COMP 4900A Mini-Project 3

Connor Stewart (101041125)
Michael Kazman (101037568)
Trevor Johns(101036054)

Majid Komeilli

December 9th, 2020

1 Abstract

Contained in the report herein is the results of the classification of neural network models with validation pipelines. The neural-network classifier was implemented to sort through sets of images representing groups of clothing items and determine their associated values, using PyTorch's Neural Network module [2]. Further on in this report, the analysis of the classifier will be discussed with regards to the network run-time of the application. Functionally - when training the network - the model with the lowest validation loss is returned in the program's output. Using the model with the lowest validation loss ensures the classification on the testing set is calculated against the most accurate models correctly. The program will be used in a course-based Kaggle competition to determine the best application programming. The program's source code was built with the spiral development model in mind, as this allows for increased modularity allowing various layer combinations in the neural network, optimizers, and loss functions to be swapped in with ease. Modularity provides the additional benefit of allowing components to be quickly changed, which reduces the risk of overfitting the model. In the following paragraphs, we will elaborate on the program's data sets, results, and significance.

2 Introduction

The classifier created was intended to sort through images consisting of multiple clothing items and match the images to a corresponding price. A convolutional neural network (CNN) made with Pythons PyTorch's Neural Network module was used to implement the stated program objective. The most accurate model will then provide predictions for an unlabelled testing set and be subsequently posted to Kaggle. These predictions will be evaluated on 30% of the testing set, alongside other research groups, to provide a leader-board score. Many findings have been made from this, such as the fact that overfitting is a problem with convolutional neural networks. Additionally, further insights into the maximum possible accuracy scores have been made and will be elaborated on further in the Discussion and Conclusion section of this report. Aspects of the program will be analyzed - such as accuracy and run-time - and will be further discussed in the following paper. To achieve the required precision, two datasets were used when training and testing the classifiers. The training set consist of two files, the labels and the targets. The first being the image files, and the second associating the image with the clothes total cost. The testing set only consists of the image files as opposed to the corresponding price labels, and further details regarding both datasets will be discussed in the following section. The goal of this application is to determine the total cost of the clothing items in each image. The section below speaks of the provided training and testing sets, along with the pre-processing steps needed for an increased score.

3 Datasets

The training dataset consists of two separate files, Train.pkl, and TrainLabels.csv. The former consists of 60,000 grey-scale images, each with a respective resolution of 64×128 totaling 8192 pixels. Each image consists of 3 objects surround by black pixels. The items within the image file represent the first five clothing items from the dataset Fashion-MNIST [5]. Each article of clothing has an associated price, with a T-shirt/top at \$1, trousers at \$2, pullover at \$3, dress at \$4, and a coat at \$5. The total cost of the image's items can range anywhere between \$5 and \$13, with the cheapest combinations being two t-shirts and a pullover, or two pairs of trousers and a t-shirt. The most expensive variants are two coats and a pullover, or two dresses and a coat. The two images below in Figure 1 illustrate the clothing articles provided within the dataset.

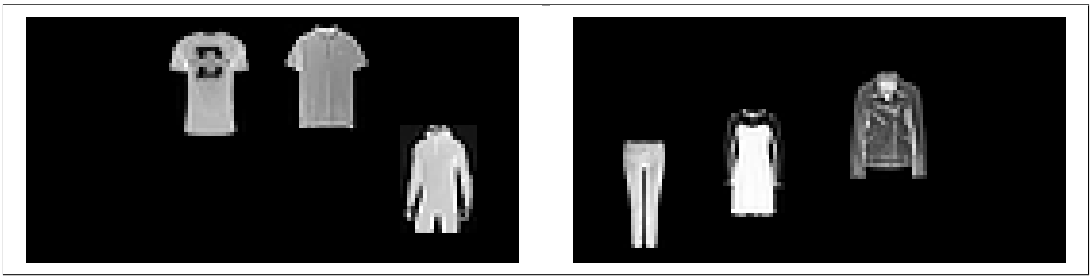

Figure 1: The images above contain, in order, two t-shirts, a pullover, trousers, a dress, and a jacket. The samples total \$5 and \$11 respectively.

