COMP 4900 Assignment 3

Tess Landry & Darren Pierre

March 25, 2020

Abstract

This assignment invesitated the use of varying supervised classification models on a modified version of the Fashion-MNIST dataset. The models used looked to identify the greatest class shown in the image, which contained 3 separate fashion items. The accuracy of a Multi-Layer Perceptron, Basic Convolutional Neural Net (CNN), LeNet, AlexNet, and VGG-16 on this dataset were compared and contrasted in this assignment. The AlexNet ended up yielding the greatest accuracy of any single model, while the greatest overall accuracy was achieved by taking the most commonly predicted class for each sample of the six best predictions.

Introduction

The purpose of this assignment was to train and validate a supervised machine learning model to classify images from a modified version of the Fashion-MNIST dataset. This modified version contained three clothing items per greyscale image. The resulting image size was also increased from the original 28x28x1 to 64x128x1 [3]. The associated class label was the label of the item in the image with the greatest class value. Five different models were attemped in this assignment: a Multi-Layer Perceptron, a Basic CNN, the LeNet, the AlexNet, and the VGG-16. The AlexNet gave the highest accuracy of any single model out of the 5 models. An averaging technique that involved taking the most predicted class for each sample of the 6 best model outputs yielded the highest overall accuracy, slightly above that of the AlexNet.

Dataset

The dataset for this assignment was based off of the Fashion-MNIST dataset. The original Fashion-MNIST dataset consisted of 28x28 greyscale images of a single clothing article from one of 10 classes [3]. This dataset was modified for the assignment to have 3 articles of clothing per image. The image size was also increased to be 64 pixels high by 128 pixels wide. The class label of each image is determined by the greatest class of the 3 clothing items in the image. The training dataset consisted of 60,000 of these images, while the test dataset was comprised of 10,000 images. As discussed below, the training data was further split into a 50,000 image training set and 10,000 image validation set for model creation, modification, and training purposes.

The data is preprocessed by being loaded into a pickle dataset, transformed using the ToTensor() function and normalized. It is then converted from image form to a single vector by the chosen models by convulutions, pooling, and other methods (described in greater detail in the next section).

Proposed Approach

In this experiement, a variety of neural net models were used for training. These models were trained and validated using a split of 50,000 images for training data and 10,000 images for validation data. The splitting of the data itself occurred by first shuffling the data, then using random selection. The use of k-fold cross validation was considered, but ultimately would have been too time-intensive for many of the models. For submission to Kaggle, these models were trained on the full training set of 60,000 images. Learning rates were started at 0.001 and then modified based on the model. The activation function used on all of the models was the ReLU function.

The first model tested was a Multi-Layer Perceptron (MLP) Neural Net. This model was used as a baseline on which to compare the other models. This model did not use any convolutional layers, and instead used only three fully connected layers.

The second model tested was a basic Convolutional Neural Net (CNN). This model used four convolutional layers, four max pooling layers, and two fully connected layers which used dropout regularization of 0.5 and 0.20, respectively. This model was used to represent a CNN that didn't follow any of the specified models.

The third model tested was the LeNet, a CNN with 3 convolutional layers, two layers of average pooling, and two fully connected layers [2]. The convolutional layers use valid convolution with filters of size 5x5, and the pooling layers have a size and stride of 2 [2]. A modified version of the LeNet was atrained in this assignment using log softmax and including dropout regularization in order to decrease variance. Since the LeNet is one of the earlier models, it is expected to outperform the Basic CNN and the MLP, but it is hypothesized that the newer models such as AlexNet and VGG-16 will yield greater accuracy.

The fourth model tested was the AlexNet, a CNN that uses dropout regularization. AlexNet has five convolutional layers and three fully connected layers [1]. The first two fully connected layers apply dropout regularization [1]. Max pooling is applied after the first, second, and fifth convlutional layers [1]. This model has more layers than the basic CNN, and applies regularization techniques to reduce overfitting. As such, it is expected to have higher accuracy and lower variance compared to the MLP, basicCNN, and LeNet. It is also expected to have a faster runtime and be less GPU-intensive compared to the VGG-16 model disccused next.

