1. # Vision Transformer (ViT) 从零详解与 PyTorch 复现教程

> 本文将带您从零开始，深入了解 Vision Transformer (ViT) 的工作原理，并通过 PyTorch 一步步实现完整的训练与推理过程。

1. 前言

本文是【系统学习LLM系列】中的一篇特殊内容，虽然讨论的模型与语言处理无直接关系，但具有重要的关联性。现代大语言模型已经进化为支持图文多模态输入的系统，被称为视觉语言模型(Vision Language Models, VLM)。在这些VLM中，Vision Transformer (ViT)作为处理图像信息的核心组件发挥着关键作用。因此，深入理解ViT是掌握多模态大模型工作原理的基础。

2020年，Google研究团队提出的Vision Transformer (ViT)彻底改变了计算机视觉领域的技术范式。不同于卷积神经网络(CNN)的局部感受野处理方式，ViT创新性地将图像分解为一系列图像块(patch)序列，并直接应用Transformer架构进行全局关系建模。这种方法不仅简化了视觉处理流程，还在各种图像分类基准测试中展现出卓越性能，证明了Transformer架构在视觉领域的强大潜力。

本文将从理论基础出发，一步步实现 ViT 模型，并进行训练和推理，帮助大家全面理解这一创新架构。

1.1 数据集介绍

在原始的 Vision Transformer 论文中，作者使用了多个大规模数据集进行实验：

1. **ImageNet**：包含约 130 万张训练图像，分为 1000 个类别
2. **JFT-300M**：Google 内部数据集，包含约 3 亿张图像，分为超过 18,000 个类别
3. **ImageNet-21k**：ImageNet 的扩展版本，包含约 1400 万张图像，分为 21,843 个类别

这些大规模数据集对计算资源要求极高，需要使用多台高性能 GPU 或 TPU 进行训练，这对于大多数个人研究者和学习者来说难以实现。

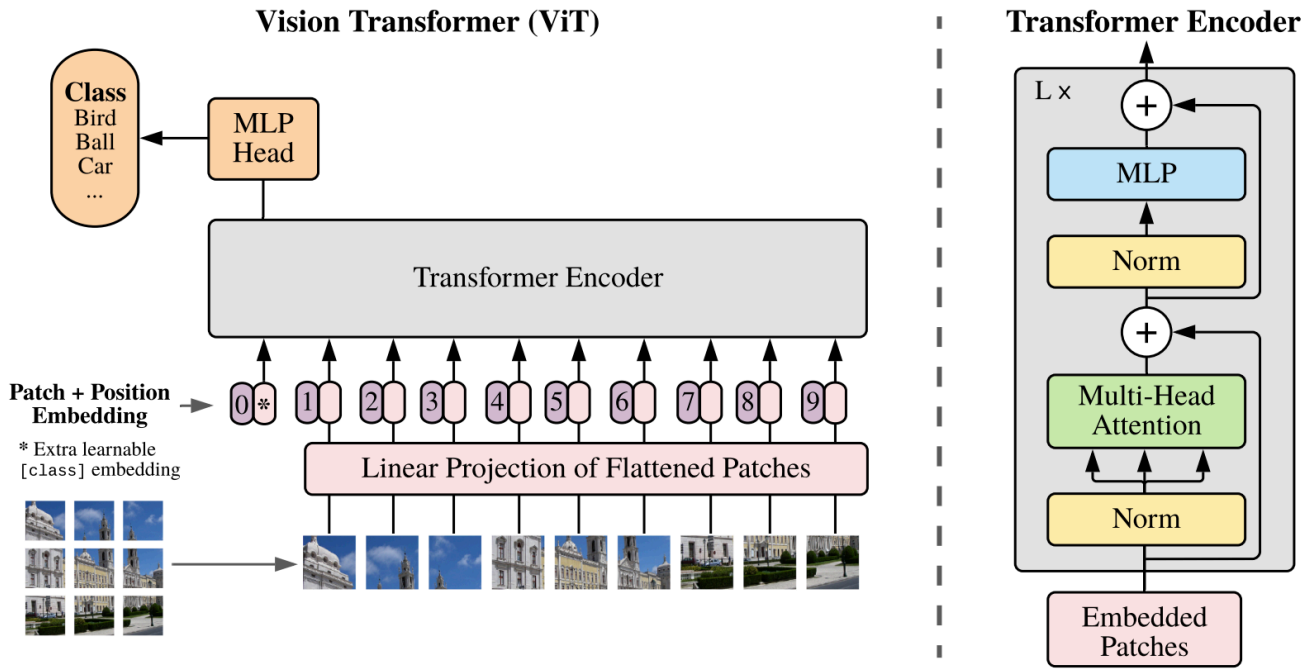
在本教程中，为了降低学习的门口，我使用 **CIFAR-10** 数据集作为演示，方便大家都能用PC上的GPU跑起来。

