Deep Learning Practical Tutorial - Github Project Reproduction and Application

School of Optics and Photonics, BIT, FuYu

目录

摘要

该文档总结了数个已经复现的深度学习项目，包括代码实现、数据集、训练细节等内容，旨在为大家提供一个全面的参考资源，帮助其更好地理解和应用深度学习技术。

1 预备知识

1.1 参考教程

此处我们只讲解代码实现层面上的基本知识，更多理论知识请参考相关教材和论文。在这里我们给出几个可供参考的学习资料：

- Dive into Deep Learning中文版
- CS231n课程笔记斯坦福大学计算机视觉课程
- 小土堆教程：PyTorch深度学习快速入门教程

如果是初学者，建议先从基本知识入手，理解神经网络的基本概念、常用的网络结构（如卷积神经网络、循环神经网络等）、损失函数和优化算法等内容。掌握这些基础知识后，再逐步深入到具体的项目实现中。参考Dive into Deep Learning和CS231n（CS231n的课后作业可不做，可以参考本文后续的实际操作）。当然不想看电子书的话，可以搜索在B站李宏毅、吴恩达等老师的教程。最后，小土堆教程给出了更加细致的代码实现细节，讲解了在实际写代码时遇到的问题和解决方法，例如张量大小的计算、如何修改模块等。我建议小土堆的教程可以多少研究一下，特别是**Dataset、Dataloader、现有模型的使用和修改**等章节，值得参考。同时，提供之前完成的各种简单例程和代码片段，方便大家理解和实践。

在实际的实现过程中，我们多多少少会遇到一些奇怪的问题和无从下手的情况，下面我将通过该文档展示2025年复现的项目，逐步总结一些常见的问题和解决思路，供大家参考。

1.2 主要过程

在初步掌握上述知识后，我们可以选择一个感兴趣的项目进行复现。建议从简单的项目开始，例如图像分类或文本分类等任务。通过阅读相关

论文和代码，理解其实现细节和设计思路。在复现过程中，建议先从数据预处理、模型构建、训练和评估等基本步骤入手，逐步完善代码实现。

在这里，我们将整个过程抽象化讲解，在我们遇到各式各样的代码风格和实现细节时，都离不开以下的过程：

- 数据集：明确任务目标，了解数据集的结构和特点。根据任务需求，对数据进行清洗、归一化、增强等处理。
- 模型构建：选择合适的模型架构，定义网络层次和参数。
- 训练模型：设置损失函数、优化器和训练参数，进行模型训练。
- 模型评估：使用验证集或测试集评估模型性能，调整超参数。
- 结果分析：分析模型输出，理解其优缺点，提出改进方案。

1.2.1 数据

数据集这一部分就有很多细节需要注意。

首先，我们拿到不同的网络架构，他的输入尺寸可能不一样，有的需要224x224，有的需要128x128等，所以我们要根据具体的模型来调整数据预处理的部分。这里我们需要去仔细阅读每一个项目的说明：readme、dataset等，如果是一个正常的能跑通的GitHub项目，这些关于数据的说明一定是完备的可操作的，**如果在复现他人GitHub项目的过程中发现这些内容是缺失的，果断去找其他的GitHub项目**，不用浪费时间。其次，不同的数据集格式也不一样，有的可能是文件夹结构，有的是csv文件等，我们要根据具体的数据集来编写相应的Dataset类。最后，数据增强也是一个重要的环节，可以提高模型的泛化能力，我们可以使用一些常见的数据增强方法，如随机裁剪、旋转、翻转等。那么在这里，无论每个项目的细节如何不同，主要的逻辑不会有差别的，我们都要经历上述的过程，那么比较直观简单的例子如下在代码中体现如下：

```
data_transform = {
    "train": transforms.Compose([transforms.RandomResizedCrop(224),
                                 transforms.RandomHorizontalFlip(),
                                 transforms.ToTensor()]),
    "val": transforms.Compose([transforms.Resize(256),
                               transforms.CenterCrop(224),
                               transforms.ToTensor()])}
```

```
    train_dataset = MyDataSet(images_path=train_images_path,
                              images_class=train_images_label,
                              transform=data_transform["train"])
```

这里我们定义了一个数据增强的transform，通过一系列操作将图像进行预处理，再将其传入自定义的MyDataSet类中，完成数据集的构建。无论是图像分类、目标检测还是语义分割等任务，数据集的构建过程都是类似的，只是具体的数据增强方法和数据集格式可能有所不同。有的代码是直接在train.py文件中直接定义并且使用DataLoader加载数据集的，也有的代码是将其归为一个单独的dataset.py文件中进行定义，在我们下面调用Dataset方法时候直接使用，这就是不同项目的代码风格不同而已，逻辑是一样的。

