$$

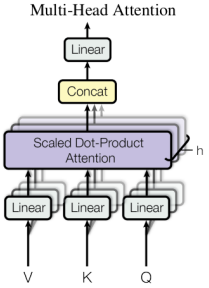
其中，d_k 是 Key 向量的维度。

3. 最终的输出是根据注意力权重对 Value 进行加权求和的结果：

$$\text{Output} = \sum_{i=1}^{n} \alpha_i V_i \tag{3}$$

其中，α_i 是通过 softmax 函数计算得到的注意力权重。

1.1.2 Multi-Head Attention



多头注意力机制（Multi-Head Attention）通过并行计算多个不同的注意力头，能够关注不同维度的不同信息。其计算公式如下：

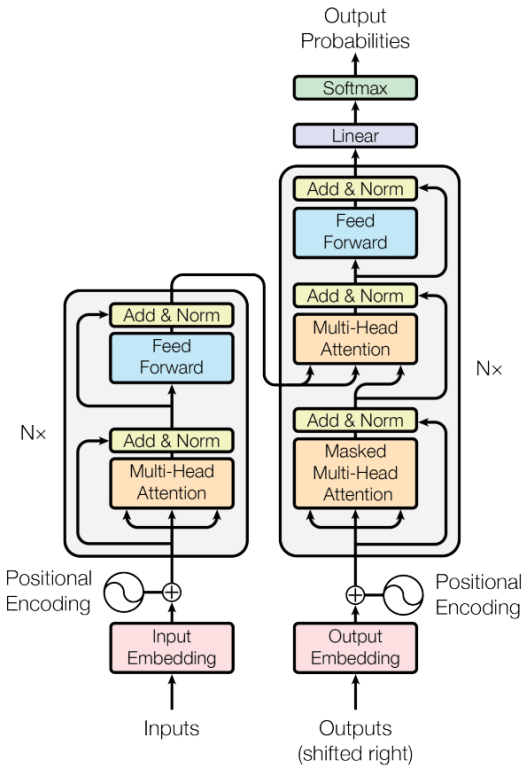
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{4}$$

其中，每个注意力头的计算方式为：

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{5}$$

这里的 W_i^Q, W_i^K, W_i^V 是不同 head 的权重矩阵。通过引入多个注意力头，使得模型在不同子空间中独立地关注不同的特征，增强了模型的表示能力。

1.2 Transformer 模型



1.2.1 Encoder

编码器由多个相同的层组成，每一层包含两个主要子层：一个多头自注意力机制和一个前馈神经网络。每个子层后都包含残差连接和层归一化。自注意力机制允许编码器在处理输入时关注不同位置的信息，从而捕捉全局依赖关系。

1.2.2 Decoder

解码器的结构与编码器类似，但在每个编码器层之后插入了一个额外的多头注意力子层，该子层在解码过程中能够访问编码器的输出。这允许解码器在生成序列时有效地使用编码后的输入信息。

1.2.3 Positional Encoding

由于不同的词语在不同的位置出现时会产生不同的语义，且每个词都有可能在任意的位置出现。Transformer 模型无法自然地捕捉输入序列中位置的信息，为了弥补这一点，使用了位置编码（Positional Encoding）。通过直接将位置信息加入到输入 embedding 中，模型能够感知输入序列中的位置信息。且在大量训练数据下，会出现无数种位置信息和词本身信息的组合，模型能学习和感知到位置信息，不会对词语原本的意思造成影响。位置编码的具体公式如下：

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

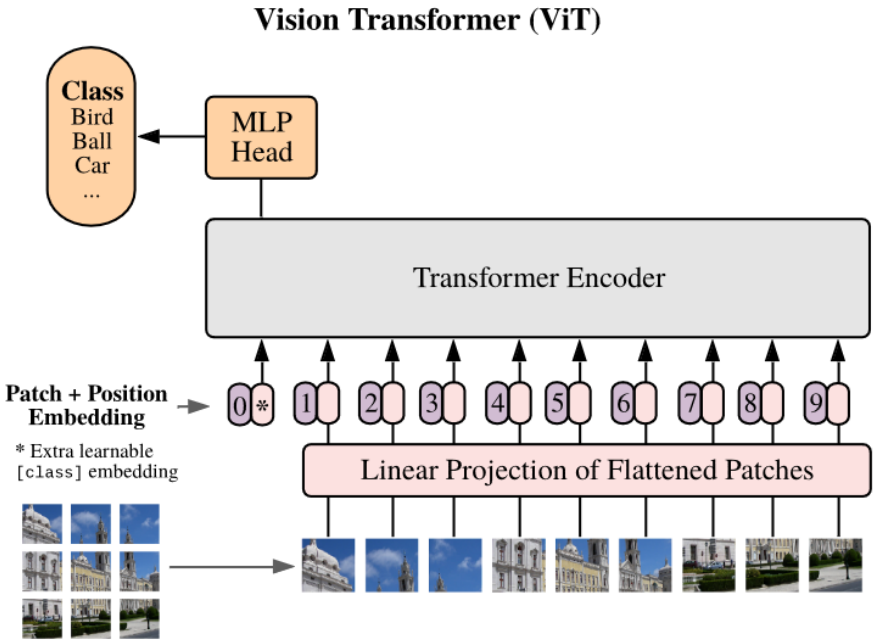
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

