COMP4900 Assignment 3

YanPeng Gao
101090653

Drew Suitor
101003158

May 21, 2020

Abstract

The task of this project was image classification on images consisting of three separate articles of clothing. The label of the image is associated with the most expensive article of clothing. We were able to show that image preprocessing and fine tuning neural network architectures can improve results and generate an accuracy over 90%.

1 Introduction

The task at hand was to, given an image of 3 separate articles of clothing, classify the image based on the most expensive article of clothing in the image. This classification ranged from 0, the lowest valued item, to 9, the highest valued item. The dataset we are using is a subset of 60,000 data points from the Fashion-MNIST dataset.

There were a handful of approaches we used when assessing this problem. We initially started with an AlexNet, but eventually decided on implementing a customized version of Residual Network as we had better success with it. We will be describing with more depth these approaches later in the report.

The key finding of this report is that we were able to achieve greater than 90 percent accuracy when classifying images from the Fashion-MNIST dataset through the preprocessing of our data and the fine tuning of our implemented Residual Network.

2 Dataset

The dataset used for this image classification task is a modified version of Fashion-MNIST. The dataset consists of 60,000 data points where each point contains an image of 3 articles clothing as well as a corresponding tag ranging from 0 to 9 which corresponds to the value of the most expensive article of clothing in the image.

The images in the dataset contained a lot of empty space between the articles of clothing. We hypothesized that we would see better classification accuracy by preprocessing the images to remove the empty space between the articles of clothing. We took advantage of K-Means clustering to achieve this. K-Means is an algorithm that partitions the dataset, or in our case an individual image, into K non-overlapping subgroups. We clustered on all pixels that were not black pixels to determine the location of individual articles of clothing and then reconstructed the image containing only the 3 articles of clothing.

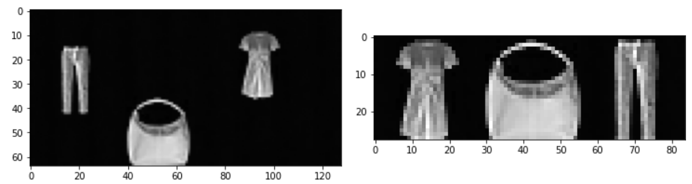

Figure 1: Image before and after preprocessing

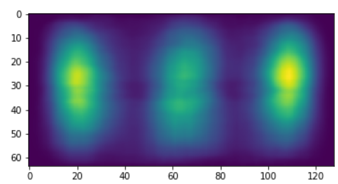

Figure 2: Heatmap of location of icons before preprocessing

We tested out our hypothesis by training and validating our AlexNet model with both the dataset of cropped and uncropped images. We compared the validation accuracy as well as the training loss and found significant improvements on the cropped dataset.

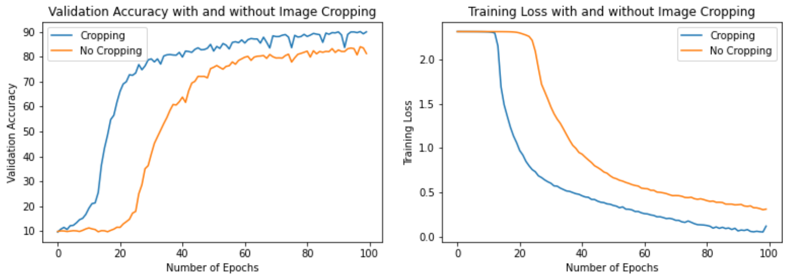

Figure 3: Validation Accuracy and Training Loss on Cropped and Uncropped Datasets

3 Proposed Approach

As with the standard of image classification tasks, convolutional neural networks were used. CNN's have the advantage over regular multilayer perceptrons as they are able to interpret the spatial arrangement of a picture's pixels.

In a CNN, the hidden neuron is made up of input neurons from hidden receptive fields. To do this, a filter (also known as a kernel) of $k_1 \times k_2 \times depth$ is slid across the image. Per each stride, the dot product of the weights in the filter and the neurons in the local receptive field becomes 1 hidden neuron in 1 activation map; and so having multiple features creates multiple activation maps. The purpose of these filters is to activate in the presence of particular shapes, edges, clusters, patterns etc and so the neural network is trying to learn the weights of these filters.

