人工神经网络 Mid-term Homework: Facial Expression Recognition (FER)

21307174 刘俊杰

May 2024

1 实验内容

1.1 任务定义及数据集

任务定义：对于给定的人脸图片，输出其表情标签（图像分类任务）；

- 数据集下载地址:https://pan.baidu.com/s/1xbCuLRwLY5GEIRHPUAhSgg?pwd=d69e

- 一共包含五种表情：Angry，Happy，Neutral，Sad，Surprise，已经进行了训练集和测试集的划分，数据规模如下（数据集中存在一定的不平衡）；

情绪标签	训练集样本数	测试集样本数
Angry	500	100
Happy	1500	100
Neutral	1000	100
Sad	1500	100
Surprise	500	100

1.2 作业要求

1.2.1 python 环境

深度学习框架使用 pytorch;

请提交 requirement.txt, 包含必要的第三方包即可;

1.2.2 模型的构建和训练

自行设计卷积神经网络对人脸特征进行抽取，并通过全连接层进行分类;

不允许加载现成的预训练模型或图像分类包;

可以参考经典 CNN 结构 (AlexNet、VGG、Resnet 等);

可以自行探索网络结构对性能的影响;

1.2.3 模型测试

使用已经划分的测试集对训练好的模型进行测试，计算准确率，和每个类别的召回率、精准率，Macro-F1 等;

请勿使用测试集进行训练（作弊);

1.3 提交内容

1.3.1 代码 +requirement.txt

1.3.2 README：使用指南（如何进行训练、评测）

1.3.3 文档：对你所实现的内容进行详细阐述；包括但不限于：

数据预处理的操作

超参数的设置

模型的架构设计

验证方法（划分验证集进行验证）

结果分析

自己的探索和心得体会；

1.4 帮助

不熟悉 pytorch 的同学可以参阅官方教程 https://pytorch.org/tutorials/

环境安装 https://pytorch.org/get-started/locally/

数据集定义 https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

模型各层的编写参考 https://pytorch.org/docs/stable/nn.html

2 实验设计

该模型参考了 VGG 网络来实现，采用 3*3 较小的卷积核既可以保证感受视野，又能够减少卷积层的参数；kernel size 均为 22，stride 为 2 的 max-pooling，小的池化核能够带来更细节的信息捕获（当时也有 average pooling，但是在图像任务上 max-pooling 的效果更好，max 更加容易捕捉图像上的变化，带来更大的局部信息差异性，更好的描述边缘纹理等，用 average-pooling 可能会使得图像模糊了，类似与数字图像处理的高斯模糊）；但本任务没有 ImageNet 任务复杂，故采用了不深的网络模型。

2.1 数据预处理的操作

首先对数据进行预处理：

```
# 数据预处理
transform = transforms.Compose([
    transforms.Resize((48, 48)), # 调整图像大小为 48x48
    transforms.ToTensor(), # 转为张量
    transforms.Normalize(mean=[0.5], std=[0.5]) # 标准化
])
```

调用 transforms 库，首先为了保持输入数据的统一尺寸将图像大小调整为统一的 48 * 48 的大小格式（VGG 将图片的输入大小统一为 224*224，但为了缩短训练时间，统一成了更小的尺寸）；再将图像数据转换为张量格式；最后调用 Normalize() 将数据的分布调整为均值为 0，标准差为 1 的标准正态分布，这有助于加速模型的收敛过程并提高模型的稳定性，避免梯度爆炸或梯度消失的问题。

2.2 超参数的设置

2.2.1 损失函数

因试验任务是多分类问题，故损失函数选择使用交叉熵损失：

```
criterion = nn.CrossEntropyLoss()
```

2.2.2 优化器和学习率

使用 Adam 优化器进行模型参数的优化，学习率设置为 0.001：

```
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

并且使用了动量学习率，利用了之前的更新步骤的动量，从而在更新参数时增加了惯性，会根据当前的梯度方向以及之前的动量值来调整参数的更新方向和速度，从而加速收敛过程并避免局部最值。

2.2.3 迭代次数

迭代次数上限：50 轮