- 包含 60,000 张 32×32 彩色图像
- 分为 10 个类别，每类 6,000 张图像
- 训练集 50,000 张，测试集 10,000 张

2. ViT 原理简介

2.1 核心思想

ViT 的核心思想是：**将图像分割成固定大小的 patch，将每个 patch 线性投影到一个嵌入向量(embedding vector)，然后使用标准 Transformer 编码器处理这些嵌入向量序列。**



2.2 关键组件

1. **Patch Embedding**：将图像切分成 patch 并嵌入到向量空间
2. **位置编码**：为嵌入的 patch 添加位置信息
3. **Transformer 编码器**：包含多层自注意力和前馈网络
4. **分类头**：将 Transformer 的输出映射到类别空间

3. 代码实现

让我们从头开始实现 ViT 模型。首先导入必要的库：

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets
from torch.utils.data import DataLoader, random_split
from einops import rearrange, repeat
from einops.layers.torch import Rearrange
import matplotlib.pyplot as plt
from tqdm import tqdm
import numpy as np
import random
from PIL import Image
```

3.1 设置随机种子

在深度学习实验中，设置随机种子是确保结果可复现的关键步骤：

```python
def seed_everything(seed=42):
    random.seed(seed)  # Python 的 random 模块
    os.environ['PYTHONHASHSEED'] = str(seed)  # Python 哈希种子
    np.random.seed(seed)  # NumPy 随机数生成器
    torch.manual_seed(seed)  # PyTorch CPU 随机数生成器
    torch.cuda.manual_seed(seed)  # 单 GPU 的随机数生成器
    torch.cuda.manual_seed_all(seed)  # 多 GPU 的随机数生成器
    torch.backends.cudnn.deterministic = True  # 确保每次返回的卷积算法是确定的
    torch.backends.cudnn.benchmark = False  # 禁用自动寻找最快算法
```

3.2 辅助函数

```python
# 将单个数值转换为元组，例如同时支持 square=224 和 rectangle=(224,196) 这两种输入方式
def pair(t):
    return t if isinstance(t, tuple) else (t, t)
```

3.3 模型组件实现

3.3.1 前馈网络 (FFN)

Transformer 中的前馈网络模块，包含两个线性层和一个 GELU 激活函数：

```python
class FeedForward(nn.Module):
    # dim: 输入的维度
    # hidden_dim: 隐藏层的维度
    # dropout: 应用的dropout率
    def __init__(self, dim, hidden_dim, dropout=0.):
        super().__init__()
        # ViT用pre-norm结构提升稳定性，Transformer用GELU激活函数而非ReLU
        self.net = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, hidden_dim), # 扩展维度
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, dim),  # 恢复原始维度
            nn.Dropout(dropout)  # 用两次dropout增强泛化能力
        )
    # 输入/输出的 shape均为: [batch, seq_len, dim]
    def forward(self, x):
        return self.net(x)
```

3.3.2 多头自注意力机制

Transformer 的核心组件，实现了 Query、Key、Value 的计算与多头注意力：

```python
class Attention(nn.Module):
    # dim: 输入的维度
    # heads: 多头注意力的头数
    # dim_head: 每个头的维度
```

