Final Project Individual Report

Yunyun Jiang

Introduction

The goal of this project is to create an optimal neural network to recognize the corresponding artist of an painting pictures based on the colors used and geometric patterns inside the pictures. The dataset we use is from Kaggle website which is a collection of artworks of the 50 most influential artists of all time. However, most of the artists have less or equal than 200 paintings available. To reduce the computation time and improve fast learning, we select top ten artists' work which includes 4060 painting images. These painting images belong to following 10 artists in alphabetical order: Albrecht Dürer, Alfred Sisley, Edgar Degas, Francisco Goya, Pablo Picasso, Paul Gauguin, Pierre-Auguste Renoir, Rembrandt, Titian, Vincent van Gogh'.

As a team, we discussed various strategies for processing the images and implementing models. Each of us has used different ways of importing image datasets, dataset pre-processing methods, and neural network methods. I have focused on convolution neural network (CNN) and pretrained VGG16 methods on Pytorch framework.

Data processing

The first step is to load the datasets. As in the original dataset, the individual artist has a directory containing all the paintings by this artist. Therefore, the datasets can be loaded using ImageFolder under torchvision.datasets. The images are resized to 224x224 which is required by the pre-trained networks. I apply transformations including random scaling, cropping, and flipping. This will help the network generalize leading to better performance. I created a function to accomplish these task. It is a good practice to test the trained network on test data, images the network has never seen either in training or validation. I split the datasets into three parts: training, validation and testing datasets. I first split the training into training , testing by a ratio of 9:1, then split the training into training and validation by a ratio of 8:2. The code is below:

```
############ define a function to transform the image data and
# split data into training and test #######


def load_split_train_test(DATA_DIR,test_train_split=0.9, val_train_split=0.2):
    train_transforms=transforms.Compose([transforms.RandomRotation(30),
                                         transforms.RandomResizedCrop(224),
                                         transforms.RandomHorizontalFlip(),
                                         transforms.ToTensor(),
                                         transforms.Normalize ([0.5,0.5,0.5],
                                                               [0.5,0.5,0.5])
                                         ])
    valid_transforms = transforms.Compose([transforms.Resize(256),
                                           transforms.CenterCrop(224),
                                           transforms.ToTensor(),
                                           transforms.Normalize ([0.5,0.5,0.5],
                                                                 [0.5,0.5,0.5])
```

```
                                        ])

    test_transforms = transforms.Compose([transforms.Resize(256),
                                          transforms.CenterCrop(224),
                                          transforms.ToTensor(),
                                          transforms.Normalize ([0.5,0.5,0.5],
                                                                [0.5,0.5,0.5])
                                        ])
    train_data = datasets.ImageFolder(DATA_DIR,
                    transform=train_transforms)
    valid_data = datasets.ImageFolder(DATA_DIR,
                    transform=valid_transforms)
    test_data = datasets.ImageFolder(DATA_DIR,
                    transform=test_transforms)


    dataset_size = len(train_data)
    indices = list ( range (dataset_size) )
    np.random.shuffle(indices)

    test_split = int(np.floor(test_train_split*dataset_size))
    train_indices_1, test_indices = indices [:test_split], indices [test_split:]
    train_size = len ( train_indices_1)
    validation_split=int(np.floor((1-val_train_split)*train_size))
    train_indices, val_indices = indices [ :validation_split],
indices[validation_split:test_split]
    train_sampler = SubsetRandomSampler ( train_indices )
    test_sampler = SubsetRandomSampler ( test_indices )
    val_sampler = SubsetRandomSampler ( val_indices )

trainloader=torch.utils.data.DataLoader(train_data,batch_size=64,sampler=train_sampler
)
    valloader = torch.utils.data.DataLoader ( valid_data, batch_size = 32, sampler =
test_sampler )
    testloader=torch.utils.data.DataLoader(test_data,sampler=val_sampler)
    return trainloader,valloader, testloader

trainloader, valloader, testloader = load_split_train_test(DATA_DIR, .8, .1)
```

Building and training the Neural Network Model in Pytorch

Self-defined CNN model

I first created a self defined neural networks based four major functions in the network and created a ArtCNNclass based on master torch.nn.Module class.

