comp309 project

ID:300595912
Name:Zihao Zhang

Introduction [4 marks]:

This project is to train a deep learning image processing model that can classify through image sets of cherries, strawberries, and tomatoes.

The goal is that when given a picture, the model can tell whether it is a picture of a cherry, a strawberry, or a tomato. The focus of this project is to build a CNN model and improve the accuracy of the model through tuning. In this project, I successfully built a CNN model and continuously tuned and trained it.

First, I build a neural network (MLP) to classify images. It is then updated to a CNN model for prediction, using some CNN methods to improve accuracy. I mainly improve accuracy by modifying the loss function and optimizer, and use pre-trained models to improve model performance.

I trained and optimized my model in jupyter notebook and finally generated the model.

Problem investigation [15 marks]:

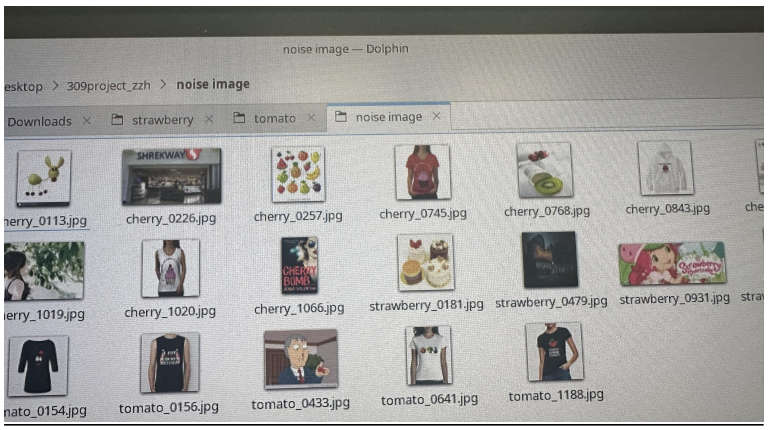
EDA:

First, I roughly observed the database and found that each of the three types of fruits has 1,500 pictures.

They are all **red fruits**. The number of fruits in the pictures contained in each fruit is more or less, the shape is large or small, so at the beginning I thought the model mainly relied on **subtle shape features for identification**.

And there are some related to Fruit-independent noise images such as some **clothes, animation images and character images**, and some such as: strawberry cake should be considered as cake instead of strawberry, so it should also be treated as a noise image.

Through further EDA analysis, I found that the size of **most noise images is not 300*300**, so in the subsequent pre-processing part, I deleted many non-300*300 images as noise images.



pre-process:

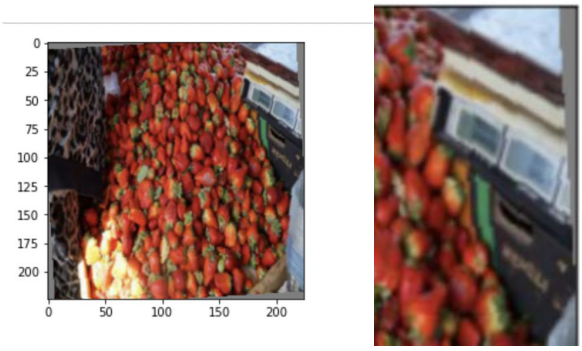
Therefore, during the preliminary data preprocessing, I placed images unrelated to the topic in a folder called ‘noise image’, and did not read them in the subsequent model training part.