3.1 AlexNet

Alexnet (Krizhevsky et al. [4]) was the first CNN architecture implemented in this project. It was an 8 layer network with 5 convolutional layers and 3 fully connected layers using RELU activation functions instead of sigmoids. While the validation accuracy after 200 epochs was around 93%, the testing accuracy on Kaggle was only 85%. We suspect, as each datapoint was constructed of 3 fashion mnist icons, there were duplication of icons in both the training and validation set hence leading to data leakage. Because of this, the validation set accuracy in these experiments may not generalize well to the testing accuracy.

3.2 Residual Networks

Residual Networks were chosen as they, and variations of ResNets, performed the best in recent years on image classification tasks (Khan et al. [3]). The unique aspect of Residual Networks is the utilization of skip connections every two convolutional layers (He et al. [1]).

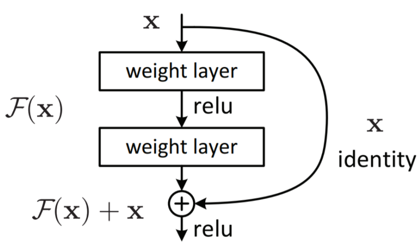

Figure 4: Residual Block

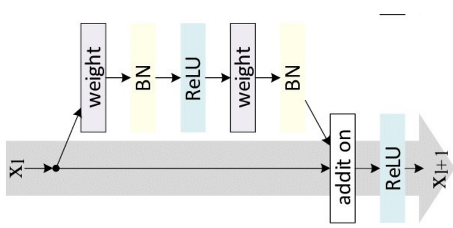

Figure 5: Detailed View of Steps in a Residual Block

The residual block in Figure 3 shows the concept of the skip connection. Fundamentally, we are still learning an underlying hypothesis mapping $H(x)$. However, instead of learning $H(x)$ directly, we learn the residual $F(x)$ and have $H(x) = F(x) + x$. The skip connection x is the identity mapping and $F(x)$ is the residual of $H(x) - x$. It is hypothesized in the original paper that it is easier to learn just the residual mapping than the original unreferenced mapping.

The problem with vanishing and exploding gradients makes deeper neural networks harder to train (Pascanu et al. [5]). Without residual blocks, one may encounter that the output mapping of a deeper convolutional layer retains less knowledge than its input. But with the skip connection, we ensure that we always retain the mappings of the previous layer. Therefore each successive layer is at least as good as the last but potentially could learn additional information as well.

The behaviour of residual networks closely resembles an ensemble of shallower neural networks (Veit et al. [6]). Just like removing one model from an ensemble does not worsen the overall accuracy by much, removing single residual layers has no significant impact on accuracy.

Also in the residual blocks, before the RELU activation function, batch normalization is first performed. Batch normalization normalizes each mini-batch of convolutional outputs with a linear transformation: $y_i = \gamma \hat{x_i} + \beta$ where $\hat{x_i}$ is the unit normalized feature and γ and β are learned features. Batch normalization helps solve the problem known as Internal Covariate Shift (Ioffe and Szegedy [2]) where the input distribution to layers deep in the network changes per mini-batch. With the elimination of the covariate shift, the model is able to use larger learning rates for quicker training.

3.3 Random Erasing

In terms of data augmentation, an approach known as Random Erasing (Zhong et al. [7]) was used in all models. With random erasing, images would have the possibility of a random rectangular portion of the image omitted from training. The authors of the paper has shown partial occlusion has the ability to reduce the variance and improve the generalization ability of CNN's.

Our images are very noisy due to the fact that of the three fashion icons per image, possibly only one of them is relevant in the decision making of our CNN. Hence adding random erasing gives the network the ability to encounter instances where some of the noise is blocked out, leading to better learning.

4 Tuning Our Residual Network

It would not make sense to directly use the residual network model presented in PyTorch as it was built for training on $224 \times 224 \times 3$ CIFAR images. The initial architecture style was taken from Zhong et al. [7]'s work as they also produced results for Fashion Mnist (unmodified). Given a depth d and $n = (d-2)/6$, they proposed an initial convolution layer of 16 filters with a stride of 1 (same convolution), followed by 3 groups of n residual blocks with 16, 32 and 64 filters per convolution respectively. The last 2 groups of residual blocks also halve the length and width by increasing the stride length to 2.

Ideally a grid search of all possible combinations would ensure the optimal network but given our time and computational restraints, the plan is to optimize one parameter, than move on to the next.