- torch.nn.Conv2d() – applies convolution
- torch.nn.relu() – applies ReLU
- torch.nn.MaxPool2d() – applies max pooling
- torch.nn.Linear() – fully connected layer (multiply inputs by learned weights)

I created ArtCNN with one class method: forward. The forward() method computes a forward pass of the ArtCNN. (Figure 1)

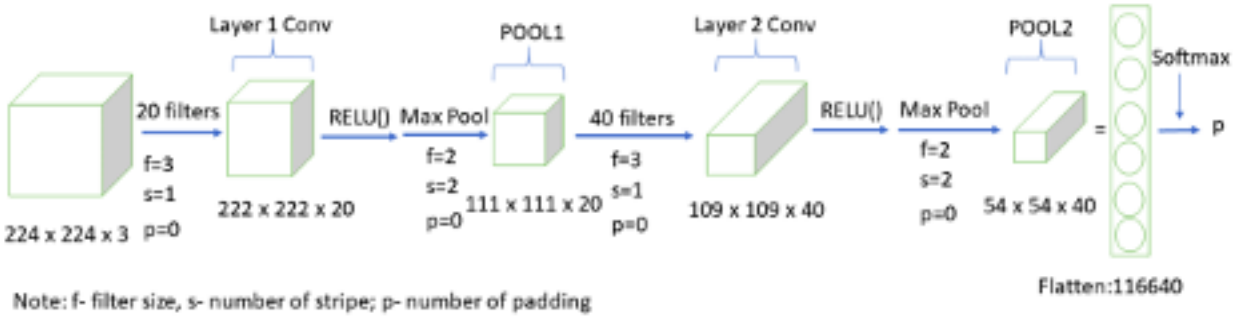
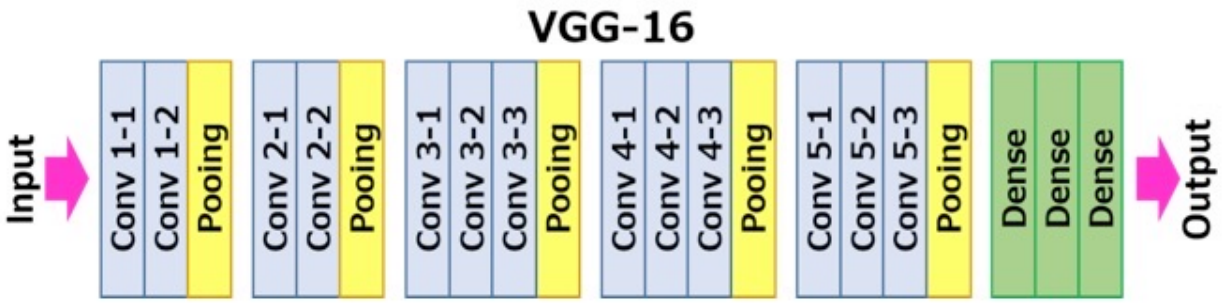


Figure 1. CNN Network for image recognition

```
# %% ------------------------------------ CNN Class ---------------------------------------------------------------
#  LeNet-5 network #:
class ArtCNN(nn.Module):
    def __init__(self):
        super(ArtCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 20, (3, 3))  # output (n_examples, 16, 48, 48)
        self.convnorm1 = nn.BatchNorm2d(20)
        self.pool1 = nn.MaxPool2d((2, 2))  # output (n_examples, 16, 24, 24)
        self.conv2 = nn.Conv2d(20, 40, (3, 3))  # output (n_examples, 32, 22, 22)
        self.convnorm2 = nn.BatchNorm2d(40)
        self.pool2 = nn.AvgPool2d((2, 2))  # output (n_examples, 32, 11, 11)
        self.linear1 = nn.Linear(40*54*54, 250)  # input will be flattened to (n_examples, 32 * 5 * 5)
        self.linear1_bn = nn.BatchNorm1d(250)
        self.drop = nn.Dropout(DROPOUT)
        self.linear2 = nn.Linear(250, 10)
        self.act = torch.relu
        #self.softmax = nn.LogSoftmax(dim =1)

    def forward(self, x):
        #x = x.reshape(1,-1)
        x = self.pool1(self.convnorm1(self.act(self.conv1(x.float()))))  #### first layer ###
        x = self.pool2(self.convnorm2(self.act(self.conv2(x.float()))))  #### second layer ###
        x = self.drop(self.linear1_bn(self.act(self.linear1(x.view(len(x), -1)))))  # fully connected layer ###
        return self.linear2(x)
```

VGG16

I also tried the pre-trained network model VGG16 from torchvision.models to build and train a new feed-forward classifier using those features. However, a new, untrained feed-forward network is used as a classifier, using ReLU activations and dropout. During the training, we also only update the weights of the feed-forward network not the model parameters from pretrained model.