By observing the characteristics of the training set images, I used the following method for preprocessing:

```
data_path = 'traindata'

transform = transforms.Compose(
    [transforms.Resize((224, 224)),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
     transforms.RandomRotation(10),
     transforms.RandomHorizontalFlip()
     ])
dataset = torchvision.datasets.ImageFolder(data_path, transform=transform)
train_set, test_set = train_test_split(dataset, train_size=0.8,random_state=309)
trainloader = torch.utils.data.DataLoader(train_set, batch_size=32, shuffle=True)
testloader = torch.utils.data.DataLoader(test_set, batch_size=32)
classes = ('cherry', 'strawberry', 'tomato')
```

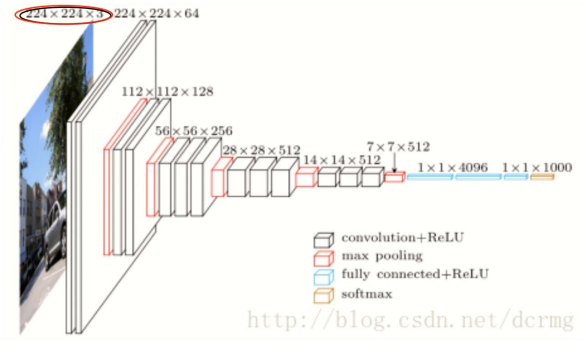
RandomRotation

This method rotates random images by a specified degree to **prevent overfitting.** Randomly rotating the image should be completed before resizing (it has been modified in the final version of my code), because this can avoid **gray** parts in the image from affecting model training, thereby reducing the complexity of model calculations.



Resize

By observing the pictures, I found that most of the pictures are 300*300 in size, and some are not. The first thing that comes to mind is that the size of the pictures should be unified. Here I want to reduce it. By considering the training CNN model theme of this project, I decided to The images are unified to 224*224. This size is also a power of 2. Adjusting the input image to 224x224 is an option in a model similar to VGGNet(see in reference). It simplifies computation by striking a balance between information abstraction and computational efficiency.



ToTensor

Input data is usually fed into the model in the form of tensors after preprocessing. For example, a color image can be represented as a **three-dimensional tensor,** where each dimension corresponds to the height, width, and color channel of the image. If we want to train the model on the image, we need to use **Tensor()** to convert the data into a tensor.

Normalize

The normalization process is applied to every image in your dataset, and these normalized images are then passed into a deep learning model for training. In the first version, I used the example values**[0.5,0.5,0.5]** linked in *5. Some Torch details to keep in mind* at the project paper.

In the final version of model training, I used the standard mean and variance of the Imagenet dataset as my normalization values. This helps the model learn more efficiently and converge faster during training.

RandomHorizontalFlip

Horizontal flipping is a simple and effective technique used to enhance data sets, improve the **generalization ability** of the model, and reduce the risk of overfitting. It is widely used in image-related tasks in deep learning, helping to enable models to handle directional changes in images.

Methodology [30 marks]:

1. Train and test split:

In the given 'traindata', I use 80% as my training set and 20% as my test set. The purpose of this is that during the subsequent training of the model, **the model prediction/test accuracy of the training set and test set can be printed out in real time to judge the performance of the model.** At the same time, **the test results of the test set can also be used to check whether the model is over-fitted**. combine. Finally, a line chart can be generated for further analysis. I did not use k-fold coss validation here because my CNN model structure is relatively complex and requires a lot of training time on the GPU. If k-fold is used, the project time will be greatly extended.

For best model proformance, In my final ‘train.py’ i use all ‘traindata’ to train my model with same cnn model structure and data preporcess which showed good performance, and the test data will be provided when in-person marking.