4.1 Optimal Depth

In the first step, we compare depths of 20, 32 and 44 layers to find the optimal depth. The criteria were the training loss of 50,000 training images, validation accuracy on 10,000 validation images and the training time per epoch. After this experimentation, a depth of 32 was chosen as it produced the lowest training loss, the highest validation accuracy and was the 2nd quickest to train.

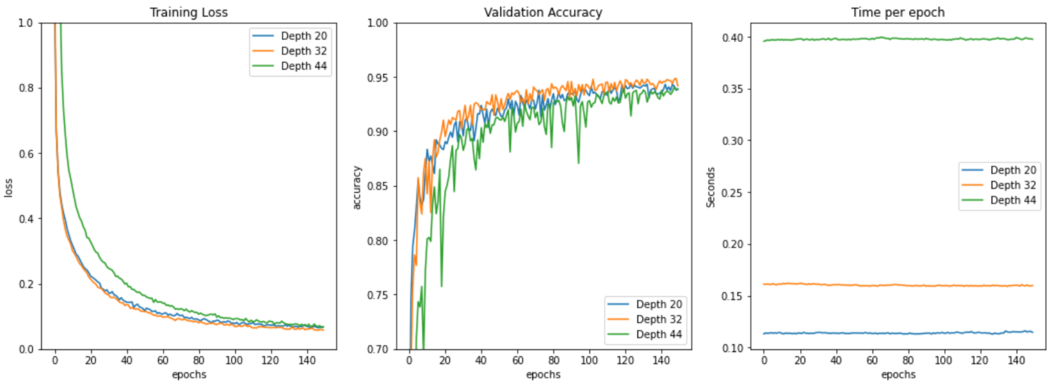

Figure 6: Investigating depth of residual network

4.2 Optimal Filters

We compared groups of blocks using [16,16,32,64] filters per convolutional layer as described above and using more filters of [32,64,128,256] for groups of residual blocks. After experimentation, it was seen that increasing the amount of convolutional filters used helped our model.

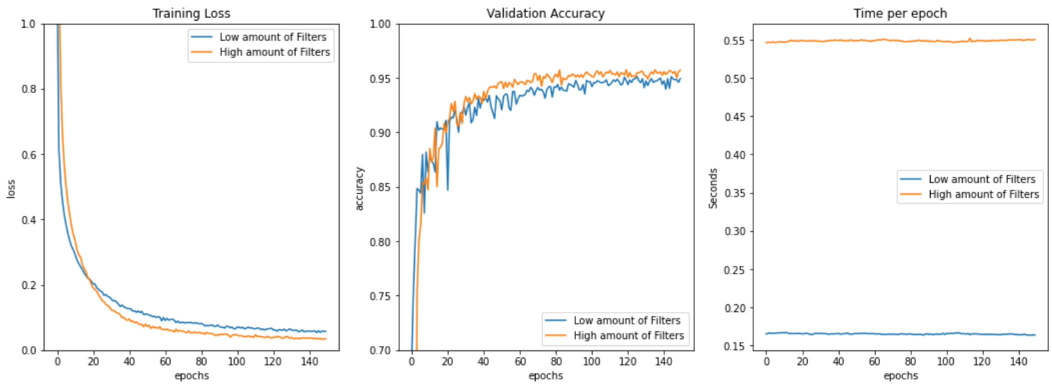

Figure 7: Investigating the amount of filters

4.3 Second Fully Connected Layer

The last layer of the original resnet is a fully connected layer where the inputs are a global average pool of each final activation map. In our case, the input would have 256 neurons representing the 256 activation maps.

The inspiration of adding a second fully connected layer comes from our own decision making of classifying the modified fashion mnist. The first step humans make is to recognize the icons while the 2nd step is to classify based on the maximum rank of the 3 icons. Therefore, the two step approached can be incorporated into our residual network with two fully connected layers at the end. We see by the training loss that indeed a second fully connected layer performs better. Validation accuracy was not used due to the data leakage problem.

