Image Classifier on the Fashion MNIST dataset

Abstract

This project shows an end-to-end implementation of image classification using Convolutional Neural Networks. The input data is a modified version of the Fashion-MNIST dataset where each image contains three articles and the sum of the prices is the label of the image. The VGG-16 structure was the most accurate amongst other types and was used to train a neural network. The *learning rate*, the *momentum* of the optimizer, the number of *epochs* and the *batch sizes* hyperparameters were tuned. The optimized values were $lr = 10^{-2}$, $momentum = 0.5$, $batch\_size = 2^7$ and $n\_epoch = 50$ which achieved 96.7% of accuracy on the validation set.

1 Introduction

In this report, an end-to-end project to implement and optimize a convolutional neural network is described. The input data used corresponds to a modified version of the Fashion MNIST dataset (1), on which each image analyzed contains three MNIST items from a group of five. The label of the observation is the sum of prices of the items on each image. To create a model that successfully predicts the targets of test images, a neural network with the architecture VGG-16 (2) is implemented and the original parameters are tuned to obtain the highest accuracy possible. The optimal parameters produced an accuracy of more than 97% on the validation set, thus providing a well-trained model for most practical tasks.

2 Dataset

The dataset used for training the proposed model corresponds to 60,000 images and their assigned labels. Each image contains three different articles of the Fashion MNIST original dataset (see some examples in Figure 1). These articles from the least to the most expensive are: T-shirt/top (\$1), Trouser (\$2), Pullover (\$3), Dress (\$4), Coat (\$5); where the number within parenthesis shows the price associated to each article. Each image is associated with a numerical label that is the sum of the articles' prices. Since each image contains articles from three different classes, the minimum value of the targets is \$5 and the maximum is \$13. A histogram with the nine different classes of the training set is shown in Figure 2. As illustrated, all classes are uniformly distributed and no re-sampling step is needed to analyze the data.

Figure 1: Examples of images on the input dataset

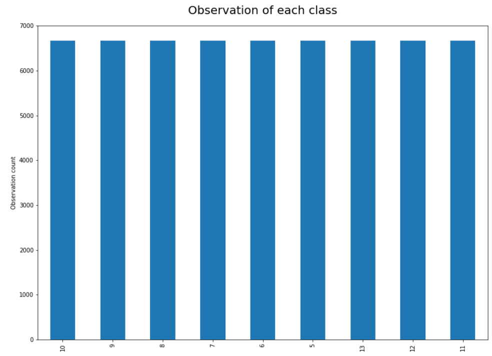


Figure 2: Histogram of training targets of the data.

3 Proposed approach

Each image contains a set of 128×64 gray-scale pixels in the (0, 255) range. The only pre-processing step conducted on the training dataset is re-scaling the range to (-1, 1). A fraction (20%) of the training dataset will be separated for testing, and the remaining data will be used for training the models.

The neural networks used in the model's development are based on the VGG-16 architecture (2) (discussed in subsection 3.1) and are implemented with *PyTorch* (3). The implementation of these neural networks is based on the tutorial (4) and (5).

3.1 Convolutional Network using VGG architecture

The architecture of VGG-16 is depicted in Figure 3 for an RGB figure of size 224×224. The image is passed through a stack of convolutional layers using a filter of size 3×3. The convolution stride is fixed to 1 pixel. Spatial pooling is carried out by five max-pooling layers, which follow some of the convolutional layers. Max pooling is performed over a 2× 2-pixel window with a stride of 2. Three fully connected layers follow the stack of convolutional layers. The final layer is a soft-max layer to provide results for the multi-class classification problem. All hidden layers are equipped with the RELU non-linearity. This convolutional network is renowned for its accuracy with image classification (6) and will be implemented for this project's task.

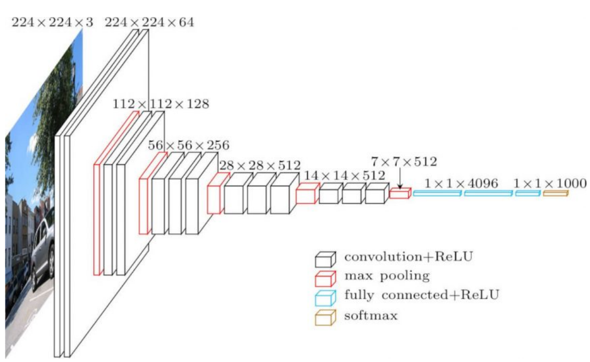


Figure 3: Architecture of VGG-16.

