山东工商学院
Shandong Technology and Business University

神经网络与深度学习综合实训

项目报告

小组编号：17 组

小组成员：刘家呈 任昱萱 孙士烜

指导教师：邓富文

日　　期：2024.7.9

一、项目介绍

本项目是一个手势识别深度学习应用项目，旨在通过训练一个可以识别手势数字（0~9）的模型，来实现智能高效的人机界面交互。手势识别技术在现代的人机交互中扮演着重要的角色，能够实现更直观、便捷的操作方式，广泛应用于手语识别、智能监控、虚拟现实等领域。

这个项目将基于手势图片数据集，以数字 0~9 为分类标签，设计并训练一个深度学习模型。通过使用 Git 进行代码管理，搭建深度神经网络模型，以及利用 VIT（Vision Transformer）和 FPN（Feature Pyramid Network）等技术，实现对手势数字的准确分类识别。

在项目的实际实现中，我们将首先进行数据集的准备和预处理工作，包括图片的读取、标签的处理等。然后，我们将搭建深度神经网络模型，利用 VIT 作为图像特征提取器，将输入的手势图片数据进行特征提取和表示学习。同时，结合 FPN 特征金字塔网络，实现多尺度的特征融合和分类预测，提高模型对手势数字的识别准确性。

在模型训练过程中，我们将使用深度学习框架进行模型的编译、训练和评估，通过调参优化模型的结构和参数，提高模型在手势数字分类任务上的性能表现。最终，在测试集上验证模型的泛化能力和准确度，评估模型在实际应用中的效果，并对模型进行部署和应用。

通过完成这个实训项目，学员将能够掌握深度学习应用的整个流程，包括数据处理、模型设计、训练调参和性能评估等环节。

该项目相关代码已上传 GitHub，由于模型较大，只上传了 VIT 模型训练模型。相关链接：Liujiacheng-0925/- (github.com)

二、项目实施过程

数据收集与准备：首先，使用包含手势数字的图片数据集，并对数据进行预处理，包括图像增强、数据划分等操作。

模型设计与训练：基于提供的初始代码，利用 VIT 和 FPN 特征金字塔等技术，设计并训练手势图片分类模型。通过调参优化模型结构，提高分类准确性。为了解决此问题，我们使用 VIT 和 FPN 特征金字塔来实现分类功能。

FPN（Feature Pyramid Network）特征金字塔是一种用于目标检测和语义分割任务的网络结构。它通过建立多尺度的特征金字塔结构，实现在不同尺度下提取图像中目标的特征。FPN 结合了底层和顶层特征信息，使得网络能够在多个尺度下进行目标检测和分割，提高了网络在处理尺度变化和不同尺寸目标时的准确性和性能。FPN 的结构包括自下而上的特征提取和自顶而下的特征融合，从而有效

地提高了网络的表征能力和准确性。

Vision Transformer（ViT）是一种全新的基于 Transformer 架构的神经网络模型，专门用于处理视觉任务。传统的神经网络模型如卷积神经网络（CNN）在处理图像任务时存在一定的局限性，而 ViT 通过将图像分割成一系列的 patches 并将它们压缩成一维向量序列，然后输入 Transformer 模型，从而在图像任务上取得了惊人的表现。与 CNN 不同，ViT 没有使用卷积操作，而是通过自注意力机制来建立全局的图像理解。该模型通过多层的 Transformer Encoder 单元来提取图像的表示，其中每个 Encoder 单元由多头自注意力机制和全连接层组成。ViT 将图像分为多个 patches，并且保持它们之间的空间关系，使得模型能够学习到图像的全局信息。Vision Transformer 已经在许多视觉任务中取得了与 CNN 相媲美甚至更好的表现，包括图像分类、目标检测、语义分割等。它的简洁架构和出色性能使得它成为当前研究领域的热门话题，Vision Transformer 的成功也证明了 Transformer 模型在处理视觉任务中的巨大潜力。

模型测试与验证：使用测试集验证模型的准确率和性能指标，在不断优化模型的过程中提高分类效果。

项目整合与部署：将训练好的模型整合到项目中，并进行部署测试，确保模型能够稳定运行并实现预期效果。

三、项目运行结果

经过模型训练和测试验证，项目取得了较好的分类效果，能够准确识别手势数字 0~9。模型在测试集上取得了高准确率，并且能够快速响应用户输入，实现了高效的手势识别功能，下面分布介绍我们使用得两个模型，由于算力问题，在跑 FPN 特征金字塔时候在 AutoDL 租用的 3090 显卡。

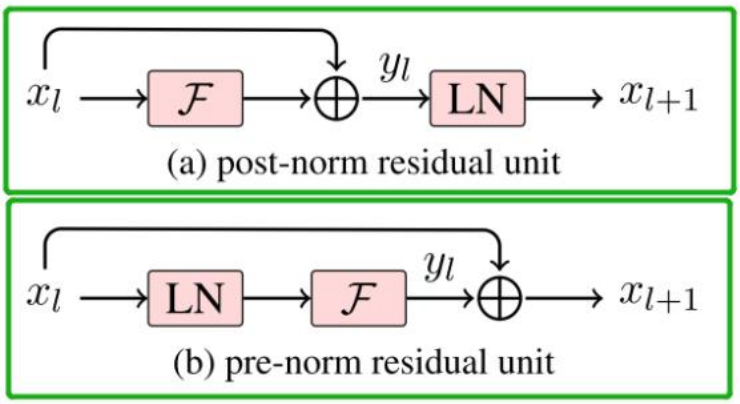
模型部分：

这段 pair 函数的作用是：判断 t 是否是元组，如果是，直接返回 t；如果不是，则将 t 复制为元组(t, t)再返回。用来处理当给出的图像尺寸或块尺寸是 int 类型时，直接返回为同值元组,如(224, 224)。

