表情识别

1. 实验目的

1.1 了解表情识别数据集

1.2 掌握 ResNet18 网络的结构

1.3 完成表情识别数据集的分类任务

1.4 完成模型的剪枝操作

1.5 完成表情识别任务的部署并调用摄像头进行表情识别

2. 实验环境

SUB KIT NX 嵌入式开发板。

3. 准备工作

3.1 文件结构

在项目文件中创建 dataset.py 文件，用于构建小批量训练集与测试集；

创建 model.py 文件，用于构建训练模型；

创建 utils.py 文件，用于存放训练用的各种脚本以及函数；

创建 main.py 文件，用于进行模型的训练与测试；

创建 logs 文件夹，用于保存 tensorboard 训练日志；

创建 config.json 文件，用于保存训练用的超参数；

创建 save_path 文件夹，用于保存训练模型参数。

3.2 构建小批量训练集和测试集

本实验所用的表情识别数据集是从表情识别公开数据集 RAF-DB 中抽取部分数据所构成的，共有 7 种表情类别，分别是 'Anger'、Disgust'、'Fear'、'Happiness'、'Neutral'、 'Sadness'、'Surprise' 每个类别的图像数据分布与真实的 RAF-DB 数据集类似。

本实验所用的表情识别数据集共有 3034 张训练图像和 779 张测试图像。为了构建小批量训练集和测试集，首先需要在 dataset.py 文件中撰写得到数据图像位置与标签的函数。

首先导入本文件所需要的包：

```
from PIL import Image
import torch
from torch.utils.data import Dataset
import matplotlib.pyplot as plt
import os
import json
import random
```

本数据集已经按照类别进行分类，撰写数据读取代码。

#参考实验三完成数据集（训练集、测试集）构建

3.3 模型文件撰写

本实验所用的模型是 Resnet18，在构建模型时，需要将模型的预训练文件放入项目内。在 model.py 文件下撰写代码，首先该文件需要导入的包如下：

```
import torch
import torch.nn as nn
import torchvision.models as models
```

构建模型，撰写如下代码，

```
class my_model(nn.Module):
    def __init__(self):
        super(my_model, self).__init__()

            # 导入 resnet18，并仅去掉全连接
            # 定义自己的全连接层，最后的输出需要与数据集类别数对应

    def forward(self, x):
        return self.get_output(self.model(x))
```

3.4 超参数文件的编写

为了让参数修改起来方便简单，将本项目中所有使用的超参数保存在一起，

方便查看与修改。在 config.json 文件中撰写所有的超参数。

```
{
    "name": "fer_new",
    "version": "1.0.0",
    "dependencies": {
    },
    "model_config": {
        "train_data_root": "/minirafdb/train",
        "test_data_root": "/minirafdb/val",
        "val_rate": 0.0,
        "num_class": 7,
        "device": "cuda:0",
        "seed": 0,
        "save_path": "save_path/",
        "batch_size": 8,
        "epochs": 60,
        "img_size": 224,
        "learning_rate": 0.000035,
        "weight_decay": 1e-4,
        "optimizer": "Adam"

    }
}
```

其中，数据集的地址需要依照具体情况修改。

3.5 脚本与函数文件撰写

接下来撰写 utils.py 文件下的代码，该文件需要导入以下包。

```
    import json
    import torch
    from tqdm import tqdm
```

```
import sys
import torch.nn as nn
```

首先撰写解析 config.json 文件的函数：

```
def read_config():
    with open("config.json") as json_file:
        config = json.load(json_file)
    return config['model_config']
#为了保证实验的可复现行，需要设置随机种子，撰写设置随机种子的函数。
def set_seed(seed=0):

    np.random.seed(seed)
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)

    # 如果用显卡运行，以下两个选项进行设置
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = True

    # 设置系统环境随机种子
    os.environ['PYTHONHASHSEED'] = str(seed)
```

然后撰写训练脚本 trainer，撰写如下代码：

```
def trainer(model, optimizer, data_loader, config, epoch):
    """
    定义训练脚本
    :param model: 训练模型
    :param optimizer: 优化器
    :param data_loader: 数据集
```

```
:param config:  超参数
:param epoch:  训练当前代数
:return:
"""
device = config['device']

model.train()

accu_loss = torch.zeros(1).to(device)
accu_num = torch.zeros(1).to(device)

optimizer.zero_grad()

sample_num = 0
data_loader = tqdm(data_loader, file=sys.stdout)
for step, data in enumerate(data_loader):
    images, labels = data
    images, labels = images.to(device), labels.to(device)

    output = model(images)
    sample_num += images.shape[0]
    criterion = nn.CrossEntropyLoss()                       # 定义损失函数
    loss = criterion(output, labels)

    pred_classes = torch.max(output, dim=1)[1]
    accu_num += torch.eq(pred_classes, labels).sum()
    loss.backward()
    accu_loss += loss.detach()
    data_loader.desc = "[train epoch {}] loss: {:.3f}, acc:{:.3f}".format(epoch,
```

```
accu_loss.item() / (step + 1),

accu_num.item() / (sample_num))

            if not torch.isfinite(loss):
                print('WARNING: non-finite loss, ending training ', loss)
                sys.exit(1)
            optimizer.step()
            optimizer.zero_grad()

        return accu_loss.item() / (step + 1), accu_num.item() / sample_num
```