The TrainLabels.csv consists of two primary columns, with the first being the identification number of the image sample and the second being the total cost of that image. The identification column ("id") consists of simple integer values, which are zero-indexed and arranged in increasing order from 0 to 59,999. The second column ("class") consists of integers representing the dollar figures of the 3 combines clothing articles. The training and testing scores were calculated on the entire dataset and corresponding labels using randomized samples based on a fixed seed for reproducible results.

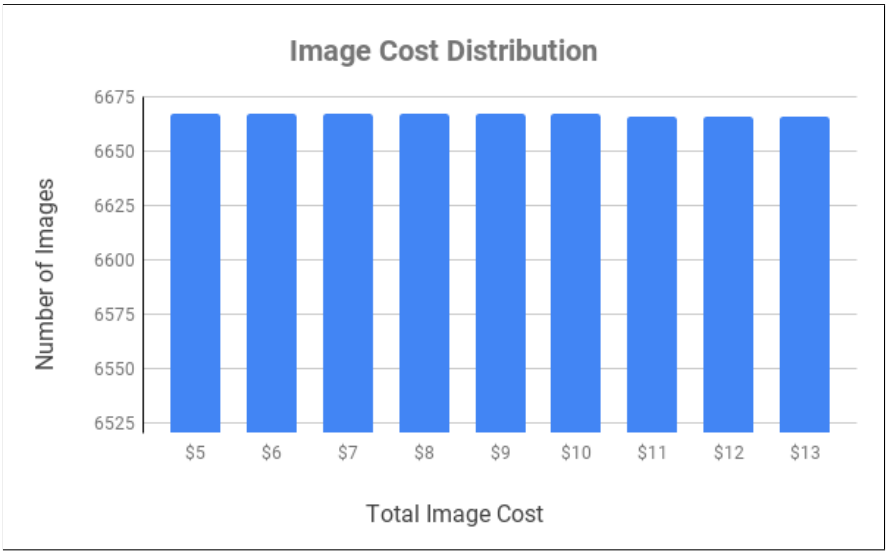

Figure 2: Histogram of Image Cost

The training dataset is used for training the models and calculating accuracy scores, while the test dataset is used to generate predictions for each image, which will be submitted on Kaggle to determine correctness.

4 Proposed Approach

Various machine learning algorithms were implemented and tested for this assignment through the use of scikit-learn. Unfortunately, scikit-learn did not support processing images with GPU support, resulting in very time-consuming CPU based calculations. This was tested using various methods like decision trees, clustering algorithms (KMeans, EM), PCA (for dimension reduction), and SVM algorithms (SVC and LinearSVC). However, all these methods took too much time to run, leading to the groups' conclusion that scikit-learn was not a viable option for this project. Since this was not a viable approach for the model, an alternate library, namely PyTorch, had to be used.

The machine learning model was not responsive to changes made in it, and convolutional layers, fully connected layers, and pooling layers had little impact on the loss and accuracy scores. Ultimately, this made it challenging to find ways to fine-tune the model to get higher accuracy scores.

For testing, the primary approach used was to edit various parameters for the neural network algorithm. These parameters include the learning rate, the batch size, and the dropout rate. Adjusting these parameters allowed the neural network to learn from the datasets in different ways, resulting in unique patterns.

The model was designed using a 3 convolutional layers. Each of which utilized max-pooling, and a ReLu activation function. Each contained a kernel size of 5, stride of 1, and padding of 1 (excluding the last layer which is 0). The convolutional layers were sized with 64, 128, and 192 channels respectively. Following a re-shape in the forward pass, the model contained 4 fully-connected layers. Each connected layer utilized dropout of 0, 50%, and 25% respectively, as well as a ReLu activation function. The fully-connected layers consisted of 192, 128, 64, and 9 neurons respectively. The final layer had an output of 9 classes to represent each article of clothing.

The architecture of the above program makes use of a convolutional neural network (CNN). A CNN is a deep learning algorithm which takes an input image, assigns importance to various aspects of the image, and differentiates the images [3]. The preprocessing stages in the CNN are much lower than other classification algorithms making it advantageous and uses data structures analogous to that of the connectivity of neurons in the human brain [3]. For the CNN, we used an architecture based of AlexNet [4] and VGG style where each convolution uses a ReLU which is then fed into a pooling layer using the maxpooling method [1]. There are three of these layers, and each of the layers doubles its depth, similar to AlexNet [4]. After all the convolutional layers, the data is passed into a set of four fully connected layers. Finally, the data is passed through a logarithmic softmax activation function [1].

