基于 CNN 的图像分类实验报告

实验目标：

1、 熟悉 pytorch 库，完成卷积神经网络相关知识和代码实现的学习

2、 完成入门教程的学习，使用 pytorch 进行图像分类等基本任务

实验目录：

1、 卷积神经网络（LeNet-5）的实现

2、宾夕法尼亚大学-复旦大学 R-CNN 行人检测模型和分段

3、计算机视觉迁移学习 – 蚂蚁和蜜蜂分类模型

实验内容 1 - 卷积神经网络（LeNet-5）的实现：

LeNet，它是最早发布的卷积神经网络之一，因其在计算机视觉任务中的高效性能而受到广泛关注，目的是识别图像中的手写数字。

当时 LeNet 取得了与支持向量机（support vector machines）性能相媲美的成果，成为监督学习的主流方法，并被广泛用于自动取款机（ATM）机中，帮助识别处理支票的数字。

总体来看，LeNet（LeNet-5）由两个部分组成：

- 卷积编码器：由两个卷积层组成；
- 全连接层密集块：由三个全连接层组成。

该架构如下图 1 所示。

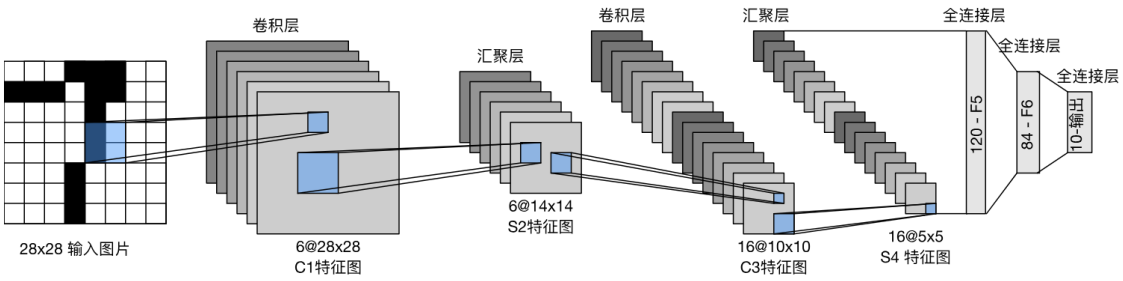

图 1 LeNet-5 组成架构

每个卷积块中的基本单元是一个卷积层、一个 sigmoid 激活函数和平均汇聚层。每个卷积层使用 5×5 卷积核和一个 sigmoid 激活函数。这些层将输入映射到多个二维特征输出，通常同时增加通道的数量。第一卷积层有 6 个输出通道（6 套卷积核），而第二个卷积层有 16 个输出通道（16 套卷积核）。每个 2×2 池操作（步幅 2）通过空间下采样将维数减少 4 倍。卷积的输出形状由批量大小、通道数、高度、宽度决定。

为了将卷积块的输出传递给稠密块（包含全连接层），我们必须在小批量中展平每个样本。换言之，我们将这个四维输入转换成全连接层所期望的二维输入。这里的二维表示的第一个维度索引小批量中的样本，第二个维度给出每个样本的平面向量表示。

LeNet 的稠密块有三个全连接层，分别有 120、84 和 10 个输出。因为我们在执行分类任务，所以输出层的 10 维对应于最后输出结果的数量。

实例化一个 Sequential 块并将需要的层连接在一起。

```
import torch
from torch import nn
from d2l import torch as d2l

net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.Linear(16 * 5 * 5, 120), nn.Sigmoid(),
    nn.Linear(120, 84), nn.Sigmoid(),
    nn.Linear(84, 10))

print("Done!")
```

```
Done!
```

图 2 实例化 Sequential 块

对原始模型做了一点小改动，去掉了最后一层的高斯激活。除此之外，这个网络与最初的 LeNet-5 一致。然后，我们将一个大小为 28×28 的单通道（黑白）图像通过 LeNet，通过在每一层打印输出的形状，我们可以检查模型，以确保其操作与我们期望的一致。

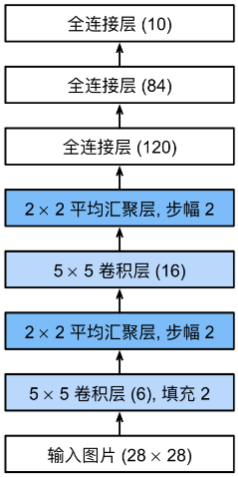

图 3 Lenet 简化版流程

对原始模型做了一点小改动，去掉了最后一层的高斯激活。除此之外，这个网络与最初的 LeNet-5 一致。然后，我们将一个大小为 28×28 的单通道（黑白）图像通过 LeNet，通过在每一层打印输出的形状，我们可以检查模型，以确保其操作与我们期望的一致。