```
    # dropout: 应用的dropout率
    # 输入 x 的 shape: [batch, seq_len, dim]
    def __init__(self, dim, heads=8, dim_head=64, dropout=0.):
        super().__init__()
        # 总注意力维度 = 头数 * 每个头的维度
        inner_dim = dim_head * heads
        # 只有当单头且dim_head=dim时才不需要最后的投影层
        project_out = not (heads == 1 and dim_head == dim)

        self.heads = heads  # 多头注意力中的头数
        self.scale = dim_head ** -0.5  # 缩放因子，sqrt(dim_head)，防止softmax梯度消失

        self.norm = nn.LayerNorm(dim)  # 同样采用Pre-norm结构
        self.attend = nn.Softmax(dim=-1)
        self.dropout = nn.Dropout(dropout)  # 应用于注意力权重

        # QKV投影，合并为一个线性层以提高效率
        self.to_qkv = nn.Linear(dim, inner_dim * 3, bias=False) # shape: [batch,
seq_len, dim] -> [batch, seq_len, inner_dim * 3]

        # 输出投影，如果需要的话
        self.to_out = nn.Sequential(
            nn.Linear(inner_dim, dim), # shape: [batch, seq_len, inner_dim] ->
[batch, seq_len, dim]
            nn.Dropout(dropout)
        ) if project_out else nn.Identity() # Identity() 表示恒等变换，不改变输入/输出的
shape

    # 输入 x 的 shape: [batch, seq_len, dim]
    # 输出 x 的 shape: [batch, seq_len, dim]
    def forward(self, x):
        x = self.norm(x)  # 先LayerNorm，Pre-norm结构

        # 一次性计算QKV并分块
        qkv = self.to_qkv(x).chunk(3, dim=-1) # shape: [batch, seq_len, dim] ->
[batch, seq_len, dim_head*heads*3] -> 3个[batch, seq_len, dim_head*heads]
        # 使用einops进行形状变换，将每个QKV分离为多头形式
        # 从 [batch, seq_len, heads*dim_head] 变为 [batch, heads, seq_len, dim_head]
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h=self.heads),
qkv)

        # 注意力计算：Q和K的矩阵乘法，然后应用缩放因子，QK^T/sqrt(dim_head)
        # shape: [batch, heads, seq_len, dim_head]大家[batch, heads, dim_head,
seq_len] -> [batch, heads, seq_len, seq_len]
        dots = torch.matmul(q, k.transpose(-1, -2)) * self.scale

        # 应用softmax得到注意力权重
        attn = self.attend(dots) # shape: [batch, heads, seq_len, seq_len] -> [batch,
heads, seq_len, seq_len]
        attn = self.dropout(attn)  # 对注意力权重应用dropout

        # 注意力权重与v相乘得到输出
```

```
        # shape: [batch, heads, seq_len, seq_len]大家[batch, heads, seq_len, dim_head]
-> [batch, heads, seq_len, dim_head]
        out = torch.matmul(attn, v)
        # 重新排列多头结果，恢复原始形状
        # shape: [batch, heads, seq_len, dim_head] -> [batch, seq_len, heads大家
dim_head]
        out = rearrange(out, 'b h n d -> b n (h d)')
        # 最后通过输出投影层
        # shape: [batch, seq_len, heads大家dim_head] -> [batch, seq_len, dim]
        return self.to_out(out)
```

关于这里的 `bias=False`：在多头注意力机制的 QKV 投影中，通常不使用偏置项，原因是：

1. 注意力层前已有 LayerNorm，减弱了偏置项的影响
2. 注意力机制关注的是位置间的相对关系，而非绝对值
3. 实践证明去掉偏置项不会降低性能，反而减少了参数量

3.3.3 Transformer 编码器

由多个自注意力层和前馈网络组成，包含残差连接：

```
class Transformer(nn.Module):
    # dim: 输入的维度
    # depth: Transformer的层数
    # heads: 多头注意力的头数
    # dim_head: 每个头的维度
    # mlp_dim: 前馈网络的维度
    # dropout: 应用的dropout率
    def __init__(self, dim, depth, heads, dim_head, mlp_dim, dropout=0.):
        super().__init__()
        self.norm = nn.LayerNorm(dim)  # 最终的LayerNorm
        self.layers = nn.ModuleList([])  # 存储多个Transformer层