其中，pos 表示位置，i 表示维度索引，d 表示嵌入的维度大小。

1.2.4 Embedding

嵌入层（Embedding）将输入序列中的每个符号转换为一个固定维度的向量表示。首先，构建一个包含所有可能输入符号的词汇表。每个符号（如单词或子词）都分配一个唯一的索引。然后将输入序列中的每个符号映射到其对应的索引。接着创建一个大小为 $|V| \times d_{model}$ 的嵌入矩阵 W_{embed}，其中 $|V|$ 是词汇表的大小，d_{model} 是嵌入向量的维度。最后使用索引序列在嵌入矩阵中查找对应的嵌入向量。每个索引对应矩阵中的一行，得到的结果是一个大小为 $L \times d_{model}$ 的嵌入向量序列，其中 L 是输入序列的长度。由嵌入层生成的嵌入向量表示包含了符号的语义信息，可以作为后续的注意力机制和前馈网络的输入。

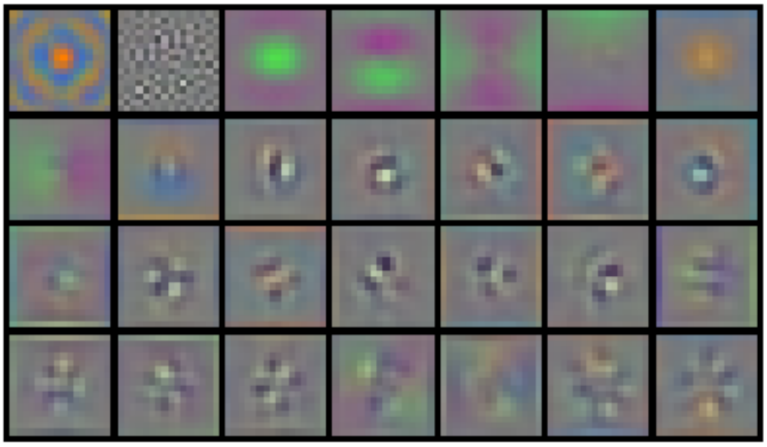
1.3 Vision Transformer 模型



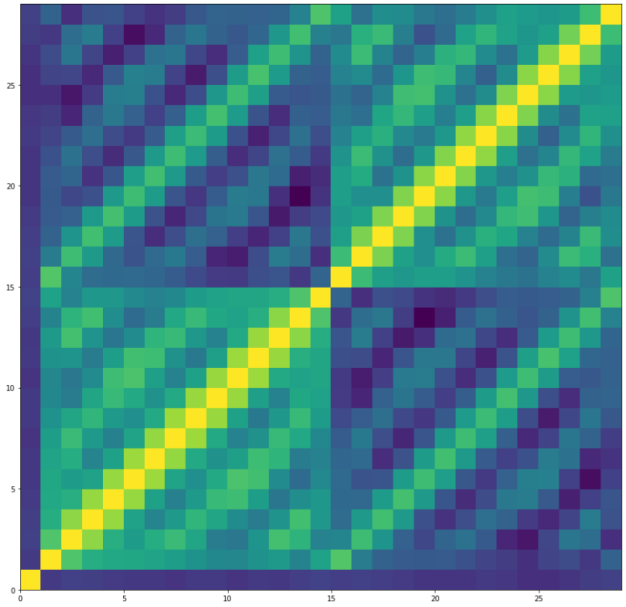
1.3.1 Model Architecture

1. 图像分块（Patch Embedding）：将输入图像划分为 16x16 的非重叠块，并展平成一个向量。
2. 线性映射 (Linear Projection)：将每个块的展平向量通过线性变换映射，生成嵌入向量。
3. 位置编码 (Position Embedding)：给每个块的嵌入向量添加位置编码，以保留块的位置信息。
4. 分类标记 (CLS Token)：引入一个额外的的分类标记，添加到输入中。
5. 编码器 (Transformer Encoder)：输入到 Transformer 编码器中，通过自注意力机制进行特征提取。
6. 感知器头 (MLP Head): 将 Encoder 处理后的数据输入到多层感知器中，输出分类结果