请注意，在整个卷积块中，与上一层相比，每一层特征的高度和宽度都减小了。

第一个卷积层使用 2 个像素的填充，来补偿 5×5 卷积核导致的特征减少。相反，第二个卷积层没有填充，因此高度和宽度都减少了 4 个像素。随着层叠的上升，通道的数量从输入时的 1 个，增加到第一个卷积层之后的 6 个，再到第二个卷积层之后的 16 个。同时，每个汇聚层的高度和宽度都减半。最后，每个全连接层减少维数，最终输出一个维数与结果分类数相匹配的输出。

```
X = torch.rand(size=(1, 1, 28, 28), dtype=torch.float32)
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape: \t',X.shape)
```

```
Conv2d output shape:     torch.Size([1, 6, 28, 28])
Sigmoid output shape:    torch.Size([1, 6, 28, 28])
AvgPool2d output shape:          torch.Size([1, 6, 14, 14])
Conv2d output shape:     torch.Size([1, 16, 10, 10])
Sigmoid output shape:    torch.Size([1, 16, 10, 10])
AvgPool2d output shape:          torch.Size([1, 16, 5, 5])
Flatten output shape:    torch.Size([1, 400])
Linear output shape:     torch.Size([1, 120])
Sigmoid output shape:    torch.Size([1, 120])
Linear output shape:     torch.Size([1, 84])
Sigmoid output shape:    torch.Size([1, 84])
Linear output shape:     torch.Size([1, 10])
```

图 4 输出结构检查

现在我们已经实现了 LeNet，让我们看看 LeNet 在 Fashion-MNIST 数据集上的表现。为了进行评估，我们对 evaluate_accuracy 函数进行轻微的修改；由于完整数据集位于内存，我们在使用 GPU 计算数据集之前将其复制到显存中。

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

def evaluate_accuracy_gpu(net, data_iter, device=None): #@save
    """使用GPU计算模型在数据集上的精度"""
    if isinstance(net, nn.Module):
        net.eval()  # 设置为评估模式
        if not device:
            device = next(iter(net.parameters())).device
    # 正确预测的数量，总预测的数量
    metric = d2l.Accumulator(2)
    with torch.no_grad():
        for X, y in data_iter:
            if isinstance(X, list):
                # BERT微调所需的（之后将介绍）
                X = [x.to(device) for x in X]
            else:
                X = X.to(device)
            y = y.to(device)
            metric.add(d2l.accuracy(net(X), y), y.numel())
    return metric[0] / metric[1]
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Using downloaded and verified file: ../data\FashionMNIST\raw\train-images-idx3-ubyte.gz
Extracting ../data\FashionMNIST\raw\train-images-idx3-ubyte.gz to ../data\FashionMNIST\raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Using downloaded and verified file: ../data\FashionMNIST\raw\train-labels-idx1-ubyte.gz
Extracting ../data\FashionMNIST\raw\train-labels-idx1-ubyte.gz to ../data\FashionMNIST\raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Using downloaded and verified file: ../data\FashionMNIST\raw\t10k-images-idx3-ubyte.gz
Extracting ../data\FashionMNIST\raw\t10k-images-idx3-ubyte.gz to ../data\FashionMNIST\raw
```

图 5 函数修改与数据集复制

为了使用 GPU，我们在进行正向和反向传播之前，需要将每一小批量数据移动到我们指定的设备（例如 GPU）上。对于多层神经网络的实现我们主要使用高级 API 创建模型，并进行相应的优化。我们使用 Xavier 随机初始化模型参数（避免梯度消失），与全连接层一样使用交叉熵损失函数和 SGD 小批量随机梯度下降。

```python
#@save
def train_ch6(net, train_iter, test_iter, num_epochs, lr, device):
    """用GPU训练模型(在第六章定义)"""
    def init_weights(m):
        if type(m) == nn.Linear or type(m) == nn.Conv2d:
            nn.init.xavier_uniform_(m.weight)
    net.apply(init_weights)
    print('training on', device)
    net.to(device)
    optimizer = torch.optim.SGD(net.parameters(), lr=lr)
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        # 训练损失之和，训练准确率之和，样本数
        metric = d2l.Accumulator(3)
        net.train()
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            optimizer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            optimizer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (train_l, train_acc, None))
        test_acc = evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(device)}')
```

图 6 训练模型 GPU 版定义

现在，我们训练和评估 Lenet-5 模型（因为本机电脑兼容问题 d2l 与 GPU 配置不符合，暂时用 CPU 跑一下）。