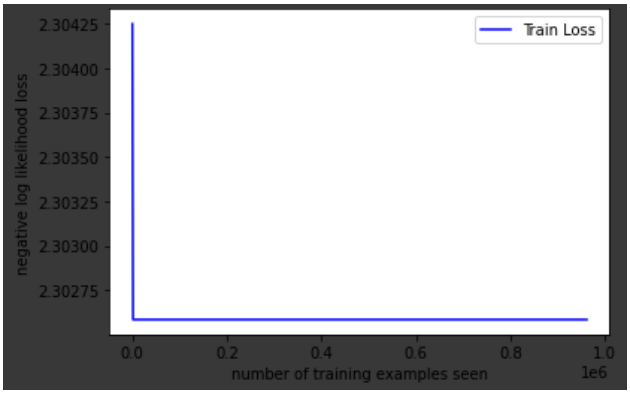
The fifth model tested was the VGG-16 Neural Net. VGG-16 is a CNN that is comprised of 13 convoluational layers and 3 fully-connected layers [4]. Max pooling with a stride of 2 occurs after the second, fourth, seventh, tenth, and thirteenth convolutional layers [4]. The convoluitional layers use same pooling to maintain image size between layers [4]. This model is known to yield high accuracy, with the downside of being extremely time and resource intensive to train [4]. It was therefore hypothesized that the VGG-16 would have the best accuracy compared to the rest of the selected models.

Finally, one attempt investigated the best 6 results that were yielded from these models. For each output sample, the most commonly outputted class was chosen for this attempt. In case of a tie, the greater of the top two tied outputs was chosen. This is due to the fact that the model is looking for the greatest class in each image, and thus choosing the greater class was the more advantageous choice in the event of a tie.

Results

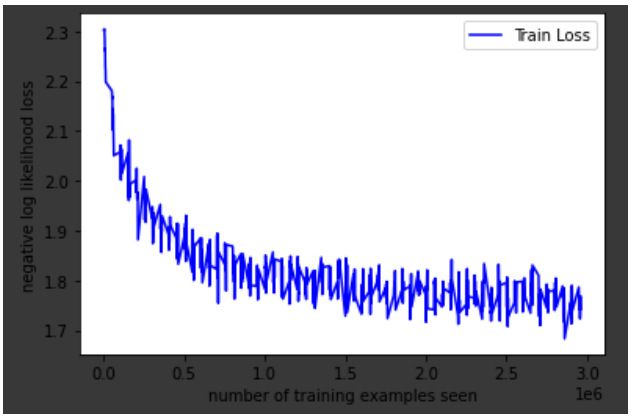
Multi-Layer Perceptron

The MLP performed the worst of the models, with a constant loss of 2.30 and accuracy of INSERT ACCURACY. The hidden size used for this test was 2000, with a batch size of 256.



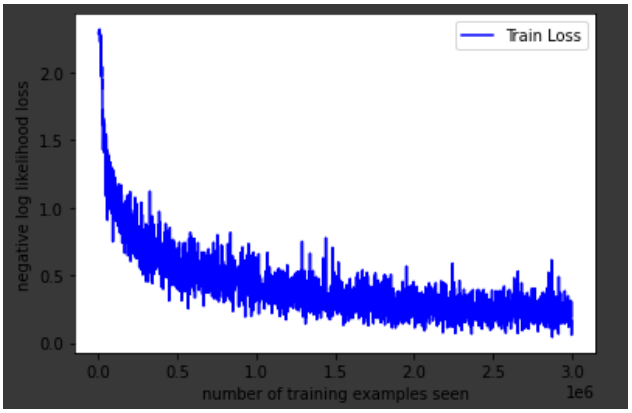
Basic CNN

The basic CNN yielded an accuracy of around 76.5%. The parameters that yielded this accuracy were a learning rate of 0.001, a batch size of 256, and 60 epochs. By the end of the 60 epochs, the loss had decreased to around 1.74, as seen in the graph below. The runtime of this version of the model was 21 minutes.



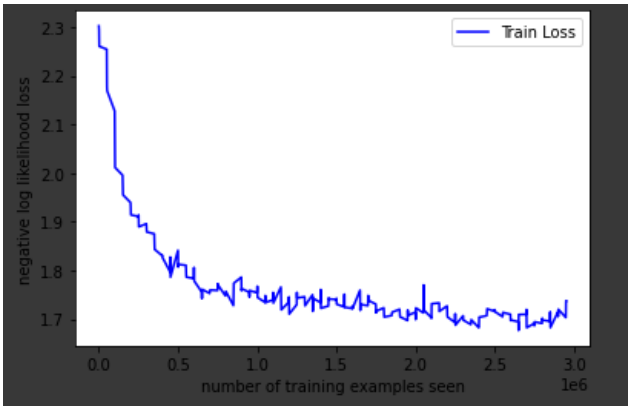
LeNet

The LeNet yielded the lowest loss of any of the tested models. This model functioned best with a learning rate of 0.001. It was first modified to use log softmax rather than regular softmax. As predicted, this caused a significant decrease in loss, however also yielded a low accuracy. It was hypothesized that this was due to overfitting. The LeNet was then modified to include dropout regularization in order to decrease variance. At dropout levels of 0.25, accuracy increased. Attempting dropout values of 0.5 yielded even higher accuracy. Decreasing the batch size also increased accuracy and decreased variance. Originally, the batch size was 256, and was decreased down to 64 to obtain the highest accuracy from this model. This version of the LeNet resulted in a validation set accuracy of 78.5%, and a Kaggle test accuracy of approxamately 76.1% after 60 Epochs. The runtime of this model was 39 minutes, and the loss obtained in the final epoch was between 0.30 and 0.05, as seen in the graph below.



