《神经网络与深度学习》课程实验作业（二）

实验内容：计算机视觉基础

何尉宁 2021213599

实验要求：

- [根据已有数据集，对training 和validation 进行处理，构造训练数据集](# one1)
- [设计transform(至少五种)](# one2)
- [对train_transform 进行可视化，将变换后的结果进行展示](# one3)
- [搭建好个人设计的训练模型，并利用tensorboard 对过程进行可视化展示](# one4)
- [在验证集上显示准确性和混淆矩阵](# one5)
- [将个人设计模型对测试集预测结果输出到ans_ours.csv中](# one6)
- [通过torchvision.models 调用VGG 系列模型，并打印网络结构](# one7)
- [使用VGG 系列模型对测试集进行预测，并将结果输出到ans_vgg.csv中](# one8)

写在最前：

> 在着手本作业代码工作之前，先看了一遍李宏毅老师的HW3视频，所以也对其中的一些代码做了参考，主要是`simple`版中的一些数据读取函数。
> 因为通原半期的缘故，加上本地算力不足的问题，我将数据集分割为了原数据集与迷你数据集，分别放在Kaggle和本地上进行测试。但由于疏忽，部分在Kaggle上运行的结果丢失了，只有一些截图，并且初版的`CNN`，`VGG16`，`VGG19`没有下载predict_ours和predict_vgg，只有本地迷你数据集做测试的predict.csv，还请见谅。
> 本次作业仍有需要整改的部分，但迫于时间原因只能仓促的交一份Acc大概在75%左右的报告。
> (🤡破防时刻：对于不同tfm参数，不同模型，不同dropout处理方式，earlystopping策略，从30-200不等的epoch，是否有交叉验证，我的准确率始终在70~75徘徊，脸都要笑烂了)

---

1. 数据集与构造

1.1 设计数据读入函数

观察文件夹中给出的照片文件格式可以发现，食物由0到10分别为：
Bread, Dairy product, Dessert, Egg, Fried food, Meat, Noodles/Pasta, Rice, Seafood, Soup, and Vegetable/Fruit.
命名格式为`0_#`，第一位为标签，第二位为编号，所以我们用python的字符串分片处理一下

```
data_dir = Path('../Assignment2_dataset/food-11')
train_dir = data_dir / 'training'
val_dir = data_dir / 'validation'
test_dir = data_dir / 'testing'

def readfile(path, own_label):
    image_dir = sorted(os.listdir(path)) #对path下的文件进行排序后输出
    x = np.zeros((len(image_dir),128,128,3),dtype=np.uint8) #训练数据
    y = np.zeros((len(image_dir)),dtype=np.uint8) #label
    for i,file in enumerate(image_dir): #将图片信息分配到x,y中
        img = Image.open(os.path.join(path,file))
        x[i,:,:] = np.array(img.resize((128,128))).astype(np.uint8)
        if own_label: #读标签
            y[i] = int(file.split("_")[0])
    if own_label: #train & validation
        return x,y
    else:
        return x #test
```

1.2 数据构成

readfile后得到数据形状：

```
Size of training data = 9866
Size of validation data = 3430
Size of Testing data = 3347
```

1.3 数据切片(本地小数据集)

因为算力的问题，所以完整数据集我是放在Kaggle跑的，本地做了一个3000张大小的小数据集对代码可行性进行验证与初步观察。
但实际上也有不少问题，在使用`Batch_Normalization`的情况下，小数据集会出现较大偏差，最终只剩下4~5种类型的输出，我猜测是标准化之后导致特征变得比较模糊

```
MINISIZE_1 = 3000
MINISIZE_2 = 900

random_sample_1 = np.random.randint(0, len(train_set), size=MINISIZE_1)
random_sample_2 = np.random.randint(0, len(val_set), size=MINISIZE_2)
train_miniset = Subset(train_set, random_sample_1)
val_miniset = Subset(val_set, random_sample_2)
# test_miniset = Subset(test_set, random_sample_2)

# 序号重排
train_miniset.indices = np.arange(len(train_miniset))
val_miniset.indices = np.arange(len(val_miniset))
# test_miniset.indices = np.arange(len(test_miniset))

print("Size of mini training data = {}".format(len(train_miniset)))
print("Size of mini validation data = {}".format(len(val_miniset)))
# print("Size of mini testing data = {}".format(len(test_miniset)))

minitrain_loader = DataLoader(train_miniset, batch_size = batch_size, shuffle =
True)
minival_loader = DataLoader(val_miniset, batch_size = batch_size, shuffle =
False)
# minitest_loader = DataLoader(test_miniset, batch_size = batch_size, shuffle =
False)
```