```
def train_model(model, train_loader, val_loader, criterion, optimizer, num_epochs=50,
output_file='training_log.txt', checkpoint_path='checkpoint.pth')
```

2.3 模型的架构设计

2.3.1 卷积层

设计的模型架构有 4 个卷积层：

```
class FER_CNN(nn.Module):
    def __init__(self):
        super(FER_CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, 3, 1, 1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(64, 128, 3, 1, 1),
            nn.BatchNorm2d(num_features=128),
           nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(128, 256, 3, 1, 1),
            nn.BatchNorm2d(num_features=256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(256, 512, 3, 1, 1),
            nn.BatchNorm2d(num_features=512),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

每个卷积层都包含一个卷积操作、批归一化 (Batch Normalization)、ReLU 激活函数和池化操作，通过卷积层 self.conv1，self.conv2，self.conv3 和 self.conv4 特征图的尺寸随着池化操作而不断减小。

2.3.2 卷积层到全连接层

最后的全连接层 self.fc 接受的输入是经过卷积层处理后展平的特征向量：

```
def forward( self , x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = x.view(x.shape[0], -1)
        y = self.fc(x)
        return y
```

2.3.3 全连接层

最后卷积的结果展平后输入到全连接层中：

```
   self.fc = nn.Sequential(
            nn.Dropout(p=0.2),
            nn.Linear(512*3*3, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 256),
           nn.ReLU(inplace=True),
            nn.Linear(256, 5),

        )
```

在全连接层中，使用了两个 Dropout 层，丢弃 20% 的神经元，以减少过拟合。然后，通过三个线性层和 ReLU 激活函数逐步减少特征的维度，直到最后输出 5 个类别的结果。最后一个线性层不使用 Softmax 激活函数的原因（在 section 4.1 中）是因为交叉熵损失函数中加上了。

2.4 验证方法（划分验证集进行验证）

因为在实验中发现在训练集上的准确率可以达到 90% 以上，但测试集上的准确率只有 50% 多，故模型发生了过拟合，故决定使用验证集验证，防止训练轮次太多导致过拟合。

2.4.1 验证集的划分

将 train 数据进行进一步的划分，划分为 20% 的验证集和 80% 的训练集，通过随机划分训练数据集中的样本为训练集和验证集，并创建相应的数据加载器：

```
# 定义验证集比例
    dataset_size = len(train_dataset)
    indices = list(range(dataset_size))
    np.random.shuffle(indices)
    split = int(np.floor(val_split * dataset_size))
    train_indices, val_indices = indices[split:], indices[:split]

    # 定义训练集和验证集的采样器
    train_sampler = SubsetRandomSampler(train_indices)
    val_sampler = SubsetRandomSampler(val_indices)

    # 加载训练集和验证集的数据加载器
    train_loader = DataLoader(train_dataset, batch_size=batch_size, sampler=train_sampler)
    val_loader = DataLoader(train_dataset, batch_size=batch_size, sampler=val_sampler)
```

首先，根据验证集所占比例，并根据此比例将样本的索引随机打乱。然后，根据划分后的索引列表，使用 SubsetRandomSampler 创建了训练集和验证集的采样器。最后，利用 PyTorch 的 DataLoader 加载器，根据采样器和指定的批次大小，创建了用于训练和验证的数据加载器。