注意，我们自定义的MyDataSet类需要继承torch.utils.data.Dataset，并实现__len__和__getitem__方法，以能被DataLoader使用。这里getitem输出数据集中每一个数据的具体内容和它的下标，len输出数据集的长度（这里不理解的话可以参考前面给出的小土堆中的相关讲解）。这个类是随着每个项目的不同而不同的，我们需要根据具体的数据集来编写相应的Dataset类。我们这里给出一个简单的例子：

```
class MyDataSet(Dataset):
    def __init__(self, images_path: list, images_class: list, transform
                                        =None):
        self.images_path = images_path
        self.images_class = images_class
        self.transform = transform

    def __len__(self):
        return len(self.images_path)

    def __getitem__(self, item):
        img = Image.open(self.images_path[item])
        if img.mode != 'RGB':
            raise ValueError("image: {} isn't RGB mode.".format(self.
                                                images_path[item]))
        label = self.images_class[item]

        if self.transform is not None:
            img = self.transform(img)

        return img, label
```

这里定义了MyDataSet类，用于加载图像数据集。该类接受图像路径列表

和对应的标签列表，并在getitem方法中读取图像文件，应用预处理变换，并返回图像和标签。有的数据增强或预处理是写入到这个方法中的，具体可以参考下面讲述的视频动作识别部分的实现，他的VideoDataset类中就包含了crop方法来进行数据增强。接下来我们只需要使用DataLoader来加载数据集即可：

```
    train_loader = torch.utils.data.DataLoader(train_dataset,
                                               batch_size=batch_size,
                                               shuffle=True,
                                               pin_memory=True,
                                               num_workers=nw,
                               collate_fn=train_dataset.collate_fn)
```

这里我们使用DataLoader来加载训练数据集，设置了批量大小、是否打乱数据、使用的线程数等参数。collate_fn参数用于指定如何将多个样本合并成一个批次，这个函数可以根据具体需求进行自定义。此时，我们就已经完成了数据集的构建和加载，在后续的过程中我们就可以使用创建好的训练集、验证集、测试集对整个模型进行训练。

1.2.2 模型

构建网络模型是深度学习项目中的核心部分。从代码层面上讲，一般的项目会有network、model、models等文件或文件夹来存放模型相关的代码。我们需要根据具体的任务选择合适的模型架构，并在代码中实现该模型。那么我们给出一个最为简单的代码进行说明：

```
import torch
import torch.nn as nn

class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 784)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

```
model = SimpleNet()
```

这里我们只需要说两个地方：

1、继承nn.Module类：在我们定义神经网络模型时，我们需要创建一个类并继承nn.Module类，以便能够使用PyTorch提供的各种功能和方法。

2、定义前向传播方法：在模型类中，我们需要实现forward方法，定义数据在网络中的前向传播过程。这是模型的核心部分，决定了输入数据如何经过各个层次进行计算并输出结果。这个一定是需要根据项目自身根据自己的输出要求来定义的。

在我们实际的代码实现中，模型的定义可能会更加复杂，包含多个层次、不同类型的层（如卷积层、池化层、全连接层等）以及各种激活函数和正则化方法，有的项目代码甚至会给出多层嵌套，多个文件来回调用，虽然减小了代码维护的难度，但是给我们带来了很大理解难度。但是我们最终关心的往往是网络的中间层的形状和最终输出，我们可以通过自定义测试代码来确定：

```
if __name__ == "__main__":
    inputs = torch.rand(1, 3, 16, 112, 112)
    net = C3D(num_classes=101, pretrained=True)

    outputs = net.forward(inputs)
    print(outputs.size())
```

这里我们以后续说明的视频动作识别中的C3D模型为例，创建了一个随机输入张量，并通过模型的forward方法进行前向传播，最后打印输出的形状。通过这种方式，我们可以验证模型的定义是否正确，并了解各层之间的数据流动情况.

如果我们正处在模型的更改阶段，我们可以通过打印每一层的输出形状来帮助我们理解模型的结构和数据流动情况（当然最直接的方法是对代码进行调试，具体的方法后续有机会补充）。例如：

```
def forward(self, x):
    print("Input shape:", x.shape)
    x = self.conv1(x)
    print("After conv1:", x.shape)
    x = self.pool1(x)
    print("After pool1:", x.shape)
    return x
```

这里我们在forward方法中添加了打印语句，输出每一层操作后的张量形状。通过这种方式，我们可以清晰地看到数据在网络中的变化，帮助我们理解模型的结构和调试潜在的问题。当然，我们在运行代码的时候，如果我们因为加入了某些模块导致模型无法正常运行，会发生某两个向量形状不匹配的情况（有时候报错信息也会直接告诉我们两个向量形状是多少，方便我们定位），我们也可以通过这种打印形状的方式来定位问题所在。到此如果对向量的形状计算不理解的话，可以参考前面给出的小土堆中的相关讲解，也可以参考这个视频，了解具体特征图张量的计算方法。那么在我们进行模型修改的时候，我们就可以通过这种方式来帮助我们理解模型的结构和数据流动情况，从而更好地进行模型的修改和优化。

在我们定义好模型后，我们在实例化模型时，还需要考虑模型的初始化、加载预训练权重等问题，这些内容在不同的项目中可能会有所不同，我们需要根据具体的项目要求来进行相应的操作。这里我们给出一个示例：