2. Transform设计

1. Resize: 将图片大小定到128*128
2. RandomResizedCrop: 随机截取图片中一部分
3. GaussianBlur: 高斯模糊
4. RandomHorizontalFlip: 随机水平翻转图片
5. RandomVerticalFlip: 随机垂直翻转图片
6. RandomRotation: 随机旋转图片
7. ColorJitter: 对比度调整
8. Normalize: 标准化图片(使用自己算出来的std和mean处理，但效果很一般。后来去网上找了一组数据)

在这些transform中，我认为`RandomResizedCrop`是最鸡肋的一个变换，需要设置裁剪的scale和ratio，否则很有可能直接把有用信息裁掉了，要么就是直接把主体给忽略了。

对于`ColorJitter`和`Normalize`，这俩是最抽象的变换，肉眼不能直观的判断变换后是否对训练有正向的帮助，他们都直接对ndarray里的数值进行数乘运算，在高对比度的情况下，人眼可能看着就是一团黑黢黢的糊糊；而在标准化的时候，mean和std的选择非常重要，很有可能导致图片能量很大，白茫茫的一片。

另外，对于没有用的什么取反色，边缘裁剪啥的，效果太过于反人类了，就根本没试。

```
train_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((128, 128)),
    transforms.RandomResizedCrop(128, scale=(0.5, 1.0), ratio=(1.0, 1.0)),
    transforms.GaussianBlur(3, sigma=(0.1, 2.0)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(20),
    transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1,
hue=0.1),
    transforms.ToTensor(),
    # transforms.Normalize(mean=mean, std=std), 仅在train中加入标准化会导致val
的acc极低
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225])
])

test_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
    # transforms.Normalize(mean=mean, std=std),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225])
])
```

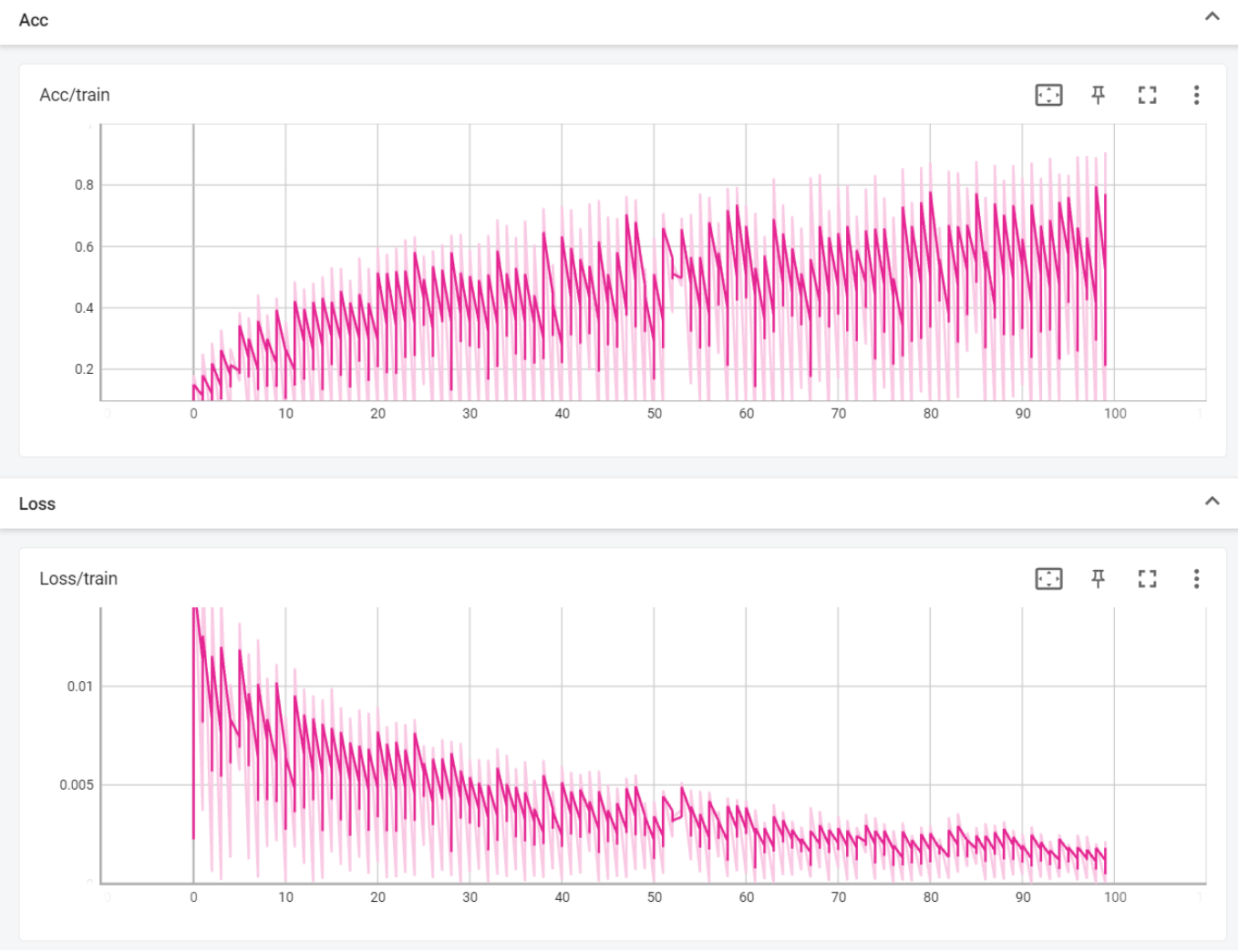
3. Transform图片可视化

```
TFM_N = 6

idx = np.random.randint(0, len(train_x))
origin_img = train_x[idx]
# origin_img = cv2.cvtColor(origin_img, cv2.COLOR_BGR2RGB)
plt.figure(figsize=(15, 3.5))
plt.title('Origin image')
```

```
plt.imshow(origin_img)
plt.axis('off')
plt.show()

plt.figure(figsize=(15, 3.5))
for i in range(TFM_N):
    tfm_img = train_transform(origin_img)
    plt.subplot(1, TFM_N, i+1)
    plt.title(f"Transformed image {i + 1}")
    plt.imshow(tfm_img.permute(1, 2, 0))
    plt.axis('off')
plt.show()
```