1：VIT(Vision Transformer)

```
import torch
from torch import nn, einsum
from einops import rearrange, repeat
from einops.layers.torch import Rearrange
import torch.nn.functional as F
import torch.utils.model_zoo as model_zoo
def pair(t):
    return t if isinstance(t, tuple) else (t, t)
```

PreNorm 层：规范化层的类封装。




```
class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(self.norm(x), **kwargs)
```

FFN 层：FeedForward 层由线性层，配合激活函数 GELU 和 Dropout 实现，Multi-Head Attention 的输出做了残差连接和 Norm 之后得数据，然后 FeedForward 做了两次线性变换，目的是更加深入的提取特征。

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

Attention 层：Attention 是 Transformer 中的核心部件，多头自注意力机制

```
class Attention(nn.Module):
    def __init__(self, dim, heads=8, dim_head=64, dropout
=0.1):
        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.attend = nn.Softmax(dim=-1)
            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):  ## 最重要的都是forword函数了
            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)
            # 分成多少个Head,与TRM生成qkv 的方式不同，要更简单,
不需要区分来自Encoder 还是Decoder
            dots = torch.matmul(q, k.transpose(-1, -2)) * sel
f.scale
            attn = self.attend(dots)
            out = torch.matmul(attn, v)
            out = rearrange(out, 'b h n d -> b n (h d)')
            return self.to_out(out)
```

构建 Transformer:

```
 class Transformer(nn.Module):
        def __init__(self, dim, depth, heads, dim_head, mlp_d
im, dropout=0.):
            super().__init__()
            self.layers = nn.ModuleList([])
            for _ in range(depth):
                self.layers.append(nn.ModuleList([
                    PreNorm(dim, Attention(dim, heads=heads,
dim_head=dim_head, dropout=dropout)),
                    PreNorm(dim, FeedForward(dim, mlp_dim, dr
opout=dropout))
                ]))
        def forward(self, x):
            for attn, ff in self.layers:
                x = attn(x) + x
                x = ff(x) + x
            return x
```

CustomNet 这个类定义了一个自定义的神经网络模型，主要用于处理图像分类任务。该模型采用了 Vision Transformer (ViT) 的架构。以下是该类的主要作用：

初始化函数：在初始化函数中设置了各种参数，包括图像尺寸、patch 大小、类别数、维度、深度、头数、MLP 维度、池化类型等。在初始化函数中还定义了一些必要的参数和层，如图像的 patch embedding、位置编码、CLS token、dropout 等。前向传播函数：在前向传播函数中，首先将输入的图像转换为 patch embedding，然后在第一个维度上添加一个 CLS token，并将位置编码与输入相加。接着将输入数据传递给 Transformer 模块进行处理，得到输出后根据池化类型进行池化操作，最后通过全连接层进行分类预测。

```
1.  class CustomNet(nn.Module):
2.      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.):
3.          super().__init__()
4.          # 初始化函数内，是将输入的图片，得到 img_size ，patch_size 的宽和高
5.          image_height, image_width = pair(image_size)  ## 224*224 *3
6.          patch_height, patch_width = pair(patch_size)  ## 16 * 16  *3
7.          # 图像尺寸必须能被patch大小整除
8.          assert image_height % patch_height == 0 and image_width % patch_width == 0, 'Image dimensions must be divisible by the patch size.'
9.          num_patches = (image_height // patch_height) * (image_width // patch_width)  ## 步骤1.一个图像 分成 N 个patch
10.         patch_dim = channels * patch_height * patch_width
11.         assert pool in {'cls', 'mean'}, 'pool type must be either cls (cls token) or mean (mean pooling)'
12.         self.to_patch_embedding = nn.Sequential(
13.             Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=patch_height, p2=patch_width),  # 步骤2.1 将patch 铺开
14.             nn.Linear(patch_dim, dim),  # 步骤2.2 然后映射到指定的embedding 的维度
15.         )
16.         self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
17.         self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
18.         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.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, num_classes)
        )
    def forward(self, img):
        x = self.to_patch_embedding(img)  ## img 1 3 224 224  输出形状x : 1 196 1024
        b, n, _ = x.shape  ##
        # 将cls 复制 batch_size 份
        cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
        # 将cls token 在维度1 扩展到输入上
        x = torch.cat((cls_tokens, x), dim=1)
        # 添加位置编码
        x += self.pos_embedding[:, :(n + 1)]
        x = self.dropout(x)
        # 输入 TRM
        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)
```