2. *The loss function(s):*

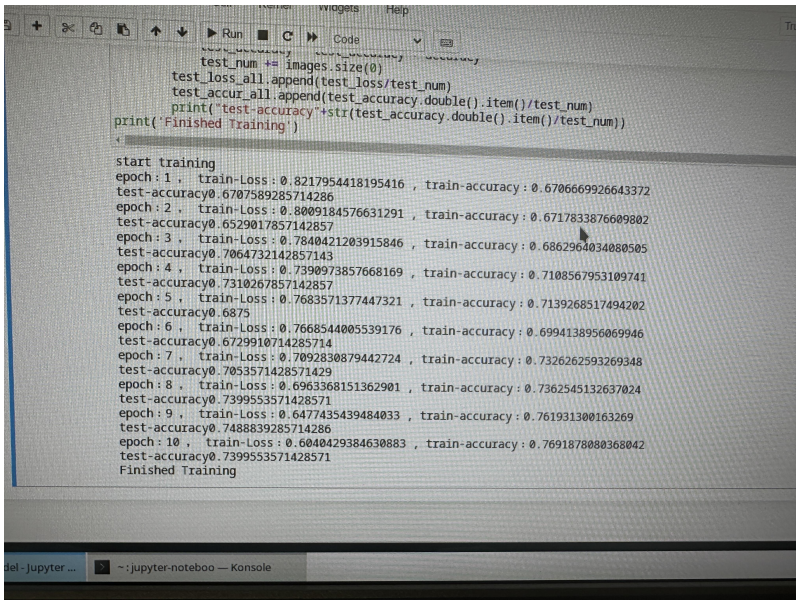
For the cnn model I chose **CrossEntropyLoss()** as my loss function, which measures the error between the model's predictions and the actual labels. If the model's prediction matches the actual label, the loss value is lower, otherwise the loss value is higher.

It is particularly effective in **this fruit classification problem**, where each sample belongs to only one category. Because it is able to compare probability distributions between multiple categories and find the predicted distribution that is closest to the true label distribution. Additionally, it incorporates **SoftMax()** to make the output smoother. In actual tests, I used it and found that the model training accuracy output did not show a large leap.

3. the optimisation method(s):

In the MLP and CNN models, I tested two optimization methods, SGD and Adam respectively: the overall performance of the MLP model is relatively weak for image recognition, and the test results show that the training results of the Adam optimizer are slightly better than SGD. In my CNN model, the performance of SGD (stochastic gradient descent) is much higher than that of Adam. The reason is that my model adopts a structure similar to the VGGNet16 model (16=13convolution layer+3Full connection layer), which is a For more complex structural designs, SGD is usually more stable and can more easily achieve smooth convergence in large models. This is very important for large deep convolutional neural networks (such as VGGNet16). Adam usually has an adaptive learning rate, which can dynamically adjust the learning rate based on the historical gradient of each parameter. For models with complex structures, too large a learning rate may cause the model to Unable to converge. SGD usually has a more clearly defined learning rate that can be better controlled. Therefore, for larger and complex models like VGGNet16, SGD may be more competitive in controlling training stability and reducing the risk of overfitting.

Adam:



SGD:

```
        test_accur_all.append(accuracy_test)
        print(f'test_accuracy: {accuracy_test}')
print('Finished Training')
```

```
[1] loss: 1.04380681103082  train_accuracy: 0.4472835659980774
test_accuracy: 0.4665178656578064
[2] loss: 0.9539408249643297  train_accuracy: 0.5298933386802673
test_accuracy: 0.5290178656578064
[3] loss: 0.8611383648079579  train_accuracy: 0.6234185099601746
test_accuracy: 0.6183035969734192
[4] loss: 0.8057223890384294  train_accuracy: 0.6735301613807678
test_accuracy: 0.7120535969734192
[5] loss: 0.7239133545953741  train_accuracy: 0.726618766784668
test_accuracy: 0.6875000596046448
[6] loss: 0.6876120469065051  train_accuracy: 0.7357975840568542
test_accuracy: 0.7098214626312256
[7] loss: 0.6462433867302523  train_accuracy: 0.7566360831260681
test_accuracy: 0.7008928656578064
[8] loss: 0.6149971552268946  train_accuracy: 0.7730092406272888
test_accuracy: 0.7522321939468384
[9] loss: 0.6114235105267  train_accuracy: 0.7742496132850647
test_accuracy: 0.7410714626312256
[10] loss: 0.586216749076777  train_accuracy: 0.7861573100090027
test_accuracy: 0.7388393282890932
[11] loss: 0.5504484968825979  train_accuracy: 0.7948400378227234
test_accuracy: 0.7500000596046448
[12] loss: 0.5428203722511501  train_accuracy: 0.8040189146995544
test_accuracy: 0.779017865657806 4
[13] loss: 0.5309083261884078  train_accuracy: 0.8154304623603821
test_accuracy: 0.7433035969734192
[14] loss: 0.5092251118197686  train_accuracy: 0.8144381642341614
test_accuracy: 0.7566964626312256
[15] loss: 0.48715559344187  train_accuracy: 0.8171669840812683
test_accuracy: 0.7767857313156128
Finished Training
```

4. *the regularisation strategy*

Batch Normalization:

BN has a certain regularization effect, which helps **prevent overfitting** and helps enhance the stability of the model. By normalizing the data for each batch.