```python
lr, num_epochs = 0.9, 10
train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.459, train acc 0.827, test acc 0.808
10750.5 examples/sec on cpu
```

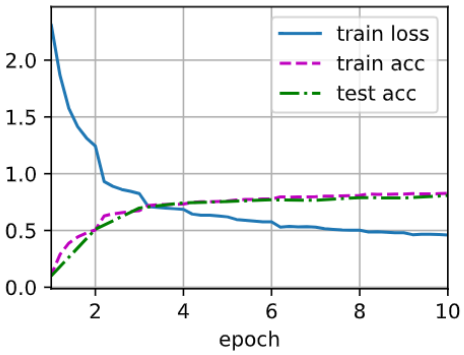

图 7 可视化训练评估

基于 LeNet 优化神经网络，提高准确率（其余代码保持不变，以用于训练和评估）：

1、替换激活函数：ReLU 通常能够提高收敛速度并减少梯度消失问题。
2、使用 MP 替换 AP：因为 MaxPooling 通常表现更好，能够捕捉更显著的特征。
3、增加 Dropout：在全连接层之间加入 Dropout，可以有效防止过拟合。

```
net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, padding=2),
    nn.ReLU(),  # 替换 Sigmoid 为 ReLU
    nn.MaxPool2d(kernel_size=2, stride=2),  # 替换 Average Pooling 为 Max Pooling
    nn.Conv2d(6, 16, kernel_size=5),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.Linear(16 * 5 * 5, 120),
    nn.ReLU(),
    nn.Dropout(0.5),  # 添加 Dropout 层
    nn.Linear(120, 84),
    nn.ReLU(),
    nn.Dropout(0.5),  # 添加 Dropout 层
    nn.Linear(84, 10)
)
```

图 8 修改部分代码

```
loss 0.447, train acc 0.839, test acc 0.839
6658.6 examples/sec on cpu
```

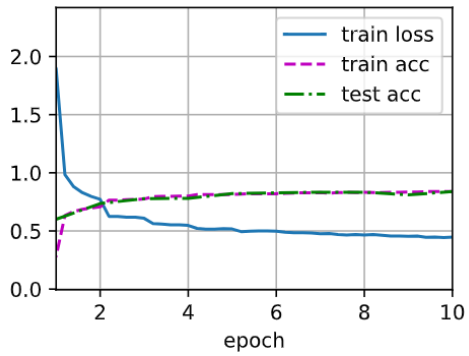

图 9 优化后可视化训练评估

实验内容 2 - 宾夕法尼亚大学-复旦大学 R-CNN 行人检测模型和分段：

在本教程中，我们将微调预训练的 Mask 宾夕法尼亚大学-复旦大学的 R-CNN 行人检测模型和分段。它包含 170 张图像，其中包含 345 个行人实例，我们将使用它来演示如何使用 TorchVision 中的新功能进行训练自定义数据集上的对象检测和实例分段模型。

首先下载数据集（PennFudan 数据集 -https://www.cis.upenn.edu/~jshi/ped_html/），然后解压缩 zip 文件至 'PennFudanPed' 目录

```
# 解压数据集
import zipfile

zip_file_path = 'PennFudanPed.zip'  # 输入 zip 文件名
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall('./PennFudanPed')  # 解压到 'PennFudanPed' 目录

print("Done!")
```

```
Done!
```

图 10 解压缩数据集文件

以下是一对图像和分割蒙版的一个示例，见图 11。

```
# 一对图像和分割蒙版的一个示例

import matplotlib.pyplot as plt
from torchvision.io import read_image

image = read_image("PennFudanPed/PennFudanPed/PNGImages/FudanPed00046.png")
mask = read_image("PennFudanPed/PennFudanPed/PedMasks/FudanPed00046_mask.png")

plt.figure(figsize=(16, 8))
plt.subplot(121)
plt.title("Image")
plt.imshow(image.permute(1, 2, 0))
plt.subplot(122)
plt.title("Mask")
plt.imshow(mask.permute(1, 2, 0))
```