2：FPN 特征金字塔

定义了一个神经网络模型 CustomNet_2，该模型包含了一个 ResNet50 的主干网络和一些额外的层，用于进行目标检测或分类任务。具体分析如下：

在 init 方法中：初始化了一个 ResNet50 模型，并将其前面的一些层作为自定义网络的一部分。定义了几个卷积层、全连接层和上采样方法。设置了神经网络模型中的一些层，如 Smooth layers、Lateral layers 和全连接层。_init_weights 方法用于初始化神经网络模型的权重和偏置，根据不同类型的层执行不同的初始化操作。_upsample_add 方法用于将两个特征图上采样并相加，以获得更高分辨率的特征图。forward 方法定义了模型前向传播的过程：

通过 ResNet50 的不同层提取特征，并进行顶层向下传播，得到不同分辨率的特征图 p2、p3、p4 和 p5。将不同层的特征图进行上采样并相加，得到多尺度特征图。经过 Smooth 层和全连接层，最终输出模型的预测结果。实现对输入数据的特征提取和分类任务预测。

```python
class CustomNet_2(nn.Module):
    def __init__(self, pretrained=False):
        super(CustomNet_2, self).__init__()
        resnet = resnet50(pretrained=pretrained)  # 是否加载预训练的权重
        self.toplayer = nn.Conv2d(2048, 256, kernel_size=1, stride=1, padding=0)  # Reduce channels
        self.toplayer_3 = nn.Conv2d(256, 40, kernel_size=1, stride=1, padding=0)  # Reduce channels
        self.layer0 = nn.Sequential(resnet.conv1, resnet.bn1, resnet.leakyrelu,
                                    resnet.maxpool)  # Sequential 是一个容器，用于按顺序组织神经网络的模块。
        self.layer1 = nn.Sequential(resnet.layer1)
        self.layer2 = nn.Sequential(resnet.layer2)
        self.layer3 = nn.Sequential(resnet.layer3)
        self.layer4 = nn.Sequential(resnet.layer4)
        # Smooth layers
        # self.smooth1 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.smooth2 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.smooth3 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        # Lateral layers
        self.latlayer1 = nn.Conv2d(1024, 256, kernel_size=1, stride=1, padding=0)
        self.latlayer2 = nn.Conv2d(512, 256, kernel_size=1, stride=1, padding=0)
        self.latlayer3 = nn.Conv2d(256, 256, kernel_size=1, stride=1, padding=0)
        # 全连接
        self.linear1 = nn.Linear(65536, 4096)
        self.linear2 = nn.Linear(4096, 10)
    # 这个函数的作用是初始化神经网络模型的权重和偏置。它遍历了模型的所有层，并根据层的类型进行不同的初始化操作。
    def _init_weights(self, m):
        if isinstance(m, nn.Linear):
            # we use xavier_uniform following official JAX ViT:
            torch.nn.init.xavier_uniform_(m.weight)
            if isinstance(m, nn.Linear) and m.bias is not None:
                nn.init.constant_(m.bias, 0)
```

```
            elif isinstance(m, nn.LayerNorm):
                nn.init.constant_(m.bias, 0)
                nn.init.constant_(m.weight, 1.0)
    def _upsample_add(self, x, y):
        _, _, H, W = y.size()
        return F.interpolate(x, size=(H, W), mode='bilinear', align_corners=False) + y
    def forward(self, x):
        c1 = self.layer0(x)
        c2 = self.layer1(c1)
        c3 = self.layer2(c2)
        c4 = self.layer3(c3)
        c5 = self.layer4(c4)
        # Top-down
        p5 = self.toplayer(c5)  # 经过一个1x1的卷积层进行降维
        p4 = self._upsample_add(p5, self.latlayer1(c4))  # p5与c4连接并上采样得到特征图p4
        p3 = self._upsample_add(p4, self.latlayer2(c3))  # p4与c3连接并上采样得到特征图p3
        p2 = self._upsample_add(p3, self.latlayer3(c2))  # p3与c2连接并上采样得到特征图p2
        # Smooth
        p2 = self.smooth3(p2)
        # Attention
        p2 = p2.view(-1, 256 * 16 * 16)
        p2 = self.linear1(p2)
        p2 = self.linear2(p2)
        return p2
```

这个类定义了一个 ResNet 模型，ResNet 是一个非常深的卷积神经网络，该模型通过堆叠多个残差块（block）来构建。在 ResNet 中，每个残差块包括多个卷积层和残差连接，使得网络可以更轻松地学习到深层特征。具体来说：

初始化函数中定义了模型的一些基本组件，如卷积层、Batch Normalization 层、LeakyReLU 激活函数、MaxPooling 层、全连接层等。同时还定义了网络的层数、类别数等参数。

_make_layer 函数用于构建多个残差块的网络层，根据输入的残差块类型、通道数、块数、步长等参数构建相应的层。

forward 函数定义前向传播过程，即输入数据经过一系列卷积、池化和全连接操作后得到分类结果。具体操作包括：卷积、BN、LeakyReLU、最大池化、多个残差块的堆叠、全局平均池化、展平、全连接。