The

```
# VGG16 network #:
for parameter in model.parameters():
  parameter.requires_grad = False
from collections import OrderedDict
classifier = nn.Sequential(OrderedDict([('fc1', nn.Linear(25088, 5000)),
                          ('relu', nn.ReLU()),
                          ('drop', nn.Dropout(p=0.5)),
                          ('fc2', nn.Linear(5000, 102))]))
```

Neural Network Training

For CNN model training, I use Cross-Entropy loss as our loss function, Adam as the optimizer. We also specify the learning rate of 0.0001. Below also contains a function to train our ArtCNN model using a simple for loop. During each epoch of training, we pass the data to the model in batches and also calculate the loss on the test dataset.

```
# %% ------------------------------------ Training Prep ------------------------------
------------------------------
model = ArtCNN().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=LR)
criterion = nn.CrossEntropyLoss()


# %% ---------------------------------- define a train classifier --------------------
----------------------------------


epochs = 20
steps = 0
print_every = 10
running_loss=0
train_losses,val_losses,val_accuracy=[],[],[]

for e in range ( epochs ):
    model.train()
    running_loss=0
    for inputs, labels in trainloader:
        steps += 1
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()  ####### every epoch needs to back to 0 ####
```

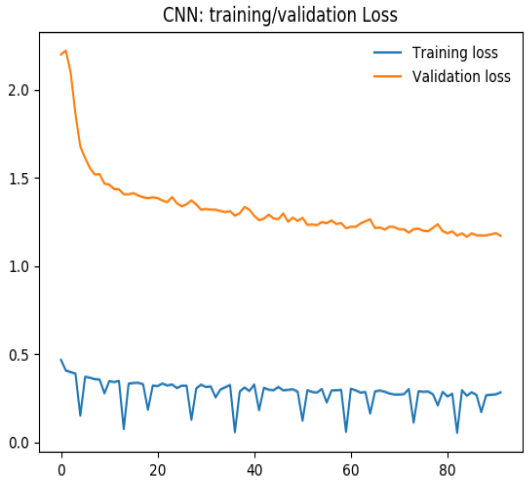
```
        output = model.forward(inputs)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

        if steps % print_every == 0:
            model.eval ()
            #val_loss=0
            accuracy=0
            # Turn off gradients for validation, saves memory and computations
            with torch.no_grad ():
                validation_loss, accuracy = validation (model,valloader,criterion )
            train_losses.append(running_loss/len(trainloader))
            val_losses.append ( validation_loss / len (valloader ) )
            val_accuracy.append(accuracy /len (valloader) )
            print ( "Epoch: {}/{}.. ".format ( e + 1, epochs ),
                    "Training Loss: {:.3f}.. ".format ( running_loss / print_every),
                    "Validation Loss: {:.3f}.. ".format (validation_loss/len
(valloader)),
                    "Validation Accuracy: {:.3f}".format ( accuracy / len
(valloader) ) )
            running_loss = 0
            model.train ()
```

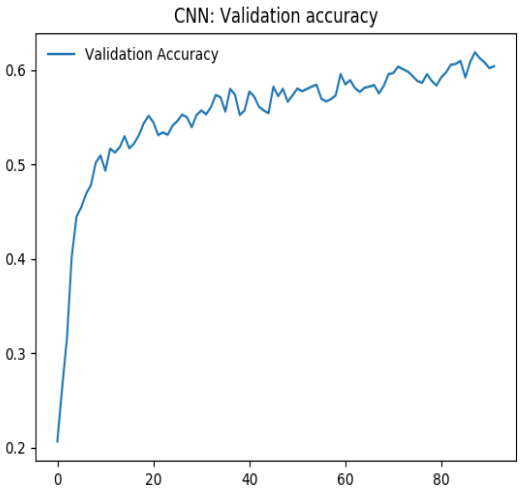
To a certain degree, the model accuracy increases with the number of filters. Also, increase in batch size also improve the performance. I also looked how other parameters change the performance of the model. The learning rate has great impact on the model performance, I varied this parameter and found that the learning rate of 0.0001 is relatively efficient. I chose optimization algorithm instead of the classical stochastic gradient descent procedure to update network weights iteratively in training data due to its advantages: computational efficiency, being appropriate for problems with very noisy or sparse gradients and models that require hyper-parameters. During each epoch of training, we pass the data to the model in batches and also calculate the loss on the validation dataset.