1.3.2 Embedding Filters

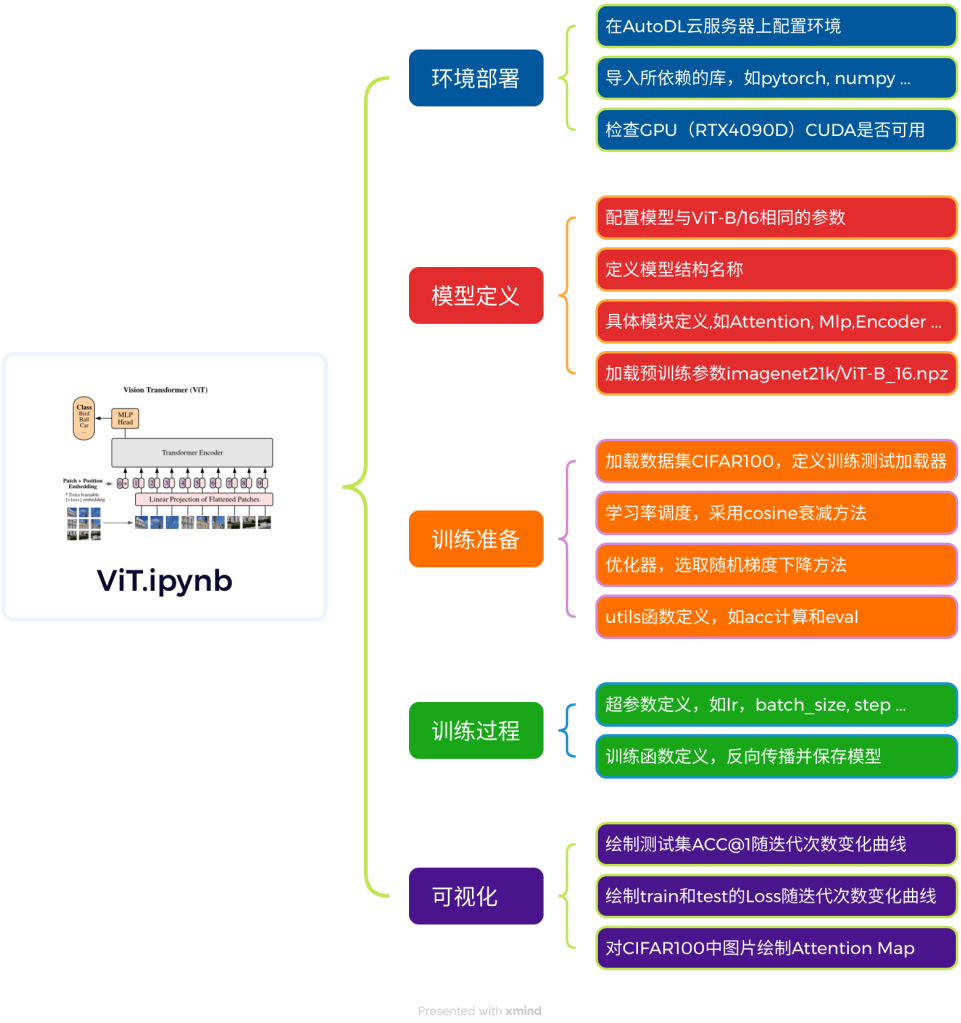


1.3.3 Cosine Position Similarity



2 实验内容

2.1 项目结构



2.2 实验环境部署

本实验在 AutoDL 平台的 RTX4090D 服务器上运行，实验工具为 Jupyter Lab

检查GPU是否可用，否则使用CPU

[4]:
```
if torch.cuda.is_available():
    device = torch.device("cuda")
    print('使用GPU进行训练，型号为: ', torch.cuda.get_device_name(0))

else:
    device = torch.device("cpu")
    print('使用CPU进行训练')
```

```
使用GPU进行训练，型号为:  NVIDIA GeForce RTX 4090 D
```

2.3 核心模块

首先编写 Vision Transformer 结构，并保持与 ViT-B/16 的模型字典一致。分别编写了 Attention 模块，Mlp 模块，Embedding 模块，Block 模块，Encoder 模块和 Vision Transformer 模型。

```
配置与 ViT-B/16 相同的参数

def get_b16_config(): ···

定义模型结构名称

logger = logging.getLogger(__name__) ···

具体模块定义

class Attention(nn.Module): ···

class Mlp(nn.Module): ···

class Embeddings(nn.Module): ···

class Block(nn.Module): ···

class Encoder(nn.Module): ···

class Transformer(nn.Module): ···
```