4. TensorBoard可视化展示训练过程

我的TensorBoard画出来好像一直有点问题，还得再研究一下



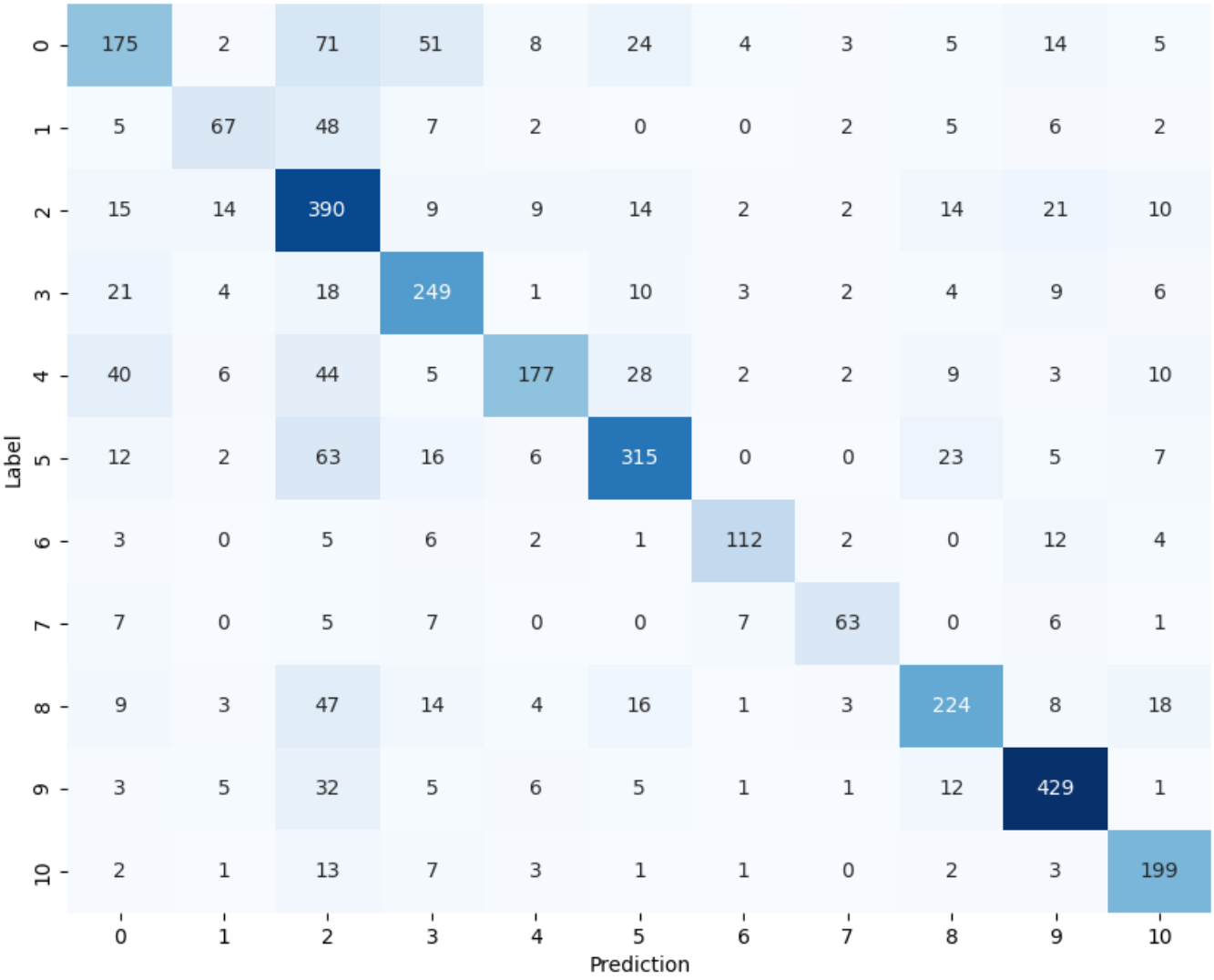
5. 验证集上显示准确性与混淆矩阵

```
import seaborn as sns
from sklearn.metrics import confusion_matrix

# confusion matrix
train_cfm = torch.zeros(11, 11)
val_cfm = torch.zeros(11, 11)

model.eval()
with torch.no_grad():
    for i, data in enumerate(tqdm(minival_loader)):
        val_pred = model(data[0].cuda())
        val_label = data[1].cuda()
        _, val_pred = torch.max(val_pred, 1)
        for j in range(len(val_label)):
            val_cfm[val_label[j], val_pred[j]] += 1
```