最后，模型初始化时对卷积层的参数进行了初始化，并保证了每个残差块的维度匹配，使得可以顺利地堆叠残差块。

ResNet 类定义了一个用于进行图像分类任务的深度卷积神经网络模型，通过残差连接解决了深度网络中的梯度消失和梯度爆炸问题，可以有效地训练深度网络并提高模型性能。

```
1.  class ResNet(nn.Module):
2.      def __init__(self, block, layers, num_classes=1000):
3.          self.inplanes = 64
4.          super(ResNet, self).__init__()
5.          self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stri
    de=2, padding=3, bias=False)
6.          self.bn1 = nn.BatchNorm2d(64)
7.          self.leakyrelu = nn.LeakyReLU(inplace=True)
8.          self.maxpool = nn.MaxPool2d(kernel_size=3, stride
    =2, padding=1)
9.          self.layer1 = self._make_layer(block, 64, layers[
    0])
10.         self.layer2 = self._make_layer(block, 128, layers
    [1], stride=2)
11.         self.layer3 = self._make_layer(block, 256, layers
    [2], stride=2)
12.         self.layer4 = self._make_layer(block, 512, layers
    [3], stride=2)
13.         self.avgpool = nn.AvgPool2d(7, stride=1)
14.         self.fc = nn.Linear(512 * block.expansion, num_cl
    asses)
15.         for m in self.modules():
16.             if isinstance(m, nn.Conv2d):
17.                 nn.init.kaiming_normal_(m.weight, mode="f
    an_out", nonlinearity="leaky_relu")
18.             elif isinstance(m, nn.BatchNorm2d):
19.                 nn.init.constant_(m.weight, 1)
20.                 nn.init.constant_(m.bias, 0)
21.     def _make_layer(self, block, planes, blocks, stride=1)
    :
22.         downsample = None
23.         if stride != 1 or self.inplanes != planes * block.
    expansion:
24.             downsample = nn.Sequential(
25.                 nn.Conv2d(self.inplanes, planes * block.e
    xpansion,
26.                           kernel_size=1, stride=stride, b
    ias=False),
```

```
                    nn.BatchNorm2d(planes * block.expansion))
        layers = []
        layers.append(block(self.inplanes, planes, stride,
    downsample))
        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))
        return nn.Sequential(*layers)
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.leakyrelu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = x.mean(3).mean(2)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

这个函数用于构建一个 ResNet-50 模型的编码器。ResNet-50 是一个深度卷积神经网络架构，可以用于图像分类，目标检测等任务。该函数通过调用 ResNet 类来构建 ResNet-50 模型，传入的参数包括 Bottleneck 的类型以及每个 stage 的卷积层数量。如果 pretrained 参数设置为 True，则会加载预训练的 ResNet-50 模型参数。最后返回构建的 ResNet-50 模型。整个函数主要实现以下功能：

构建 ResNet-50 模型，使用的是 Bottleneck 结构，包括 4 个 stage，每个 stage 的卷积层数量分别是 3、4、6、3。根据 pretrained 参数的值，加载预训练的 ResNet-50 模型参数。返回构建的 ResNet-50 模型。

这个函数的作用是方便用户快速构建并使用 ResNet-50 模型，同时还提供了加载预训练参数的功能，方便用户在自己的数据集上进行微调或者迁移学习。

```
def resnet50(pretrained=False, **kwargs):
    """Constructs a ResNet-50 model Encoder"""
    model = ResNet(Bottleneck, [3, 4, 6, 3], **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url("https:/
/download.pytorch.org/models/resnet50-19c8e357.pth"))
    return model
```

BasicBlock 这个类是一个基本的残差块，用于构建 ResNet 网络中的基本模块。在深度学习中，由于网络层数较深，容易出现梯度消失或梯度爆炸的问题，为了解决这个问题，ResNet 提出了残差学习的概念。基本的残差块中包含了两个 3x3 卷积层和一个 BatchNormalization 层，以及一个 LeakyReLU 激活函数，这些层构成了基本的残差学习单元。

在这个类中，构造函数__init__中定义了 BasicBlock 的结构，包括两个卷积层和两个 BatchNormalization 层，以及 LeakyReLU 激活函数。同时也定义了残差连接的 downsample 和 stride 参数。

在 forward 方法中，实现了残差块的前向传播过程。首先将输入 x 保存为 residual，然后经过 conv1、bn1 和 LeakyReLU 激活函数操作，再经过 conv2 和 bn2 操作。如果残差连接不为空，即 downsample 不为 None，就对输入 x 进行 downsample 操作，然后将两部分结果相加，最后再经过 LeakyReLU 激活函数得到最终输出 out。总的来说，这个 BasicBlock 类定义了 ResNet 中的基本 Residual Block 结构，用于构建深度残差网络。