The group tested multiple different methodologies to determine the final approach used in the neural network. Permutations for the neural networks' coefficients were tested to find the most accurate neural network model. For example, the group experimented with changing many parameters, such as the number of epochs, dropout rate, batch size, and the learning rate. Moreover, the structure of the neural network was tuned by changing the number of convolution layers, fully connected layers, stride, padding, and pooling.

Matplotlib was used to visualize both the accuracy and loss values for each batch of the training and validation sets to resolve some of the issues encountered when making the neural network. Using that data, we were able to construct graphs to visualize what the neural network was doing. The graphs from matplotlib helped by making it easier to determine overfitted models. As illustrated in the image below, the graph shown depicts a model with a steady increase in accuracy for both sets and a steady decrease in loss. However it should be noted that in the validation set there are a few instances of the model's prediction be way out of range for what it should be, this is likely do to the over fitting as mentioned before.

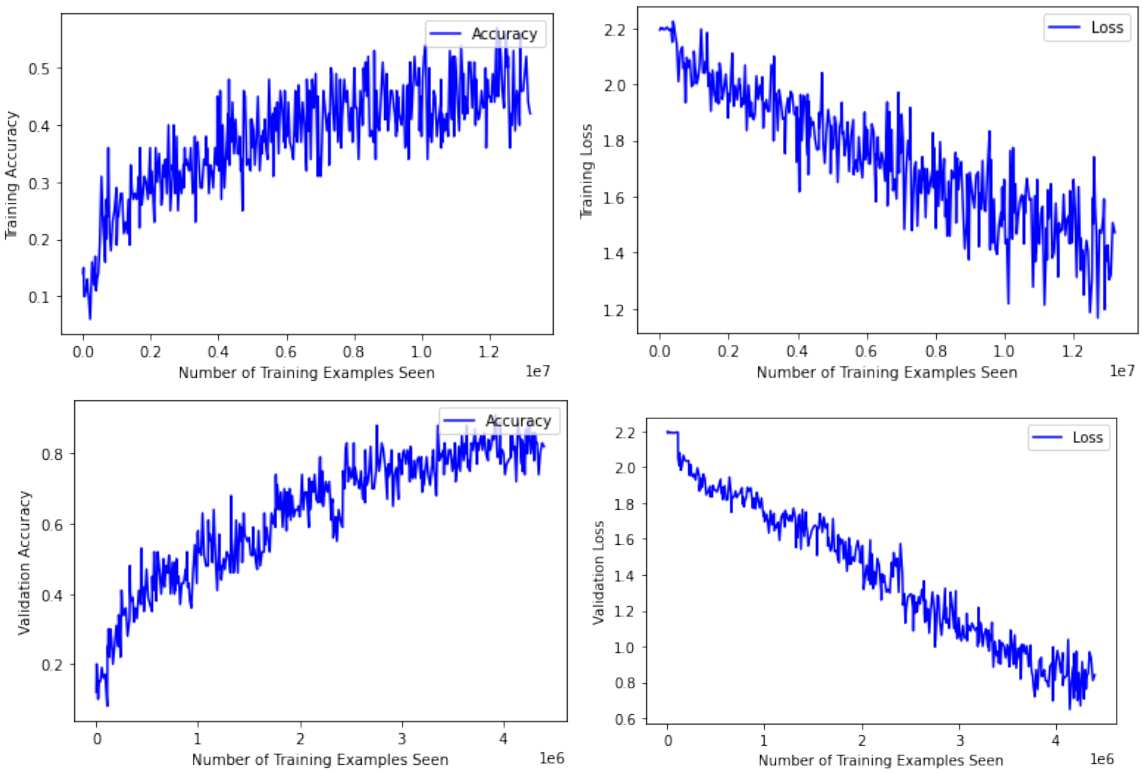

Figure 3: Accuracy and loss of the training and validation sets

5 Results

The neural network model has a training accuracy of 45%, and it also has a validation accuracy of 80%. The accuracy values are yielded from a neural network model with 40 epochs. The model has a loss of 1.4 for the training and 0.8 for the validations. Each epoch in the neural network model took roughly 1 minute to be computed, and this represents a decent run-time for a CNN with 3 convolutional layers as well as 4 fully-connected layers.