```
<matplotlib.image.AxesImage at 0x2d4c0dbe1a0>
```

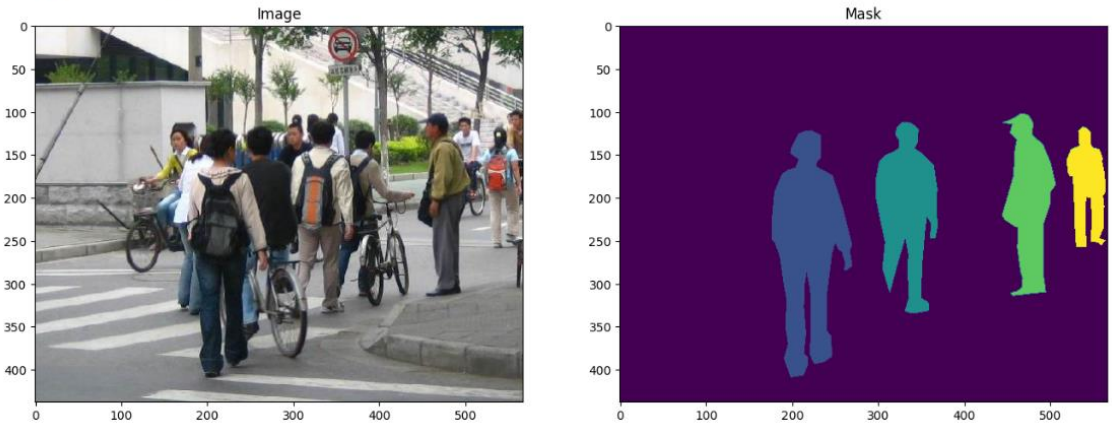

图 11 图像和分割蒙版示例

因此，每幅图像都有一个相应的分割掩码（Segmentation Mask），其中每种颜色对应一个不同的实例。让我们为这个数据集编写一个 torch.utils.data.Dataset 类。

1、图像张量将由 torchvision.tv_tensors.Image 封装；
2、边界框将由 torchvision.tv_tensors.BoundingBoxes 封装；
3、遮罩将由 torchvision.tv_tensors.Mask 封装。

在本教程中，我们将使用 Mask R-CNN，它基于 Faster R-CNN 之上。

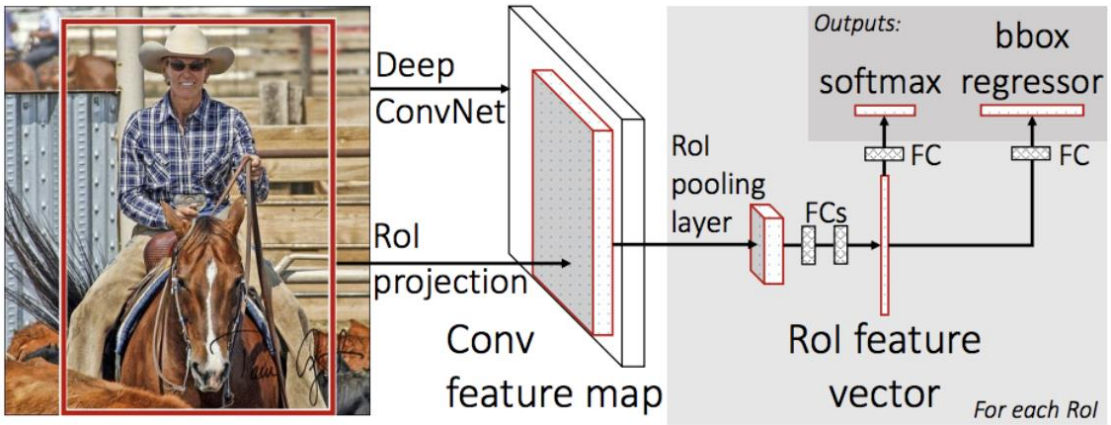

图 12 Faster R-CNN 示例

Mask R-CNN 添加了一个额外的分支 转换为 Faster R-CNN，它还预测每个实例。

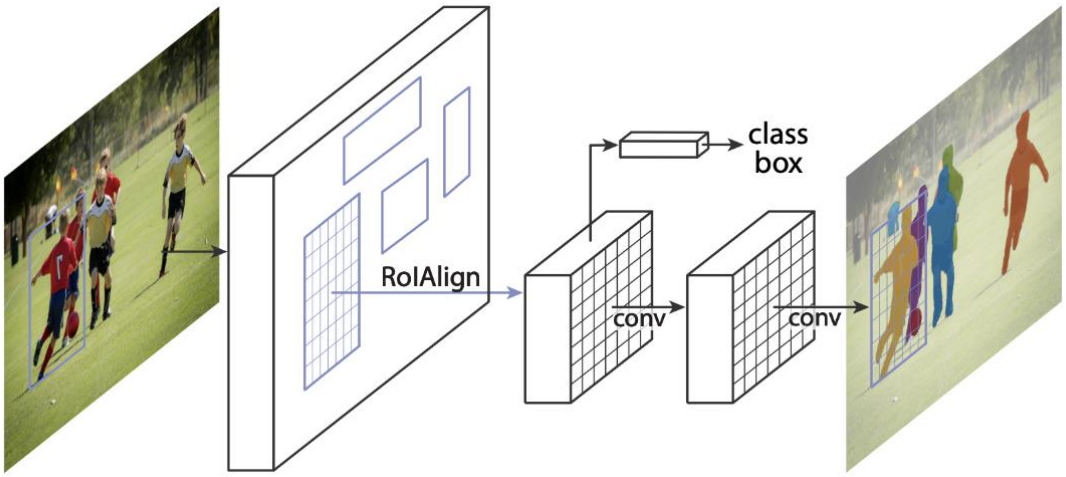

图 13 Mask R-CNN 示例

有两种常见的情况当人们可能想要修改 TorchVision Model Zoo 中的可用模型之一。第一种是当我们想要从预先训练的模型开始，然后微调最后一层。另一种是当我们想替换模型主干为其他模型（例如：为了更快地进行预测）。

假设您想从 COCO 上预训练的模型开始并希望针对您的特定类对其进行微调。这是一个可能的方法：

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor

# load a model pre-trained on COCO
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(weights="DEFAULT")

# replace the classifier with a new one, that has
# num_classes which is user-defined
num_classes = 2  # 1 class (person) + background
# get number of input features for the classifier
in_features = model.roi_heads.box_predictor.cls_score.in_features
# replace the pre-trained head with a new one
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
```