Results

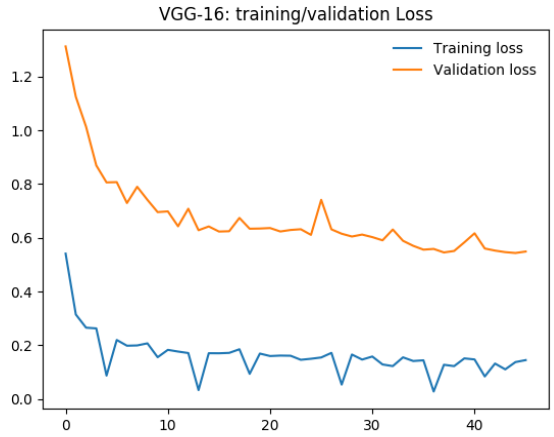
Training/validation loss

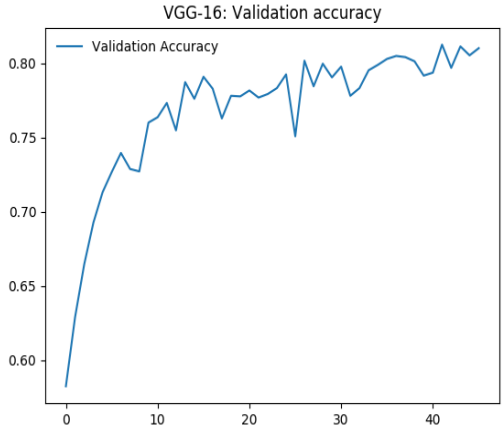
CNN





VGG





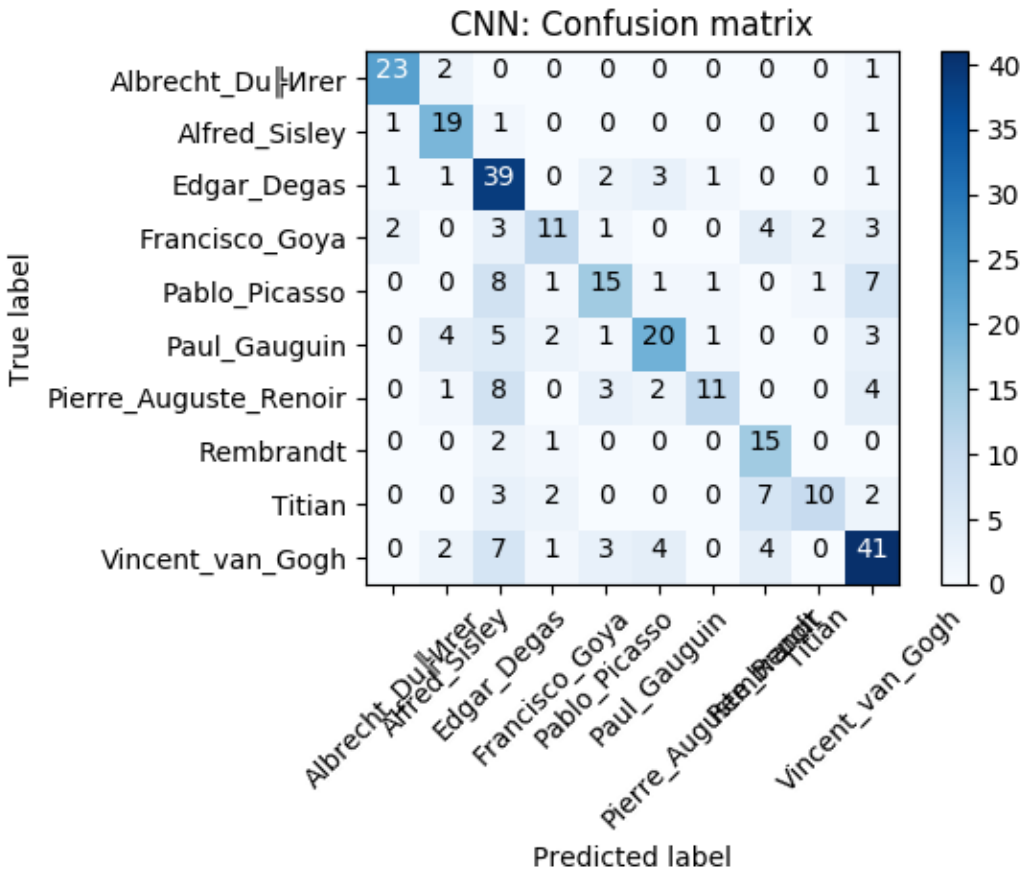
This plot is to show the training/validation accuracy, and training and validation losses. As we can see, the validation loss went down very quickly and become flattened toward the end of the epoch. The training loss as expected is low. The validation accuracy increases with epochs.