```
1.  class BasicBlock(nn.Module):
2.      expansion = 1
3.      def __init__(self, inplanes, planes, stride=1, downsa
    mple=None):
4.          super(BasicBlock, self).__init__()
5.          self.conv1 = conv3x3(inplanes, planes, stride)
6.          self.bn1 = nn.BatchNorm2d(planes)
7.          self.leakyrelu = nn.LeakyReLU(inplace=True)
8.          self.conv2 = conv3x3(planes, planes)
9.          self.bn2 = nn.BatchNorm2d(planes)
10.         self.downsample = downsample
11.         self.stride = stride
12.     def forward(self, x):
13.         residual = x
14.         out = self.conv1(x)
15.         out = self.bn1(out)
16.         out = self.leakyrelu(out)
17.         out = self.conv2(out)
18.         out = self.bn2(out)
19.         if self.downsample is not None:
20.             residual = self.downsample(x)
21.         out += residual
22.         out = self.leakyrelu(out)
23.         return out
```

Bottleneck 这个类实现了 ResNet 中的瓶颈残差块结构。瓶颈残差块主要用于构建深层神经网络，在减少参数数量的同时保持模型的性能。

在该类中，首先通过三个卷积层（分别是 1x1, 3x3, 1x1）对输入进行特征提取和降维操作，然后通过 Batch Normalization 对特征进行归一化处理，最后通过 LeakyReLU 激活函数进行非线性变换。

瓶颈残差块的核心在于通过 1x1 卷积先将特征降维，然后再通过 3x3 卷积进行特征提取，最后再通过 1x1 卷积将特征维度扩展回来。这种结构可以有效减少计算量和参数数量，同时提高网络的表达能力。

另外，瓶颈残差块还包含了一个 downsample 参数，用于对输入进行下采样操作，以适应不同尺寸的输入。在前向传播过程中，通过残差连接将输入与经过卷积和 Batch Normalization 处理后的特征相加，然后再经过 LeakyReLU 激活函数得到最终的输出。

总的来说，瓶颈残差块的作用是构建深层神经网络中的基本模块，用于提高网络的性能和表达能力，同时保持较少的参数数量和计算量。通过堆叠多个瓶颈残差块可以构建深层的 ResNet 模型，用于解决复杂的计算机视觉任务。

```
1.  class Bottleneck(nn.Module):
2.      expansion = 4
3.      def __init__(self, inplanes, planes, stride=1, downsa
   mple=None):
4.          super(Bottleneck, self).__init__()
5.          self.conv1 = nn.Conv2d(inplanes, planes, kernel_s
   ize=1, bias=False)
6.          self.bn1 = nn.BatchNorm2d(planes)
7.          self.conv2 = nn.Conv2d(
8.              planes, planes, kernel_size=3, stride=stride,
    padding=1, bias=False
9.          )
10.         self.bn2 = nn.BatchNorm2d(planes)
11.         self.conv3 = nn.Conv2d(
12.             planes, planes * self.expansion, kernel_size=
   1, bias=False
13.         )
14.         self.bn3 = nn.BatchNorm2d(planes * self.expansion
   )
15.         self.leakyrelu = nn.LeakyReLU(inplace=True)
16.         self.downsample = downsample
17.         self.stride = stride
18.     def forward(self, x):
19.         residual = x
20.         out = self.conv1(x)
```

```
        out = self.bn1(out)
        out = self.leakyrelu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out = self.leakyrelu(out)
        out = self.conv3(out)
        out = self.bn3(out)
        if self.downsample is not None:
            residual = self.downsample(x)
        out += residual
        out = self.leakyrelu(out)
        return out
```

训练部分：

该函数定义了一个训练循环，包括多个 epoch 的训练过程。具体作用如下：

1.将模型置为训练模式。

2.在每个 epoch 中，对数据加载器中的每个 batch 进行训练。

3.对每个 batch 进行以下操作：

4.将输入数据和标签移动到指定的设备（CPU 或 GPU）上。

5.将输入数据输入模型，得到模型的输出。

6.根据模型输出和真实标签计算损失值。

7.清空优化器中之前的梯度。

8.根据损失值进行反向传播，更新模型参数。

9.记录当前 batch 的损失值。

10.每个 epoch 结束时计算并保存平均损失值，输出当前 epoch 的训练进度。

11.保存模型参数到文件中。

12.在每个 epoch 结束时调用测试函数对模型进行评估。

13.可视化训练过程中的损失值变化，将损失值随 epoch 数目的变化保存为图像文件。

14.打印并显示训练过程中的损失值变化图像。

总体来说，该函数的作用是定义了一个包含多个 epoch 的训练循环，负责模型的训练、参数更新、损失值计算、模型评估和可视化