图 14 使用 COCO 预训练模型

修改模型以添加不同的主干（部分修改代码）。

```
import torchvision
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection.rpn import AnchorGenerator

# load a pre-trained model for classification and return
# only the features
backbone = torchvision.models.mobilenet_v2(weights="DEFAULT").features
# ``FasterRCNN`` needs to know the number of
# output channels in a backbone. For mobilenet_v2, it's 1280
# so we need to add it here
backbone.out_channels = 1280
```

图 15 添加不同的主干

然后我们进行 PennFudan 数据集的目标检测和实例分割模型，在我们的例子中，我们希望从预训练模型进行微调，因为我们的数据集非常小，因此我们将遵循第 1 种方法。

在这里，我们还想计算实例分段掩码，因此我们将使用 Mask R-CNN：

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor


def get_model_instance_segmentation(num_classes):
    # load an instance segmentation model pre-trained on COCO
    model = torchvision.models.detection.maskrcnn_resnet50_fpn(weights="DEFAULT")

    # get number of input features for the classifier
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    # replace the pre-trained head with a new one
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    # now get the number of input features for the mask classifier
    in_features_mask = model.roi_heads.mask_predictor.conv5_mask.in_channels
    hidden_layer = 256
    # and replace the mask predictor with a new one
    model.roi_heads.mask_predictor = MaskRCNNPredictor(
        in_features_mask,
        hidden_layer,
        num_classes
    )

    return model
```

图 16 使用 Mask R-CNN

在 references/detection/ 中，我们有许多辅助函数来简化检测模型的训练和评估。在这里，我们将使用 references/detection/engine.py 和 references/detection/utils.py 。只需将下的所有内容下载到您的文件夹中并在此处使用它们（我个人觉得使用 Python 的 request 库更具可移植性和跨平台性，因此将代码进行了适当修改）。

```
import requests

urls = [
    "https://raw.githubusercontent.com/pytorch/vision/main/references/detection/engine.py",
    "https://raw.githubusercontent.com/pytorch/vision/main/references/detection/utils.py",
    "https://raw.githubusercontent.com/pytorch/vision/main/references/detection/coco_utils.py",
    "https://raw.githubusercontent.com/pytorch/vision/main/references/detection/coco_eval.py",
    "https://raw.githubusercontent.com/pytorch/vision/main/references/detection/transforms.py"
]

for url in urls:
    response = requests.get(url)
    filename = url.split("/")[-1]  # 从URL中提取文件名
    with open(filename, "wb") as f:
        f.write(response.content)  # 写入文件

print("Done!")
```

```
Done!
```

图 17 下载并写入辅助函数

从 v0.15.0 开始，torchvision 提供了新的 Transforms API，可以轻松地为对象检测和分割任务编写数据增强管道。我们编写一些用于数据增强的辅助函/转型（get_transfrom）。

现在让我们编写执行训练的 main 函数和验证，并通过输出来展示模型训练效果：

[13]:
```
import torch
from engine import train_one_epoch, evaluate