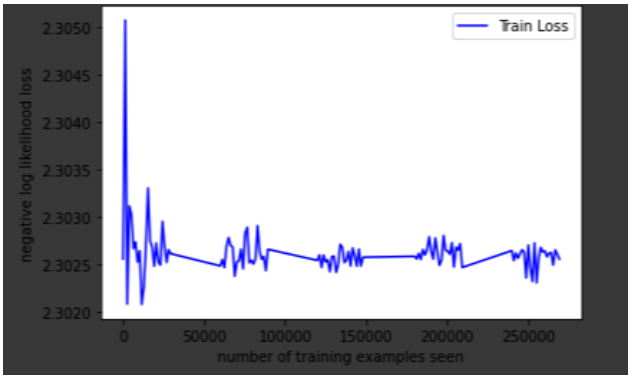
AlexNet

The AlexNet yielded the highest accuracy of the tested models, with a Kaggle test result of 84.1%. This model used a batch size of 1024, a learning rate of 0.001, and 60 epochs. In order to achieve this accuracy, the AlexNet was modified from its original form to include dropout regularization at all 3 fully connected layers with values of 0.25 for the first layer and 0.20 for the last two layers. Batch normalization was also attempted on the original model, however in combination with the dropout regularization, it had a negative effect on accuracy and was not used in this modified version. This model took 34 minutes to run and had a loss at the final epoch of roughly 1.73.



VGG-16

The VGG-16 model took the longest to train of any model, and only yielded a near-random accuracy. This was further emphasized by recording the class estimates and seeing that all samples were classified to be from a single class (the value of which varied in different runs). Different learning rates were attempted (anywhere from 0.1 to 0.000001) and different batch sizes (512 to 64), but the change seemed to have no impact. Part of the reasoning for this could be that the original VGG-16 model was trained with extremely strong GPUs over a long period of time. As such, the current model used may not have been trained long enough, and may have required more extensive resources to train a smaller learning rate over a more extended period of time. The near-stagnant loss for this model can be seen below. This was seen across different values for learning rate, epochs, and batch size.



Summary of Results

Model	Epochs	Accuracy
MLP	20	0
Basic CNN	60	76.5
LeNet	60	76.1
AlexNet	60	84.1
VGG-16	60	≈ 10
Top 6 Average	-	84.8

Table 1: Top accuricies yielded for each model. Learning rate was 0.001 and activation was ReLU unless otherwise indicated. Accuracies are from Kaggle where applicable, and are otherwise from the validation set.

The top Kaggle submission (with accuracy of 84.8%) was achieved by using a modified model inspired by the final step of bagging. The output of the best 6 models was obtained, and then the most commonly outputted class was chosen as the final output. If multiple classes were the highest chosen class (for example, if 3 models outputted class 4 and 3 outputted class 6), then the largest label was chosen since it is possible that the other assigned class was simply another clothing article in the image, and not the maximum class in the image.

In addition to these results, a transfer learning model was also attempted. Due to limitations of google colab and GPU resources, it was not feasable and could not be fully tested.

Discussion and Conclusions

The three most successful models in this assignment were the Basic CNN, the LeNet, and the AlexNet. The MLP and the VGG-16 were the least successful models for this experiement, and were not used at all in the averaging section due to their near-random outputs. The singular model which yielded the highest accuracy in this experiement was the AlexNet. The highest total accuracy was achieved by using the averaging model described in the 'Summary of Results' section, with an accuracy of 84.8%. The fact that this percentage was only slightly higher than our next best percentage (of 84.1%) seems to suggest that our models have been misclassifying many of the same samples.

The use of log softmax resulted in lower losses compared to just softmax. The tradeoff for this was that, over a greater number of epochs, log softmax was far more likely to overfit to the training data. Models which used regular softmax seemed to perform better over a greater number of epochs (such as 60) compared to log softmax. Future investigations could look into the effect of using log softmax over a beginning set of training epochs (for example, 20) and then switching to softmax after this for any remaining epochs.

As predicted, the use of regularization (such as dropout) greatly reduced model overfitting and increased accuracy. Modifiying the classic version of the LeNet to include dropout regularization made it more competitive in comparison with the best model.

The high runtimes recorded in the above results indicate that K-Fold validation would have been extremely costly time-wise. As such, a single, random train-validtion split was performed in order to train all of the data. Future attempts could look into a middle ground between these two options to get a better train-validation model.