```
import torch
from matplotlib import pyplot as plt
from torch import nn
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader
from dataset import CustomDataset
```

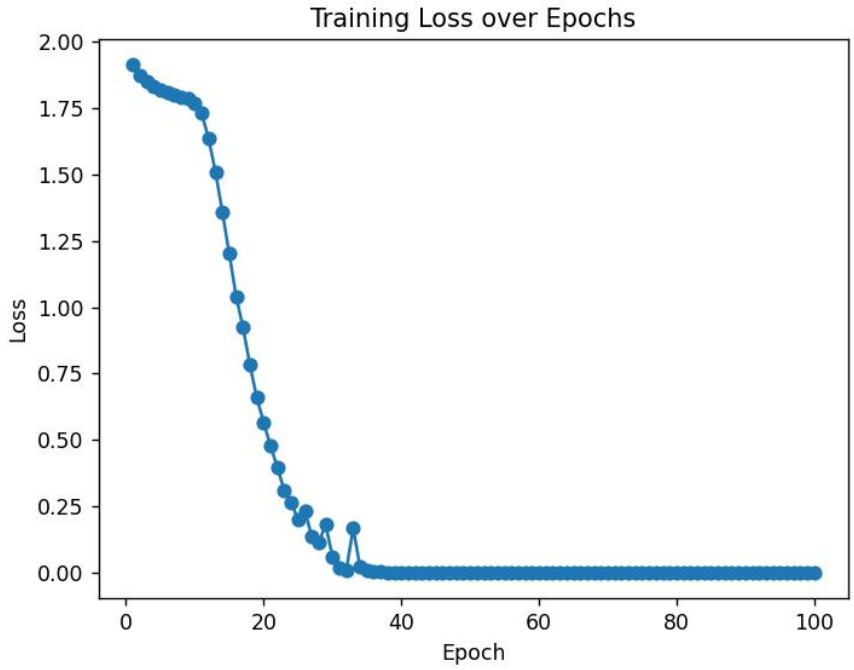
```
 from model import CustomNet, CustomNet_2
 from test import test
 def train_loop(epoch, dataloader, model, loss_fn, optimizer, device):
    """定义训练流程。
    :param epoch: 定义训练的总轮次
    :param dataloader: 数据加载器
    :param model: 模型，需在 model.py 文件中定义好
    :param loss_fn: 损失函数
    :param optimizer: 优化器
    :param device: 训练设备，即使用哪一块 CPU、GPU 进行训练
    """
    # 将模型置为训练模式
    model.train()
    # 记录每个 epoch 的损失值
    all_losses = []
    for e in range(epoch):
        print(f"Epoch {e + 1}/{epoch}")
        running_loss = 0.0
        for batch_idx, batch in enumerate(dataloader):
            inputs, targets = batch['image'].to(device), batch['label'].to(device)
            # 前向传播 + 计算损失
            outputs = model(inputs)
            loss = loss_fn(outputs, targets)
            # 将梯度清零
            optimizer.zero_grad()
            # 反向传播 + 更新模型参数
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        average_loss = running_loss / len(dataloader)
        all_losses.append(average_loss)
        print(f"Epoch {e + 1}/{epoch} -> Loss: {average_loss:.4f}")
        # 保存模型
        torch.save(model, rf'D:\pycham\pythonProject20\nndl_project-master\models\model_epoch_{e + 1}.pkl')
        # 测试数据加载器
        test_dataloader = DataLoader(CustomDataset('./images/test.txt', './images/test', ToTensor),
                                     batch_size=32)
        test(test_dataloader, model, device)
    plt.figure()
```

```
    plt.plot(range(1, epoch + 1), all_losses, '-o')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training Loss over Epochs')
    plt.savefig('training_loss.png')  # 保存图像文件
    plt.show()  # 显示图像
if __name__ == "__main__":
    # 定义模型超参数
    BATCH_SIZE = 32
    LEARNING_RATE = 5e-2
    EPOCH = 100
    model = CustomNet(
        image_size=64,
        patch_size=8,
        num_classes=10,
        dim=128,
        depth=3,
        heads=4,
        mlp_dim=512,
        dropout=0.1,
        emb_dropout=0.1
    )
    # fpn = CustomNet_2()
    # 实例化
    if torch.cuda.is_available():
        device = torch.device("cuda")
        print(f"Using GPU: {torch.cuda.get_device_name(device)}")
    else:
        device = torch.device("cpu")
        print("Using CPU")
    model.to(device)
    # 训练数据加载器
    train_dataloader = DataLoader(CustomDataset('./images/train.txt', './images/train', ToTensor),batch_size=BATCH_SIZE)
    # 损失函数
    loss_fn = nn.CrossEntropyLoss()
    # 学习率和优化器
    optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)
    # 调用训练方法
    train_loop(EPOCH, train_dataloader, model, loss_fn, optimizer, device)
```

模型一 VIT 的训练结果：

```
Using GPU: GeForce GTX 1650
Epoch 1/100
Epoch 1/100 -> Loss: 1.9138
Test Acc: 0.1667
Epoch 2/100
Epoch 2/100 -> Loss: 1.8765
Test Acc: 0.1667
Epoch 3/100
Epoch 3/100 -> Loss: 1.8552
Test Acc: 0.1667
Epoch 4/100
Epoch 4/100 -> Loss: 1.8419
Test Acc: 0.1667
```

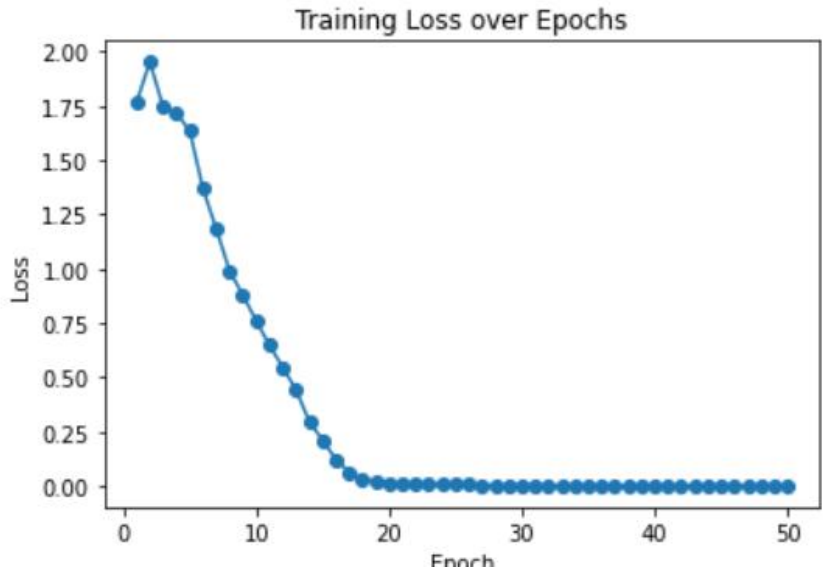
```
Epoch 97/100
Epoch 97/100 -> Loss: 0.0002
Test Acc: 0.9688
Epoch 98/100
Epoch 98/100 -> Loss: 0.0002
Test Acc: 0.9688
Epoch 99/100
Epoch 99/100 -> Loss: 0.0002
Test Acc: 0.9688
Epoch 100/100
Epoch 100/100 -> Loss: 0.0001
Test Acc: 0.9688
```



模型二 FPN 的训练结果：