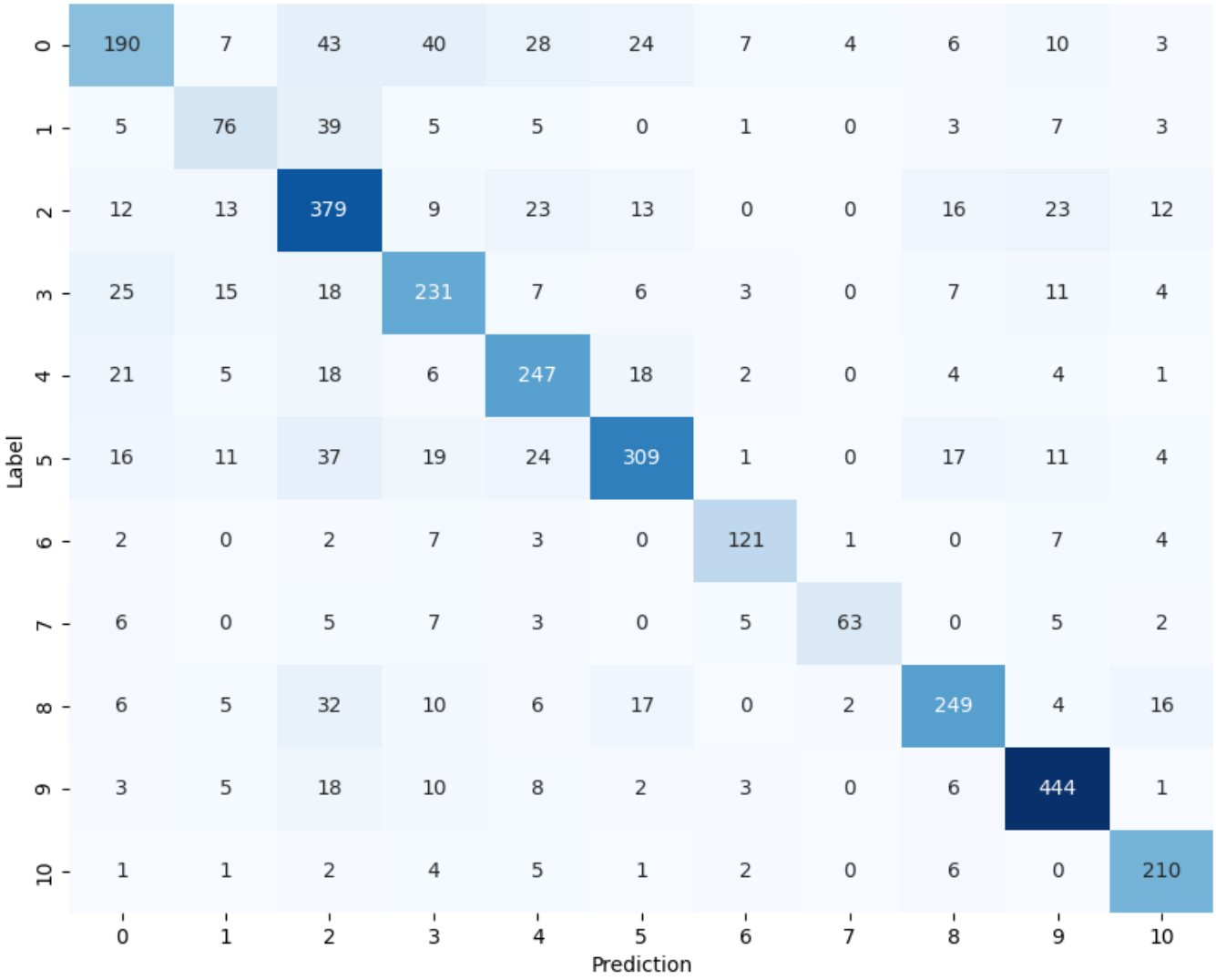
```
plt.figure(figsize=(10, 8))
sns.heatmap(val_cfm, annot=True, cmap='Blues', fmt='g', cbar=False)
plt.xlabel('Prediction')
plt.ylabel('Label')
plt.show()
```