```
def create_model(num_classes: int = 21843, has_logits: bool = True)
                                    :
model = VisionTransformer(img_size=224,
                          patch_size=16,
                          embed_dim=768,
                          depth=12,
                          num_heads=12,
                          representation_size=768 if has_logits
                                                  else
                                                  None,
                          num_classes=num_classes)
return model

model = create_model(num_classes=args.num_classes, has_logits=False
                                  ).to(device)
```

这里我们定义了一个create_model函数，用于创建Vision Transformer模型，并根据传入的参数设置模型的类别数和是否包含logits层。然后我们实例化模型并将其移动到指定的设备上（如GPU）。在实际的项目中，我们可能还需要加载预训练权重，这可以通过torch.load()函数来实现，具体的加载方式会根据预训练权重的格式和存储位置有所不同。这个实例就是稍微复杂的案例，将模型的创建封装在一个函数中，方便我们根据不同的需求来创建模型实例。我们需要注意的点就是：加载哪个定义好的模型、是否需要加载预训练权重、模型的输入输出设定等，一般这个会在readme中有说明，**如果没有相关说明，或者GitHub项目给有给出预训练权重下载地址，我们**

就需要果断放弃该项目，去找其他的项目。最后还需要注意的点是否所有参数都需要训练、是否需要冻结某些层，这些内容会根据具体的项目要求有所不同，我们需要根据实际情况来进行相应的操作。

1.2.3 训练

在我们完成数据集的构建和模型的定义后，接下来就是训练模型的过程了。首先，我们需要一些必要的定义，这些内容在不同的项目中可能会有所不同。这里我们给出一个简单的示例：

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(train_params, lr=lr, momentum=0.9,
                                    weight_decay=5e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10,
                                      gamma=0.1)
```

这里我们定义了损失函数、优化器和学习率调度器。损失函数用于衡量模型预测结果与真实标签之间的差距，优化器用于更新模型参数以最小化损失函数，学习率调度器用于调整学习率以提高训练效果。在这里我们需要注意的点是，optimizer可能有多个不同的优化器可供选择，例如Adam、RMSprop等，如果我们自行搭建了网络模型或者加入了自己的模块，大多数情况可以不改变原有代码的优化器，但是小部分情况我们需要根据具体的任务和模型选择合适的优化器，可以自行查阅不同的优化器之间的差别。另外，optimizer的参数同样需要注意，我们在这里给他传递我们需要训练的模型的参数，学习率等，一般会在训练模型的方法或者调用文件的传参对这些参数进行设置，可以参考项目中已给出的文件说明（readme等文件）。同样地，损失函数也有很多种类，例如均方误差、交叉熵等，我们需要根据具体的任务选择合适的损失函数。最后，学习率调度器也是一个重要的环节，可以帮助我们更好地控制学习率的变化，提高训练效果。不同的学习率调度器有不同的调整策略，我们需要根据具体的任务和模型选择合适的学习率调度器。这里可以自行查阅相关资料了解不同的学习率调度器之间的差别和使用方法。

完成了这些步骤，我们就要开始写循环进行网络的训练，下面我们给出最主要的的训练代码：

```
scheduler.step()
model.train()
optimizer.zero_grad()
```

```
    outputs = model(inputs)
    probs = nn.Softmax(dim=1)(outputs)
    preds = torch.max(probs, 1)[1]
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
```

这里我们首先调用scheduler.step()来更新学习率，然后将模型设置为训练模式。接着，我们清零优化器的梯度，进行前向传播计算模型输出，并通过损失函数计算损失值。然后，我们进行反向传播计算梯度，最后调用优化器的step()方法来更新模型参数。在实际的训练过程中，我们通常会将上述代码放在一个循环中，遍历整个训练数据集多次，以不断优化模型参数，提高模型性能。在实际使用过程中，无论是有的项目将训练过程写成类似train_one_epoch()的方法，还是直接写在train.py文件中进行训练，逻辑是一样的，我们都需要经历上述的过程。在具体的实现过程中，可能有一些细节需要注意，例如批量大小、迭代次数等，这些内容会根据具体的项目要求有所不同，我们需要根据实际情况来进行相应的操作。

1.2.4 可视化

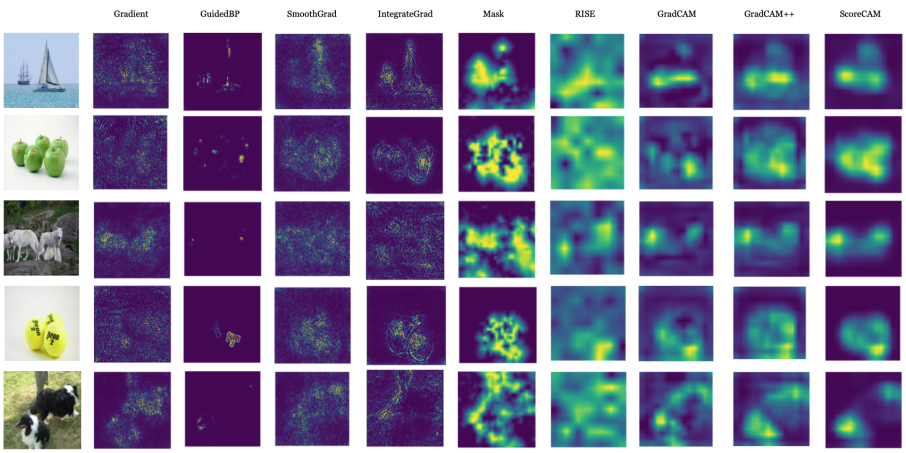


图 1: 不同可视化方法效果示例

可视化的部分在不同的项目中可能会有所不同，我们需要根据具体的项目要求来进行相应的操作。这里，我给出一个已经复现成功的可视化示例，访问这个链接下载GitHub项目，然后按照readme中的说明进行操作即