Future investigations could compare the accuricies of the pre-trained versions of the neural nets used in this assignment. Specifically, it would be worth investigating the default VGG-16 model with pretrained weights and seeing if it outperforms any models used in this investigation. In order to do this, the images would need to be converted to 3 channels to mimic RGB, and resized to 224x224 [4]. Finally, additional attempts could also look further into the option of transfer learning. Having undergone a similar problem to the VGG-16 model of being restricted by GPU availability, more resources would need to be allocated to make this attempt worthwhile.

Statement of Contributions

Tess Landry was responsible for the LeNet, the VGG-16, the written report, the final averaging, and the random sampling validation. **Darren Pierre** was responsible for the code for the loading, converting, and management of the datasets. He was also responsible for the original train and test code, the MLP, the Basic CNN, the AlexNet, transfer learning, and the code to yield a csv for Kaggle.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. Technical report, University of Toronto, 2012.

[2] Max Pechyonkin. Key Deep Learning Architectures: LeNet-5, 2018.

[3] Kashif Rasul and Han Xiao. Fashion-MNIST.

[4] Muneeb ul Hassan. VGG16 - Convolutional Network for Classification and Detection, 2018.

Appendix

Set Up

```
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True )
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 torch.utils.data import random_split
from PIL import Image
import torch
import pandas as pd
from google.colab import files
from torch.utils.tensorboard import SummaryWriter
from sklearn.model_selection import KFold
from sklearn.utils import shuffle
!pip install ipython-autotime
%load_ext autotime

# Writer will output to ./runs/ directory by default
writer = SummaryWriter()
```

```
ROOT_DIR="./gdrive/My Drive/Colab Notebooks/COMP4900A3/Data/"
TRAIN_P=ROOT_DIR+"Train.pkl"
TRAIN_L=ROOT_DIR+"TrainLabels.csv"
TEST_P=ROOT_DIR+"Test.pkl"
Kaggle=ROOT_DIR + "kaggleSubmission.csv"
# 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( TRAIN_P, 'rb' ), encoding='bytes')
targets = np.genfromtxt(TRAIN_L, delimiter=',')

for x in range(10):
  plt.imshow(data[877,:,:])
print(data[0,:,:].shape)
```

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

```
# img_file: the pickle file containing the images
# label_file: the .csv file containing the labels
# transform: We use it for normalizing images (see above)
# idx: This is a binary vector that is useful for creating training and validation set.
# It return only samples where idx is True
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

# Data set used for kaggle submission
class kaggleDS(Dataset):
    def __init__(self, img_file, transform=None,):
        self.data = pickle.load( open( img_file, 'rb' ), encoding='bytes')
        self.transform=transform


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

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

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

        return img

dataset = MyDataset(TRAIN_P, TRAIN_L,transform=img_transform, idx=None)
batch_size = 256
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

Train & Test For Full Dataset

```
def train(model,epoch,optimizer,use_gpu=False):
    train_losses = []
    train_counter = []
    if use_gpu:
        model.cuda() # put our model on the gpu
    model.train()

    loss_f = torch.nn.CrossEntropyLoss() # Loss function
    for epoch in range(1,epoch+1):
        for batch_idx,(data) in enumerate(dataloader):
        [data,label]=data
        if use_gpu:
               data, label = data.cuda(), label.cuda()
        model.zero_grad() # sets gradients to 0 before loss calc.
```

```
        output = model(data) # model makes predictions
        loss = loss_f(output, label) # calc and grab the loss value

        loss.backward() # apply this loss backwards thru the network's parameters
        optimizer.step() # attempt to optimize weights to account for loss/gradients

        if batch_idx % 20 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
            epoch, batch_idx * len(data), len(dataloader.dataset),
            100. * batch_idx / len(dataloader), loss.item()))
            train_losses.append(loss.item())
            train_counter.append(
            (batch_idx*64) + ((epoch-1)*len(dataloader.dataset)))
            torch.save(model.state_dict(), '/model.pth')
            torch.save(optimizer.state_dict(), '/optimizer.pth')

    fig = plt.figure()
    plt.plot(train_counter, train_losses, color='blue')
    plt.legend(['Train Loss'], loc='upper right')
    plt.xlabel('number of training examples seen')
    plt.ylabel('negative log likelihood loss')
    fig.show()
```