It normalizes the input data distribution to make it close to the standard normal distribution with mean 0 and variance 1. This helps reduce the problem of gradient disappearance or explosion, allows the network to converge to the optimal solution faster, and accelerates model training. Since my customized network structure is relatively complex, the model training speed is significantly accelerated after using BN.

Here I batch normalize the convolutional layer output of a 2D convolutional neural network. Applied to a convolutional layer with **64** output channels.

```
class cnn(nn.Module):
    def __init__(self):
        #16=13convolution layer+3Full connection layer
        super(cnn, self).__init__()

        self.Conv1 = nn.Sequential(
            #color image RGB three channel data set, input channel = 3, image size = 224*224
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, stride=1, padding=1),
            #批量归一化是一种用于提高训练稳定性和加速训练的技术。它通常在卷积层中使用。
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),

            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            # 池化层
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

Dropout:

Dropout is another regularization technique that **must be placed after the pooling layer to effectively reduce overfitting.**

A Dropout layer is added between the fully connected layers, which randomly deactivates some neurons to help the model generalize better to new data.

Here I set **0.5** to make half of the neurons ineffective to prevent overfitting caused by too many parameters.

```
# Full connection layer
#全连接层：全连接层用于将卷积层的输出转换为最终的分类结果。
#全连接层包括几个线性层，它们负责将卷积层的特征映射转化为类别概率。
self.fc = nn.Sequential(

    nn.Linear(512*7*7, 256, bias=False),
    nn.ReLU(inplace=True),

    nn.Dropout(0.5),

    nn.Linear(256, 128),
    nn.ReLU(inplace=True),
    #Dropout是一种正则化技巧，用于减少过拟合。
    #在全连接层之间添加了Dropout层，它会随机使一些神经元无效，从而帮助模型更好地泛化到新数据。
    # Make half of the neurons ineffective to prevent over fitting caused by excessive parameter quantity
    nn.Dropout(0.5),