可。里面已经包含了几个较为简单的网络模型的可视化实现细节，大家可以参考学习。

我们这里的可视化一般用于对网络模型的注意力区域进行分析，帮助我们理解模型的决策过程。通过可视化，我们可以看到模型在处理输入数据时关注的区域，从而更好地理解模型的行为和性能。在实际的项目中，我们可能需要对模型的中间层的输出进行可视化，帮助我们理解模型的决策过程，如何实现可以参考每个可视化工具的具体实现和使用方法。有时我们需要根据具体的项目要求来选择合适的可视化方法和工具，并进行相关的操作。

1.2.5 其他

后续的操作，包括保存模型参数、评估模型性能等内容，在不同的项目中可能会有所不同，我们需要根据具体的项目要求来进行相应的操作。可以参考下文中提到的图像分类ViT的实现细节，这里的代码相对简洁利于理解，建议仔细研究。再往后，我们还需要对模型进行消融实验来验证我们添加的模块是否有效，这些内容会根据具体的项目要求有所不同，我们需要根据实际情况来进行相应的操作。

2 图像分类

在编辑该章节时，我们已经成功复现github项目中有关ViT的部分。

这里我们就不再过多讲述上面已经提到的各种实现细节，我们在这里讨论在实现过程中遇到的一些问题和解决思路。

2.1 添加新模块

在成功复现原有代码的基础之上，我们尝试对模型进行一些修改和扩展，例如添加新的模块或更改现有模块的结构。那么在此过程中，我们需要确认我们的添加的模块的输入输出形状是正确的，并且与模型的其他部分兼容。我们可以通过打印每一层的输出形状来帮助我们理解模型的结构和数据流动情况，具体方法前面已经讲过。

在这里，我们需要强调的是，尽管这个黑盒过程无法确定我们想要添加的模块是否正确，仅仅从输入输出形状上来看是无法保证模块的添加或者修改是绝对有效的，但是至少可以保证模型能够正常运行，不会因为形

状不匹配而报错，不报错是我们进行下一步调试和优化的前提条件。就像做生物化学实验一样，实验能够成功运行是我们进行下一步分析的前提条件，实验结果是否符合预期是另外一回事。当然，这并不是说我们完全无法解释模块的有效性，我们可以通过分析模块的功能和作用，结合具体的任务需求，来判断模块是否合理和有效。例如，如果我们添加的是一个卷积层，我们可以分析该卷积层的滤波器大小、步幅等参数，结合输入数据的特点，来判断该卷积层是否能够有效地提取特征。同样地，如果我们添加的是一个注意力机制模块，我们可以分析该模块的计算方式和作用，结合具体的任务需求，来判断该模块是否能够提高模型的性能。因此，在添加新模块时，我们不仅需要关注输入输出形状的匹配，还需要结合具体的任务需求和模块的功能，来判断模块的合理性和有效性。就像我在自己的实验中，添加了频域分析和增强机制，是具有很强的可解释性的，并且也有相关的文献支持其有效性。所以，在添加新模块时，我们需要综合考虑多个因素作为先验条件，来判断模块的合理性和有效性。我们这里给出一个示例：

```
def complexinit(weights_real, weights_imag, criterion):
    output_chs, input_chs, num_rows, num_cols = weights_real.shape
    fan_in = input_chs
    fan_out = output_chs

    if criterion == 'glorot':
        s = 1. / np.sqrt(fan_in + fan_out) / 4.
    elif criterion == 'he':
        s = 1. / np.sqrt(fan_in) / 4.
    else:
        raise ValueError('Invalid criterion: ' + criterion)

    rng = RandomState()
    kernel_shape = weights_real.shape

    modulus = rng.rayleigh(scale=s, size=kernel_shape)
    phase = rng.uniform(low=-np.pi, high=np.pi, size=kernel_shape)

    weight_real = modulus * np.cos(phase)
    weight_imag = modulus * np.sin(phase)

    weights_real.data = torch.Tensor(weight_real)
    weights_imag.data = torch.Tensor(weight_imag)

class FCB(nn.Module):
```

```
    def __init__(self, input_chs: int, num_rows: int, num_cols: int,
                                        stride=1, init='he'):
        super(FCB, self).__init__()
        self.weights_real = nn.Parameter(torch.Tensor(1, input_chs,
                                            num_rows, int(num_cols // 2
                                             + 1)))
        self.weights_imag = nn.Parameter(torch.Tensor(1, input_chs,
                                            num_rows, int(num_cols // 2
                                             + 1)))

        complexinit(self.weights_real, self.weights_imag, init)

        self.size = (num_rows, num_cols)
        self.stride = stride

    def forward(self, x):
        x = torch.fft.rfftn(x, dim=(-2, -1), norm=None)
        print(x.shape)  # torch.Size([1, 32, 50, 26])
        x_real, x_imag = x.real, x.imag

        # (a+bi)(c+di) = (ac-bd) + (ad+bc)i
        y_real = torch.mul(x_real, self.weights_real) - torch.mul(
                                            x_imag, self.weights_imag)
        y_imag = torch.mul(x_real, self.weights_imag) + torch.mul(
                                            x_imag, self.weights_real)

        x = torch.fft.irfftn(torch.complex(y_real, y_imag), s=self.size
                                            , dim=(-2, -1), norm=None)

        if self.stride == 2:
            x = x[..., ::2, ::2]

        return x

    def loadweight(self, ilayer):
        weight = ilayer.weight.detach().clone()
        fft_shape = self.weights_real.shape[-2]

        weight = torch.flip(weight, [-2, -1])

        pad = torch.nn.ConstantPad2d(
            padding=(0, fft_shape - weight.shape[-1], 0, fft_shape -
                                                weight.shape[-2]),
            value=0
        )
        weight = pad(weight)
```

```
        weight = torch.roll(weight, (-1, -1), dims=(-2, -1))

        weight_kc = torch.fft.fftn(weight, dim=(-2, -1), norm=None).
                                            transpose(0, 1)

        weight_kc = weight_kc[..., :weight_kc.shape[-1] // 2 + 1]

        self.weights_real.data = weight_kc.real
        self.weights_imag.data = weight_kc.imag

if __name__ == "__main__":
    x = torch.randn(1, 32, 50, 50)
    model = FCB(input_chs=32, num_rows=50, num_cols=50)
    output = model(x)
    print(f"x.shape")
    print(f"output.shape")
```