# train on the GPU or on the CPU, if a GPU is not available
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

# our dataset has two classes only - background and person
num_classes = 2
# use our dataset and defined transformations
dataset = PennFudanDataset('PennFudanPed/PennFudanPed', get_transform(train=True))
dataset_test = PennFudanDataset('PennFudanPed/PennFudanPed', get_transform(train=False))

# split the dataset in train and test set
indices = torch.randperm(len(dataset)).tolist()
dataset = torch.utils.data.Subset(dataset, indices[:-50])
dataset_test = torch.utils.data.Subset(dataset_test, indices[-50:])

# define training and validation data loaders
data_loader = torch.utils.data.DataLoader(
    dataset,
    batch_size=2,
    shuffle=True,
    collate_fn=utils.collate_fn
)

data_loader_test = torch.utils.data.DataLoader(
    dataset_test,
    batch_size=1,
    shuffle=False,
    collate_fn=utils.collate_fn
)

# get the model using our helper function
model = get_model_instance_segmentation(num_classes)

# move model to the right device
model.to(device)

# construct an optimizer
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(
    params,
    lr=0.005,
    momentum=0.9,
    weight_decay=0.0005
)

# and a learning rate scheduler
lr_scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer,
    step_size=3,
    gamma=0.1
)

# let's train it just for 2 epochs
num_epochs = 2

for epoch in range(num_epochs):
    # train for one epoch, printing every 10 iterations
    train_one_epoch(model, optimizer, data_loader, device, epoch, print_freq=10)
    # update the learning rate
    lr_scheduler.step()
    # evaluate on the test dataset
    evaluate(model, data_loader_test, device=device)

print("That's it!")
```

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.816
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.650
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.738
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.825
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.745
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.987
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.922
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.217
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.523
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.759
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.336
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.787
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.787
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.550
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.750
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.794
That's it!
```

So after one epoch of training, we obtain a COCO-style mAP > 50, and a mask mAP of 65.

图 18 模型训练

因此，经过一个 epoch 的训练，我们获得了 COCO 风格的 mAP > 50，并且 Mask mAP 为 65。但预测是什么样的呢？让我们在应用到数据集并验证，使用 matplotlib.pyplot 可视化输出以便对验证结果有更好的理解。

```python
import matplotlib.pyplot as plt

from torchvision.utils import draw_bounding_boxes, draw_segmentation_masks


image = read_image("PennFudanPed/PennFudanPed/PNGImages/FudanPed00046.png")
eval_transform = get_transform(train=False)

model.eval()
with torch.no_grad():
    x = eval_transform(image)
    # convert RGBA -> RGB and move to device
    x = x[:3, ...].to(device)
    predictions = model([x, ])
    pred = predictions[0]


image = (255.0 * (image - image.min()) / (image.max() - image.min())).to(torch.uint8)
image = image[:3, ...]
pred_labels = [f"pedestrian: {score:.3f}" for label, score in zip(pred["labels"], pred["scores"])]
pred_boxes = pred["boxes"].long()
output_image = draw_bounding_boxes(image, pred_boxes, pred_labels, colors="red")

masks = (pred["masks"] > 0.7).squeeze(1)
output_image = draw_segmentation_masks(output_image, masks, alpha=0.5, colors="blue")


plt.figure(figsize=(12, 12))
plt.imshow(output_image.permute(1, 2, 0))
```

```
<matplotlib.image.AxesImage at 0x2d672f03610>
```