2.4.2 验证集避免过拟合

为了避免训练轮次过多而导致的过拟合，设计验证集验证来避免，每一轮训练同时在验证集上次测试，如果连续 3 次在验证集上的准确率下降，

那么说明模型训练有可能发生了过拟合：

```
model.eval()
            val_loss = 0.0
            val_correct = 0
            val_total = 0

            with torch.no_grad():
                for images, labels in tqdm(val_loader):
                    outputs = model(images)
                    loss = criterion(outputs, labels)
                    val_loss += loss.item()
                    _, predicted = torch.max(outputs.data, 1)
                    val_total += labels.size(0)
                    val_correct += (predicted == labels).sum().item()

            val_loss /= len(val_loader)
            val_accuracy = 100 * val_correct / val_total
            f.write(f'Validation Loss:
            {val_loss:.4f}, Validation Accuracy:
            {val_accuracy:.2f}%\n')
            print(f'Validation Loss:
            {val_loss:.4f}, Validation Accuracy:
            {val_accuracy:.2f}%\n')

            if val_loss > best_val_loss or val_accuracy < best_val_accuracy:
                f.write("Validation loss increased or accuracy decreased.\n")
                print("Validation loss increased or accuracy decreased.\n")
                bad_epochs += 1
                if bad_epochs >= 3:
                    f.write("Stopping training due to consecutive bad epochs.\n")
                    print("Stopping training due to consecutive bad epochs.\n")

                    torch.save({
```

```
                    'epoch': epoch,
                    'model_state_dict': model.state_dict(),
                    'optimizer_state_dict': optimizer.state_dict(),
                    'val_loss': val_loss,
                    'val_accuracy': val_accuracy
                }, checkpoint_path)

                break
            else:
                bad_epochs = 0

            best_val_loss = val_loss
            best_val_accuracy = val_accuracy
```

2.5 训练函数

```
def train_model(model, train_loader, val_loader, criterion, optimizer, num_epochs=36,
output_file='training_log.txt', checkpoint_path='checkpoint.pth'):
    start_time = time.time()
    best_val_loss = float('inf')
    best_val_accuracy = 0.0
    bad_epochs = 0

    with open(output_file, 'w') as f:
        for epoch in range(num_epochs):
            f.write(f'Epoch [{epoch+1}/{num_epochs}]\n')
            print(f'Epoch [{epoch+1}/{num_epochs}]\n')

            model.train()
            running_loss = 0.0
            correct = 0
            total = 0

```

```
    for i, (images, labels) in enumerate(tqdm(train_loader)):
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    train_loss = running_loss / len(train_loader)
    train_accuracy = 100 * correct / total
    f.write(f'Train Loss: {train_loss:.4f},
    Train Accuracy: {train_accuracy:.2f}%\n')
    print(f'Train Loss: {train_loss:.4f}, Train
    Accuracy: {train_accuracy:.2f}%\n')

    model.eval()
    val_loss = 0.0
    val_correct = 0
    val_total = 0

    with torch.no_grad():
        for images, labels in tqdm(val_loader):
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            val_total += labels.size(0)
            val_correct += (predicted == labels).sum().item()

```

```
            val_loss /= len(val_loader)
            val_accuracy = 100 * val_correct / val_total
            f.write(f'Validation Loss: {val_loss:.4f},
            Validation Accuracy: {val_accuracy:.2f}%\n')
            print(f'Validation Loss: {val_loss:.4f},
            Validation Accuracy: {val_accuracy:.2f}%\n')

            if val_loss > best_val_loss or val_accuracy < best_val_accuracy:
                f.write("Validation loss increased or accuracy decreased.\n")
                print("Validation loss increased or accuracy decreased.\n")
                bad_epochs += 1
                if bad_epochs >= 3:
                    f.write("Stopping training due to consecutive bad epochs.\n")
                    print("Stopping training due to consecutive bad epochs.\n")

                    torch.save({
                        'epoch': epoch,
                        'model_state_dict': model.state_dict(),
                        'optimizer_state_dict': optimizer.state_dict(),
                        'val_loss': val_loss,
                        'val_accuracy': val_accuracy
                    }, checkpoint_path)

                    break
            else:
                bad_epochs = 0

            best_val_loss = val_loss
            best_val_accuracy = val_accuracy

        end_time = time.time()
        elapsed_time = end_time - start_time
        f.write(f'Training time: {elapsed_time:.2f} seconds\n')
```

```
84          print(f'Training time: {elapsed_time:.2f} seconds\n')
```