    nn.Linear(128, 3, bias=False)
)
```

5. *the activation function*

In the model I use **nn.ReLU(inplace=True)** as my activation function between each layer. It converts the input signal into a non-negative value. If the input is greater than zero, the output is equal to the input; if the input is less than or equal to zero, the output is zero. Its role is to introduce nonlinearity, allowing the neural network model to learn nonlinear relationships.

Its advantage is that it is fast in calculation and simple to implement. Compared with saturated activation functions such as **Sigmoid**, ReLU has a larger gradient in the

positive area, so it helps to alleviate the vanishing gradient problem. This makes training deep neural networks more stable.

6. hyper-parameter settings

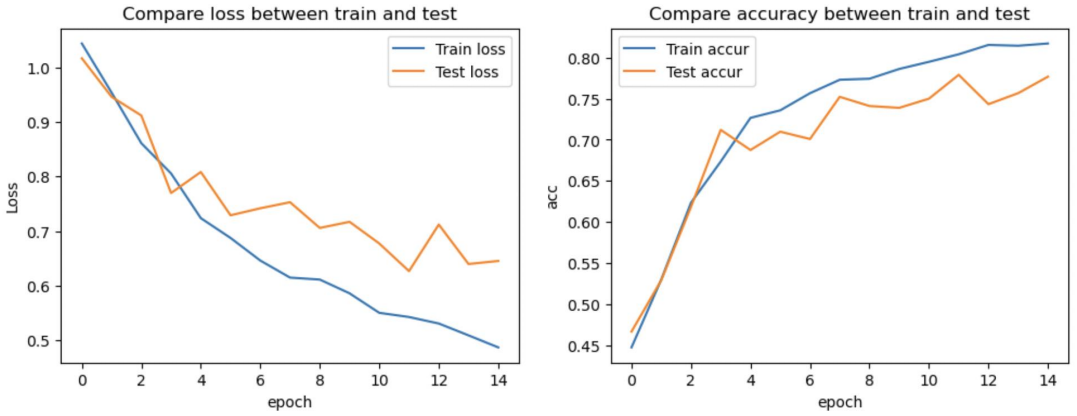
batch size:

At the beginning, I chose to use batch size=32. In the first mlp model training, I found that a larger batch size can speed up model training. However, when training my CNN model, the terminal of the school lab computer reported an error: **"CUDA out of memory error"**. The reason is that the structure of my customized CNN model is relatively complex, with 16 levels, which will further increase the demand for GPU memory. Therefore, without changing the model structure, I chose to modify the **batch size=10** to train the model. This will cause the training speed of my model to be very slow. It takes about half an hour to train 10 times. Therefore, reducing the batch size can reduce the storage consumption of the GPU.

learning rate:

There is no need to set the learning rate in the Adam optimizer that I tried at the beginning, because Adam usually has an adaptive learning rate, which can dynamically adjust the learning rate based on the historical gradient of each parameter. But since my model is more suitable for the SGD optimizer, I set **lr=0.01** at the beginning but the visualization part of the model results found that the **TestLoss fluctuated greatly and the trainLoss did not have a steady downward trend**, so I modified it to **lr=0.001** It was later found that the model training results were reasonable and normal.

result of modified:




```
PATH = './cnn_model.pth'
torch.save(net.state_dict(), PATH)
```

epoch:

Before using transfer learning, I set **epoch=30**, which meant that the model was trained 30 times. The line chart of the prediction results was intuitive and normal. After using transfer learning as described below, my model training time was very long, so I had to Change **epoch=10** to save time. I also think this is reasonable because after I use transfer learning, the prediction accuracy of the second epoch model has reached about 90%, so there is no need for too many loop trainings. At the same time It will not affect the visualization of training results.

7. the use of existing models.

In the transfer learning part, I used the **pre-trained features** (including convolution layer

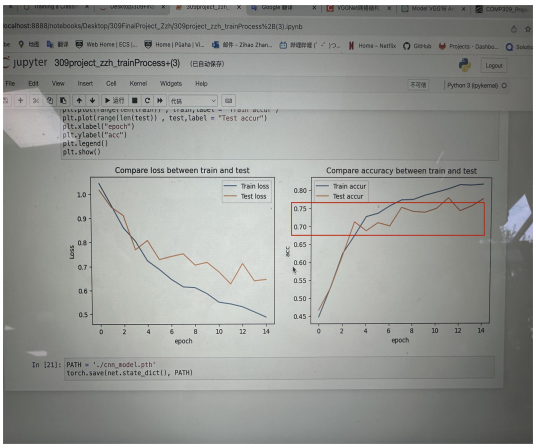
and pooling layer) of VGGNet16 (Visual Geometry Group) (VGGNet won the runner-up in ImageNet in 2014) to add to my model structure, and customized three fully connected layers.