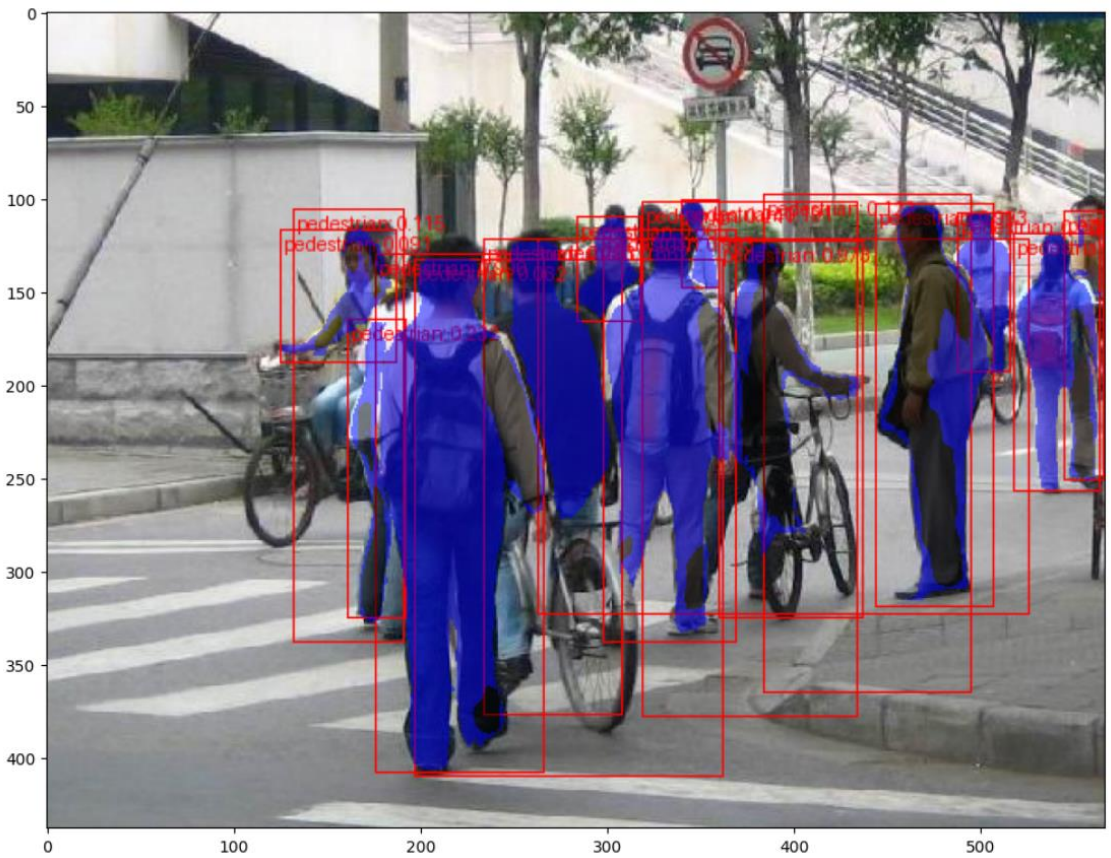

图 19 验证与结果可视化

这结果看上去还不错！

实验内容 3 - 计算机视觉迁移学习 - 蚂蚁和蜜蜂分类模型：

在本教程中，您将学习如何训练卷积神经网络并使用迁移学习进行图像分类。

在实践中，很少有人从头开始（随机初始化）训练整个卷积网络，因为它很少有足够大小的数据集。相反，它通常是在非常大的数据集（例如 ImageNet，其包含 120 万张图像和 1000 个类别），然后使用 ConvNet 作为初始化或固定特征提取器来完成感兴趣的任务。

这两个主要的迁移学习场景如下所示：

1、微调 ConvNet：不使用随机初始化，而是使用预先训练的网络来初始化，例如在 ImageNet 1000 数据集上训练好的网络。其余的训练看起来和正常训练一样。

2、ConvNet 作为固定特征提取器：在这里，我们将冻结除最终全连接的网络之外的所有网络层的权重。最后一个全连接层将替换为新层并赋予随机权重，并且仅训练此层。

此处要解决的问题是训练一个模型来对蚂蚁和蜜蜂进行分类。我们为蚂蚁和蜜蜂各准备了大约 120 张训练图像，每个类别有 75 张验证图像。通常情况下，如果从头开始训练，这个数据集的泛化范围非常小。但我们使用的是迁移学习，因此我们应该能够很好地进行泛化。

```
# 解压数据集
import zipfile

zip_file_path = 'hymenoptera_data.zip'  # 输入 zip 文件名
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall('./hymenoptera_data')  # 解压到 'hymenoptera_data' 目录

print("Done!")
```

```
Done!
```

```
# Data augmentation and normalization for training - 用于训练的数据增强和标准化
# Just normalization for validation - 仅对验证进行规范化

data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

data_dir = 'hymenoptera_data/hymenoptera_data'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=4,
                                              shuffle=True, num_workers=4)
               for x in ['train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

图 20 解压数据集并进行数据增强和标准化

让我们可视化一些训练图像，以便理解数据增强。

```
# 让我们可视化一些训练图像，以便理解数据增强

def imshow(inp, title=None):
    """Display image for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)  # pause a bit so that plots are updated


