# 实验二：基于 ViT 的 CIFAR10 图像分类

## 一、实验目的

1. 学习如何使用深度学习框架来实现和训练一个 ViT 模型，以及 ViT 中的 Attention 机制。
2. 进一步掌握使用深度学习框架完成任务的具体流程：如读取数据、构造网络、训练模型和测试模型等。

## 二、实验要求

1. 基于 Python 语言和任意一种深度学习框架（实验指导书中使用 PyTorch 框架进行介绍），从零开始一步步完成数据读取、网络构建、模型训练和模型测试等过程，最终实现一个可以完成基于 ViT 的 CIFAR10 图像分类任务的程序。
2. 在 CIFAR10 数据集上进行训练和评估，实现测试集准确率达到 90%以上。
3. 按照规定时间在课程网站上提交实验报告，代码和 PPT。

## 三、实验原理

ViT 相关概念和原理参考《深度学习》课程讲授内容，VIT 首次将 Transformer 模型运用到计算机视觉领域并且取得了不错的分类效果，模型架构图如图 1 所示。从图 1 可以看出 VIT 只用了 Transformer 模型的编码器部分，并未涉及解码器。ViT 架构由三部分组成：（1）图像特征嵌入模块；（2）Transformer 编码器模块；（3）MLP 分类模块。 ViT 的组成模块详细介绍如下：

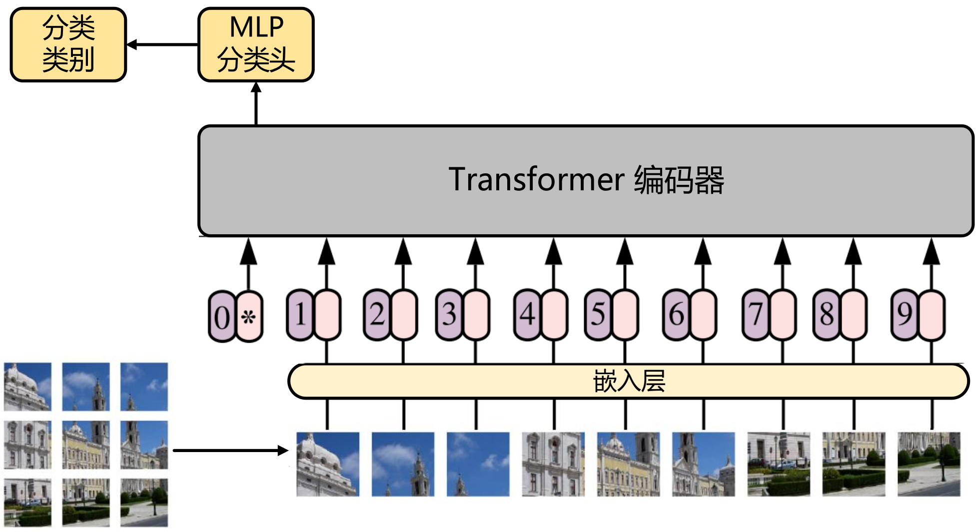


图 1 ViT 的架构

（1）图像特征嵌入模块：标准的 ViT 模型对图像的输入尺寸有要求，必须为224*224，图像输入之后首先是需要进行 Patch 分块，一般设置 Patch 的尺寸为16*16，那么一共能生成(224/16)*(224/16)=196 个 Patch 块。

（2）Transformer 编码器模块：主要由 LayerNorm 层、多头注意力机制、MLP 模块、残差连接这 5 个部分组成。其中多头注意力如图 2 所示。