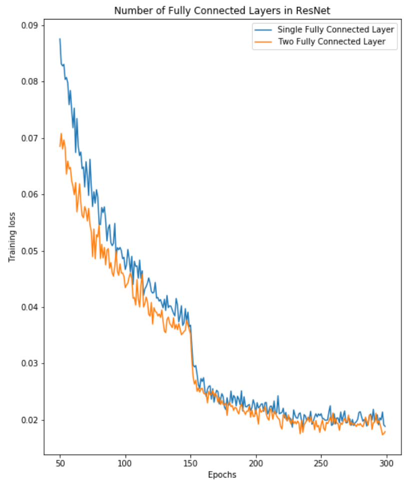

Figure 8: Investigating a second fully connected layer Zhong et al. [7]

5 Results

Initially training was done using 50,000 images whilst a 10,000 set of validation images were used to evaluate accuracy and prevent overfitting. But due to the data leakage problem explained above, a validation set was not deemed able to properly do the two tasks. It was determined to be more useful to train with the full 60,000 training set and use the training loss to evaluate model performance.

Model	Training Loss	Public Test Accuracy
AlexNet UnCropped	0.481	76.466%
AlexNet Cropped	0.239	85.100%
ResNet20- max filters:64	0.117	90.033%
ResNet32- max filters:64	0.108	90.966%
ResNet44- max filters:64	0.125	91.066%
ResNet32- max filters:256, 1FC	0.020	92.36%
ResNet33- max filters:256, 2FC	0.018	92.766%

6 Discussion and Conclusion

The first noticeable jump in our experimentation was by switching from AlexNet to ResNet. We believe that this truly shows the capabilities of skip connections and residual blocks. Image augmentation by using Kmeans to extract the icons helped reduce noise while implementing random erasing improved our models' ability to generalize to the test set. It was learned through experimentation that fine tuning the specific architecture of residual networks to suit your specific classication task also improved results.

Future work can be done on implementing improved versions of ResNets such as DenseNets, PyramidNets or Wide ResNets.

7 Statement of Work

Both YanPeng Gao and Drew Suitor contributed to the report and experimentation. YanPeng Gao also researched and set up the code base for this projected.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. pages 770–778, 06 2016. doi: 10.1109/CVPR.2016.90.

[2] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 37:448–456, 07–09 Jul 2015. URL http://proceedings.mlr.press/v37/ioffe15.html.

[3] Riaz Khan, Xiaosong Zhang, Rajesh Kumar, and Emelia Opoku Aboagye. Evaluating the performance of resnet model based on image recognition. 11 2018. doi: 10.1145/3194452.3194461.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. pages 1097–1105, 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[5] Razvan Pascanu, Tomas Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. *30th International Conference on Machine Learning, ICML 2013*, 11 2012.

[6] Andreas Veit, Michael Wilber, and Serge Belongie. Residual networks behave like ensembles of relatively shallow networks. *Advances in Neural Information Processing Systems*, 05 2016.

[7] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random erasing data augmentation. *CoRR*, abs/1708.04896, 2017. URL http://arxiv.org/abs/1708.04896.

8 Appendix

8.1 Kmeans Segmentation

```
import pickle
import matplotlib.pyplot as plt
import numpy as np
from torchvision import transforms
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from PIL import Image
import torch

data = pickle.load( open( ’./data/Test.pkl’, ’rb’ ), encoding=’bytes’)
targets = np.genfromtxt(’./data/TrainLabels.csv’, delimiter=’,’)


from sklearn import cluster

kmeans_cluster = cluster.KMeans(n_clusters=3, init=’k-means++’)
newdatalist = []
for index in range(data.shape[0]):
  arr1 = np.argwhere(data[index,:,:] > 20) #2d array where each row has x and y coordinates of non black values
  kmeans_cluster.fit(arr1)
  cluster_centers = kmeans_cluster.cluster_centers_
  newimagelist = []
  for i in range(len(cluster_centers)):
    row = int(round(cluster_centers[i][0]))
    col = int(round(cluster_centers[i][1]))
    if row<=13:
      row = 14
    if row >= 50:
      row = 49
    if col<=13:
      col = 14
    if col >= 114:
      col = 113
    newimagelist.append(data[index,row-14:row+14,col-14:col+14])
  newimage = np.concatenate(newimagelist,axis=1)
  newdatalist.append(newimage)
newdata = np.stack(newdatalist)
newdata.dump("./data/NewTest.pkl")
```