从 Google 的 API 下载 ViT-B/16 模型，该模型是在 ImageNet-21k 数据集上训练得到的，参数量 86M，patch size = 16，并将预训练参数加载到自己所编写的模型中作为参数初始化。

```
加载预训练参数

[13] !wget https://storage.googleapis.com/vit_models/imagenet21k/ViT-B_16.npz

输出已折叠 ···

[14] config = CONFIGS["ViT-B_16"]
     model = VisionTransformer(config, img_size=224, num_classes=21843, zero_head=False, vis=True)
     model.load_from(np.load("ViT-B_16.npz"))
     model.to(device)
```

随后为微调模型做准备，下载 CIFAR 100 数据集并分成训练集和测试集，学习率调度采用余弦衰减，优化器采用随机梯度下降，并编写了 utils 函数（如模型初始化，保存等）以及测试评估函数 eval。

```
加载数据集

def get_loader(local_rank, img_size, dataset, train_batch_size, eval_batch_size): ···

学习率调度 cosine衰减

class WarmupCosineSchedule(LambdaLR): ···

utils函数

class AverageMeter(object): ···

测试集ACC和Loss计算

def evaluation(eval_batch_size, local_rank, device, model, writer, test_loader, global_step): ···
```

训练的超参数如下：

- **train_batch_size = 64**：训练过程中每个批次的样本数目
- **eval_batch_size = 64**：评估过程中每个批次的样本数目
- **eval_every = 10**：每训练 10 步后进行一次模型评估。
- **learning_rate = 3e-2**：学习率设置为 0.03，控制权重调整的速度
- **weight_decay = 0**：权重衰减，0 表示不使用权重衰减。
- **num_steps = 1000**：总训练步数设置为 1000 步。
- **decay_type = "cosine"**：学习率衰减策略，使用余弦衰减。
- **warmup_steps = 100**：预热步数设置为 100，学习率会在这期间线性增加到初始设定值。
- **max_grad_norm = 1.0**：梯度裁剪的最大范数值，设置为 1.0，用于防止梯度爆炸。
- **seed = 42**：随机种子设置为 42，确保实验可复现性。

修改 **num class** 为 100 后开始训练，并保存最佳模型至 output_dir

开始训练

```
def get_world_size(): ···
```

```
def train(local_rank, output_dir, name, train_batch_size, eval_batch_size, seed, n_gpu, ···
```