        # 创建多个Transformer层
        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                Attention(dim, heads=heads, dim_head=dim_head, dropout=dropout),  # 多
头注意力
                FeedForward(dim, mlp_dim, dropout=dropout)  # 前馈网络
            ]))

    # 输入 x 的 shape: [batch, seq_len, dim]
    # 输出 x 的 shape: [batch, seq_len, dim]
    def forward(self, x):
        # 按顺序应用每个Transformer层
        for attn, ff in self.layers:
            # 注意这里使用了残差连接，是Transformer的关键设计
            x = attn(x) + x  # 注意力层 + 残差连接
            x = ff(x) + x  # 前馈层 + 残差连接

        # 最后应用LayerNorm
```

```
        return self.norm(x)
```

3.3.4 Vision Transformer 主模型

完整的 ViT 模型实现，包含 patch 嵌入、位置编码、Transformer 编码器和分类头：

```
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.):
        """
        image_size: 输入图像的大小
        patch_size: 每个图像块的大小
        num_classes: 分类类别数
        dim: 模型的隐层维度
        depth: Transformer的层数
        heads: 多头注意力的头数
        mlp_dim: 前馈网络的隐层维度
        pool: 池化方式, 'cls'或'mean'
        channels: 输入图像的通道数
        dim_head: 每个注意力头的维度
        dropout: 模型内部使用的dropout率
        emb_dropout: 应用于嵌入层的dropout率
        """
        super().__init__()

        # 处理图像和块的尺寸
        image_height, image_width = pair(image_size)
        patch_height, patch_width = pair(patch_size)

        # 确保图像尺寸能被块尺寸整除
        assert image_height % patch_height == 0 and image_width % patch_width == 0, \
            'Image dimensions must be divisible by the patch size.'

        # 计算块的数量和每个块的维度
        num_patches = (image_height // patch_height) * (image_width // patch_width)
        patch_dim = channels * patch_height * patch_width

        # 确保池化方式有效
        assert pool in {'cls', 'mean'}, 'pool type must be either cls (cls token) or
mean (mean pooling)'

        # Patch Embedding
        # ViT的关键部分，将2D图像转换为1D序列，相当于对每个patch进行线性投影
        # 输入 x 的 shape: [batch, channel, height, width]
        # 输出 x 的 shape: [batch, patches, dim]
        self.to_patch_embedding = nn.Sequential(
            # 将图像重新排列成块序列
            # 从 [batch, channel, height, width] 变为 [batch, patches, patch_dim]
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=patch_height,
p2=patch_width),
```

```
            nn.LayerNorm(patch_dim),  # 对每个块进行LayerNorm
            nn.Linear(patch_dim, dim),  # 线性投影到模型维度, shape: [batch, patches, patch_dim] -> [batch, patches, dim]
            nn.LayerNorm(dim),  # 再次LayerNorm
        )

        # 位置编码，用于提供序列位置信息
        # +1是为了额外的分类标记
        # nn.Parameter 表示这是一个可训练的参数
        # torch.randn(1, num_patches + 1, dim) 表示生成一个形状为 (1, num_patches + 1, dim) 的张量，其中 num_patches + 1 是序列的长度，dim 是每个位置的维度
        # 之所以 num_patches + 1，是因为需要一个额外的分类标记，用于表示整个序列的特征
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))

        # 分类标记，类似于BERT的[CLS]标记
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))

        # 嵌入层dropout
        self.dropout = nn.Dropout(emb_dropout)

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

        # 池化方式：cls或mean
        self.pool = pool

        # 输出前的恒等变换，可用于额外处理
        self.to_latent = nn.Identity()

        # 最终的分类头
        self.mlp_head = nn.Linear(dim, num_classes)

    def forward(self, img):
        # 将图像变为块嵌入
        # 输入 x 的 shape: [batch, channel, height, width]
        # 输出 x 的 shape: [batch, patches, dim]
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape  # 批次大小，块数量，特征维度