# Get a batch of training data
inputs, classes = next(iter(dataloaders['train']))

# Make a grid from batch
out = torchvision.utils.make_grid(inputs)

imshow(out, title=[class_names[x] for x in classes])
```

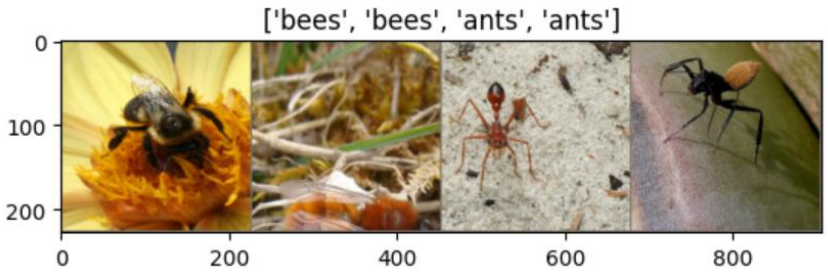

图 21 可视化训练图像

编写一个通用函数 train_model 来训练模型，再编写可视化模型预测的通用函数来显示一些图像的预测。然后加载预训练模型并重置最终的全连接层。

```
# 微调 ConvNet - 加载预训练模型并重置最终的全连接层。

model_ft = models.resnet18(weights='IMAGENET1K_V1')
num_ftrs = model_ft.fc.in_features
# Here the size of each output sample is set to 2.
# Alternatively, it can be generalized to ``nn.Linear(num_ftrs, len(class_names))``.
model_ft.fc = nn.Linear(num_ftrs, 2)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
```

```
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to C:\Users\WDMX/.cache\torch\hub\checkpoints\resnet18-f37072fd.pth
100.0%
```

图 22 加载预训练模型

训练与评估（内容输出太长，此处只截取第 1 轮和最后一轮）：

```
model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                       num_epochs=25)
```

```
Epoch 0/24
----------
train Loss: 0.7131 Acc: 0.6762
val Loss: 0.2375 Acc: 0.8954

Epoch 24/24
----------
train Loss: 0.3453 Acc: 0.8484
val Loss: 0.2098 Acc: 0.9150

Training complete in 3m 12s
Best val Acc: 0.921569
```

图 23 训练与评估

部分预测结果可视化：

```
visualize_model(model_ft)
```

图 24 预测结果可视化

接着我们完成以 ConvNet 作为固定特征提取器的情况。我们需要冻结除最后一层之外的所有网络。使用 requires_grad = False 以设置冻结参数，以便梯度不在 backward()中计算。

```
model_conv = torchvision.models.resnet18(weights='IMAGENET1K_V1')
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)
```

图 25 修改模型设置

训练与评估（内容输出太长，此处只截取最后几轮）：

```
[13]: model_conv = train_model(model_conv, criterion, optimizer_conv,
                               exp_lr_scheduler, num_epochs=25)
```

```
Epoch 22/24
----------
train Loss: 0.3498 Acc: 0.8279
val Loss: 0.1604 Acc: 0.9608

Epoch 23/24
----------
train Loss: 0.3608 Acc: 0.8566
val Loss: 0.1636 Acc: 0.9542

Epoch 24/24
----------
train Loss: 0.3504 Acc: 0.8566
val Loss: 0.1768 Acc: 0.9477

Training complete in 3m 1s
Best val Acc: 0.967320
```

图 26 训练与评估

部分预测结果可视化：

```
visualize_model(model_conv)

plt.ioff()
plt.show()
```

图 27 预测结果可视化

此时我们可以使用经过训练的模型对自定义图像进行预测并可视化预测的类标签以及图像。

```
def visualize_model_predictions(model,img_path):
    was_training = model.training
    model.eval()

    img = Image.open(img_path)
    img = data_transforms['val'](img)
    img = img.unsqueeze(0)
    img = img.to(device)

    with torch.no_grad():
        outputs = model(img)
        _, preds = torch.max(outputs, 1)

        ax = plt.subplot(2,2,1)
        ax.axis('off')
        ax.set_title(f'Predicted: {class_names[preds[0]]}')
        imshow(img.cpu().data[0])

        model.train(mode=was_training)
```

图 28 可视化模型预测设置

```
visualize_model_predictions(
    model_conv,
    img_path='hymenoptera_data/hymenoptera_data/val/bees/72100438_73de9f17af.jpg'
)

plt.ioff()
plt.show()
```

图 29 可视化模型预测