这里我们给出了一个即插即用的模块：给出一个输入向量，经过该模块他的输出形状和输入形状是完全相同的，我们可以将其简单理解为在频域进行特征增强。那我们可以根据之前的讲述的方法，打印原网络模型每一层的输出形状来帮助我们理解模型的结构和数据流动情况，在某一层中插入该模块，设置该FCB模块的输入输出形状与准备插入的层的输入输出形状一致，batchSize是不用动的，我们需要设置input_chs: int, num_rows: int, num_cols: int这三个参数，使得该模块的输入输出形状与插入层的输入输出形状一致。这样我们就完成了模块的添加。当然，模块的有效性还需要通过后续的训练和评估来验证，这里我们主要关注的是如何正确地添加模块并确保模型能够正常运行。当然，同样需要强调的是，添加模块时，我们不仅需要关注输入输出形状的匹配，还需要结合具体的任务需求和模块的功能，来判断模块的合理性和有效性，我们需要综合考虑多个因素作为先验条件，如果在插入之前我们甚至无法说服自己这样是合理的，我们就没有必要去插入这个无法解释的模块，更不用费劲去研究输入输出形状是否匹配的问题了。

3 视频动作识别

在编辑该章节时，我们已经成功复现github项目,成功实现了包括并不限于C3D模型在UCF-101和HMDB-51数据集上的动作识别任务。

这里我们就不再过多讲述上面已经提到的各种实现细节，我们在这里讨论在实现过程中遇到的一些问题和解决思路。

3.1 模型不收敛

在实际复现该项目的过程中，我们遇到了模型不收敛的问题。虽然他的训练能够成功运行，但是在其过程中，训练损失并没有明显下降，一直在4.7左右（这个数值是有原因的，经过查阅资料并从公式的推导结果来看，这个数值说明基本是完全没有训练效果的），验证准确率也没有提升。经过仔细分析代码和调试，我们将问题锁定在了学习率上，如果我们使用预训练模型，1e-5的学习率是可以的，但是如果我们不使用预训练模型，直接从头开始训练的话，这个学习率显然是过小的。经过多次尝试，我们将学习率调整为1e-3，模型终于开始收敛，训练损失逐渐下降，验证准确率也有所提升。在这个过程中，还调试了是否为初始化的问题、是否为数据预处理的问题、是否为优化器的问题，等，最终发现问题确实出在学习率上。通过这个过程，我们深刻体会到在复现他人项目时，学习率等超参数的选择是至关重要的，尤其是在使用预训练模型和从头开始训练时，学习率的设置可能会有很大差异。因此，在复现项目时，我们需要根据具体情况灵活调整超参数，以确保模型能够有效训练和收敛。当然，根据本人经验，如果代码能够正常运行和训练，但是损失和准确率一直没有变化的话，学习率应该是首先需要考虑的因素，其次是初始化，优化器等。

3.2 数据集处理

由于该任务是面向视频数据的，在处理数据集上遇到了不小的麻烦。首先，在实际获取数据集的过程中，UCF-101和HMDB-51数据集的下载和解压过程比较繁琐，需要按照指定的目录结构进行组织。并且，视频数据集往往都比较大，在实际使用中也会带来不小的麻烦。那么我们在选中要复现的项目之前，就要关注数据集是否合适。同样的，如果项目没有对数据、训练超参数等必要的元素进行详细的讲解，只给出了模型的代码（甚至没有模型代码或GitHub链接），不要犹豫直接放弃。视频的数据处理目前主流有两种方法：