```
def test(model):
    correct = 0
    total = 0
    output=[]
    model.eval()
    model.cuda()
    classPredictions = [0,0,0,0,0,0,0,0,0,0]
    with torch.no_grad():
        for data in dataloader:
            [data,label]=data
            data=data.cuda()
            label=label.cuda()
            output = model(data)

            for idx, i in enumerate(output):

                if torch.argmax(i) == label[idx]:
                    correct += 1
                total += 1
                classPredictions[torch.argmax(i)] += 1

    print("Accuracy: ", round(correct/total, 3))
    print("Class Predictions: ")

    for i in range(len(classPredictions)):
        print(i, classPredictions[i])
```

```
# function to test for submission
def testKaggle(model):
    fileName="kaggle submission.csv"
    submissionPD=pd.read_csv(Kaggle,header=0)
    output=[]
    kaggle_ds= kaggleDS(TEST_P,img_transform)
    kaggleDSLoader=DataLoader(kaggle_ds,10)
```

```
    for data in kaggleDSLoader:
        data=data.cuda()
        p_labels=model(data)
        output.append([torch.argmax(x) for x in p_labels])
    output=torch.flatten(torch.IntTensor(output),end_dim=1)
    output=torch.reshape(output,(1,10000))

    answer=list()

    for x in range(len(output[0])):
        answer.append(output[0][x].item())
    submissionPD.output=answer
    submissionPD.to_csv(fileName,index=False)
    files.download(fileName)
```

```
# Checking GPU availability
use_gpu = torch.cuda.is_available()
```

Train & Test For Train / Validation Split

```
def train2(model,epoch,optimizer,dataLoaderToUse, use_gpu=False):
    train_losses = []
    train_counter = []
    if use_gpu:
        model.cuda() # put our model on the gpu
    model.train()

    loss_f = torch.nn.CrossEntropyLoss() # Loss function
    for epoch in range(1,epoch+1):
        for batch_idx,(data) in enumerate(dataLoaderToUse):
        [data,label]=data
        if use_gpu:
               data, label = data.cuda(), label.cuda()
        model.zero_grad() # sets gradients to 0 before loss calc.
        output = model(data) # model makes predictions
        loss = loss_f(output, label) # calc and grab the loss value

        loss.backward() # apply this loss backwards thru the network's parameters
        optimizer.step() # attempt to optimize weights to account for loss/gradients

        if batch_idx % 20 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
            epoch, batch_idx * len(data), len(dataLoaderToUse.dataset),
            100. * batch_idx / len(dataLoaderToUse), loss.item()))
            train_losses.append(loss.item())
            train_counter.append(
            (batch_idx*64) + ((epoch-1)*len(dataLoaderToUse.dataset)))
            torch.save(model.state_dict(), '/model.pth')
            torch.save(optimizer.state_dict(), '/optimizer.pth')

    fig = plt.figure()
    plt.plot(train_counter, train_losses, color='blue')
    plt.legend(['Train Loss'], loc='upper right')
    plt.xlabel('number of training examples seen')
    plt.ylabel('negative log likelihood loss')
    fig.show()
```

```
def test2(model, dataLoaderToUse):
    correct = 0
    total = 0
    output=[]
    model.eval()
    model.cuda()
    classPredictions = [0,0,0,0,0,0,0,0,0,0]
    with torch.no_grad():
        for data in dataLoaderToUse:
            [data,label]=data
            data=data.cuda()
            label=label.cuda()
            output = model(data)

            for idx, i in enumerate(output):

                if torch.argmax(i) == label[idx]:
                    correct += 1
                total += 1
                classPredictions[torch.argmax(i)] += 1

    print("Accuracy: ", round(correct/total, 3))
    print("Class Predictions: ")

    for i in range(len(classPredictions)):
        print(i, classPredictions[i])

    return round(correct/total, 3)
```

Multi-Layer Perceptron

```
        import torch.nn as nn
        import torch.nn.functional as F
        class Net(nn.Module):
            def __init__(self, hidden_size):
                INPUT_SIZE=64*128
                OUTPUTSIZE=10
                super(Net, self).__init__()               # Inherited from the parent class
                    nn.Module
                self.fc1 = nn.Linear(INPUT_SIZE, hidden_size) # nn.Linear multiples input by weights
                    and add a bias
                self.fc2 = nn.Linear(hidden_size, hidden_size)
                self.fc3 = nn.Linear(hidden_size, OUTPUTSIZE)

            def forward(self, x):
                x=torch.flatten(x,start_dim=1)   #flatten our image to a vector                #
                    Forward pass: stacking each layer together
                out = F.relu(self.fc1(x))     #activation function uses relu
                out = F.relu(self.fc2(out))
                out = F.relu(self.fc3(out))
                return out
```

Basic CNN