准确率：

```
...

...
[196/200] 37.53 sec(s) Train Acc:0.956619 Loss:0.001064 | Val Acc:0.746647
loss:0.008651
[197/200] 37.84 sec(s) Train Acc:0.952058 Loss:0.001196 | Val Acc:0.730904
loss:0.008909
[198/200] 37.51 sec(s) Train Acc:0.937766 Loss:0.001546 | Val Acc:0.757726
loss:0.008164
[199/200] 38.11 sec(s) Train Acc:0.947598 Loss:0.001274 | Val Acc:0.750437
loss:0.008518
[200/200] 37.67 sec(s) Train Acc:0.953983 Loss:0.001136 | Val Acc:0.762974
loss:0.008249
```



还是能看出来乳制品，鸡蛋啥的浅色食物容易被误分类为面包等面食。



去掉对比度调整之后成了非常均匀的误判。

6. 测试集预测输出

```
with open("predict_resnet18.csv",'w') as f:
    f.write('Id,Categroy\n')
    for i,y in enumerate(prediction):
        f.write('{},{}\n'.format(i,y))
```

7. 打印VGG网络模型

VGG19和VGG16都用了，后来看到网上还有人说VGG11对这个问题的处理效果比较好，但没时间去试了

```
# VGG19
from torchvision.models import vgg19
```

```
class Classifier_VGG19(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.CNN = vgg19(weights=None, num_classes=11)
        self.CNN = nn.Sequential(
            nn.Conv2d(3, 64, 3, 1, 1),
            nn.BatchNorm2d(64),
            nn.Dropout(0.1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),

            nn.Conv2d(64, 128, 3, 1, 1),
            nn.BatchNorm2d(128),
            nn.Dropout(0.1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),

            nn.Conv2d(128, 256, 3, 1, 1),
            nn.BatchNorm2d(256),
            nn.Dropout(0.1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),

            nn.Conv2d(256, 512, 3, 1, 1),
            nn.BatchNorm2d(512),
            nn.Dropout(0.1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),

            nn.Conv2d(512, 512, 3, 1, 1),
            nn.BatchNorm2d(512),
            nn.Dropout(0.1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2, 0),
        )

        # fully connected layers
        self.fc = nn.Sequential(
            nn.Linear(512*4*4, 1024),
            nn.Dropout(0.25),
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.Dropout(0.25),
```

```
            nn.ReLU(),
            nn.Linear(512, 11)
        )

    def forward(self, x):
        #
        out = self.CNN(x)
        out = out.view(out.size()[0], -1)
        out = self.fc(out)

        return out
```

▶ VGG19 Classifier网络结构

8. VGG预测输出

同6

9. 创新与探索

这次整个实验的基调就是紧张刺激草率，虽然很早就把李宏毅老师的HW3那节课看完了，但实际上写的时候还是有点发怵的。\ 这次的难点主要是数据读入，transform变换，然后就是无穷无尽的调参与训练。\ 最恶心的一点就是我的所有优化都收效甚微，基本上最后都是75%的ACC，然后因为备考通信原理，每天挤出来的时间里只能东改一下西改一下，最后也不太尽人意，离预期甚远。