3.2 Models implementation

3.2.1 VGG-16 using *PyTorch*

The implementation of this neural network can be seen in Appendix A using *PyTorch*. It has been conducted such that the neural network architecture is easily adaptable for other versions of the VGG architecture (e.g. VGG-11), which accelerates the comparison between models.

3.2.2 Optimization algorithms - Stochastic Gradient Descent

For the neural network's training steps, *PyTorch*'s stochastic gradient descent (SGD) optimizer has been utilized. This well-known algorithm avoids local minima by randomly assigning initial weights for each training step. The SGD class receives as input the *learning rate* of the descent and its *momentum*, which can be used as hyperparameters for tuning the optimal model.

3.2.3 Hyperparameter tuning

The following parameters are tuned to find the optimal combination with the highest accuracy (smallest loss):

Learning rate: The learning rate of the SGD optimization algorithm, which controls how much to change the model in response to the estimated error each time the model weights are updated, is crucial. For example, a too-small value might lead to a slow convergence on loss, while a too big rate could yield an unstable training procedure. Thus, the learning rate is tested with $lr \in \{10^{-1}, 10^{-2}, 10^{-3}\}$ to confirm this hypothesis.

Momentum: The momentum of the SGD optimization algorithm, which accelerates gradients vectors, leads to faster converging. A system with momentum is proved to perform better in (7). Its value is set to $momentum = 0.5$.

Batch size: As SGD is sequential and uses small batches, the training time is inversely proportional to batch size in general. For a system with reasonable training time, a large batch size may be favourable. The $batch\_size \in \{2^5, 2^7, 2^9\}$ were compared to confirm this hypothesis.

Number of epochs: In general, as the number of epochs increases, the model goes from underfitting, then to optimal state, then overfitting. Therefore, identifying a fair number of epochs is essential. The number of epochs in the range $n\_epoch \in \{40, 60\}$ were compared.

As a baseline, the VGG-16 architecture an a SGD optimizer with the following hyperparameters value was used ($lr = 10^{-2}$, $momentum = 0.5$, $batch\_size = 2^7$, $n\_epoch = 60$). From this base case, each parameter was changed one at a time and the loss and accuracy curves were analyzed for both datasets. Only a portion of these results is presented to highlight these experiments' main features due to space limits.

4 Results

The results discussed in this report are only a fraction of what is shown in the attached `mini_project_3_team_20`.

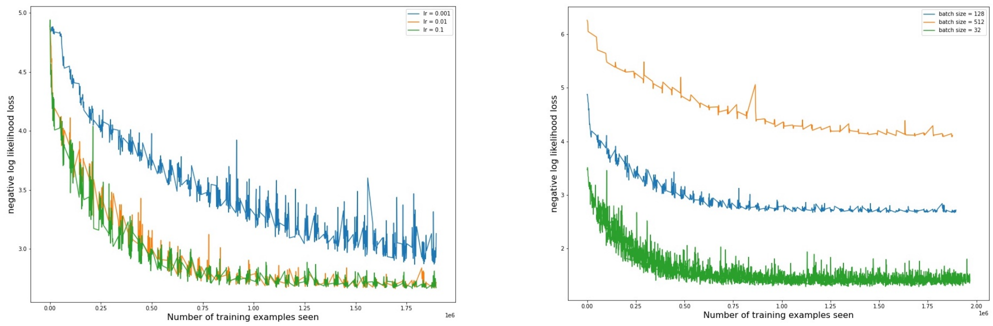
4.1 Analysis of hyperparameter tuning

4.1.1 Accuracy versus different learning rates

The values for the optimizer learning rate $lr \in \{10^{-1}, 10^{-2}, 10^{-3}\}$ are used for the same system to evaluate their impact. The results can be observed in Figure 4a. It can be seen that the loss curve for $lr = 10^{-3}$ converges very slowly, and after 40 epochs are still higher than the other two values. In addition, the loss curve of $lr = 10^{-1}$ is more fluctuating when compared to the curve of $lr = 10^{-2}$. Thus, the results agree with our previous hypothesis, and it can be concluded that the optimal learning rate among the tested values is $lr = 10^{-2}$.