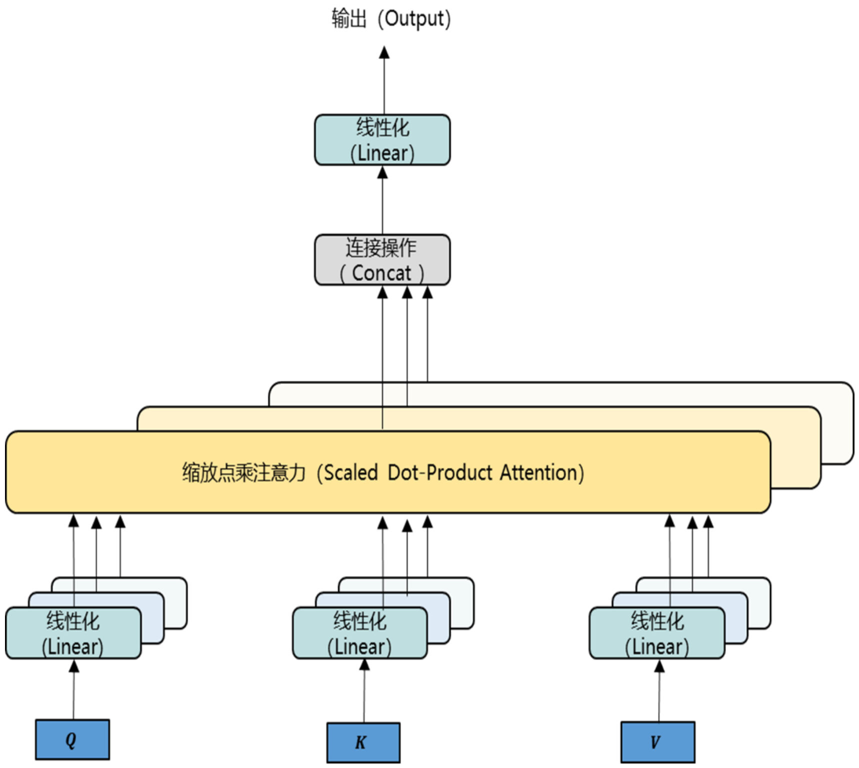


图 2 多头注意力

（3）MLP 模块：由两个全连接层加上 Dropout 层实现 。

## 四、实验所需工具和数据集

### 1. 数据集

CIFAR-10（Canadian Institute for Advanced Research-10）是一个常用的计算机视觉数据集，由 60000 张 32*32 像素的彩色图片组成，分为 10 个类别，每个类别有 6000 张图片。这个数据集包含飞机、汽车、鸟类、猫、鹿、狗、青蛙、马、船和卡车等类别。其中，训练集包含 50000 张图片，测试集包含 10000 张图片。

CIFAR-10 是一个用于测试图像分类算法性能的标准基准数据集之一，由于图像尺寸小且类别丰富，因此在计算资源有限的情况下，它通常用于快速验证和原型设计。

下载地址：[https://www.cs.toronto.edu/~kriz/cifar.html](https://www.cs.toronto.edu/~kriz/cifar.html)

2. 实验环境

- 一台电脑
- Python3.X
- PyTorch 深度学习框架

## 五、实验步骤和方法

1. 下载数据集和数据预处理

```python
# 加载和预处理数据集
trans_train = transforms.Compose(
    [transforms.RandomResizedCrop(224),  # 将给定图像随机裁剪为不同的大小和宽高比，
     # 然后缩放所裁剪得到的图像为制定的大小；
     # （即先随机采集，然后对裁剪得到的图像缩放为同一大小） 默认scale=(0.08, 1.0)
     transforms.RandomHorizontalFlip(),  # 以给定的概率随机水平旋转给定的PIL的图像，默认为0.5；
     transforms.ToTensor(),
     transforms.Normalize(mean=[0.485, 0.456, 0.406],
                          std=[0.229, 0.224, 0.225])])

trans_valid = transforms.Compose(
    [transforms.Resize(256),  # 是按照比例把图像最小的一个边长放缩到256，另一边按照相同比例放缩。
     transforms.CenterCrop(224),  # 依据给定的size从中心裁剪
     transforms.ToTensor(),  # 将PIL Image或者 ndarray 转换为tensor，并且归一化至[0-1]
     # 归一化至[0-1]是直接除以255，若自己的ndarray数据尺度有变化，则需要自行修改。
     transforms.Normalize(mean=[0.485, 0.456, 0.406],
                          std=[0.229, 0.224, 0.225])])
                          # 对数据按通道进行标准化，即先减均值，再除以标准差，注意是 hwc

trainset = torchvision.datasets.CIFAR10
(root="./cifar10", train=True, download=True, transform=trans_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=256, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./cifar10', train=False,
                                       download=False, transform=trans_valid)
testloader = torch.utils.data.DataLoader(testset, batch_size=256,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
# 随机获取部分训练数据
dataiter = iter(trainloader)
images, labels = dataiter.next()
```

2. 构建模型：包括 Attention 结构和整体结构

- Attention 结构

```python
class Attention(nn.Module):
    def __init__(self, dim, heads = 8, dim_head = 64, dropout = 0.):
        super().__init__()
        inner_dim = dim_head *  heads
        project_out = not (heads == 1 and dim_head == dim)

        self.heads = heads
        self.scale = dim_head ** -0.5

        self.norm = nn.LayerNorm(dim)

        self.attend = nn.Softmax(dim = -1)
        self.dropout = nn.Dropout(dropout)

        self.to_qkv = nn.Linear(dim, inner_dim * 3, bias = False)

        self.to_out = nn.Sequential(
            nn.Linear(inner_dim, dim),
            nn.Dropout(dropout)
        ) if project_out else nn.Identity()

    def forward(self, x):
        x = self.norm(x)

        qkv = self.to_qkv(x).chunk(3, dim = -1)
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h = self.heads), qkv)

        dots = torch.matmul(q, k.transpose(-1, -2)) * self.scale

        attn = self.attend(dots)
        attn = self.dropout(attn)

        out = torch.matmul(attn, v)
        out = rearrange(out, 'b h n d -> b n (h d)')
        return self.to_out(out)
```