```
super(VGGNet16, self).__init__()
#using vgg16 feature(including convolution layer and pooling layer)
self.features = torchvision.models.vgg16_bn(pretrained=True).features
self.classifier = nn.Sequential(
    nn.Linear(512*7*7, 256, bias=False),
    nn.ReLU(inplace=True),
    nn.Dropout(0.5),

    nn.Linear(256, 128),
    nn.ReLU(inplace=True),
    nn.Dropout(0.5),

    nn.Linear(128, 3, bias=False)
)
```

Output and visualize the results:

without pre-train:

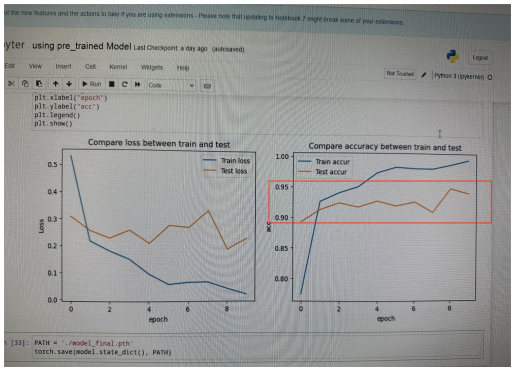
```
[1] loss: 1.04380681103082  train_accuracy: 0.4472835659980774
test_accuracy: 0.4665178656578064
[2] loss: 0.9539408249643297  train_accuracy: 0.5298933386802673
test_accuracy: 0.5290178656578064
[3] loss: 0.8611383648079579  train_accuracy: 0.6234185099601746
test_accuracy: 0.6183035969734192
[4] loss: 0.8057223890384294  train_accuracy: 0.6735301613807678
test_accuracy: 0.7120535969734192
[5] loss: 0.7239133545953741  train_accuracy: 0.726618766784668
test_accuracy: 0.68750005960646448
[6] loss: 0.6876120469065051  train_accuracy: 0.7357975840568542
test_accuracy: 0.7098214626312256
[7] loss: 0.6462433867302523  train_accuracy: 0.7566360831260681
test_accuracy: 0.7008928656578064
[8] loss: 0.6149971552268946  train_accuracy: 0.7730092406272888
test_accuracy: 0.7522321939468384
[9] loss: 0.6114235105267  train_accuracy: 0.7742496132850647
test_accuracy: 0.7410714626312256
[10] loss: 0.586216749076777  train_accuracy: 0.7861573100090027
test_accuracy: 0.738839328289032
[11] loss: 0.5504484968825979  train_accuracy: 0.7948400378227234
test_accuracy: 0.7500000596046448
[12] loss: 0.5428203722511501  train_accuracy: 0.8040189146995544
test_accuracy: 0.7790178656578064
[13] loss: 0.5309083261884078  train_accuracy: 0.8154304623603821
test_accuracy: 0.7433035969734192
[14] loss: 0.5092251118197686  train_accuracy: 0.8144381642341614
test_accuracy: 0.7566964626312256
[15] loss: 0.48715559344187  train_accuracy: 0.8171669840812683
test_accuracy: 0.7767857313156128
Finished Training
```



After pre-train:

```
        print("test-accuracy"+str(test_accuracy.double().item()/test_num))
print('Finished Training')
```