1、将视频进行抽帧并保存在文件夹中，

2、保存视频并在生成Dataloader时再进行抽帧（其本质实际上是在列表中存放多个视频帧，大部分不会将整个视频全部用于训练，是一个[batch，

channel，time，h，w]的五维数组）。

这个github项目就是使用的第一种方法，大家可以自行阅读代码去学习是如何进行操作的。视频数据的预处理、增强一般需要考虑视频的帧率、分辨率、模型的输入等因素，这一点在代码中也有所体现。处理好了数据，后续的步骤就能继续开展。

3.3 数据集迁移

如果在之前的过程中，有同学已经完成了对这个github项目的复现，那么目前我们需要做的就是考虑数据集的迁移使用和模型的修改。同样的，我们需要注意的地方就是在生成训练、验证、测试数据集的时候，将我们已经实现了的项目的数据做些许修改（具体来说有数据的形状、数据的增强等），使得它们能在新模型中能够能进行正常的读取和运用，并且将新模型添加到这个已经实现的模型中。这里的实际操作比较有价值，可以带着大家具体看一下到底该怎么改、应该注意什么内容，详情请参考附录内容：“过程记录”。

3.4 修改模型

同样的，在复现了原有代码后，我们尝试对模型进行一些修改和扩展，例如添加新的模块或更改现有模块的结构。具体的修改方法和图像分类部分类似，这里不再赘述。需要注意的是，视频数据的输入通常是一个五维张量，我们在添加新模块时，需要确保模块能够正确处理这种输入格式。例如，如果我们添加的是一个卷积层，我们需要使用三维卷积（Conv3D）而不是二维卷积（Conv2D），以适应视频数据的时间维度。

3.5 Python的Registry机制

我们在学习别的GitHub项目代码的时候，有的代码会为了封装和其他需求，将代码写的一层嵌套一层导致我们的理解出现困难。这里我们介绍一种比较常见的有点难以理解的Python机制：装饰器。首先看下面这一段代码：

```
def a_decorator(func):

    def wrapTheFunction():
        print('Do some work before calling!')
```

```
        func()

        print('Do some work after calling')

    return wrapTheFunction

def math():
    print('I love math!')

math()
# outputs: I love math!

math = a_decorator(math)
math()
# output:
# Do some work before calling!
# I love math!
# Do some work after calling
```

在这个例子我们封装了一个函数，并且用另一个函数去修改这个函数的行为，这个功能其实就是Python装饰器所做的事情，它提供了更简洁的方式来实现同样的功能，装饰器的写法是在被装饰的函数前使用@+装饰器名。下面就引出一种更加简约的写法：

```
def a_decorator(func):

    def wrapTheFunction():
        print('Do some work before calling!')

        func()

        print('Do some work after calling')

    return wrapTheFunction

@a_decorator
def math():
    print('I love math!')

math()
# output:
# Do some work before calling!
# I love math!
# Do some work after calling
```

```
print(math.__name__)
# output: wrapTheFunction
```

不仅仅只有函数可以构建装饰器，类也可以用于构建装饰器，在构建装饰器类时，需要将原本装饰器函数的部分实现于__call__函数中即可:

```
from functools import wraps
class a_decorator(object):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        @wraps(self.func)
        def wrapTheFunction():
            print('Do some work before calling!')
            return self.func(*args, **kwargs)
        return wrapTheFunction

@a_decorator
def math(*args, **kwargs):
    return 'I love math!'

m = math()
print(m())
# output:
# Do some work before calling!
# I love math!
```

接下来，就是注册器Registry。Python的注册器本质上就是用装饰器的原理实现的。Registry提供了字符串到函数或类的映射，这个映射会被整合到一个字典中，开发者只要输入相应的字符串（为函数或类起的名字）和参数，就能获得一个函数或初始化好的类。下面请看：

```
register = {}

def func():
    pass

f = lambda x : x

class cls(object):
    pass

register[func.__name__] = func
register[f.__name__] = f
register[cls.__name__] = cls
```

```
print(register)
# output:
# {'func': <function func at 0x7fec95a1add0>, '<lambda>': <function <
                                    lambda> at 0x7fec95a27a70>, 'cls':
                                    <class '__main__.cls'>}
```