```
IMG_DIM=(64,128)
class BasicCNN(nn.Module):
  def __init__(self):
    super(BasicCNN,self).__init__()
    self.conv1= nn.Conv2d(in_channels=1,out_channels=80,kernel_size=7,stride=1,padding=1)
    self.conv2= nn.Conv2d(80,40,5)
    self.conv3= nn.Conv2d(40,20,3)
    self.conv4= nn.Conv2d(20,10,3)
    self.pool1=nn.MaxPool2d(2)
    self.drop1=nn.Dropout(0.5)
    self.drop2=nn.Dropout(0.20)
    self.fc1 = nn.Linear(50, 128)
    self.fc2 = nn.Linear(128, 10)

  def forward(self, X):
    X = F.relu(self.pool1(self.conv1(X)))
    X = F.relu(self.pool1(self.conv2(X)))
    X = F.relu(self.pool1(self.conv3(X)))
    X = F.relu(self.pool1(self.conv4(X)))
    X = X.flatten(1)
    X = F.relu(self.drop1(self.fc1(X)))
    X = self.drop2(self.fc2(X))
    return F.softmax(X)
```

LeNet

```
IMG_DIM=(64,128)
class LeNetCNN(nn.Module):
  def __init__(self):
    super(LeNetCNN, self).__init__()
    self.conv1 = nn.Conv2d(1, 6, kernel_size=5) #out: 60 x 124 x 6
    #Max Pool out: 30 x 62 x 6
    self.conv2 = nn.Conv2d(6, 16, kernel_size=5) #out: 26 x 58 x 16
    #Max Pool out: 13 x 29 x 16
    self.conv3 = nn.Conv2d(16, 120, kernel_size=5) #out: 9 x 25 x 120

    #self.dropout1 = nn.Dropout(0.7)
    self.dropout2 = nn.Dropout(0.7)

    self.fc1 = nn.Linear(27000, 120)
    self.fc2 = nn.Linear(120, 10)

  def forward(self, x):

    x = F.relu(F.avg_pool2d(self.conv1(x), 2))
    #x = nn.BatchNorm2d(x)
    x = F.relu(F.avg_pool2d(self.conv2(x), 2))
    #x =nn.BatchNorm2d(x)
    x = F.relu(self.conv3(x), 2)


    # Imaginary Layer (11*25*120 = 4096)
    x = x.view(-1, 27000)
```

```
    #Feed forward layers
    x = self.dropout2(F.relu(self.fc1(x)))
    #x = F.relu(self.dropout2(self.fc2(x)))
    x = F.relu(self.fc2(x))
    #Output layer
    return F.log_softmax(x)
```

AlexNet

```
#Original
class Alexnet_Original(nn.Module):
  def __init__(self):
    super(Alexnet_Original,self).__init__()
    self.conv1= nn.Conv2d(in_channels=1,out_channels=96,kernel_size=7,stride=1,padding=1)
    self.conv2= nn.Conv2d(96,256,5)
    self.conv3= nn.Conv2d(256,384,3)
    self.conv4= nn.Conv2d(384,384,3)
    self.conv5= nn.Conv2d(384,256,3)
    self.pool1=nn.MaxPool2d(2)
    self.batch1=nn.BatchNorm2d(96)
    self.batch2=nn.BatchNorm2d(256)

    self.drop1=nn.Dropout(0.25)
    self.drop2=nn.Dropout(0.20)
    self.fc1 = nn.Linear(8448, 800)
    self.fc2 = nn.Linear(800, 200)
    self.fc3 = nn.Linear(200, 10)


  def forward(self, X):
    X = F.relu(self.pool1(self.batch1(self.conv1(X))))
    X = F.relu(self.pool1(self.batch2((self.conv2(X)))))
    X = F.relu((self.conv3(X)))
    X = F.relu((self.conv4(X)))
    X = F.relu(self.pool1(self.conv5(X)))
    X = X.flatten(1)
    X = F.relu((self.fc1(X)))
    X = F.relu((self.fc2(X)))
    X = self.drop2(self.fc3(X))
    return F.softmax(X)
```

```
# Modified Alexnet
class Alexnet_M(nn.Module):
  def __init__(self):
    super(Alexnet_M,self).__init__()
    self.conv1= nn.Conv2d(in_channels=1,out_channels=40,kernel_size=7,stride=1,padding=1)
    self.conv2= nn.Conv2d(40,80,5)
    self.conv3= nn.Conv2d(80,160,3)
    self.conv4= nn.Conv2d(160,320,3)
    self.conv5= nn.Conv2d(320,400,3)
    self.pool1=nn.MaxPool2d(2)
    self.drop1=nn.Dropout(0.25)
    self.drop2=nn.Dropout(0.20)
    self.fc1 = nn.Linear(13200, 800)
    self.fc2 = nn.Linear(800, 200)
    self.fc3 = nn.Linear(200, 10)
```