2.6 测试函数及指标计算

```
1  def test_model(model, test_loader, criterion, output_file='training_log.txt'):
2      with open(output_file, 'a') as f:
3          test_loss = 0.0
4          test_correct = 0
5          test_total = 0
6          test_class_correct = list(0. for i in range(5))
7          test_class_total = list(0. for i in range(5))
8          with torch.no_grad():
9              for images, labels in tqdm(test_loader):
10                 outputs = model(images)
11                 loss = criterion(outputs, labels)
12                 test_loss += loss.item()
13                 _, predicted = torch.max(outputs, 1)
14                 test_total += labels.size(0)
15                 test_correct += (predicted == labels).sum().item()
16                 c = (predicted == labels).squeeze()
17                 for i in range(len(labels)):
18                     label = labels[i]
19                     test_class_correct[label] += c[i].item()
20                     test_class_total[label] += 1
21
22         test_loss /= len(test_loader)
23         test_accuracy = 100 * test_correct / test_total
24         f.write(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2f}%\n')
25         print(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2f}%\n')
26
27         for i in range(5):
28             recall = 100 * test_class_correct[i] / test_class_total[i]
29             precision = 100 * test_class_correct[i] / test_total
```

```
            f1 = 2 * ( precision * recall ) / ( precision + recall )
            f.write(f'Class {i} - Recall: {recall:.2f}%,
            Precision: {precision:.2f}%, F1: {f1:.2f}\n')
            print(f'Class {i} - Recall: {recall:.2f}%,
            Precision: {precision:.2f}%, F1: {f1:.2f}\n')
```

3 结果分析

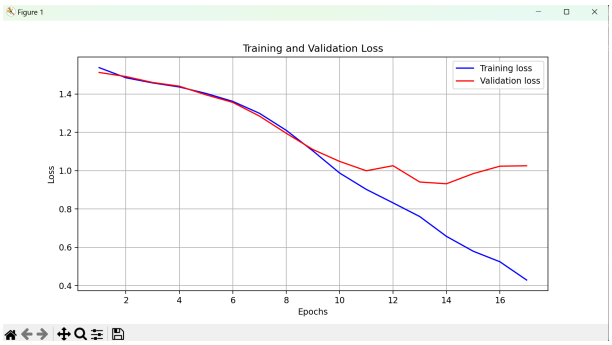
训练过程日志：

```
Epoch [1/50]
Train Loss: 1.5378, Train Accuracy: 30.88%
Validation Loss: 1.5125, Validation Accuracy: 35.60%
Epoch [2/50]
Train Loss: 1.4850, Train Accuracy: 36.40%
Validation Loss: 1.4912, Validation Accuracy: 35.70%
Epoch [3/50]
Train Loss: 1.4580, Train Accuracy: 38.20%
Validation Loss: 1.4603, Validation Accuracy: 38.00%
Epoch [4/50]
Train Loss: 1.4368, Train Accuracy: 38.55%
Validation Loss: 1.4409, Validation Accuracy: 38.70%
Epoch [5/50]
Train Loss: 1.4031, Train Accuracy: 40.30%
Validation Loss: 1.3957, Validation Accuracy: 40.00%
Epoch [6/50]
Train Loss: 1.3616, Train Accuracy: 42.73%
Validation Loss: 1.3563, Validation Accuracy: 43.60%
Epoch [7/50]
Train Loss: 1.2993, Train Accuracy: 45.52%
Validation Loss: 1.2848, Validation Accuracy: 45.70%
Epoch [8/50]
Train Loss: 1.2107, Train Accuracy: 50.95%
```