这里我们用一个字典存放字符串到函数的映射,但缺点是我们需要手动维护register这个字典，当增加新的函数或类，或者删除某些函数或类时，我们也要手动修改register这个字典，因此我们需要一个可以自动维护的字典，在我们定义一个函数或类的时候就自动把它整合到字典中。为了达到这一目的，这里就使用到了装饰器，在装饰器中将我们新定义的函数或类存放的字典中，这个过程我们称之为注册。我们可以定义一个装饰器类Register，其中核心部分就是成员函数register，它作为一个装饰器函数，快速注册函数：

```
class Register(dict):
    def __init__(self, *args, **kwargs):
        super(Register, self).__init__(*args, **kwargs)
        self._dict = {}

    def __call__(self, target):
        return self.register(target)

    def register(self, target):
        def add_item(key, value):
            if not callable(value):
                raise Exception(f"Error:{value} must be callable!")
            if key in self._dict:
                print(f" {value.__name__} already exists and will be
                                                    overwritten!")
            self[key] = value
            return value

        if callable(target):
            return add_item(target.__name__, target)
        else:
            return lambda x : add_item(target, x)

    def __setitem__(self, key, value):
        self._dict[key] = value

    def __getitem__(self, key):
        return self._dict[key]
```

```
    def __contains__(self, key):
        return key in self._dict

    def __str__(self):
        return str(self._dict)

    def keys(self):
        return self._dict.keys()

    def values(self):
        return self._dict.values()

    def items(self):
        return self._dict.items()


register_func = Register()

@register_func
def add(a, b):
    return a + b

@register_func
def multiply(a, b):
    return a * b

@register_func('matrix multiply')
def multiply(a, b):
    pass

@register_func
def minus(a, b):
    return a - b

for k, v in register_func.items():
    print(f"key: {k}, value: {v}")
# output:
# key: add, value: <function add at 0x7fdedd53cb90>
# key: multiply, value: <function multiply at 0x7fdedd540200>
# key: matrix multiply, value: <function multiply at 0x7fdedd5403b0>
# key: minus, value: <function minus at 0x7fdedd540320>
```

到这里，我们就可以试着去理解相关的模型注册的python写法了，请大家参考相关代码（或者直接阅读ViedoMamba的代码加深理解。）

4 实战目标

这里给大家提供一个可行的实战目标，供大家参考和实践。

首先，复现上面提到的图像分类的项目。为了确保可复现性，大家访问上面给出的github链接并且找到pytorch_classification vision_transformer,这里给出了ViT的代码（其他的网络模型大家可以在完成ViT模型的复现后根据自己的目标或者兴趣自行进行复现，由于我只复现了ViT模型，所以真敢保证这个模型的可行性，大家在判断其他模型是否可行时最简单的方法就是看流程是否完整，包括数据集、训练超参数等，另外还需观察是否给出模型预训练链接，都有的话是很有保障复现的）。

成功复现后再尝试对模型进行修改，可以使用上面2.1提到的模块，也可以自行添加其他的模块或网络层。观察添加前后的模型性能变化，提升或下降都无所谓，修改完模型能收敛就是成功。最后，可以尝试将修改后的模型应用到其他图像分类数据集上，例如CIFAR-10、CIFAR-100等，观察模型在不同数据集上的表现。

到此，我们就完成了一个完整的实战项目，从复现代码到修改模型再到应用新数据集，全面提升了对深度学习项目的理解和实践能力。大家可以根据自己的课题和自己的兴趣，自行选择其他的项目进行复现和实践，过程大同小异，关键是要理解每一步的细节和背后的原理。

如果在看了这一部分仍有困难，请参考附录中的“过程记录”进行实际操作学习。

5 总结

如果大家能成功复现代码，距离成功已经完成了65%的工作，那么剩下的35%才是：理解代码、分析结果、改进模型。在后续过程中，我会在网站上持续更新该文档，希望大家能有所收获。祝科研顺利!

A 过程记录

这里我们带来一个简单的过程记录，供大家参考，具体的训练代码我上传到了funcCodeExample文件夹中，可以边查看trainvimucf边学习，并且和原来的train.py文件进行比较，可以更好的理解代码的运行流程。

我们这里实现的是将ViedoMamba应用在这里的视频动作识别项目中，因为我们已经完成了对该项目的复现，所以我们仅需要修改模型部分的代码即可。请大家确保学习该部分时已经可以复现上述的视频动作识别项目。

A.1 数据处理

由于我们已经能跑通了这个视频动作识别项目，所以数据处理部分我们是已经实现了的。由于我们是调用VideoMamba的模型架构代码，所以我们需要确保数据处理部分和VideoMamba的要求是一致的。我们可以找到图2 的说明，我们点开script指向了一个微调的启动命令，是一个sh命令，我们可以找到这个文件中对数据处理的超参数input_size,这里就是用的224，当时回到我们的视频动作识别项目中，找到数据处理部分，我们发现它的数据大小并不是这个值，并且他还做了crop用于数据增强，那么我们现在就用两种处理思路：1、直接将数据处理部分改为VideoMamba的要求，这样就能保证数据处理的一致性，2、重新生成视频帧，这样就需要重新跑数据处理代码生成新的数据集。这里我选择了第一种方法，直接将数据处理部分改为VideoMamba的要求，这样省得再跑一遍数据生成（当然，这里是考虑到视频数据集比较大，如果新生成数据集比较简单，我建议重新生成数据集，这样可以减少对代码的改动）。通过阅读代码，我们将resize_height resize_width crop_size改成256、344、224（这样改的原理是由于我们最终需要进行图像增强，所以crop之后的图像大小一定是224×224，我们将原图像使用插值放大到256×344尺寸后增加数据集的鲁棒性），之后，我们将DataLoader的传参改成sh文件中的参数，这样就能保证数据处理的一贯性，到此，我们就完成了新数据集的构建（可以参考funcCodeExample文件夹中的datasetvimucf文件参考学习）。