超参数定义

```
name = "vit-cifar100" ···
```

主函数

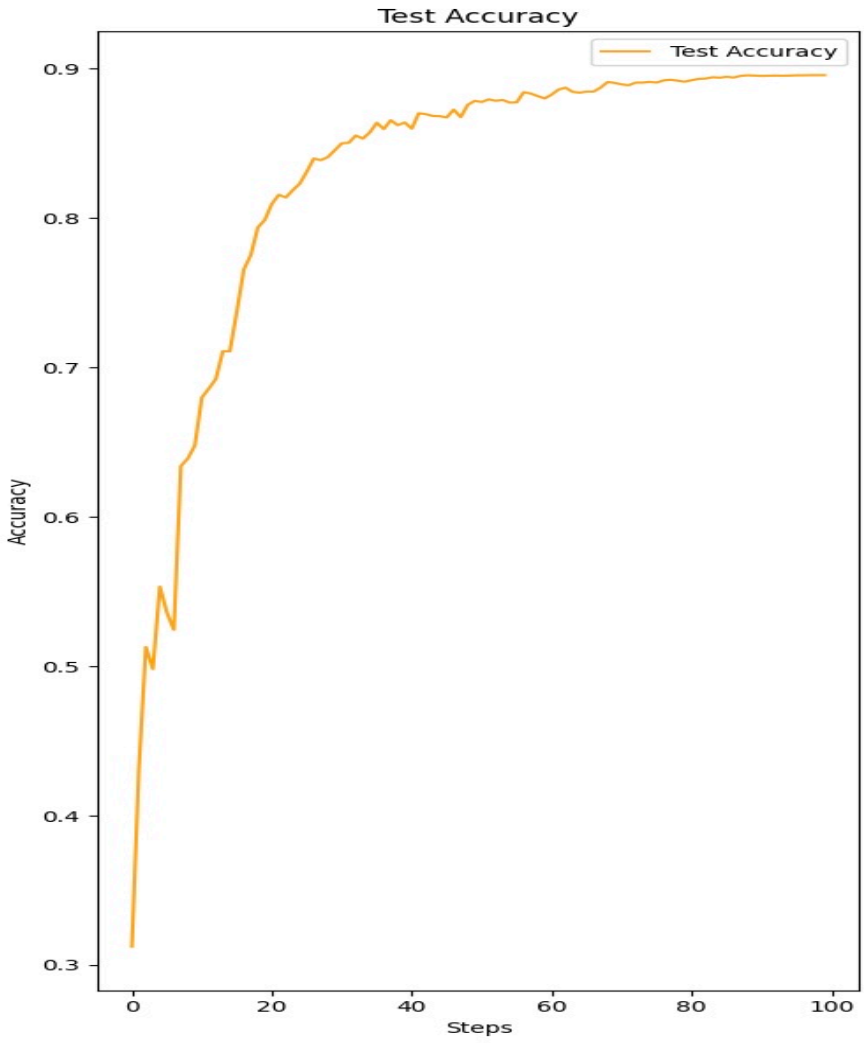
```
test_accs = [] ···
```

```
···  Files already downloaded and verified
     Files already downloaded and verified
     训练中 (10 / 1000 Steps) (loss=4.60298):   1%|| 9/782 [00:03<03:49,  3.37it/s]INFO:__main__:

     INFO:__main__:***** 开始评估 *****
     INFO:__main__:

     INFO:__main__:测试结果
     INFO:__main__:当前迭代步数: 10
     INFO:__main__:损失函数值: 4.60187
     INFO:__main__:准确率: 0.31230
     训练中 (20 / 1000 Steps) (loss=4.59234):   2%|| 19/782 [00:20<05:49,  2.18it/s]INFO:__main__:
```

3 实验结果

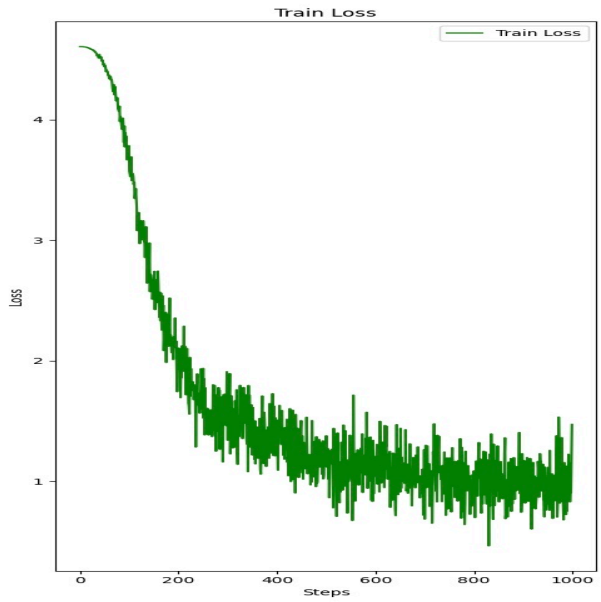
3.1 评价指标

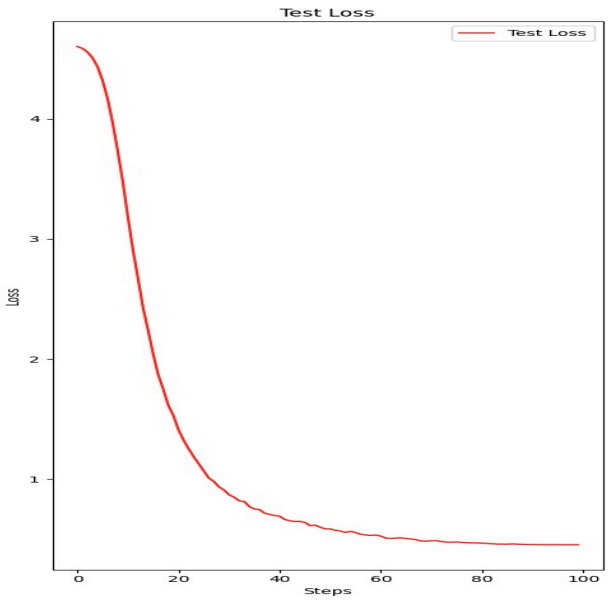
最终的测试准确率达到 89.57%




```
INFO:__main__:***** 开始评估 *****
INFO:__main__:

INFO:__main__:测试结果
INFO:__main__:当前迭代步数: 1000
INFO:__main__:损失函数值: 0.45434
INFO:__main__:准确率: 0.89570
训练中 (1000 / 1000 Steps) (loss=1.47234):  28%|| 217/782 [06:19<16:28,  1.75s/it]
INFO:__main__:最终测试准确率:      0.895700
INFO:__main__:结束训练
```