4.1.2 Accuracy versus different batch sizes and time

The values $batch\_size \in \{2^5, 2^7, 2^9\}$ are used on the same system to observe their effect, shown in Figure 4b. It can be observed that the smaller the batch size, the faster the convergence of the loss curve. Besides, the training versus batch size is given in Table 1. The training time is not proportional to batch size, and $batch\_size = 2^7$ yields the best training time. As a model with good accuracy and reasonable training time is wanted, it can be concluded that the optimal batch size among the tested values is $batch\_size = 2^7$.



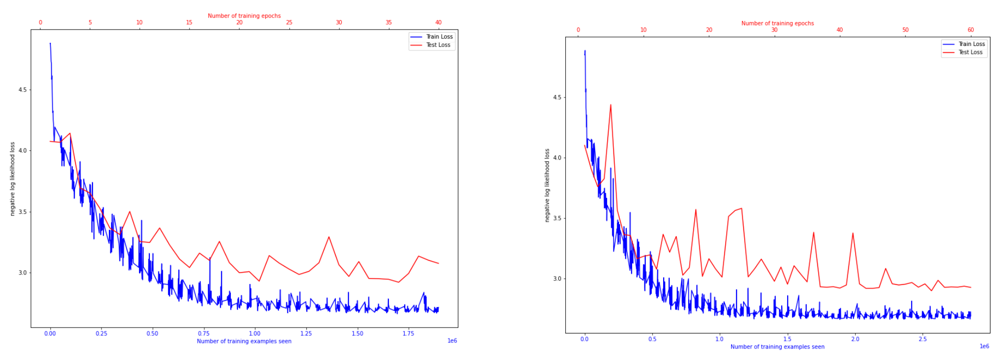
(a) Versus different learning rates

(b) Versus different batch sizes

Figure 4: Train loss variation

Table 1: Average time per epoch of different batch size

Batch size (samples)	32	128	512
Total time for 40 epochs (secs)	6402.2	5603.5	6182.6
Total time for 60 epochs (secs)	9605.1	8401.0	9278.9
Average time per epoch (secs)	160.0	139.9	154.5



(a) $n\_epoch = 40$

(b) $n\_epoch = 60$

Figure 5: Train and test losses with n_epoch

4.1.3 Accuracy versus different numbers of epochs

The previous analyses showed the behaviour of the *training* loss curve with different hyperparameters. Yet, there is no use in comparing only training curves when the model predicts unknown values. To select optimal models, curves of losses and accuracies on a test set should be explored. Figure 5 shows these accuracies curves for the cases $n\_epoch \in \{40, 60\}$. In fact, both training loss curves fluctuation decrease after 30 epochs; however, the test loss curves remain steady only after 50 epochs. Therefore, to minimize the risk of over-fitting, the optimal value for the number of epochs should be $n\_epoch = 50$.

4.2 Analysis of the final result

Based on previous observations and analysis, it can be concluded that the optimal hyperparameters for the VGG-16 network is $lr = 10^{-2}$, $momentum = 0.5$, $batch\_size = 2^7$ and $n\_epoch = 50$. A new CNN is trained with these hyperparameters, and the corresponding accuracy is illustrated in Figure 6, which achieved 96.7% over a test set of $12,000$ images. Besides, from Figure 7, it can be observed that most error classes are mislabeled only to the nearest class, which suggests that our model only mistakes between very similar images.

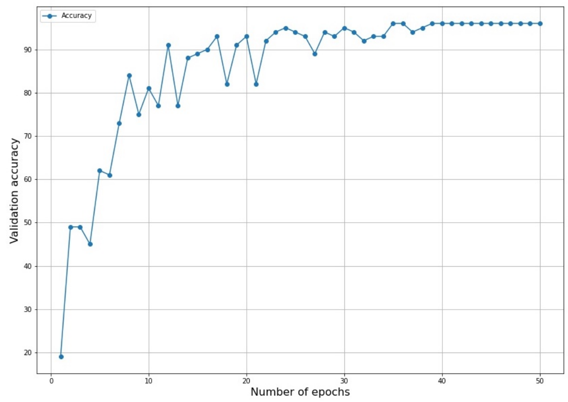


Figure 6: Accuracy over iteration obtained obtained with $lr = 10^{-2}$, $momentum = 0.5$, $batch\_size = 2^7$ and $n\_epoch = 50$.