```
  def forward(self, X):
    X = F.relu(self.pool1(self.conv1(X)))
    X = F.relu(self.pool1(self.conv2(X)))
    X = F.relu((self.conv3(X)))
    X = F.relu((self.conv4(X)))
    X = F.relu(self.pool1(self.conv5(X)))
    X = X.flatten(1)
    X = F.relu(self.drop1(self.fc1(X)))
    X = F.relu(self.drop2(self.fc2(X)))
    X = self.drop2(self.fc3(X))
    return F.softmax(X)
```

```
# Neural net for experimenting with layers and hyperparameters
class Temp(nn.Module):
  def __init__(self):
    super(Temp,self).__init__()
    self.conv1= nn.Conv2d(in_channels=1,out_channels=96,kernel_size=7,stride=1,padding=1)
    self.conv2= nn.Conv2d(96,256,5)
    self.conv3= nn.Conv2d(256,384,3)
    self.conv4= nn.Conv2d(384,384,3)
    self.conv5= nn.Conv2d(384,256,3)
    self.pool1=nn.MaxPool2d(2)
    self.batch1=nn.BatchNorm2d(96)
    self.batch2=nn.BatchNorm2d(256)

    self.drop1=nn.Dropout(0.25)
    self.drop2=nn.Dropout(0.20)
    self.fc1 = nn.Linear(8448, 800)
    self.fc2 = nn.Linear(800, 200)
    self.fc3 = nn.Linear(200, 10)


  def forward(self, X):
    X = F.relu(self.pool1(self.batch1(self.conv1(X))))
    X = F.relu(self.pool1(self.batch2((self.conv2(X)))))
    X = F.relu((self.conv3(X)))
    X = F.relu((self.conv4(X)))
    X = F.relu(self.pool1(self.conv5(X)))
    X = X.flatten(1)
    X = F.relu((self.fc1(X)))
    X = F.relu((self.fc2(X)))
    X = self.drop2(self.fc3(X))
    return F.softmax(X)
```

VGG-16

```
IMG_DIM=(64,128)
class VGGNet(nn.Module):
  #Define Layers
  def __init__(self):
    super(VGGNet,self).__init__()
    #If this does not work, change 1 -> 3 and convert our greyscale to have 3 channels
    self.conv1_1 = nn.Conv2d(1, 64, kernel_size=3, padding=1)
    self.conv1_2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
```

```
        self.conv2_1 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv2_2 = nn.Conv2d(128, 128, kernel_size=3, padding=1)

        self.conv3_1 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.conv3_2 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.conv3_3 = nn.Conv2d(256, 256, kernel_size=3, padding=1)

        self.conv4_1 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
        self.conv4_2 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.conv4_3 = nn.Conv2d(512, 512, kernel_size=3, padding=1)

        self.conv5_1 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.conv5_2 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.conv5_3 = nn.Conv2d(512, 512, kernel_size=3, padding=1)

        #Modified Feed Forward Layers
        self.fc1 = nn.Linear(4096, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, 10)


    def forward(self, x):
        #x: 64, 128, 1 (change to 3 if this doesnt work and modify input files)

        x = F.relu(self.conv1_1(x))
        x = F.relu(F.max_pool2d(self.conv1_2(x), 2))
        #Output: 32, 64, 64

        x = F.relu(self.conv2_1(x))
        x = F.relu(F.max_pool2d(self.conv2_2(x), 2))
        #Output: 16, 32, 128

        x = F.relu(self.conv3_1(x))
        x = F.relu(self.conv3_2(x))
        x = F.relu(F.max_pool2d(self.conv3_3(x), 2))
        #Output: 8, 16, 256

        x = F.relu(self.conv4_1(x))
        x = F.relu(self.conv4_2(x))
        x = F.relu(F.max_pool2d(self.conv4_3(x), 2))
        #Output: 4, 8, 512

        x = F.relu(self.conv5_1(x))
        x = F.relu(self.conv5_2(x))
        x = F.relu(F.max_pool2d(self.conv5_3(x), 2))
        #Output: 2, 4, 512

        # Imaginary Layer (2*4*512 = 4096)
        x = x.view(-1, 4096)


        #Feed forward layers
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        #Output layer
        return F.softmax(x)
```

Train / Validation Split

```
#Split into train and validation
train_data, test_data = random_split(dataloader.dataset, (50000, 10000))

batch_size = 512 #Change this as needed

train_dataLoader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
test_dataLoader = DataLoader(test_data, batch_size=batch_size, shuffle=True)


valModel = LeNetCNN()
optimizer=torch.optim.Adam(valModel.parameters(),lr=0.001)
train2(valModel,30,optimizer,train_dataLoader,use_gpu)
testAccuracy = test2(valModel, test_dataLoader)
testKaggle(valModel)
```

Training, Saving, and Loading Model