We made multiple checks to determine which of the models resulted from overfitting. These methods include looking at graphs of the accuracy and loss rates for each epoch. Oscillations, bumps, and non-curved graphs indicated that overfitting occurred in the model. The results of overfitted models were checked for accuracy by sampling a couple of data sets and discarded if they were incorrect. Still, overfitting was a major obstacle in this assignment, as the model would commonly generate incorrect predictions. It's hypothesized that the model would tend to 'recognize' or assume patterns in the data that were erroneous and not important in the classification problem. It is believed that the high rate of false-positives was the cause of poor performance scores. This caused the model to provide high loss and accuracy scores on the train and validation sets and low scores on the test set. Subsequently, this over fitting lead to a poor position on the Kaggle leaderboards. Specifically a test score of 63.86% and a leaderboard position of 20.

6 Discussion and Conclusion

Many challenges were faced during the creation of the neural network model. These challenges were mostly the result of model overfitting, incorrectly implemented neural network models, and the possibility of incorrectly derived mathematical formulas. In contrast to previous projects made during this course, the group had difficulty finding the source of these errors or finding ways to fix them.

The group attempted to solve these problems by implementing modifications to neural network coefficients, the neural network stride parameters, the neural network padding methods, and the neural network layers. Ultimately, the desired accuracy was not achieved for the neural network model due to a lack of insight into ways to correctly implement or fix the model. However, the model partially worked, resulting in significantly greater accuracy scores than the baseline accuracy for random guessing.

The use of too many layers in the machine learning model (three or more convolutional layers) leads to overfitting. Maxpooling and the activation functions like ReLu and Log_softmax were crucial to creating an accurate CNN, especially when dealing with image recognition. Future investigation into the dropout rate is advisable for making a more accurate learning model. Maxpooling allowed for the use of up to five layers, but there was little improvement past the use of three layers. Increasing the data depth by more than ten layers resulted in no improvements for accuracy.

Models with more connections in the last layer almost always performed worse on validation sets. Kernal size and maxpolling mattered as well when attempting to generalize the datasets. Adding maxpooling to layers improved memory usage and training speed during the learning phase. Smaller batch sizes when loading data had higher accuracy and speed for training results. The batch sizes likely affected model overfitting and reducing the batch size helped with this.

Ultimately - with more time - the model's correctness and accuracy scores could have been improved greatly. The primary problem with the model is with its coefficient tuning, which resulted in overfitting / overpredicting with the datasets. Some more insight into what parameters to adjust and the root cause of the model's inaccuracy would have aided the group's effort in this project. More time, tests, and insight would have the best way to have completed the neural network model.

7 Statements of Contribution

While all team members were responsible for research, debugging, and designing fundamentals of the neural network, the core functionality and Python implementation was completed by Michael Kazman and Trevor Johns. Connor Stewart completed a majority of the theoretical aspects such as research, analysis, and data visualization. Lastly, further documentation and construction of the report was a joint effort between all three members on the team.

References

[1] Aramis et al. *ImageNet: VGGNet, ResNet, Inception, and Xception with Keras*. June 2020. URL: https://www.pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/.

[2] *PyTorch 1.7.0 documentation*. URL: https://pytorch.org/docs/stable/index.html.

[3] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks-the ELI5 way*. Dec. 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[4] Jerry Wei. *AlexNet: The Architecture that Challenged CNNs*. Sept. 2020. URL: https://towardsdatascience.com/alexnet-the-architecture-that-challenged-cnns-e406d5297951.

[5] Zalandoresearch. *zalandoresearch/fashion-mnist*. URL: https://github.com/zalandoresearch/fashion-mnist.

8 Appendix

Appendix To facilitate the grading process, attach the codes for your implementations to the end of your report. This does not count towards the page limit of the report. You must also submit your code as a .zip file.