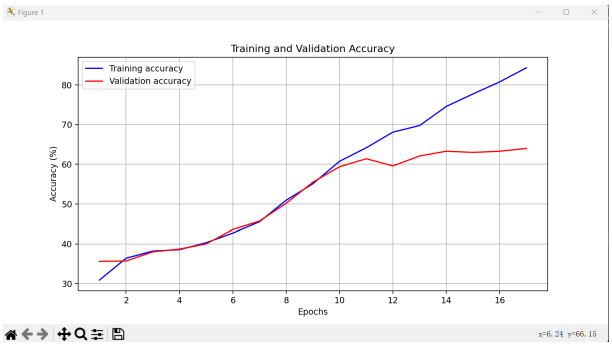
```
Validation Loss: 1.1953, Validation Accuracy: 50.20%
Epoch [9/50]
Train Loss: 1.1031, Train Accuracy: 55.10%
Validation Loss: 1.1103, Validation Accuracy: 55.50%
Epoch [10/50]
Train Loss: 0.9876, Train Accuracy: 60.77%
Validation Loss: 1.0482, Validation Accuracy: 59.40%
Epoch [11/50]
Train Loss: 0.9020, Train Accuracy: 64.17%
Validation Loss: 0.9993, Validation Accuracy: 61.40%
Epoch [12/50]
Train Loss: 0.8317, Train Accuracy: 68.12%
Validation Loss: 1.0258, Validation Accuracy: 59.60%
Validation loss increased or accuracy decreased.
Epoch [13/50]
Train Loss: 0.7602, Train Accuracy: 69.78%
Validation Loss: 0.9406, Validation Accuracy: 62.10%
Epoch [14/50]
Train Loss: 0.6568, Train Accuracy: 74.60%
Validation Loss: 0.9315, Validation Accuracy: 63.30%
Epoch [15/50]
Train Loss: 0.5793, Train Accuracy: 77.75%
Validation Loss: 0.9843, Validation Accuracy: 63.00%
Validation loss increased or accuracy decreased.
Epoch [16/50]
Train Loss: 0.5242, Train Accuracy: 80.80%
Validation Loss: 1.0232, Validation Accuracy: 63.30%
Validation loss increased or accuracy decreased.
Epoch [17/50]
Train Loss: 0.4289, Train Accuracy: 84.33%
Validation Loss: 1.0254, Validation Accuracy: 64.00%
Validation loss increased or accuracy decreased.
Stopping training due to consecutive bad epochs.
```

```
Training time: 991.89 seconds
Test Loss: 1.2011, Test Accuracy: 60.00%
Class 0 — Recall: 41.00%, Precision: 8.20%, F1: 13.67
Class 1 — Recall: 87.00%, Precision: 17.40%, F1: 29.00
Class 2 — Recall: 62.00%, Precision: 12.40%, F1: 20.67
Class 3 — Recall: 39.00%, Precision: 7.80%, F1: 13.00
Class 4 — Recall: 71.00%, Precision: 14.20%, F1: 23.67
```

训练集和验证集损失的变化：



训练集和验证集准确度的变化：



可以看到当训练到第 15 轮之后，模型在验证集上连续 3 轮的损失增大了，模型判断可能出现了过拟合，故停止训练

测试集结果:

Test Loss: 1.7005, Test Accuracy: 57.60%

Class 0 - Recall: 56.00%, Precision: 11.20%, F1: 18.67

Class 1 - Recall: 76.00%, Precision: 15.20%, F1: 25.33

Class 2 - Recall: 31.00%, Precision: 6.20%, F1: 10.33

Class 3 - Recall: 71.00%, Precision: 14.20%, F1: 23.67

Class 4 - Recall: 54.00%, Precision: 10.80%, F1: 18.00

4 自己的探索心得体会;

4.1 自己的探索

4.1.1 我的模型中多层感知机最后一层用 softmax 激活函数的效果没有直接 linear 的效果好

查询资料得知: 常用的图像分类的模型 vgg、resent 都不在最后一层使用 softmax 将概率之和调为 1, 而是直接加一个 FC 层, 因为随着深度学习框架的发展, 为了更好的性能, 部分框架选择了在使用交叉熵损失函数时默认加上 softmax, 包括 pytorch, 这样无论你的输出层是什么, 只要用了 nn.CrossEntropyLoss 就默认加上了 softmax。而如果像我之前一样多层感知机最后一层再加了 softmax, 相当于两个 softmax 合在一起, 就可能效果不好。

4.2 心得体会

通过本次实验，我对卷积神经网络有了更深的了解，通过学习 AlexNet、VGG 和 ResNet 的网络结构和优点，我意识到了设计一个好的网络架构是至关重要的, 故我学习借鉴了 VGG 网络的创新之处来应用到了自己的模型中，取得了不错的效果，并且对 VGG 网络的特点和创新之处设计原因有了更深的了解。

同时在本次实验中，我也熟悉了数据预处理、验证集划分和使用验证集验证、学习率的调整、网络模型架构的设计以及 pytorch 的相关库和函数的实现方法和细节，强化了自己对人工神经网络的使用技巧。

这次实验也加深了我对人工神经网络的学习兴趣，其魅力在于可以解决各

种各样复杂的问题，同时也启发我要加强对典型模型的原理和工作方式的学习，同时进行实践，提高自己利用人工神经网络解决实际问题的能力。