        # 扩展cls标记到批次大小
        # shape: [1, 1, dim] -> [batch, 1, dim]
        cls_tokens = repeat(self.cls_token, '1 1 d -> b 1 d', b=b)

        # 将cls标记添加到序列开头
        # shape: [batch, patches, dim] -> [batch, patches + 1, dim]
        x = torch.cat((cls_tokens, x), dim=1)

        # 加入位置编码
        # 这里使用切片是为了处理可能的位置编码和输入序列长度不匹配的情况
        # shape: [batch, patches + 1, dim] -> [batch, patches + 1, dim]
        x += self.pos_embedding[:, :(n + 1)]
```

```
        # 应用dropout
        # shape: [batch, patches + 1, dim] -> [batch, patches + 1, dim]
        x = self.dropout(x)

        # 通过Transformer编码器
        # shape: [batch, patches + 1, dim] -> [batch, patches + 1, dim]
        x = self.transformer(x)

        # 池化：平均池化或使用cls标记
        # shape: [batch, patches + 1, dim] -> [batch, dim]
        x = x.mean(dim=1) if self.pool == 'mean' else x[:, 0]

        # 通过潜在空间投影
        # shape: [batch, dim] -> [batch, dim]
        x = self.to_latent(x)

        # 通过分类头得到最终输出
        # shape: [batch, dim] -> [batch, num_classes]
        return self.mlp_head(x)
```

关于 `pool='cls'`：当设置为 'cls' 时，模型将使用添加的分类标记 (CLS token) 的特征作为整个图像的表示。这个标记通过自注意力机制与所有图像块交互，能够聚合整个图像的信息，是从 BERT 等 NLP 模型借鉴的设计。

关于 `self.to_latent = nn.Identity()`：这个恒等变换看似多余，但它提供了一个接口占位符，未来可以在这里插入额外的处理层，增强模型的灵活性和扩展性。

3.4 训练与评估功能

3.4.1 训练

```
def train_epoch(model, dataloader, criterion, optimizer, device):
    model.train()  # 设置为训练模式，启用dropout等
    running_loss = 0.0
    correct = 0
    total = 0

    # 使用tqdm创建进度条
    pbar = tqdm(dataloader)
    for inputs, targets in pbar:
        # 将数据移动到指定设备(CPU/GPU)
        inputs, targets = inputs.to(device), targets.to(device)

        # 梯度清零，防止梯度累积
        optimizer.zero_grad()

        # 前向传播
        outputs = model(inputs)

        # 计算损失
        loss = criterion(outputs, targets)
```

```
        # 反向传播
        loss.backward()

        # 参数更新
        optimizer.step()

        # 累积统计信息
        running_loss += loss.item()
        _, predicted = outputs.max(1)  # 获取最高概率的类别
        total += targets.size(0)  # 累计样本数
        correct += predicted.eq(targets).sum().item()  # 累计正确预测数

        # 更新进度条显示
        pbar.set_description(f'Loss: {running_loss/(pbar.n+1):.4f} | Acc:
{100.*correct/total:.2f}%')

    # 返回整个epoch的平均损失和准确率
    return running_loss/len(dataloader), 100.*correct/total
```

3.4.2 验证函数

```
def validate(model, dataloader, criterion, device):
    model.eval()  # 设置为评估模式，禁用dropout等
    running_loss = 0.0
    correct = 0
    total = 0

    # 不需要计算梯度，节省内存并加速计算
    with torch.no_grad():
        pbar = tqdm(dataloader)
        for inputs, targets in pbar:
            inputs, targets = inputs.to(device), targets.to(device)

            # 前向传播
            outputs = model(inputs)
            loss = criterion(outputs, targets)

            # 累积统计信息
            running_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

            # 更新进度条
            pbar.set_description(f'Loss: {running_loss/(pbar.n+1):.4f} | Acc:
{100.*correct/total:.2f}%')

    # 返回整个验证集的平均损失和准确率
    return running_loss/len(dataloader), 100.*correct/total
```

3.4.3 推理函数

```
def predict(model, image_path, transform, device, class_names=None):
    model.eval()  # 评估模式