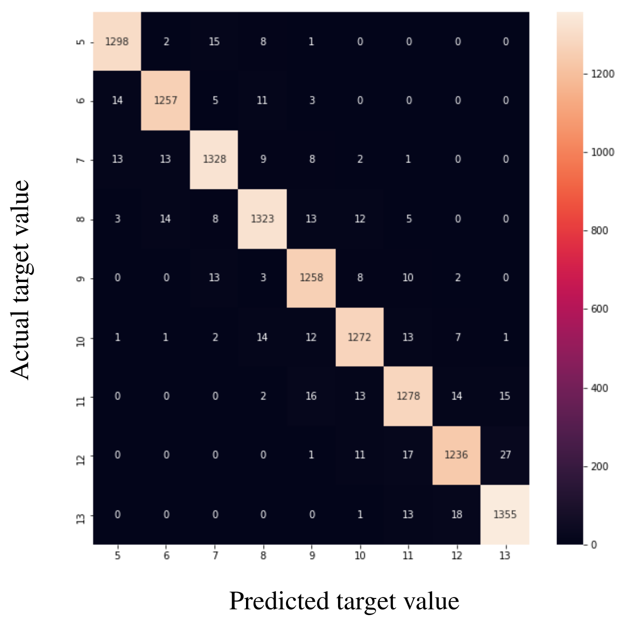


Figure 7: Heat Map of the Confusion Matrix for results obtained with $lr = 10^{-2}$, $momentum = 0.5$, $batch\_size = 2^7$ and $n\_epoch = 50$.

5 Conclusion

This project presents the results of applying a dense neural network of architecture VGG-16 to the task of image recognition and classification. The pre-processing of the images and the analysis of their distribution was shown. The composition of this popular neural network was briefly discussed. Several figures of interest are analyzed throughout the project: running time of the network, loss and accuracy vs observed data for both test and training set. Different hyperparameters for the model influenced these figures. These experiments and comparisons were used to select an optimal model with the highest accuracy possible on the validation set.

References

[1] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[3] FaceBook, "*PyTorch*," 2020. `https://pytorch.org/` [Accessed: 11/28/2020].

[4] A. Persson, "*Pytorch VGG implementation from scratch*," 2020. `https://www.youtube.com/watch?v=ACmuBbuXn20` [Accessed: 11/29/2020].

[5] A. Kak and C. Bouman, "*Lecture Notes on Deep Learning*," February 2020. `https://engineering.purdue.edu/DeepLearn/pdf-kak/week5.pdf` [Accessed: 11/30/2020].

[6] J. Wei, "Vgg neural networks: The next step after alexnet," July 2019. `https://towardsdatascience.com/vgg-neural-networks-the-next-step-after-alexnet-3f91fa9ffe2c` [Accessed: 11/29/2020].

[7] G. D. Ilya Sutskever1, James Martens and G. Hinton, "On the importance of initialization and momentum in deep learning," *Proceedings of the 30th International Conference on Machine Learning*, pp. 1139–1147, 2013.

A Appendix

Complete notebook with figures can be found in the attached files MiniProject3Team20.ipynb.