```
Using GPU: NVIDIA GeForce RTX 3090
Epoch 1/50
Epoch 1/50 -> Loss: 1.7670
Test Acc: 0.1667
Epoch 2/50
Epoch 2/50 -> Loss: 1.9531
Test Acc: 0.2604
Epoch 3/50
Epoch 3/50 -> Loss: 1.7462
Test Acc: 0.1875
Epoch 4/50
Epoch 4/50 -> Loss: 1.7168
Test Acc: 0.3208
Epoch 5/50
Epoch 5/50 -> Loss: 1.6363
Test Acc: 0.3958
```

```
Epoch 46/50
Epoch 46/50 -> Loss: 0.0016
Test Acc: 0.9604
Epoch 47/50
Epoch 47/50 -> Loss: 0.0015
Test Acc: 0.9604
Epoch 48/50
Epoch 48/50 -> Loss: 0.0015
Test Acc: 0.9604
Epoch 49/50
Epoch 49/50 -> Loss: 0.0014
Test Acc: 0.9604
Epoch 50/50
Epoch 50/50 -> Loss: 0.0014
Test Acc: 0.9604
```



测试部分：

这个函数的作用是针对给定的数据加载器 dataloader 以及训练好的模型 model 进行测试，评估模型在测试数据上的性能表现。具体流程如下：

1.将模型置为评估（测试）模式，通过 model.eval()实现。

2.获取测试集的样本总数 size = len(dataloader.dataset)。

3.初始化 correct_num 为 0，用于记录预测正确的样本数。

4.使用 torch.no_grad()上下文管理器，关闭自动求导，避免在测试阶段产生不必要的梯度计算。

5.遍历数据加载器 dataloader，逐个获取测试数据样本进行测试。

6.将测试数据样本 inputs 和对应标签 targets 转移到设备 device 上。

7.通过前向传播得到模型的输出 output。

8.根据输出结果获取预测结果 pred，并将预测结果与真实标签进行比较，累计预测正确的样本数 correct_num。

9.最终计算测试准确率 accuracy = correct_num / size，并打印输出测试准确率信息。

10.返回测试准确率结果。

总结来说，该函数的作用是对模型在测试数据集上进行测试，计算模型的准确率并返回结果。通过该函数可以评估模型在未见数据上的泛化能力和性能。

```
import torch
from torch.utils.data import DataLoader
from torchvision.transforms import ToTensor
from dataset import CustomDataset
def test(dataloader, model, device):
    """定义测试流程。
    :param dataloader: 数据加载器
    :param model: 训练好的模型
    :param device: 测试使用的设备，即使用哪一块 CPU、GPU 进行模型测试
    """
    # 将模型置为评估（测试）模式
    model.eval()
    size = len(dataloader.dataset)  # 测试集样本总数
    correct_num = 0  # 预测正确的样本数
    with torch.no_grad():
        for batch_idx, batch in enumerate(dataloader):
            inputs, targets = batch['image'].to(device), batch['label'].to(device)
            # 前向传播
            output = model(inputs)
```

```
            # 获取预测结果
            pred = output.argmax(dim=1, keepdim=True)
            # 累计预测正确的样本数
            correct_num += pred.eq(targets.view_as(pred)).sum().item()
    accuracy = correct_num / size
    print(f'Test Acc: {accuracy:.4f}')
    return  accuracy
if __name__ == "__main__":
    # 加载训练好的模型
    model = torch.load('./models/较好模型/vit.pkl')
    if torch.cuda.is_available():
        device = torch.device("cuda")
    else:
        device = torch.device("cpu")
    model.to(device)
    # 测试数据加载器
    test_dataloader = DataLoader(CustomDataset('./images/test.txt', './images/test', ToTensor),
                                 batch_size=32,shuffle=True)
    # 运行测试函数
    test(test_dataloader, model, device)
```

1：VIT 测试结果：

```
Test Acc: 0.9688
```

2：FPN 特征金字塔测试结果：

```
Test Acc: 0.9604
```

推理应用：

inference 这个函数是一个模型推理的流程，接受一个输入图片的路径、训练好的模型和设备作为参数。函数首先将模型置为评估模式，然后打开输入图片，并将其转换为张量。接着将图片张量移动到指定的设备上。使用 torch.no_grad() 禁用梯度计算，对图片进行推理，得到推理输出。在这里假设输出是一个分类结果，通过 torch.max 函数找到输出中概率最大的类别，并返回预测结果。

最后，函数会显示输入图片，并在标题上显示模型对图片的预测结果。最后返回预测结果。整个流程完成了对给定图片的推理，并返回预测的结果。

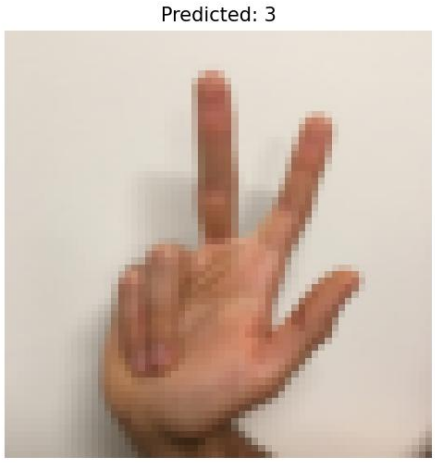
```
import torch
from PIL import Image
```