8.1 mini-project-3.py

```
from google.colab import drive
drive.mount('/content/gdrive')
%cd '/content/gdrive/My Drive/mini-project-3'

# pytorch libraries
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader

# data science libraries (pandas, matplotlib, etc.)
import pandas as pd
import matplotlib.pyplot as plt

# custom classes
from dataset import Dataset
from network import Network
from transformer import Transformer

# python system libraries
import time

# set seed for reproducible randomness
SEED = 4538964893498
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
torch.set_num_threads(10)

# initialize datasets (using custom class inherited from torch dataset)
train_dataset = Dataset('./data/Train.pkl', './data/TrainLabels.csv', transform=
    ↪ Transformer().transform, index=None, height=64, width=128)
test_dataset = Dataset('./data/Test.pkl', None, transform=Transformer().transform, index=
    ↪ None, height=64, width=128)

# generate validation set (from half the dataset)
evens = list(range(0, len(train_dataset), 3))
valid_dataset = torch.utils.data.Subset(train_dataset, evens)

# display sizes of datasets
print('Train dataset size: {}'.format(len(train_dataset)))
print('Validation dataset size: {}'.format(len(valid_dataset)))
print('Test dataset size: {}'.format(len(test_dataset)))

# show sample images
```

```
show_images = False
if (show_images is True):
    # setup plot for images to be displayed on
    num_images, image_width, image_height = 3, 24, 40
    fig = plt.figure(figsize=(image_width, image_height))
    plot = []
    # go through images
    for i in range(0, num_images):
        plot.append(fig.add_subplot(num_images, 1, i + 1))
        train_dataset.show_image(i)

num_epochs = 10
num_classes = 9
dropout = [0.5, 0.25]
batch_size = 500
learning_rate = 0.0001

# training set (shuffled with a select seed)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True);

# validation set (based on half the training set)
valid_loader = torch.utils.data.DataLoader(dataset=valid_dataset,
                                           batch_size=batch_size,
                                           shuffle=True);

# testing set (unshuffled)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                           batch_size=batch_size,
                                           shuffle=False);

# get GPU if available
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# initialize CNN for classification
model = Network(num_classes=num_classes, dropout=dropout).to(device)

# initialize criterion and optimization algorithms (used in CNN)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
#optimizer = torch.optim.Adam(model.parameters(), lr=learning

# constants for training and testing
COST_REDUCTION = -5
LOSS_PERCENTAGE = 10

# lists for generating graphs and analytics
train_loss, train_acc, train_samples = [], [], []
valid_loss, valid_acc, valid_samples = [], [], []

# calculate time of each epoch
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
```

```
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = round(float(elapsed_time - (elapsed_mins * 60)), 2)
    return elapsed_mins, elapsed_secs

# plot figure (for displaying accuracy and loss)
def plot_figure(value, counter, legend, caption):
    fig = plt.figure()
    plt.plot(counter, value, color='blue')
    plt.legend([legend], loc='upper right')
    plt.xlabel('Number of Training Examples Seen')
    plt.ylabel(caption)
    fig

def train(model, data_loader, optimizer, criterion):
    # set model to training mode
    model.train()

    # determines at what value batch number the split happens
    split_percentage = (len(data_loader) * (0.01 * LOSS_PERCENTAGE))

    # go through training batches
    for batch_num, (images, labels) in enumerate(data_loader):
        # convert images and labels to pytorch objects
        images = images.to(device)

        # normalize labels (0 - 8 representing $5 - $13)
        corrected_labels = torch.add(labels, COST_REDUCTION).to(device)

        # zero out gradient (for optimization)
        optimizer.zero_grad()

        # generate predictions
        output = model(images)

        # calculate loss and corectness
        #print(output.shape, corrected_labels.shape)
        loss = criterion(output, corrected_labels)

        # backward propogation and step forward
        loss.backward()
        optimizer.step()

        # determine number of correct samples (in this batch)
        predictions = output.data.max(1, keepdim=True)[1]
        correct = 0
        for i in range(predictions.shape[0]):
            if (predictions[i] == corrected_labels[i]):
                correct += 1
        acc = correct / len(corrected_labels)

        # print out analysis (after going through TRAIN_LOSS_PERCENTAGE of batches)
        if (batch_num % split_percentage == 0):
            # calculate number of passed samples
            passed_samples = int((batch_num + split_percentage) * batch_size)
```

```
            total_samples = int(len(data_loader) * len(images))
            percentage_passed = (passed_samples / total_samples * 100)

            # add losses, acc, and number of samples observed to lists
            # (for future graphing over all epochs)
            train_loss.append(loss.item())
            train_acc.append(acc)
            if (len(train_samples) == 0):
                train_samples.append(passed_samples)
            else:
                train_samples.append(train_samples[-1] + passed_samples)

            # print message
            print('\t[{} / {}\t({:.2f}%)]\tLoss: {:.4f}\tAcc: {:.2f}%'
                  .format(