3.2 训练 & 测试损失曲线





3.3 可视化分析

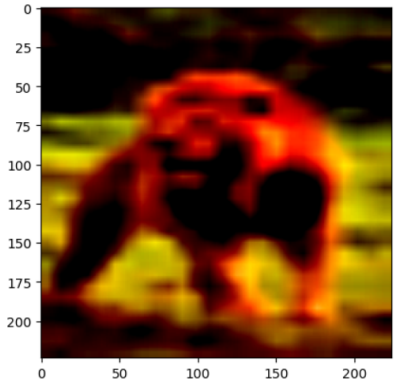
3.3.1 数据集展示

最初从数据集中提取图片后，可视化程度非常低。后续发现是因为在数据预处理步骤时，为图片添加了均值和方差的校正。通过标准化函数恢复后再展示的图片，具有很好的可视化特性。

从CIFAR100中提取图片并显示

```
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

train_loader, test_loader = get_loader(local_rank=-1, img_size=224,
                                       dataset="cifar100", train_batch_size=64,
                                       eval_batch_size=64)
dataiter = iter(train_loader)
images, labels = next(dataiter)

imshow(torchvision.utils.make_grid(images[0]))
```

```
Files already downloaded and verified
Files already downloaded and verified
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
```



标准化处理后重新展示

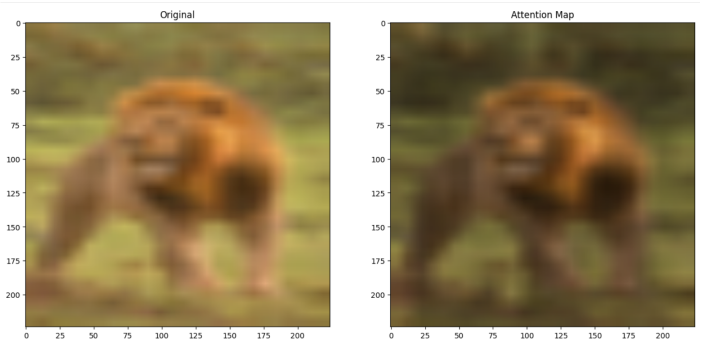
```
def std_imshow(img):
    img = img.detach().cpu().numpy()
    img = (img * np.array([0.2675, 0.2565, 0.2761])[:, None, None]
           + np.array([0.5071, 0.4867, 0.4408])[:, None, None])

    img = np.transpose(img, (1, 2, 0))  # 从CHW转换为HWC
    img = np.clip(img, 0, 1)  # 确保图像值在[0, 1]范围内
    img_pil = Image.fromarray((img * 255).astype(np.uint8))
    img_pil.save("saved_image.png")

    plt.imshow(img)
    plt.show()

std_imshow(torchvision.utils.make_grid(images[0]))
```

3.3.2 绘制注意力图

将这张图片传入 output_dir 中保存的模型，得到输出并绘制出注意力图与原图比较。发现注意力图的背景相较于原图亮度更低，且画面前景部分（狮子）的轮廓更加清晰，主体的对比度增强了。

Attention Map绘制

```
x = images[0].unsqueeze(0).to(device)
model = model.to(device)
logits, att_mat = model(x)
att_mat = torch.stack(att_mat).squeeze(1)

att_mat = torch.mean(att_mat, dim=1)

residual_att = torch.eye(att_mat.size(1)).to(device)
aug_att_mat = att_mat + residual_att
aug_att_mat = aug_att_mat / aug_att_mat.sum(dim=-1).unsqueeze(-1)

joint_attentions = torch.zeros(aug_att_mat.size()).to(device)
joint_attentions[0] = aug_att_mat[0]

for n in range(1, aug_att_mat.size(0)):
    joint_attentions[n] = torch.matmul(aug_att_mat[n], joint_attentions[n-1])

v = joint_attentions[-1]
grid_size = int(np.sqrt(aug_att_mat.size(-1)))
mask = v[0, 1:].reshape(grid_size, grid_size).detach().cpu().numpy()
im = im = Image.open("saved_image.png").convert('RGB')
mask = cv2.resize(mask / mask.max(), im.size)[..., np.newaxis]
result = (mask * im).astype("uint8")
```