9.1 CrossValidation

```
for fold, (train_idx, val_idx) in enumerate(skf.split(train_x, train_y)):
    print(f"Fold {fold + 1}/{num_splits}")

    model = Classifier_ResNet18().cuda()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

    # 划分数据为训练集和验证集
    train_features_fold = train_x[train_idx]
    train_labels_fold = train_y[train_idx]
    val_features_fold = train_x[val_idx]
    val_labels_fold = train_y[val_idx]

    # 创建数据加载器
    train_fold_dataset = ImgDataset(train_features_fold, train_labels_fold,
```

```
transform = train_transform)
    val_fold_dataset = ImgDataset(val_features_fold, val_labels_fold, transform
= test_transform)
    train_fold_loader = DataLoader(train_fold_dataset, batch_size=batch_size,
shuffle=True)
    val_fold_loader = DataLoader(val_fold_dataset, batch_size=batch_size,
shuffle=False)
```

K-Fold CrossValidation Output

```
Fold 1/5
[001/020] 49.93 sec(s) Train Acc:0.270026 Loss:0.021960 | Val Acc:0.196992
loss:0.017890
[002/020] 49.36 sec(s) Train Acc:0.337157 Loss:0.019961 | Val Acc:0.255263
loss:0.015858
[003/020] 49.62 sec(s) Train Acc:0.393757 Loss:0.018807 | Val Acc:0.334211
loss:0.014607
[004/020] 49.05 sec(s) Train Acc:0.439545 Loss:0.018088 | Val Acc:0.344361
loss:0.014918
[005/020] 48.93 sec(s) Train Acc:0.464930 Loss:0.017424 | Val Acc:0.402632
loss:0.013695
[006/020] 49.56 sec(s) Train Acc:0.511188 Loss:0.016626 | Val Acc:0.385714
loss:0.013769
[007/020] 48.71 sec(s) Train Acc:0.519932 Loss:0.016386 | Val Acc:0.446992
loss:0.012423
[008/020] 49.96 sec(s) Train Acc:0.559891 Loss:0.015556 | Val Acc:0.468045
loss:0.012153
[009/020] 49.68 sec(s) Train Acc:0.581328 Loss:0.014983 | Val Acc:0.474060
loss:0.011744
[010/020] 49.40 sec(s) Train Acc:0.613294 Loss:0.014455 | Val Acc:0.517669
loss:0.010815
[011/020] 49.27 sec(s) Train Acc:0.630876 Loss:0.014005 | Val Acc:0.498496
loss:0.011475
[012/020] 48.66 sec(s) Train Acc:0.650056 Loss:0.013572 | Val Acc:0.534586
loss:0.010372
[013/020] 48.56 sec(s) Train Acc:0.679767 Loss:0.012902 | Val Acc:0.564286
loss:0.009993
[014/020] 48.31 sec(s) Train Acc:0.685314 Loss:0.012640 | Val Acc:0.587594
loss:0.009371
[015/020] 48.58 sec(s) Train Acc:0.714554 Loss:0.012064 | Val Acc:0.608647
loss:0.008702
[016/020] 48.22 sec(s) Train Acc:0.736179 Loss:0.011609 | Val Acc:0.572932
loss:0.009767
```

```
[017/020] 48.92 sec(s) Train Acc:0.748966 Loss:0.011262 | Val Acc:0.636090
loss:0.008178
[018/020] 48.91 sec(s) Train Acc:0.769744 Loss:0.010751 | Val Acc:0.632331
loss:0.008079
[019/020] 49.22 sec(s) Train Acc:0.780369 Loss:0.010654 | Val Acc:0.630451
loss:0.008424
[020/020] 49.74 sec(s) Train Acc:0.787326 Loss:0.010404 | Val Acc:0.646992
loss:0.007865
Fold 2/5
[001/020] 49.65 sec(s) Train Acc:0.228918 Loss:0.022863 | Val Acc:0.229786
loss:0.017475
[002/020] 49.01 sec(s) Train Acc:0.325562 Loss:0.020055 | Val Acc:0.300489
loss:0.015679
[003/020] 47.98 sec(s) Train Acc:0.376516 Loss:0.018931 | Val Acc:0.343362
loss:0.014511
[004/020] 48.07 sec(s) Train Acc:0.427094 Loss:0.018215 | Val Acc:0.334336
loss:0.014904
[005/020] 47.58 sec(s) Train Acc:0.453699 Loss:0.017783 | Val Acc:0.379090
loss:0.013993
[006/020] 47.37 sec(s) Train Acc:0.466015 Loss:0.017424 | Val Acc:0.397142
loss:0.013673
[007/020] 47.29 sec(s) Train Acc:0.500893 Loss:0.016842 | Val Acc:0.424220
loss:0.012974
[008/020] 48.10 sec(s) Train Acc:0.522516 Loss:0.016507 | Val Acc:0.446032
loss:0.012825
[009/020] 47.32 sec(s) Train Acc:0.553163 Loss:0.015743 | Val Acc:0.476871
loss:0.012067
[010/020] 47.65 sec(s) Train Acc:0.579863 Loss:0.015024 | Val Acc:0.492290
loss:0.011548
[011/020] 47.00 sec(s) Train Acc:0.609288 Loss:0.014472 | Val Acc:0.488153
loss:0.011745
[012/020] 47.33 sec(s) Train Acc:0.618407 Loss:0.014114 | Val Acc:0.528770
loss:0.011000
[013/020] 47.14 sec(s) Train Acc:0.645859 Loss:0.013709 | Val Acc:0.508838
loss:0.011144
[014/020] 47.79 sec(s) Train Acc:0.672370 Loss:0.013126 | Val Acc:0.518240
loss:0.010636
[015/020] 47.09 sec(s) Train Acc:0.690326 Loss:0.012619 | Val Acc:0.555848
loss:0.009975
[016/020] 47.47 sec(s) Train Acc:0.706120 Loss:0.012221 | Val Acc:0.559985
loss:0.009977
[017/020] 46.95 sec(s) Train Acc:0.710915 Loss:0.012108 | Val Acc:0.606619
loss:0.008764
```

```
[018/020] 47.10 sec(s) Train Acc:0.732913 Loss:0.011558 | Val Acc:0.587815
loss:0.009200
[019/020] 46.92 sec(s) Train Acc:0.750588 Loss:0.011353 | Val Acc:0.608123
loss:0.008650
[020/020] 46.94 sec(s) Train Acc:0.771270 Loss:0.010828 | Val Acc:0.602858
loss:0.008732
Fold 3/5
[001/020] 47.28 sec(s) Train Acc:0.259754 Loss:0.022444 | Val Acc:0.207597
loss:0.018895
[002/020] 47.53 sec(s) Train Acc:0.346996 Loss:0.019913 | Val Acc:0.296728
loss:0.015900
```