It follows the code of implementing a VGG-16 network to this project.

```python
# -*- coding: utf-8 -*-
"""Mason_Mini-Project_3_group_20.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1P6Xhs3G0G0Rq8lDVDak96oweIxY8X94M

# 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.
"""

# Commented out IPython magic to ensure Python compatibility.
from google.colab import drive
from google.colab import files

drive.mount('/content/gdrive')
# %cd '/content/gdrive/MyDrive/Colab Notebooks/Mini Project 3 ECSE 551 Team 20'

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

path_lib = '/content/gdrive/MyDrive/Colab Notebooks/Mini Project 3 ECSE 551 Team 20/'
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
#device = 'cpu'

"""
Let's see how the data looks like.
"""

import pandas as pd
import matplotlib.pyplot as plt

train = pd.read_csv("./data/TrainLabels.csv")

plt.subplots(figsize=(14, 10))
train['class'].value_counts().plot.bar()
plt.ylabel("Observation count")
plt.title('Observation of each class', fontsize=20, pad=20)
plt.savefig(path_lib + 'graph/class_hist.png')
plt.show()

print((train['class'].value_counts()))

print("Each class has the same number of observation +-1 data")

# Read a pickle file and dispaly 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/Train.pkl', 'rb' ), encoding='bytes' )
targets = np.genfromtxt('./data/TrainLabels.csv', delimiter=',', skip_header=1)[:,1:]
# plt.subplots(figsize=(14, 10))
plt.imshow(data[1234,:,:],cmap='gray', vmin=0, vmax=256)
print(data.shape, targets.shape)

"""# 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.
"""

training_phase = True
export = False
idx_input = None
re_train = True

"""
If in training phase, set the following parameters
"""

train_size = 48000
test_size = 12000
batch_size = 2 ** 7 #feel free to change it

#Optimizer calibratable hyper parameters
c_lr = 0.001
c_momentum = 0.5

#Network iteration calibratable paramerters
n_epochs = 40

# 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, targets, transform=None, idx = None):
        self.data = pickle.load( open( img_file, 'rb' ), encoding='bytes')
        self.targets = targets
```

```
        # self.targets = np.genfromtxt(label_file, delimiter=',', skip_header=1)[:,1:]
        if idx is not None:
          self.targets = self.targets[idx]
          self.data = self.data[idx]
        if transform is not None:
          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

class MyValidationSet(Dataset):
    def __init__(self, img_file, transform=None, idx = None):
        self.data = pickle.load( open( img_file, 'rb' ), encoding='bytes')
        # self.targets = np.genfromtxt(label_file, delimiter=',', skip_header=1)[:,1:]
        if idx is not None:
          self.data = self.data[idx]
        if transform is not None:
          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

# Read image data and their label into a Dataset class

if training_phase:
  print("Training phase dataset generation")
  # Split the data into the training set and the test set
  from sklearn.model_selection import ShuffleSplit
  SS = ShuffleSplit(n_splits=1, train_size=train_size, test_size=test_size, random_state=28)
  targets = np.genfromtxt('./data/TrainLabels.csv', delimiter=',', skip_header=1)[:,1:]
  train_index, test_index = next(SS.split(targets))

  #Create the train dataset
  target_csv = np.genfromtxt('./data/TrainLabels.csv', delimiter=',', skip_header=1)[:,1:]
  min_target = min(target_csv)
  target_csv = target_csv - min_target

  train_dataset = MyDataset('./data/Train.pkl', target_csv, idx=train_index, transform=img_transform)
  test_dataset = MyDataset('./data/Train.pkl', target_csv, idx=test_index, transform=img_transform)