    # 加载并转换图像
    # 转换为RGB确保与模型训练时的输入一致
    image = Image.open(image_path).convert('RGB')

    # 应用图像变换并添加批次维度
    image_tensor = transform(image).unsqueeze(0).to(device)

    # 进行推理
    with torch.no_grad():
        outputs = model(image_tensor)
        _, predicted = outputs.max(1)

    # 获取预测类别索引
    predicted_class = predicted.item()

    # 如果提供了类别名称，返回对应名称而非索引
    if class_names is not None:
        return class_names[predicted_class]

    return predicted_class
```

3.5 主函数：完整训练流程

```
def main():
    # 设置随机种子确保可复现性
    seed_everything(42)

    # 配置训练参数
    batch_size = 64  # 批次大小，较大值可加速训练但需要更多内存
    num_epochs = 10  # 训练轮数
    learning_rate = 1e-4  # 学习率，Transformer通常使用较小的学习率
    image_size = 224  # 输入图像尺寸，标准的ImageNet预处理尺寸
    patch_size = 16  # 图像块大小，ViT标准设置
    num_classes = 10  # CIFAR-10有10个类别

    # 设置计算设备，优先使用GPU
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f'使用设备：{device}')

    # 数据增强和LayerNorm
    # 数据增强对防止过拟合非常重要，特别是对小数据集
    transform_train = transforms.Compose([
        transforms.RandomResizedCrop(image_size),  # 随机裁剪并缩放到指定大小
        transforms.RandomHorizontalFlip(),  # 随机水平翻转增加多样性
        transforms.ToTensor(),  # 转换为张量，同时将像素值缩放到[0,1]
```

```
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # LayerNorm到[-1,1]区
间
    ])

    # 测试集不需要数据增强，但需要相同的LayerNorm
    transform_test = transforms.Compose([
        transforms.Resize(image_size),  # 缩放到指定大小
        transforms.CenterCrop(image_size),  # 中心裁剪确保一致性
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    # 加载CIFAR-10数据集
    dataset = datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform_train)

    # 分割训练集和验证集
    train_size = int(0.8 * len(dataset))  # 80%用于训练
    val_size = len(dataset) - train_size  # 20%用于验证
    train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

    # 加载测试集
    test_dataset = datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform_test)

    # 创建数据加载器
    # num_workers=2使用多进程加速数据加载
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
num_workers=2)
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,
num_workers=2)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,
num_workers=2)

    # CIFAR-10的类别名称
    class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck']

    # 创建ViT模型实例
    model = ViT(
        image_size=image_size,
        patch_size=patch_size,
        num_classes=num_classes,
        dim=512,  # 模型维度
        depth=6,  # Transformer层数
        heads=8,  # 注意力头数
        mlp_dim=1024,  # 前馈网络维度，通常是dim的2-4倍
        dropout=0.1,  # 模型内部dropout
        emb_dropout=0.1  # 嵌入层dropout
    ).to(device)  # 移动到指定设备

    # 设置损失函数，分类任务标准选择
```

```
    criterion = nn.CrossEntropyLoss()

    # 设置优化器，Adam对Transformer通常效果较好
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # 学习率调度器，使用余弦退火策略
    # 这可以使学习率从初始值逐渐下降，有助于模型收敛
    scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)

    # 初始化训练记录容器
    train_losses = []
    train_accs = []
    val_losses = []
    val_accs = []

    # 开始训练循环
    print('开始训练')
    for epoch in range(num_epochs):
        print(f'\nEpoch {epoch+1}/{num_epochs}')

        # 训练阶段
        train_loss, train_acc = train_epoch(model, train_loader, criterion,
optimizer, device)
        train_losses.append(train_loss)
        train_accs.append(train_acc)

        # 验证阶段
        val_loss, val_acc = validate(model, val_loader, criterion, device)
        val_losses.append(val_loss)
        val_accs.append(val_acc)

        # 更新学习率
        scheduler.step()

        # 打印当前epoch的结果
        print(f'Train Loss: {train_loss:.4f} | Train Acc: {train_acc:.2f}%')
        print(f'Val Loss: {val_loss:.4f} | Val Acc: {val_acc:.2f}%')