这里的需要抓住的核心就是数据集的传输和模型想要的输入一致，找到相关代码确认即可。

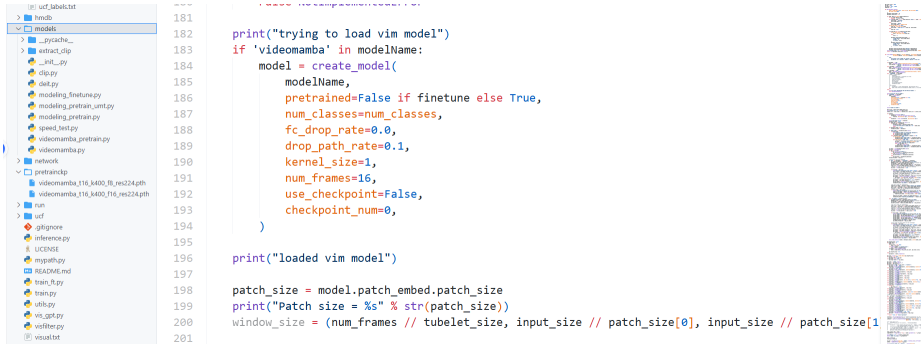
A.2 确认模型

- 🔥 **2024/03/12: All the code and models are released.**
  - image_sm: Single-modality Image Tasks
    - Image Classification: script & model
  - video_sm: Single-modality Video Tasks
    - Short-term Video Understanding: script & model
    - Long-term Video Understanding: script & model
    - Masked Modeling: script, model
  - video_mm: Multi-modality Video Tasks
    - Video-Text Retrieval: script & model

图 2: readme截图

图 3: 调用文件截图

第一步我们要做的就是确认哪些是VideoMamba的模型架构代码，并且将涉及到的所有文件都下载下来。首先我们需要看的就是readme文件中是否有明显的指向运行模型文件代码的说明：同样的，我们可以找到图2 的说明，我们点开script指向了一个微调的启动命令，里面的文件一定会包含一个调用的python文件，如图3所示，还有其他必要的超参数，包括视频帧数、数据集类别数、输入数据尺寸、学习率等，我们后续还需要使用这些参数



```
print("trying to load vim model")
if 'videomamba' in modelName:
    model = create_model(
        modelName,
        pretrained=False if finetune else True,
        num_classes=num_classes,
        fc_drop_rate=0.0,
        drop_path_rate=0.1,
        kernel_size=1,
        num_frames=16,
        use_checkpoint=False,
        checkpoint_num=0,
    )

print("loaded vim model")

patch_size = model.patch_embed.patch_size
print("Patch size = %s" % str(patch_size))
window_size = (num_frames // tubelet_size, input_size // patch_size[0], input_size // patch_size[1
```

图 4: 复制后截图

进行模型搭建。为了方便理解，我们尽量少的复制文件，将一些必要的文件和参数进行整合即可。我们进入这个run_class_finetuning.py文件，找到构建模型的部分（参考上面1.2的讲解），我们发现它使用了create_model()方法来创建模型，这里使用的是timm库中的方法，这里我们直接将models文件夹进行复制到我们需要加入的项目代码之中，如图4所示，然后同样使用create_model()方法即可。复制过去之后，我们需要考虑的事情就是超参数的设置，我们需要根据上面的sh文件中的参数进行设置，包括类别数、输入尺寸等，这里我使用了一个非常偷懒的方法，不建议学习，但是为了展示过程我还是给出，如图5所示：我们直接对照sh文件将里面的参数直接复制到我们需要调用的方法中即可，这样就避免了传参的问题。当然，这种偷懒的方法是**极其不利于维护和更新**的，强烈建议自行书写argparse进行传参，有的集成好的函数就在主程序中调用函数时进行方法传参即可。这里可以参考主页中关于这部分传参方法的使用。 完成了这些步骤，我们下一步需要完成的是模型的预训练模型的加载。同样的，这里我们需要去看sh文件和run_class_finetuning.py文件，从这两个文件找到线索。我们需要确定的是预训练模型的下载地址和加载方法。我们可以在readme找到预训练模型的下载地址，并且在run_class_finetuning.py文件中的337行开始，到503行，是加载预训练模型的代码，因为第503行出现了我们在1.2中提到的加载模型的代码load_state_dict（如果对这些命令不敏感的话，可以去funcCodeExample文件夹找到里面的trainvimucf代码对这些相关的方法函数进行学习），别看中间100多行代码，如果让我去讲解的话也道不清其中的过程，但我们只需要抓住重点实现代码，把中间这些一块复制过去就好。到此，我们就完成了模型的构建。

图 5: 复制后截图

A.3 训练设置

走到这一步，我们就要去查看原来VideoMamba中对优化器等的设置是怎么做的了。通过前面的学习，我们知道我们需要确定模型哪些参数是参加训练的。通过查阅代码，我们可以找到一个关键的参数n_parameters、optimizer，以此为线索我们发现它使用了create_optimizer建立的optimizer，同样的我们复制这个方法，并且进入这个方法，为了直观我们直接复制方法内容到我们的代码中（建议复制方法并且引用方法，有较高的代码阅读性和可维护性），根据sh文件中的超参数设置优化器，并将原文optimizer = optim.SGD备注即可。

A.4 训练

开始训练。