                      passed_samples,
                      total_samples,
                      percentage_passed,
                      loss.item(),
                      acc))

    # calculate loss over entire epoch
    num_steps = int(100 / LOSS_PERCENTAGE)
    average_train_loss = (sum(train_loss[-num_steps:]) / num_steps)
    average_train_acc = (sum(train_acc[-num_steps:]) / num_steps)
    return average_train_loss, average_train_acc

def evaluate(model, data_loader, criterion):
    # set model to training mode
    model.eval()

    # determines at what value batch number the split happens
    split_percentage = (len(data_loader) * (0.01 * LOSS_PERCENTAGE))

    # disable gradient calculations (for optimization when testing)
    with torch.no_grad():
        # go through training batches
        for batch_num, (images, labels) in enumerate(data_loader):
            # convert images and labels to pytorch objects
            images = images.to(device)

            # normalize labels (0 - 8 representing $5 - $13)
            corrected_labels = torch.add(labels, COST_REDUCTION).to(device)

            # generate predictions
            output = model(images)

            # calculate loss and corectness
            loss = criterion(output, corrected_labels)

            # determine number of correct samples (in this batch)
            predictions = output.data.max(1, keepdim=True)[1]
            correct = 0
            for i in range(predictions.shape[0]):
```

```
                if (predictions[i] == corrected_labels[i]):
                    correct += 1
            acc = correct / len(corrected_labels)

            # print out analysis (after going through TRAIN_LOSS_PERCENTAGE of batches)
            if (batch_num % split_percentage == 0):
                # calculate number of passed samples
                passed_samples = int((batch_num + split_percentage) * batch_size)
                total_samples = int(len(data_loader) * len(images))
                percentage_passed = (passed_samples / total_samples * 100)

                # add losses, acc, and number of samples observed to lists
                # (for future graphing over all epochs)
                valid_loss.append(loss.item())
                valid_acc.append(acc)
                if (len(valid_samples) == 0):
                    valid_samples.append(passed_samples)
                else:
                    valid_samples.append(valid_samples[-1] + passed_samples)

                # analysis message
                print('\t[{} / {}\t({:.2f}%)]\tLoss: {:.4f}\tAcc: {:.2f}%'
                      .format(
                          passed_samples,
                          total_samples,
                          percentage_passed,
                          loss.item(),
                          acc))

    # calculate loss over entire epoch
    num_steps = int(100 / LOSS_PERCENTAGE)
    average_valid_loss = (sum(valid_loss[-num_steps:]) / num_steps)
    average_valid_acc = (sum(valid_acc[-num_steps:]) / num_steps)
    return average_valid_loss, average_valid_acc

# current best loss (to determine most accurate model)
best_valid_loss = float('inf')

# iterate over n epochs
for epoch in range(num_epochs):
    # begin timer
    start_time = time.time()

    # train model
    print('Training Epoch {}:'.format(epoch + 1))
    epoch_training_loss, epoch_training_acc = train(model, train_loader, optimizer,
        ↪ criterion)

    # test validation set
    print('Validation Epoch {}:'.format(epoch + 1))
    epoch_validation_loss, epoch_validation_acc = evaluate(model, valid_loader, criterion
        ↪ )

    # calculate duration of epoch
```

```
    epoch_mins, epoch_secs = epoch_time(start_time, time.time())

    # save model with lowest validation loss
    if epoch_validation_loss < best_valid_loss:
        best_valid_loss = epoch_validation_loss
        torch.save(model.state_dict(), 'model.pt')

    # analysis message
    print('\nEpoch {}: Took {}m {}s'.format(epoch + 1, epoch_mins, epoch_secs))
    print('Train Loss: {:.2f}\tVal. Loss: {:.2f}'.format(epoch_training_loss,
        ↪ epoch_validation_loss))
    print('Train Acc: {:.2f}%\tVal. Acc: {:.2f}%'.format(epoch_training_acc,
        ↪ epoch_validation_acc))
    print
        ↪ ('----------------------------------------------------------------------------------------------------
        ↪


# plot figures for the corresponding loss and acc of each dataset
plot_figure(train_loss, train_samples, 'Loss', 'Training Loss')
plot_figure(train_acc, train_samples, 'Accuracy', 'Training Accuracy')
plot_figure(valid_loss, valid_samples, 'Loss', 'Validation Loss')
plot_figure(valid_acc, valid_samples, 'Accuracy', 'Validation Accuracy')

# load in best model (from training step)
model.load_state_dict(torch.load('model.pt'))
model.to(torch.device('cpu'))

# set to evaluation mode
model.eval()

# list for storing predictions
prediction_list = []

# go through test set
for images, _ in test_loader:
    # get predictions (and choose most likely option)
    outputs = model(images)
    _, predictions = torch.max(outputs.data, 1)

    # un-normalize labels (back to proper cost)
    corrected_predictions = torch.add(predictions, -1 * COST_REDUCTION)

    # add to prediction list
    for prediction in corrected_predictions:
        prediction_list.append(prediction.item())

# create in pandas dataframe
dataframe = pd.DataFrame({ 'class': prediction_list }, columns=['class'])

# convert to csv
output_file = 'test_results.csv'
dataframe.to_csv(output_file, index=True, index_label='id')
```

```
# export message
print('~ Predictions exported to ', output_file)
```