```
start training
epoch: 1 ,  train-Loss: 0.5320094437157762 , train-accuracy: 0.7750487923622131
test-accuracy0.8928571428571429
epoch: 2 ,  train-Loss: 0.21778070778663317 , train-accuracy: 0.9263187050819397
test-accuracy0.9129464285714286
epoch: 3 ,  train-Loss: 0.18098054969999883 , train-accuracy: 0.9405525922775269
test-accuracy0.9241071428571429
epoch: 4 ,  train-Loss: 0.15047682488144226 , train-accuracy: 0.9506000280380249
test-accuracy0.9174107142857143
epoch: 5 ,  train-Loss: 0.0944351564109039 , train-accuracy: 0.9734858870506287
test-accuracy0.9274553571428571
epoch: 6 ,  train-Loss: 0.056081415173603516 , train-accuracy: 0.9829751253128052
test-accuracy0.9196428571428571
epoch: 7 ,  train-Loss: 0.06342029747868057 , train-accuracy: 0.9810214638710022
test-accuracy0.9263392857142857
epoch: 8 ,  train-Loss: 0.06474748999658343 , train-accuracy: 0.9807423949241638
test-accuracy0.9095982142857143
epoch: 9 ,  train-Loss: 0.040672668279680475 , train-accuracy: 0.9874406456947327
test-accuracy0.9486607142857143
epoch: 10 ,  train-Loss: 0.019580372559144917 , train-accuracy: 0.9952553510665894
test-accuracy0.9408482142857143
Finished Training
```



analysis:

It is clearly seen from the image: the test accuracy loss and accuracy are negatively correlated, and before and after using transfer learning, the test accuracy increases slowly as the model training accuracy increases, which means that the model is not overfitting. After using the features of the pre-trained model, compared with the CNN structure (13 convolutional layers) defined by myself, the training accuracy of the model in the line chart improved very quickly, and finally reached an accuracy of more than 90%, and The accuracy of training and testing of my custom CNN model is about 70%-80%.

Since I have tried other lightweight pre-training models (such as mobileNet), compared to VGGNet16, using lightweight models for transfer learning will indeed greatly improve the speed of model training, but the final performance of the model is weaker. In the end After comparison, I found that the features of the VGGNet16 model are more suitable for processing images, and the results are more stable and smooth, so I finally chose it. The only disadvantage is that the GPU storage requirements are relatively high during training.

Summary and discussion [7 marks]:

MLP:

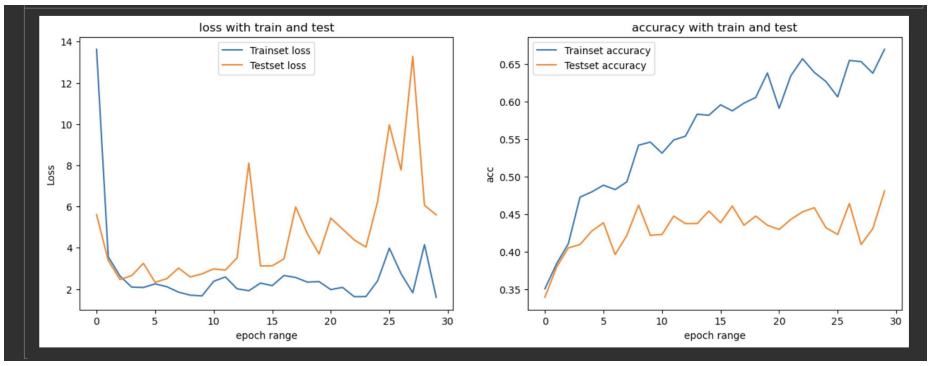
Structure:

```
#structure of MLP
class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet,self).__init__()
        self.fc1=nn.Linear(3*224*224,512) # hidden layer
        self.act1=nn.ReLU()
        self.fc2 = nn.Linear(512,3)
        self.dropout = nn.Dropout(0.2)

    def forward(self,x):
        x = torch.flatten(x,1)#2d-1d
        x=self.fc1(x)
        x=self.act1(x)#activeFuction
        x=self.fc2(x)
        x = self.dropout(x)