#接着撰写测试脚本 evaluater，与训练脚本类似，这里请自己尝试撰写。

3.6 模型训练函数的撰写

最后，在 main.py 文件中撰写训练函数，首先需要导入：**（建议网络训练在计算机上进行）**

```
import torch
import utils
import numpy as np
import random
import os
from torch.utils.tensorboard import SummaryWriter
from dataset import MyDataSet, read_split_data
import time
from torchvision import transforms
from model import my_model
```

#解析撰写的 config.json 文件：

```
config = utils.read_config()
#设置随机种子
utils.set_seed(config[‘seed’])
#以当前时间戳的形式保存 tensorboard 日志文件
t = time.localtime()
log_path = 'logs/' + str(t.tm_year) + '_' + str(t.tm_mon) + '_' + str(t.tm_mday) + '_' +
str(t.tm_hour) + '_' + str(t.tm_min) + '_' + str(t.tm_sec)
os.makedirs(log_path)
tb_writer = SummaryWriter(log_dir=log_path)
#读取训练集中所有图像的地址：
train_images_path, train_images_label, val_images_path, val_images_label =
read_split_data(root=config['train_data_root'], val_rate=config['val_rate']) # 如
果需要设置验证集请将 config 中 val_rate 设置成大于 0；
```

#读取测试集中所有图像地址的方法与上述相同，请自己撰写。

```
#设置批次大小与工作核心数：
batch_size = config['batch_size']
nw = min([os.cpu_count(), batch_size if batch_size > 1 else 0, 8])
数据预处理：
data_transform = {
        "train":        transforms.Compose([transforms.Resize([config['img_size'],
config['img_size']]),
                        transforms.ToTensor()]),

        "val":          transforms.Compose([transforms.Resize([config['img_size'],
config['img_size']]),
                        transforms.ToTensor()])}
构建训练数据集：
train_dataset = MyDataSet(images_path=train_images_path,
```

```
                                images_class=train_images_label,
                                transform=data_transform["train"])
```

构建训练 dataloader：

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

#测试的数据集构建与 dataloader 构建与之类似，请自己撰写。

#如果有验证集，也请自己撰写。

```
#实例化自己的训练模型：
train_model = my_model()
train_model = train_model.cuda()        # 将训练模型设置在显卡上
```

设置优化器：

```
if config['optimizer'] == 'Adam':
        optimizer = torch.optim.Adam(train_model.parameters(),
                                     lr=config['learning_rate'],
                                    betas=(0.9, 0.999),
                                    eps=1e-08,
                                    weight_decay=0,
                                    amsgrad=False)
elif config['optimizer'] == 'SGD':
        optimizer = torch.optim.SGD(train_model.parameters(),
                                    lr=config['learning_rate'],
                                    momentum=0.9,
                                    dampening=0,
```

```
                                        weight_decay=0,
                                        nesterov=False)
else:
        raise ValueError("Optimizer must be Adam or SGD, got {}".format(config['optimizer']))
#撰写学习率衰减：
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
                                                       T_max=config['epochs'],
                                                       eta_min=0)
#调用训练脚本进行训练：
    for epoch in range(config['epochs']):
        train_loss, train_acc = utils.trainer(train_model,
                                              optimizer,
                                              train_loader,
                                              config,
                                              epoch)
```

#调用测试脚本进行测试的代码请自己撰写。

#如果有验证集也请自己撰写测试验证集的代码。

```
#记录每一代的训练日志，并保存在 tensorboard 文件内：
tags = ['train_loss', 'train_acc', 'test_loss', 'test_acc', 'learning_rate']
tb_writer.add_scalar(tags[0], train_loss, epoch)
tb_writer.add_scalar(tags[1], train_acc, epoch)
tb_writer.add_scalar(tags[2], test_loss, epoch)
tb_writer.add_scalar(tags[3], test_acc, epoch)
tb_writer.add_scalar(tags[4], optimizer.param_groups[0]['lr'], epoch)
#并在每一代训练完成后执行一次学习率衰减：
scheduler.step()
```

#最后保存最后一次训练得到的模型参数：

```
torch.save({'model_state_dict': train_model.state_dict()},
           config['save_path'] + "model-{}-{}-last.pth".format(epoch, test_acc))
```

4. 模型剪枝操作

为了在不损失较高精度的前提下，减少模型的参数量与运算量，需要对模型进行剪枝操作。该操作需要导入的包如下：

```
Import torch_pruning as tp
```