```
3.  from torchvision.transforms import ToTensor
4.  import matplotlib.pyplot as plt
5.  from model import CustomNet, CustomNet_2,Bottleneck
6.  def inference(image_path, model, device):
7.      """定义模型推理应用的流程。
8.      :param image_path: 输入图片的路径
9.      :param model: 训练好的模型
10.     :param device: 模型推理使用的设备，即使用哪一块 CPU、GPU 进行模型推理
11.     """
12.     # 将模型置为评估（测试）模式
13.     model.eval()
14.     # START----------------------------------------------------------
15.     # 打开图片并转换为张量
16.     image = Image.open(image_path)
17.     image_tensor = ToTensor()(image).unsqueeze(0)  # 增加一个 batch 的维度
18.     # 将图片张量移到指定设备上
19.     image_tensor = image_tensor.to(device)
20.     # 禁用梯度计算，进行推理
21.     with torch.no_grad():
22.         output = model(image_tensor)
23.     # 输出处理（假设输出是分类结果）
24.     _, predicted = torch.max(output, 1)
25.     # END------------------------------------------------------------
26.     # 显示图片
27.     plt.imshow(image)
28.     plt.title(f'Predicted: {predicted.item()}')
29.     plt.axis('off')  # 隐藏坐标轴
30.     plt.show()
31.     # 返回预测结果
32.     print(predicted)
33.     return predicted.item()
34. if __name__ == "__main__":
35.     # 指定图片路径
36.     image_path = "./images/test/signs/img_0006.png"
37.     # 加载训练好的模型
38.     model = torch.load(r'D:\pycham\pythonProject20\nndl_project-master\models\较好模型\models.circ')
39.     if torch.cuda.is_available():
40.         device = torch.device("cuda")
41.     else:
```

```
        device = torch.device("cpu")
    model.to(device)
    # 显示图片，输出预测结果
    inference(image_path, model, device)
```

1：VIT 应用结果：

```
D:\python\envs\MAE\python.exe D:/pyc
tensor([3], device='cuda:0')
```

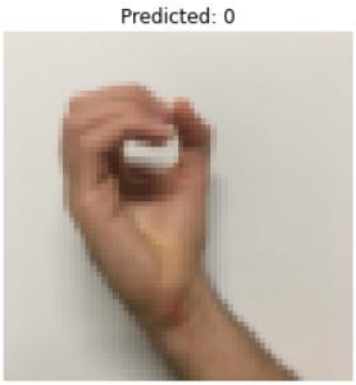


2：FPN 特征金字塔应用结果：

[17]:
```
image_path = "./images/test/signs/img_0002.png"

# 加载训练好的模型
model = torch.load(r'models.pkl')
if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")
model.to(device)

# 显示图片，输出预测结果
inference(image_path, model, device)
```




```
tensor([0], device='cuda:0')
```

四、总结与体会

在本次实训项目中，我们使用了 VIT（Vision Transformer）和 FPN（Feature Pyramid Network）两种深度学习模型来实现手势识别任务。首先，我们对这两种模型进行了深入的了解和研究，分析了它们各自的优势和特点，然后根据手势图片数据集的特点和任务要求选择了合适的模型进行实现。

在使用 VIT 模型时，我们首先将手势图片数据集进行预处理，包括数据增强、归一化等操作，然后将处理后的数据输入到 VIT 模型中进行训练。VIT 模型的核心思想是将图片分割成固定大小的图块，然后通过 Transformer 模块来学习图块之间的全局关系，最终实现图像分类任务。通过调整超参数和优化算法，我们不断优化模型，最终取得了不错的分类效果。

另外，我们还尝试了使用 FPN 模型来实现手势识别任务。FPN 模型是一种特征金字塔网络，能够有效地处理不同尺度的特征信息，提高模型对目标检测和分类任务的准确性。我们将手势图片数据集输入到 FPN 模型中进行训练，通过特征金字塔网络的层级结构，不断提取和融合不同尺度的特征信息，最终实现了准确的手势识别结果。

在完成实训项目的过程中，我们遇到了一些挑战和困难，比如过拟合，模型训练速度慢等问题，但通过不断尝试和调整，最终成功地解决了这些问题，取得了相对满意的实验结果。在参数修改之前在测试集上准确率仅仅只有 0.86 左右，而在训练集合达到了 1，这是典型的过拟合现象，我们通过修改参数，将其在测试集的准确率提升到 0.96 左右，这是不小的进步。通过这次实训项目，我们不仅掌握了使用深度学习模型实现图像分类任务的技能，还学会了团队协作、问题解决等实用技能，对深度学习和人工智能领域有了更深入的了解和认识。

总的来说，这次实训项目让我们收获颇丰，不仅提升了我们的技术能力，还培养了我们的实际操作能力和团队合作精神。我们相信在未来的学习和工作中，这些经验和技能将对我们产生积极的影响，让我们能够更好地应对各种挑战和机遇，迈向更高的技术高度。