```
def trainModel():
    model=Temp()
    optimizer=torch.optim.Adam(model.parameters(),lr=5e-5) #5e-4
    train(model,20,optimizer,use_gpu)
    test(model)

def saveModel(model,name="bestModel.dict"):
    PATH= ROOT_DIR+name
    torch.save(model.state_dict(), PATH)
    print(f"your model has been saved to {PATH}")

def loadModel(ModelClass,name):
    PATH=ROOT_DIR + name
    model = ModelClass()
    model.load_state_dict(torch.load(PATH))
    return model
```

Transfer Learning

```
    from PIL import Image
    """
    This functions is used to transform a PIL image into a rgb image
    We created this function because we need to format our images into something
    """
    def rgb(pil):
      copy=np.zeros((64,128,3))
      a = np.asarray(pil)

      for x in range(64):
        for y in range(128):
          value=a[x][y]
          copy[x][y]=[value,value,value]

      return Image.fromarray(copy.astype('uint8'), 'RGB')
```

```
transform_learning_transform=transforms.Compose([
    transforms.Lambda(rgb),
    transforms.Resize(224),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
```

```
import torchvision.models as models
alexnet = models.alexnet(True)
dataset = MyDataset(TRAIN_P, TRAIN_L,transform=transform_learning_transform, idx=None)
batch_size = 32
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```
    from collections import OrderedDict
    classifier = nn.Sequential(OrderedDict([
                          ('fc1', nn.Linear(9216, 4096)),
                          ('relu', nn.ReLU()),
                          ('fc2', nn.Linear(4096, 10)),
                          ('output', nn.LogSoftmax(dim=1))
                          ]))
```

```
    alexnet.classifier=classifier
    optimizer=torch.optim.Adam(alexnet.parameters(),lr=5e-5) #5e-4

    train(alexnet,10,optimizer,use_gpu)
    test(alexnet)
```

Averaging Code

```
    from google.colab import drive
    drive.mount('/content/drive')
    ROOT_DIR="/content/drive/My Drive/Colab Notebooks/COMP4900A3/Data/ens/"
    ENS_1=ROOT_DIR+"ens761.csv"
    ENS_2=ROOT_DIR+"ens779.csv"
    ENS_3=ROOT_DIR+"ens798.csv"
    ENS_4=ROOT_DIR+"ens820.csv"
    ENS_5=ROOT_DIR+"ens828.csv"
    ENS_6=ROOT_DIR+"ens841.csv"
```

```
    import os
    import pandas as pd
    from pandas import DataFrame
    import numpy as np
    from google.colab import files

    #Convert to pandas
    ens1 = pd.read_csv(ENS_1)
    ens2 = pd.read_csv(ENS_2)
    ens3 = pd.read_csv(ENS_3)
    ens4 = pd.read_csv(ENS_4)
    ens5 = pd.read_csv(ENS_5)
    ens6 = pd.read_csv(ENS_6)
```

```
#get IDs
ids = ens1.drop('output', 1)

#drop our IDs
ens1 = ens1.output
ens2 = ens2.output
ens3 = ens3.output
ens4 = ens4.output
ens5 = ens5.output
ens6 = ens6.output
```

```
predictions = []

#Loop through each row of the dataset outputs
for i in range(len(ens1)):
  classTotals = [0,0,0,0,0,0,0,0,0,0]

  #Add 1 for each predicted value
  indexToAdd = ens1[i]
  classTotals[indexToAdd] += 1

  indexToAdd = ens2[i]
  classTotals[indexToAdd] += 1

  indexToAdd = ens3[i]
  classTotals[indexToAdd] += 1

  indexToAdd = ens4[i]
  classTotals[indexToAdd] += 1

  indexToAdd = ens5[i]
  classTotals[indexToAdd] += 1

  indexToAdd = ens6[i]
  classTotals[indexToAdd] += 1

  #Choose the GREATEST of the max values, since we are looking for the highest value in pic
  maxValue = max(classTotals)

  indexToChoose = 0
  for j in range(len(classTotals)):
    if (classTotals[j] == maxValue):
      indexToChoose = j

  #if we only have one of each, choose from best model
  if(maxValue == 1):
    indexToChoose = ens6[i]

  predictions.append(indexToChoose)

#check accuracy compared to best file (ens6)
#Get accuracy
total = len(ens6)
correct = 0
k=0
while k < total:
  if predictions[k] == ens6[k]:
```

```
    correct = correct+1
  k=k+1

print("Accuracy compared to best predictions is: " + str(correct/float(total)))

ids['output'] = predictions

predictTotals = [0,0,0,0,0,0,0,0,0,0]
for prediction in predictions:
    predictTotals[prediction] +=1
print(predictTotals)

#Convert to CSV
#ids.to_csv('kaggleSubmission.csv', index=False)
#files.download('kaggleSubmission.csv')
```