    # 绘制训练历史图表
    # plot_training_history(train_losses, train_accs, val_losses, val_accs)

    # 在测试集上进行最终评估
    test_loss, test_acc = validate(model, test_loader, criterion, device)
    print(f'\nTest Loss: {test_loss:.4f} | Test Acc: {test_acc:.2f}%')

    # 保存训练好的模型以便将来使用
    torch.save(model.state_dict(), 'vit_model.pth')
    print('模型已保存到 vit_model.pth')

    # 如果有示例图像，进行测试推理
    print('\n示例推理: ')
    sample_image_path = 'sample_image.jpg'
```

```
    if os.path.exists(sample_image_path):
        predicted_class = predict(model, sample_image_path, transform_test, device,
class_names)
        print(f'预测类别: {predicted_class}')
```

3.6 模型推理功能

```
def load_and_predict(image_path, model_path='vit_model.pth', num_classes=10,
image_size=224, patch_size=16):
    # 设置计算设备
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # 重新创建模型架构，必须与保存时一致
    model = ViT(
        image_size=image_size,
        patch_size=patch_size,
        num_classes=num_classes,
        dim=512,
        depth=6,
        heads=8,
        mlp_dim=1024
    ).to(device)

    # 加载保存的模型权重
    # map_location参数确保模型能在当前设备上加载
    model.load_state_dict(torch.load(model_path, map_location=device))

    # 定义与训练时相同的图像变换
    transform = transforms.Compose([
        transforms.Resize(image_size),
        transforms.CenterCrop(image_size),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    # CIFAR-10的类别名称
    class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck']

    # 进行预测
    predicted_class = predict(model, image_path, transform, device, class_names)
    return predicted_class
```

3.7 程序入口点

```
if __name__ == '__main__':
    main()
```

4. 模型训练与测试

使用上述代码，我们可以在 CIFAR-10 数据集上训练 ViT 模型。训练过程会显示每个 epoch 的损失和准确率，并在验证集上评估模型性能。

在一台单卡 A10 的机器上，训练 10 个 epoch 大约需要 15 分钟，最多占用4GB显存。

5. ViT 的优缺点分析

5.1 优点

1. **全局感受野**：Transformer 的自注意力机制能够捕捉图像的全局依赖关系，而 CNN 需要多层堆叠才能获得大的感受野。
2. **可扩展性**：ViT 模型易于扩展到更大规模，性能随着模型和数据规模的增加而持续提升。
3. **与 NLP 的统一**：使用同一种架构处理视觉和语言任务，为多模态学习奠定基础。

5.2 缺点

1. **数据饥饿**：原始 ViT 需要大量数据才能达到良好的性能，在小数据集上容易过拟合。
2. **计算成本**：自注意力机制的计算复杂度是序列长度的平方，对于高分辨率图像计算成本高。
3. **缺少归纳偏置**：ViT 缺少 CNN 内置的平移不变性等归纳偏置，需要从数据中学习这些性质。

6. ViT 的改进与变种

基于基础 ViT，研究人员提出了许多改进版本，如：

1. **DeiT**：通过知识蒸馏减少对大量数据的依赖
2. **Swin Transformer**：使用移动窗口的层次化设计，降低计算复杂度
3. **MobileViT**：为移动设备优化的轻量级设计
4. **MAE**：通过掩码自编码预训练提升性能

7. 总结与展望

Vision Transformer 通过将 Transformer 架构引入计算机视觉领域，展示了一种全新的处理视觉数据的范式。虽然传统 CNN 仍有其优势，但 ViT 及其衍生模型的成功表明，自注意力机制在视觉任务中具有巨大潜力。

随着硬件性能的提升和算法的优化，我们可以期待 Transformer 在计算机视觉领域发挥更大的作用，特别是在多模态学习和大规模预训练模型方面。

本文详细讲解了 ViT 的原理和实现，希望能帮助大家更好地理解这一创新架构，并在自己的项目中应用和改进它。

参考资料：

1. Dosovitskiy, A., et al. (2020). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.
2. https://github.com/lucidrains/vit-pytorch