4 实验总结

4.1 实验特点

本实验主要以复现 Vision Transformer 为主，将预训练的模型参数导入自己编写的 ViT-B/16 模型中，并通过一系列方法从 ImageNet-21k 迁移到 CIFAR100 数据集上。本实验全程在云平台（AutoDL）上进行，为增强对模型的理解，采用 ipynb 进行编写增强可视化，实验代码已上传到 Github。

4.2 仍然存在的问题

从论文查得，ViT 最高可在 CIFAR100 数据集上实现 94.1% 的准确率，但该微调方式所需的计算量过于大，是在 ImageNet-21k 上预训练 3000 个 epoch，且模型为 L/16，参数量 307M，远大于本实验的 86M。

再对比本实验所使用的 ImageNet-21k 预训练 30 个 epoch，模型为 B/16，论文中的 CIFAR100 最佳的正确率达到 91.6%，而本实验结果为 89.57%，十分接近。后续的改进可以由以下两个出发点：数据集增强和正则化方法，来进一步提高实验的准确率。另外，增大 batch size 也是一个可行的方法。

		Caltech101	CIFAR-100	DTD	Flowers102	Pets	Sun397	SVHN	Mean	Camelyon	EuroSAT	Resisc45	Retinopathy	Mean	Clevr-Count	Clevr-Dist	DMLab	dSpr-Loc	dSpr-Ori	KITTI-Dist	sNORB-Azim	sNORB-Elev	Mean
ImageNet-1k (300ep)	R+Ti/16	91.6	81.9	68.0	94.0	91.9	70.6	95.6	84.8	85.2	98.4	94.8	80.4	89.7	96.1	89.8	67.4	99.9	86.9	81.9	25.1	46.3	74.2
	S/32	92.7	86.4	70.7	93.6	91.2	72.9	95.8	86.2	83.6	98.6	95.5	79.6	89.3	94.2	88.4	65.8	99.9	86.1	80.7	24.9	68.2	76.0
	B/32	92.6	87.6	72.7	94.4	92.2	73.8	95.8	87.0	82.7	98.6	94.9	79.8	89.0	94.0	89.6	66.1	99.8	84.7	80.3	24.7	62.4	75.2
	Ti/16	92.7	84.0	68.9	93.8	92.5	72.0	96.1	85.7	83.7	98.7	95.6	81.6	89.9	98.0	91.9	68.5	99.7	83.2	82.0	26.5	65.9	77.0
	R26+S/32	90.2	86.2	74.0	95.5	94.3	74.5	95.6	87.2	84.5	98.6	96.0	83.4	90.6	99.7	91.6	73.3	100	84.8	84.5	28.2	51.3	76.7
	S/16	93.1	86.9	72.8	95.7	93.8	74.3	96.2	87.5	84.1	98.7	95.9	82.7	90.3	98.7	91.5	69.8	100	84.3	79.6	27.3	58.0	76.1
	R50+L/32	90.7	88.1	73.7	95.4	93.5	75.6	95.9	87.6	85.8	98.4	95.4	83.1	90.7	99.8	90.4	71.1	100	87.5	82.4	23.5	53.0	76.0
	B/16	93.0	87.8	72.4	96.0	94.5	75.3	96.1	87.9	85.1	98.9	95.7	82.5	90.5	98.1	91.8	69.5	99.9	84.5	84.0	25.9	53.9	76.0
	L/16	91.0	86.2	69.5	91.4	93.0	75.3	94.9	85.9	81.0	98.7	93.8	81.6	88.8	94.3	88.3	63.9	98.5	85.1	81.3	25.3	51.2	73.5
ImageNet-21k (30ep)	R+Ti/16	92.4	82.7	69.5	98.7	88.0	72.4	95.1	85.6	83.6	98.8	94.9	80.7	89.5	95.7	90.2	66.6	99.9	87.0	80.3	24.4	47.0	73.9
	S/32	92.7	88.5	72.4	98.9	90.5	75.4	95.4	87.7	83.5	98.7	95.0	79.5	89.2	94.5	89.8	64.4	99.8	87.9	81.2	24.9	57.7	75.0
	B/32	93.6	90.5	74.5	99.1	91.9	77.8	95.7	89.0	83.5	98.8	95.1	78.8	89.1	93.6	90.1	62.9	99.8	89.0	78.3	24.1	55.9	74.2
	Ti/16	93.3	85.5	72.6	99.0	90.0	74.3	95.1	87.1	85.5	98.8	95.5	81.6	90.4	97.7	91.7	67.4	99.9	83.8	81.2	26.3	55.1	75.4