Model accuracy

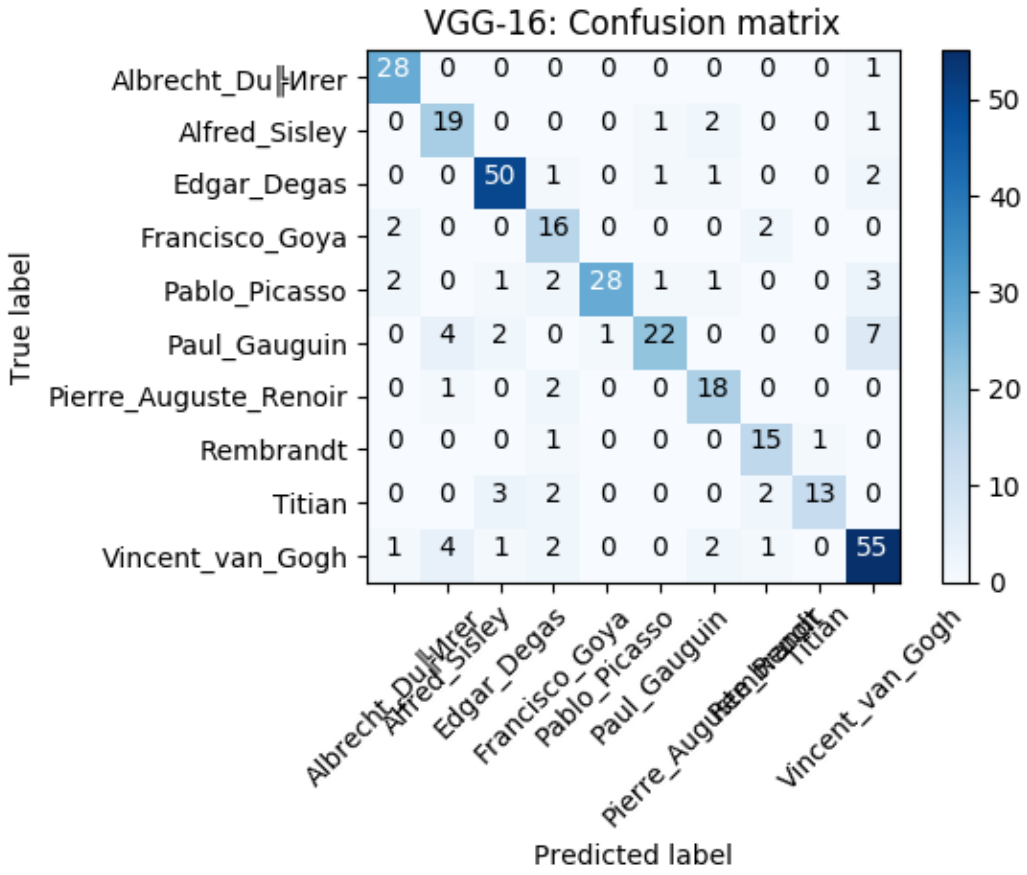
Model	Accuracy
CNN	0.64
VGG16	0.81

Clearly, the VGG model has better accuracy as compared to CNN model.

Confusion matrix



I created a confusion matrix to describe the performance of a CNN classification model on the test dataset. The diagonal elements represent the number of points for which the predicted labels are equal to the true label, the off diagonal elements are those that are not equal. The performance of the model is relatively well as 64% of the pairs of predicted and targeted labels are correctly matched.



For VGG-16 model, where the accuracy is 81%, the numbers on the diagonal line are very high, which reflects the high accuracy.

Summary

In this project, I explore the difference between self-defined and pre-trained model based on convolution neural network. The percentage of the code I found from the internet is about 40%. In future, I would like to use these model make inference on future unseen image data and also incorporate the text information to make better prediction if possible.

References

https://www.kaggle.com/ikarus777/best-artworks-of-all-time
https://help.healthycities.org/hc/en-us/articles/219556208-How-are-the-different-age-groups-defined-
https://keras.io/preprocessing/image/
https://keras.io/applications/#vgg16
https://keras.io/applications/#resnet
https://en.wikipedia.org/wiki/Convolutional_neural_network

https://en.wikipedia.org/wiki/Multilayer_perceptron
https://wiki.python.org/moin/PyQt4
https://scikit-learn.org/stable/
http://en.wikipedia.org/wiki/Vincent_van_Gogh
http://en.wikipedia.org/wiki/Pablo_Picasso
http://en.wikipedia.org/wiki/Francisco_Goya
http://en.wikipedia.org/wiki/Edgar_Degas