8.2 dataset.py

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

# 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.

class Dataset(TorchDataset):
    def __init__(self, img_file, label_file=None, transform=None, index=None, height=64,
        ↪ width=128):
        # save image dimensions
        self.height = height
        self.width = width

        # load in data (using pickle) and targets (csv of [id, class] columns)
        # Note that image data are stored as unit8 so each element is an integer value
            ↪ between 0 and 255
        self.data = pickle.load(open(img_file, 'rb'), encoding='bytes')
        if label_file is not None:
            self.targets = np.genfromtxt(label_file, delimiter=',', skip_header=1)[:, 1:]

        # if index is a list of indices, use those as data/targets
        if index is not None:
          self.data = self.data[index]
          if self.targets is not None:
            self.targets = self.targets[index]

        # set image transformations (for normalization purposes)
        self.transform = transform

    # return the length of the dataset (based on target set)
    def __len__(self):
        return int(self.data.size / (self.height * self.width))

    # retrieve item from the dataset
    def __getitem__(self, index):
        # get image and target, and convert image to human-readable format
        img = self.data[index]
        img = Image.fromarray(img.astype('uint8'), mode='L')

        # if dataset has labels (retrieve it)
        target = []
        if hasattr(self, 'targets'):
```

```
            target = int(self.targets[index])

        # normalize image if transformer is specified
        if self.transform is not None:
           img = self.transform(img)

        # tuple of image
        return img, target

    # size of target matrix e.g. (60000, 1)
    def data_shape(self):
        return self.data.shape

    # size of target matrix e.g. (60000, 64, 128)
    def target_shape(self):
        return self.targets.shape

    # display a specific image (index) from the pickle dataset
    def show_image(self, index):
        #print(self.__getitem__(index)[0])
        image = np.squeeze(self.__getitem__(index)[0])
        plt.imshow(image, cmap='twilight_shifted', vmin=-1, vmax=1)
```

8.3 network.py

```
import torch
from torch import nn
import torch.nn.functional as F
import time

class Network(nn.Module):
    def __init__(self, num_classes, dropout):
        super().__init__()

        self.conv1 = nn.Conv2d(1, 64, kernel_size=5, stride=1, padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=1, padding=1)
        self.conv3 = nn.Conv2d(128, 192, kernel_size=4, stride=1, padding=0)

        self.fc1 = nn.Linear(192*6*14, 192)
        self.fc2 = nn.Linear(192, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, num_classes)

        self.dropout = dropout

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), kernel_size=2, stride=2))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size=2, stride=2))
        x = F.relu(F.max_pool2d(self.conv3(x), kernel_size=2, stride=2))

        x = x.view(-1, 192*6*14)
```

```
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.dropout(x, p=self.dropout[0], training=self.training)
        x = F.relu(self.fc4(x))
        x = F.dropout(x, p=self.dropout[1], training=self.training)

        return F.log_softmax(x, dim=1)
```

8.4 transformer.py

```
from torchvision import transforms

class Transformer():
    def __init__(self):
        self.transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ])
```