Listing 1: Image Cropping KMeans

8.2 ResNet

```python
# -*- coding: utf-8 -*-
"""ResNet.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1j8RtFmvL0Abm1i9sr30i1xZVXamfTewN

# Introduction
In the following you will see how to read the provided files for the mini-project 3.
First you will see how to read each of the provided files. Then, you will see a more elegant way of using this data for training neural networks.
"""

from google.colab import drive
drive.mount('/content/gdrive' )

# Commented out IPython magic to ensure Python compatibility.
# %cd '/content/gdrive/My Drive/Comp4900A3'
!ls './data/'

import pickle
import matplotlib.pyplot as plt
import numpy as np
from torchvision import transforms
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from PIL import Image
import torch
import math

# Read a pickle file and disply its samples
# Note that image data are stored as unit8 so each element is an integer value between 0 and 255
data = pickle.load( open( './data/NewTrain.pkl', 'rb' ), encoding='bytes')
targets = np.genfromtxt('./data/TrainLabels.csv', delimiter=',')
plt.imshow(data[1234,:,:],cmap='gray', vmin=0, vmax=256)

"""# Dataset class
*Dataset* class and the *Dataloader* class in pytorch help us to feed our own training data into the network. Dataset class is used to provide an interface for accessing all the training or testing samples in your dataset. For your convinance, we provide you with a custom Dataset that reads the provided data including images (.pkl file) and labels (.csv file).

# Dataloader class
Although we can access all the training data using the Dataset class, for neural networks, we would need batching, shuffling, multiprocess data loading, etc. DataLoader class helps us to do this. The DataLoader class accepts a dataset and other parameters such as batch_size.
"""

# Transforms are common image transformations. They can be chained together using Compose.
# Here we normalize images img=(img-0.5)/0.5
img_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5]),
    transforms.RandomErasing(p=0.5,ratio=(0.5,2.0)),
])

test_transform = transforms.Compose([
```

```
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])
])

class MyDataset(Dataset):
    def __init__(self, img_file, label_file, transform=None, idx = None):
        self.data = pickle.load( open( img_file, 'rb' ), encoding='bytes')
        self.targets = np.genfromtxt(label_file, delimiter=',')
        if idx is not None:
          self.targets = self.targets[idx]
          self.data = self.data[idx]
        self.transform = transform

    def __len__(self):
        return len(self.targets)

    def __getitem__(self, index):
        img, target = self.data[index], int(self.targets[index])
        img = Image.fromarray(img.astype('uint8'), mode='L')

        if self.transform is not None:
           img = self.transform(img)

        return img, target

# Read image data and their label into a Dataset class
dataset = MyDataset('./data/NewTrain.pkl', './data/TrainLabels.csv',transform=
    img_transform, idx=None)
train_set, val_set = torch.utils.data.random_split(dataset, [59999, 1])

batch_size = 256 #feel free to change it
trainloader = DataLoader(train_set, batch_size=batch_size, shuffle=True)

device = 'cuda' #CUDA is GPU

import torch.nn as nn
import torch.nn.functional as F
#code from:
# https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py
# https://github.com/zhunzhong07/Random-Erasing/blob/master/models/fashion/resnet.py

def conv3x3(in_planes, out_planes, stride=1):
    "3x3 convolution with padding"
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
                     padding=1, bias=False)

class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = nn.Conv2d(planes, planes * 4, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(planes * 4)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