- ViT 整体结构

```python
class ViT(nn.Module):
    def __init__(self, *, image_size, patch_size, num_classes, dim, depth,
    heads, mlp_dim, pool = 'cls', channels = 3, dim_head = 64, dropout = 0., emb_dropout = 0.):
        super().__init__()
        image_height, image_width = pair(image_size)
        patch_height, patch_width = pair(patch_size)

        assert image_height % patch_height == 0 and image_width % patch_width == 0,
        'Image dimensions must be divisible by the patch size.'

        num_patches = (image_height // patch_height) * (image_width // patch_width)
        patch_dim = channels * patch_height * patch_width
        assert pool in {'cls', 'mean'},
        'pool type must be either cls (cls token) or mean (mean pooling)'

        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = patch_height, p2 = patch_width),
            nn.LayerNorm(patch_dim),
            nn.Linear(patch_dim, dim),
            nn.LayerNorm(dim),
        )

        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)

        self.transformer = Transformer(dim, depth, heads, dim_head, mlp_dim, dropout)

        self.pool = pool
        self.to_latent = nn.Identity()

        self.mlp_head = nn.Linear(dim, num_classes)
```

```python
    def forward(self, img):
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape

        cls_tokens = repeat(self.cls_token, '1 1 d -> b 1 d', b = b)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n + 1)]
        x = self.dropout(x)

        x = self.transformer(x)

        x = x.mean(dim = 1) if self.pool == 'mean' else x[:, 0]

        x = self.to_latent(x)
        return self.mlp_head(x)
```

```python
class Transformer(nn.Module):
    def __init__(self, dim, depth, heads, dim_head, mlp_dim, dropout = 0.):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.layers = nn.ModuleList([])
        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                Attention(dim, heads = heads, dim_head = dim_head, dropout = dropout),
                FeedForward(dim, mlp_dim, dropout = dropout)
            ]))

    def forward(self, x):
        for attn, ff in self.layers:
            x = attn(x) + x
            x = ff(x) + x

        return self.norm(x)
```

- 前向 MLP 网络

```python
class FeedForward(nn.Module):
    def __init__(self, dim, hidden_dim, dropout = 0.):
        super().__init__()
        self.net = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, dim),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        return self.net(x)
```

3. 模型训练

```python
def train(epoch):
    print('\nEpoch: %d' % epoch)
    net.train()
    train_loss = 0
    correct = 0
    total = 0
    for batch_idx, (inputs, targets) in enumerate(trainloader):
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        sparse_selection()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()

        progress_bar(batch_idx, len(trainloader), 'Loss: %.3f | Acc: %.3f%% (%d/%d)'
            % (train_loss/(batch_idx+1), 100.*correct/total, correct, total))
    return train_loss/(batch_idx+1)
```

4. 模型验证

```python
def test(epoch):
    global best_acc
    net.eval()
    test_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(testloader):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

            progress_bar(batch_idx, len(testloader), 'Loss: %.3f | Acc: %.3f%% (%d/%d)'
                % (test_loss/(batch_idx+1), 100.*correct/total, correct, total))
```

```python
# Update scheduler
if not args.cos:
    scheduler.step(test_loss)

# Save checkpoint.
acc = 100.*correct/total
if acc > best_acc:
    print('Saving..')
    state = {
        'net': net.state_dict(),
        'acc': acc,
        'epoch': epoch,
    }
    if not os.path.isdir('checkpoint'):
        os.mkdir('checkpoint')
    torch.save(state, './checkpoint/'+args.net+'-{}-ckpt.t7'.format(args.patch))
    best_acc = acc

os.makedirs("log", exist_ok=True)
content =
time.ctime() + ' ' + f'Epoch {epoch}, lr: {optimizer.param_groups[0]["lr"]:.7f}, val loss: {test_loss:.
print(content)
with open(f'log/log_{args.net}_patch{args.patch}.txt', 'a') as appender:
    appender.write(content + "\n")
return test_loss, acc
```