9.2 VGG16

```
[001/030] 25.98 sec(s) Train Acc: 0.264591 Loss: 0.016628
[002/030] 26.21 sec(s) Train Acc: 0.352512 Loss: 0.014444
[003/030] 26.23 sec(s) Train Acc: 0.404106 Loss: 0.013292
[004/030] 26.25 sec(s) Train Acc: 0.438779 Loss: 0.012474
[005/030] 26.17 sec(s) Train Acc: 0.474654 Loss: 0.011746
[006/030] 26.28 sec(s) Train Acc: 0.503685 Loss: 0.011087
[007/030] 26.24 sec(s) Train Acc: 0.528430 Loss: 0.010594
[008/030] 26.04 sec(s) Train Acc: 0.548736 Loss: 0.010147
[009/030] 26.36 sec(s) Train Acc: 0.572879 Loss: 0.009660
[010/030] 26.18 sec(s) Train Acc: 0.588598 Loss: 0.009312
[011/030] 26.24 sec(s) Train Acc: 0.594088 Loss: 0.009008
[012/030] 26.25 sec(s) Train Acc: 0.614997 Loss: 0.008684
[013/030] 25.99 sec(s) Train Acc: 0.631769 Loss: 0.008290
[014/030] 26.24 sec(s) Train Acc: 0.637485 Loss: 0.008160
[015/030] 26.12 sec(s) Train Acc: 0.644254 Loss: 0.008043
[016/030] 26.18 sec(s) Train Acc: 0.649594 Loss: 0.007925
[017/030] 25.93 sec(s) Train Acc: 0.666892 Loss: 0.007454
[018/030] 26.02 sec(s) Train Acc: 0.670954 Loss: 0.007353
[019/030] 26.06 sec(s) Train Acc: 0.683363 Loss: 0.007055
[020/030] 26.25 sec(s) Train Acc: 0.693592 Loss: 0.006900
[021/030] 26.11 sec(s) Train Acc: 0.699534 Loss: 0.006755
[022/030] 26.16 sec(s) Train Acc: 0.703896 Loss: 0.006643
[023/030] 26.87 sec(s) Train Acc: 0.718637 Loss: 0.006399
[024/030] 27.10 sec(s) Train Acc: 0.729468 Loss: 0.006107
[025/030] 26.86 sec(s) Train Acc: 0.726008 Loss: 0.006121
[026/030] 26.77 sec(s) Train Acc: 0.737891 Loss: 0.005882
[027/030] 26.27 sec(s) Train Acc: 0.741652 Loss: 0.005761
[028/030] 26.12 sec(s) Train Acc: 0.749549 Loss: 0.005578
[029/030] 25.99 sec(s) Train Acc: 0.756167 Loss: 0.005497
[030/030] 26.31 sec(s) Train Acc: 0.760153 Loss: 0.005390
```