        return x
```

Layer1:	*Layer2*
Input Feature: 3*224*224	Input Feature: 512
Output Feature: 512	Output Feature: 3
Activation Fuction: ReLU()	Activation Fuction: ReLU()
	Dropout(p=0.2, inplacement=False)

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

Result(accuracy and loss with train&test):



CNN:

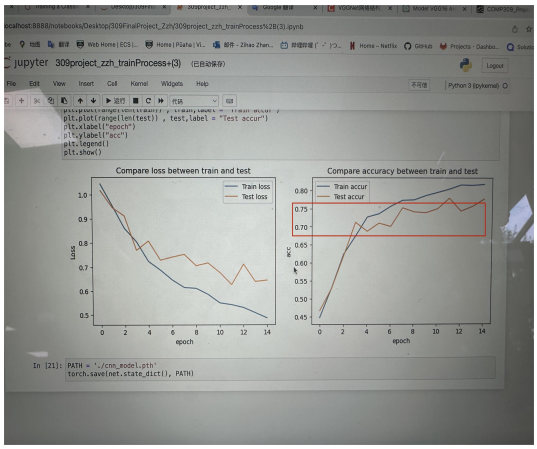
Structure:

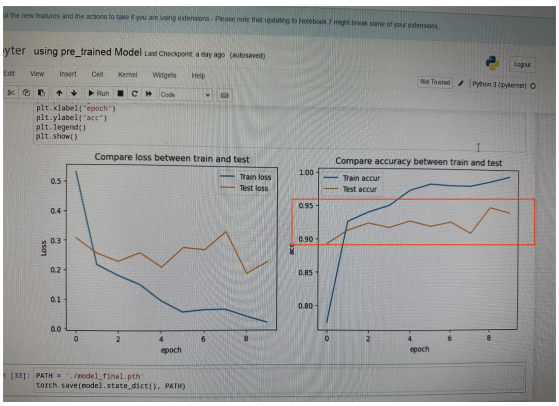
without transfer learing:

A total of 16 layers (excluding Max pooling layer and softmax layer), 16=13convolution layer+3Full connection layer.

There are 5-segment convolutions in the 13 convolutional layers, and the 5-segment convolutions are: 2, 2, 3, 3, 3. That is, 13=2+2+3+3+3. (Similar to the structure of

VGGNet16).
The depth of the convolution layer is 64 -> 128 -> 256 -> 512 ->512
All convolution kernels use a size of 3*3, and pooling kernels use a size of 2*2.
after transfer learning:
Use the features of the pre-trained model VGGNet16 and define three fully connected layers by myself.
Result(accuracy and loss with train&test):
without transfer learing(Left) & after transfer learning(Right)





<u>training time and the classification performance:</u>
With the same data preprocessing. The training time of my CNN model is much longer than that of MLP. The reason is that my CNN model has a complex structure, using transfer learning will extend the training time even more.There are 16 layers of convolutional layer and fully connected layer, while MLP only has a two-layer structure.
It can be concluded from the above line chart results that the performance of the CNN model is much better than that of MLP. From the image of mlp, we can see that as the epoch increases, the training accuracy of the model rises to about 65%, but the test accuracy remains at only 40%-45%. The test accuracy of the CNN model increases steadily with the training accuracy, from about 75% before transfer learning to 90%+ after using transfer learning.

<u>Conclusions and future work [4 marks]:</u>

Through this project, I came to the conclusion that the main reason why CNN performs better than MLP in image classification tasks is that CNN has the following advantages:
1. CNN can effectively capture local features in images, which is very important for tasks such as fruit image classification.
2. CNN has translation invariance and can detect similar features at different locations.
3. Multi-layer convolution and pooling layers allow CNN to learn abstract features layer by layer, improving performance.
4. Technologies such as data enhancement and transfer learning increase the generalization ability and efficiency of the model.
Combining these factors, CNN is more suitable for processing image data.
Among the tuning methods for CNN models, it was concluded that the most effective

one is transfer learning. Training accuracy change increased from 79% to 90%+. The reason is that the excellent features of the pre-trained model are introduced to greatly improve the model performance.

In terms of optimization technology, I think we can also try to add more pictures. For example: because the data set contains strawberry cakes, strawberry-patterned clothes, etc., they have strawberry features and can be used for training, but a large number of such pictures will affect the overall judgment of the model, so you need to enrich the data set and try to add more pictures. to further improve the generalization ability of the model.

reference:

Data preprocessing normalization: using the standard mean and variance of the Imagenet dataset.
https://blog.csdn.net/tugouxp/article/details/123213423
The activation function idea
https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html
VGGNet16 model structure (good at processing image structures).
https://zhuanlan.zhihu.com/p/79258431
4 Pre-Trained CNN Models to Use for Computer Vision with Transfer Learning.
https://towardsdatascience.com/4-pre-trained-cnn-models-to-use-for-computer-vision-with-transfer-learning-885cb1b2dfc
Some model structure idea & model save/load.
https://pytorch.org/tutorials/beginner/saving_loading_models.html