```

```
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out

class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out

class ResNet(nn.Module):

    def __init__(self, depth=20, num_classes=10):
        super(ResNet, self).__init__()
        # Model type specifies number of layers for CIFAR-10 model
        assert (depth - 2) % 6 == 0, 'depth should be 6n+2'
        n = int((depth - 2) / 6)

        block = Bottleneck if depth >=44 else BasicBlock

        self.inplanes = 32
        self.conv1 = nn.Conv2d(1, self.inplanes, kernel_size=3, padding=1,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(self.inplanes)
        self.relu = nn.ReLU(inplace=True)
```

```
        self.layer1 = self._make_layer(block, 64, n) #maybe this should have been 32
        self.layer2 = self._make_layer(block, 128, n, stride=2)
        self.layer3 = self._make_layer(block, 256, n, stride=2)
        # self.avgpool = nn.AvgPool2d(7)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc1 = nn.Linear(256 * block.expansion, 220)
        self.fc2 = nn.Linear(220, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample))
        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.fc2(x)
        return x

net = ResNet(num_classes=10,depth=32)
net.to(device)
import torch.optim as optim

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=0.1, momentum=0.9)

# valloader = DataLoader(val_set, batch_size=batch_size, shuffle=True)
trainingloss = []
for epoch in range(400):  # loop over the dataset multiple times
    net.train()
    running_loss = 0.0

    for i, data in enumerate(trainloader, 0):

        # get the inputs; data is a list of [inputs, labels]
```

```
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # print statistics
        running_loss += loss.item()

    print('[%d] training loss: %.5f' % (epoch + 1, running_loss / i))
    trainingloss.append(running_loss / i)

    if epoch == 99:
      PATH = './saved_models/resnet6_100epochs.pth'
      torch.save(net.state_dict(), PATH)
    if epoch == 199:
      PATH = './saved_models/resnet6_200epochs.pth'
      torch.save(net.state_dict(), PATH)
    if epoch == 299:
      PATH = './saved_models/resnet6_300epochs.pth'
      torch.save(net.state_dict(), PATH)
    if epoch == 399:
      PATH = './saved_models/resnet6_400epochs.pth'
      torch.save(net.state_dict(), PATH)


    if epoch == 150:
      for param_group in optimizer.param_groups:
          param_group['lr'] = 0.01
    if epoch == 225:
      for param_group in optimizer.param_groups:
          param_group['lr'] = 0.001
    if epoch == 300:
      for param_group in optimizer.param_groups:
          param_group['lr'] = 0.0005

print('Finished Training')



testdata = MyDataset('./data/NewTest.pkl','./data/temptestlabels.csv',transform=
    test_transform, idx=None)
testloader = DataLoader(testdata, batch_size=batch_size, shuffle=False)

tempdata = pickle.load( open( './data/NewTest.pkl', 'rb' ), encoding='bytes')
print(tempdata.shape)
plt.imshow(tempdata[5555],cmap='gray', vmin=0, vmax=256)

correct = 0
total =0
loopcount=0
preds = []
net.eval()
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        preds.append(predicted.cpu().numpy())
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

```
        loopcount+=1
print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))

import numpy as np
preds = np.concatenate(preds,axis=0)

import pandas as pd
indexlist = list(range(0,10000))
df = pd.DataFrame(list(zip(indexlist, preds)),
               columns =['id', 'output'])

df.to_csv("./Submissions/resnet34morefilters_200epochs.csv",index=False)

#ResNet
```

Listing 2: ResNet

8.3 AlexNet

```python
#AlexNet
img_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])
])
class MyDataset(Dataset):
    def __init__(self, img_file, label_file, transform=None, idx = None):
        self.data = pickle.load( open( img_file, 'rb' ), encoding='bytes')
        self.targets = np.genfromtxt(label_file, delimiter=',')
        if idx is not None:
          self.targets = self.targets[idx]
          self.data = self.data[idx]
        self.transform = transform

    def __len__(self):
        return len(self.targets)

    def __getitem__(self, index):
        img, target = self.data[index], int(self.targets[index])
        img = Image.fromarray(img.astype('uint8'), mode='L')

        if self.transform is not None:
           img = self.transform(img)

        return img, target

dataset = MyDataset('./data/NewTrain.pkl', './data/TrainLabels.csv',transform=img_transform, idx=None)
old_dataset = MyDataset('./data/Train.pkl', './data/TrainLabels.csv',transform=img_transform, idx=None)

train_set, val_set = torch.utils.data.random_split(dataset, [50000, 10000])
old_train_set, old_val_set = torch.utils.data.random_split(old_dataset, [50000, 10000])

batch_size = 256 #feel free to change it
trainloader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
old_trainloader = DataLoader(old_train_set, batch_size=batch_size, shuffle=True)

import torch.nn as nn
import torch.nn.functional as F


class AlexNet(nn.Module):

    def __init__(self, num_classes=1000):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=6, stride=2, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
```

```
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x


net = AlexNet(num_classes=10)
net.to(device)
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)


valloader = DataLoader(val_set, batch_size=batch_size, shuffle=True)
training_loss = [0] * 100
validation_acc = [0] * 100
for epoch in range(100):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # print statistics
        running_loss += loss.item()

    training_loss[epoch] = (running_loss/i)
    print('[%d] training loss: %.5f' % (epoch + 1, running_loss / i))

    correct = 0
    total = 0
    with torch.no_grad():
        for data in valloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs, 1)
            # _, predicted = torch.topk(input=outputs,k=3,dim=1)
            # predicted,_ = torch.max(predicted,dim=1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    validation_acc[epoch] = (100 * correct/total)
    print('[%d] Validation acc: %.5f' % (epoch + 1, (100 * correct / total)))

```

```python
print(training_loss)
print(validation_acc)

print('Finished Training')
```
Listing 3: AlexNet