首先需要定义一个评估指标，输出一个一维的重要性得分向量，来评估每个通道的重要性，在 utils.py 文件中撰写：

```
class MySlimmingImportance(tp.importance.Importance):
    def __call__(self, group, **kwargs):
        # 1. 首先定义一个列表用于存储分组内每一层的重要性
        group_imp = []    # (num_bns, num_channels)
        # 2. 迭代分组内的各个层，对 BN 层计算重要性
        for dep, idxs in group:   # idxs 是一个包含所有可剪枝索引的列表，用于处理 DenseNet 中的局部耦合的情况
            layer = dep.target.module   # 获取 nn.Module
            prune_fn = dep.handler   # 获取 剪枝函数
            # 3. 对每个 BN 层计算重要性
            if isinstance(layer, (nn.BatchNorm1d, nn.BatchNorm2d, nn.BatchNorm3d)) and layer.affine:
                local_imp = torch.abs(layer.weight.data)   # 计算 scale 参数的绝对值大小
                group_imp.append(local_imp)   # 将其保存在列表中
        if len(group_imp) == 0: return None   # 跳过不包含 BN 层的分组
        # 4. 按通道计算平均重要性
        group_imp = torch.stack(group_imp, dim=0).mean(dim=0)
        return group_imp
```

接下来对 BN 层进行稀疏训练，在 utils.py 中撰写：

```
class MySlimmingPruner(tp.pruner.MetaPruner):
    def regularize(self, model, reg):  # 输入参数一般是模型和正则项的权重，这里可以任意修改
        for m in model.modules():  # 遍历所有层
            if isinstance(m, (nn.BatchNorm1d, nn.BatchNorm2d, nn.BatchNorm3d)) and m.affine == True:
                m.weight.grad.data.add_(reg * torch.sign(m.weight.data))
```

对所有 BN 逐层更新稀疏训练梯度，在 main.py 文件中模型实例化后，遍历模型，找到模型的线性层和最后的输出层，将其在剪枝中忽略。

```
ignored_layers = []
for m in train_model.modules():
    if isinstance(m, torch.nn.Linear) and m.out_features == 7:
        ignored_layers.append(m)
#定义一个测试输入：
example_inputs = torch.randn(1, 3, 224, 224).cuda()
#使用上述定义的重要性评估：
imp = utils.MySlimmingImportance()
#依据训练代数，设置迭代次数：
iterative_steps = config['epochs']
#初始化剪枝器：
pruner = tp.pruner.MetaPruner(
    train_model,
    example_inputs,
    importance=imp,
    iterative_steps=iterative_steps,
    ch_sparsity=0.5,        # 目标稀疏性
    ignored_layers=ignored_layers,
)
#统计原始网络的参数量与计算量：
base_macs, bnase_nparams = tp.utils.count_ops_and_params(train_model,
```

example_inputs)

#在训练中，执行模型剪枝操作，并统计剪枝后网络的参数量与计算量。

pruner.step()

macs, nparams = tp.utils.count_ops_and_params(train_model, example_inputs)

print("Params: {:.2f} M => {:.2f} M".format(bnase_nparams / 1e6, nparams / 1e6))

print("MACs: {:.2f} G => {:.2f} G".format(base_macs / 1e9, macs / 1e9))

"""

#剪枝后再训练模型

"""

最后比较剪枝前后网络的参数量与计算量。

5. 表情识别任务部署

在计算机上完成模型的训练，并保存网络的参数文件。之后在嵌入式开发板上进行模型的部署，完成模型的测试，并调用摄像头完成实时表情识别。

撰写如下代码

```
from model import my_model
import torch
import utils
import cv2
import numpy as np
from torchvision import transforms
from PIL import Image

config = utils.read_config()

#  实例化自己的网络并载入训练参数

data_trans = transforms.Compose([transforms.Resize([224, 224]),
                                 transforms.ToTensor()])
```

```
label_name = ['Surprise', 'Fear', 'Disgust', 'Happiness', 'Sadness', 'Anger', 'Neutral']


device = config['device']
cap = cv2.VideoCapture(0)
face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
model.eval()
while True:
    ret, frame = cap.read()
    if ret:
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        faces_rects = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5)
        for (x, y, w, h) in faces_rects:
            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
            face_img = gray[y: y + h, x: x + w]
            face_img = np.tile(face_img, 3).reshape((w, h, 3))
            face_img = Image.fromarray(face_img)
            face_img = data_trans(face_img)
            face_img = face_img.unsqueeze(0)
            face_img = face_img.to(device)
            label_pd = model(face_img)
            predict_np = np.argmax(label_pd.cpu().detach().numpy(), axis=1)
            fer_text = label_name[predict_np[0]]
            font = cv2.FONT_HERSHEY_SIMPLEX
            pos = (x, y)
            font_size = 1.5
            color = (0, 0, 255)
            thickness = 2
            cv2.putText(frame, fer_text, pos, font, font_size, color, thickness, cv2.LINE_AA)
```

```
            cv2.imshow('Face Detection', frame)
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
        else:
            break
# 使用 release()方法释放摄像头，并使用 destroyAllWindows()方法关闭所有窗口
cap.release()
cv2.destroyAllWindows()
```