9.3 ResNet18 with Dropout

1. 在CNN网络中使用Dropout，设置`p = 0.1`
2. 在全连接层中使用Dropout，设置`p = 0.25`

然而效果都很一般

有Dropout的ResNet18:

```
[095/100] 36.43 sec(s) Train Acc:0.833063 Loss:0.003919 | Val Acc:0.711370 loss:0.006936
[096/100] 36.61 sec(s) Train Acc:0.834786 Loss:0.003760 | Val Acc:0.708163 loss:0.007305
[097/100] 36.32 sec(s) Train Acc:0.836712 Loss:0.003859 | Val Acc:0.690962 loss:0.007789
[098/100] 36.48 sec(s) Train Acc:0.833874 Loss:0.003948 | Val Acc:0.694169 loss:0.007449
[099/100] 36.43 sec(s) Train Acc:0.832556 Loss:0.003878 | Val Acc:0.702332 loss:0.007226
[100/100] 36.75 sec(s) Train Acc:0.849382 Loss:0.003530 | Val Acc:0.713994 loss:0.006691
```

无Dropout的ResNet18:

```
[196/200] 37.53 sec(s) Train Acc:0.956619 Loss:0.001064 | Val Acc:0.746647
loss:0.008651
[197/200] 37.84 sec(s) Train Acc:0.952058 Loss:0.001196 | Val Acc:0.730904
loss:0.008909
[198/200] 37.51 sec(s) Train Acc:0.937766 Loss:0.001546 | Val Acc:0.757726
loss:0.008164
[199/200] 38.11 sec(s) Train Acc:0.947598 Loss:0.001274 | Val Acc:0.750437
loss:0.008518
[200/200] 37.67 sec(s) Train Acc:0.953983 Loss:0.001136 | Val Acc:0.762974
loss:0.008249
```

9.4 EarlyStopping

使用早停法控制训练，避免出现Train的Acc在涨，但Validation的Acc开始降低的情况。但实际来看的话其实作用不大，因为Adam里面自带了正则化，在最后的决策点上距离差不多，也不需要早停来控制距离。

```python
# Early Stopping
class EarlyStopping:
    def __init__(self, patience=5, verbose=False, delta=0,
path='checkpoint.pt'):
        self.patience = patience  # 没有改善的训练轮数
        self.verbose = verbose  # 是否打印出每一次改善
        self.counter = 0  # 记录没有改善的训练轮数
        self.best_score = None  # 记录最好的验证集性能
        self.early_stop = False  # 是否停止训练
        self.delta = delta  # 改善的阈值
```

```
        self.path = path  # 保存模型的文件路径

    def __call__(self, val_loss, model):
        score = -val_loss  # 如果是准确率，将其变为负数

        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
        elif score < self.best_score + self.delta:
            self.counter += 1
            if self.verbose:
                print(f'EarlyStopping counter: {self.counter} out of
{self.patience}')
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
            self.counter = 0

    # def early_stop(self):
    #    return self.early_stop

    def save_checkpoint(self, val_loss, model):
        if self.verbose:
            print(f'Validation loss decreased ({self.best_score:.6f} -->
{val_loss:.6f}).  Saving model ...')
        torch.save(model.state_dict(), self.path)  # 保存最好的模型权重
```

9.5 Ensemble

因为时间不够了所有没有用ensemble这个方法，这个方法是我认为非常自然的提升预测准确率的方法，对多个模型或者不同处理方式处理后得到的model进行一个加权融合。
扔一个简单的模板在这里

```
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import accuracy_score

models是已经训练好的多个模型的列表
models = [model1, model2, ..., model_n]

ensemble_model = VotingClassifier(estimators=[('model_'+str(i), model) for i,
```

```
model in enumerate(models)], voting='hard')

ensemble_model.fit(features_train, labels_train)
ensemble_predictions = ensemble_model.predict(features_test)

ensemble_accuracy = accuracy_score(labels_test, ensemble_predictions)
print(f'Ensemble Accuracy: {ensemble_accuracy:.6f}')
```