	R26+S/32	94.7	89.9	76.5	99.5	93.0	79.1	95.9	89.8	86.3	98.6	96.1	83.1	91.0	99.7	92.0	73.4	100	88.7	84.8	26.2	53.3	77.3
	S/16	94.3	89.4	76.2	99.3	92.3	78.1	95.7	89.3	84.5	98.8	96.3	81.7	90.3	98.4	91.5	68.3	100	86.5	82.8	25.9	52.7	75.8
	R50+L/32	95.4	92.0	79.1	99.6	94.3	81.7	96.0	91.1	85.9	98.7	95.9	82.9	90.9	99.9	90.9	72.9	100	86.3	82.6	25.4	57.4	76.9
	B/16	95.1	91.6	77.9	**99.6**	94.2	80.9	96.3	90.8	84.8	99.0	96.1	82.4	90.6	98.9	90.9	72.1	100	88.3	83.5	26.6	69.6	78.7
	L/16	95.7	93.4	79.5	99.6	94.6	82.3	96.7	91.7	**88.4**	98.9	96.5	81.8	91.4	99.3	91.8	72.1	100	88.5	83.7	25.0	62.9	77.9
ImageNet-21k (300ep)	R+Ti/16	93.2	85.3	71.5	99.0	90.3	74.7	95.2	87.0	85.2	98.3	95.3	81.3	90.0	95.5	90.5	67.4	99.9	87.4	78.2	24.5	45.2	73.6
	S/32	93.2	89.7	75.3	99.2	92.0	78.1	96.1	89.1	84.0	98.5	95.4	80.6	89.6	96.9	88.7	68.1	100	91.0	79.6	26.2	55.0	75.7
	B/32	95.2	92.3	77.2	99.5	92.8	81.2	96.6	90.7	87.0	98.8	96.0	81.3	90.8	97.7	89.8	70.5	100	**92.3**	82.7	25.9	**83.1**	80.2
	Ti/16	93.7	87.2	73.1	99.2	91.0	77.3	95.7	88.2	86.0	98.5	95.8	81.9	90.6	98.3	89.7	70.8	100	86.0	82.6	26.8	49.9	75.5
	R26+S/32	94.8	90.9	78.9	99.5	94.1	81.3	96.7	90.9	87.5	98.7	96.4	**84.2**	**91.7**	99.9	**92.4**	**77.0**	**100**	87.1	83.4	**28.6**	56.0	78.1
	S/16	95.2	90.8	77.8	99.6	93.2	80.6	96.6	90.5	86.7	98.8	96.4	82.9	91.2	99.1	89.8	73.9	100	87.6	85.1	26.8	61.1	77.9
	R50+L/32	95.7	93.9	**81.6**	99.5	94.9	**83.6**	97.1	**92.3**	85.8	98.7	96.7	84.2	91.3	**100**	92.0	76.8	**100**	87.2	85.2	26.8	61.8	78.7
	B/16	**96.0**	93.2	79.1	99.6	94.7	83.0	97.0	91.8	87.4	98.7	**96.8**	83.5	91.6	99.7	89.0	76.0	100	86.7	**85.7**	28.3	68.2	79.2
	L/16	95.5	**94.1**	80.3	99.6	**95.0**	83.4	**97.4**	92.2	86.4	**99.0**	96.6	83.3	91.3	99.8	91.7	75.6	100	90.4	84.7	27.5	76.5	**80.8**

图 1: 不同 ViT 模型在不同数据集上的表现

4.3 实验心得

我曾经使用过 CNN 对 CIFAR10 数据集进行分类，当时编写一个简单的卷积网络对我来说很难，但最终还是编写成功了。我也试过 CLIP 模型对 CIFAR10 数据集进行分类，但结果十分差，不过也为我提供了许多经验。这次课程作业要求使用 ViT 对 CIFAR100 分类，一开始我认为比较简单，但真正开始做才发现比较费功夫。我的想法先是在 CIFAR100 上从头训练一个 ViT 分类器，但模型在训练集上已经发生过拟合，在测试集的准确率也只有 60% 左右。之后我又想自己租服务器，在 ImageNet 上训练，将模型参数和迭代次数增大来增加模型的泛化能力。但计算资源消耗太大，最终也以失败告终。于是我开始查阅论文，查看方法和各种实验结果，最终选择了迁移学习这条路，且没有再无脑增大模型参数和迭代次数，而是选择比较合适的小模型来进行任务。在阅读了几篇相关文献后我便确定了道路，于是开始实践。中间也遇到了许许多多的问题，但通过不断尝试都解决了。最终证明，必须先确立合适的战略，然后再实践才是正确的道路。