```

```
  print(test_dataset.targets)
  #test_dataset = MyDataset('./Train.pkl', './TrainLabels.csv', idx=test_index, transform=img_transform)

  #Apply the dataloader
  train_loader = DataLoader(train_dataset ,batch_size=batch_size, shuffle=True)
  test_loader = DataLoader(test_dataset ,batch_size=batch_size, shuffle=False)

  examples = enumerate(test_loader)
  batch_idx, (example_data, example_targets) = next(examples)
  example_data = np.squeeze(example_data)

  fig = plt.figure()
  for i in range(6):
    plt.subplot(2,3,i+1)
    #plt.tight_layout()
    plt.imshow(example_data[i].cpu().numpy(), cmap='gray', vmin=-1, vmax=1, interpolation='none')
    plt.title("Ground Truth: {}".format(example_targets[i]))
    plt.xticks([])
    plt.yticks([])
  fig
  print(example_data.shape)

else:
  print("Not in training phase")
  #Create the train dataset
  target_csv = np.genfromtxt('./data/TrainLabels.csv', delimiter=',', skip_header=1)[:,1:]
  min_target = min(target_csv)
  target_csv = target_csv - min_target

  train_dataset = MyDataset('./data/Train.pkl', target_csv, idx=None, transform=img_transform)
  train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

  #Create the test dataset
  test_dataset = MyValidationSet('./data/Test.pkl', idx=None, transform=img_transform)
  test_loader = DataLoader(test_dataset ,batch_size=batch_size, shuffle=True)

"""# Define our NN
Define the VGG16 to be used in this project
"""

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

class Net(nn.Module):
    # This part defines the layers
    def __init__(self):
        super(Net, self).__init__()
        # At first there is only 1 channel (greyscale). The next channel size will be 10.
        input_sz_h = 128
        input_sz_v = 64
        fltr_sz_cv_1 = 3
        fltr_num_cv_1 = 10
        fltr_sz_cv_2 = 3
        fltr_num_cv_2 = 10
```

```python
        final_sz_h = np.floor(((input_sz_h-fltr_sz_cv_1+1)/2-fltr_sz_cv_2+1)/2)
        final_sz_v = np.floor(((input_sz_v-fltr_sz_cv_1+1)/2-fltr_sz_cv_2+1)/2)
        self.img_sz = int(fltr_num_cv_2 * final_sz_h * final_sz_v)

        self.conv1 = nn.Conv2d(1, fltr_num_cv_1, kernel_size=fltr_sz_cv_1)
        self.conv2 = nn.Conv2d(fltr_num_cv_1, fltr_num_cv_2, kernel_size=fltr_sz_cv_2)
        self.conv2_drop = nn.Dropout2d()


        NN_neurons = 120
        self.fc1 = nn.Linear(self.img_sz, NN_neurons)
        self.fc2 = nn.Linear(NN_neurons, 9)



    # And this part defines the way they are connected to each other
    # (In reality, it is our foreward pass)
    def forward(self, x):


        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        #x = x.view(-1, self.img_sz)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)

        return F.log_softmax(x,dim=0)

class Net_VGG16(nn.Module):
    # This part defines the layers
    def __init__(self):
        super(Net_VGG16, self).__init__()

# architecture based on https://engineering.purdue.edu/DeepLearn/pdf-kak/week5.pdf
#VGG16 = {’model’:[64, 64, ’M’, 128, 128, ’M’, 256, 256, 256, ’M’, 512, 512, 512,’M’, 512, 512, 512, ’M’], ’name’:’VGG11’}

        self.conv_seqn = nn.Sequential(
        # Conv Layer block 1:
        nn.Conv2d(in_channels=1, out_channels=64, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU(inplace=True),
        nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
        # Conv Layer block 2:
        nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(128),
        nn.ReLU(inplace=True),
        nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(128),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
        # Conv Layer block 3:
```

```
        nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(256),
        nn.ReLU(inplace=True),
        nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(256),
        nn.ReLU(inplace=True),
        nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(256),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
        # Conv Layer block 4:
        nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(inplace=True),
        nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(inplace=True),
        nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
        # Conv Layer block 5:
        nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(inplace=True),
        nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(inplace=True),
        nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.to_linear = 4096
        c_num_classes = 9

        self.fc_seqn = nn.Sequential(
            nn.Linear(self.to_linear,4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096,4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096,c_num_classes)
        )


    # And this part defines the way they are connected to each other
    # (In reality, it is our foreward pass)
    def forward(self, x):

        x = self.conv_seqn(x)
        x = x.view(x.size(0), -1)
        x = self.fc_seqn(x)

        return F.log_softmax(x,dim=0)
```

```python

#network = Net().to(device)
network = Net_VGG16().to(device)

optimizer = optim.SGD(network.parameters(), lr=c_lr, momentum=
    c_momentum)

train_losses = []
train_counter = []
test_losses = []
test_counter = range(1,n_epochs+1)
predi = []

def train(epoch):
  network.train()
  for batch_idx, (data, target) in enumerate(train_loader):
    target = target.to(device)
    data = data.to(device)
    optimizer.zero_grad()
    output = network(data)
    loss = F.nll_loss(output, target) #negative log liklhood loss
    loss.backward()
    optimizer.step()

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

  torch.save(network.state_dict(), './models/model.pth')
  torch.save(optimizer.state_dict(), './models/optimizer.pth')

def test():
  network.eval()
  test_loss = 0
  correct = 0
  predi = []
  with torch.no_grad():
    for batch_idx, (data, target) in enumerate(test_loader):

      target = target.to(device)
      data = data.to(device)
      output = network(data)
      test_loss += F.nll_loss(output, target, reduction='sum').item()
      pred = output.data.max(1, keepdim=True)[1]
      correct += pred.eq(target.data.view_as(pred)).sum()
      predi.append(pred)
      #print(pred)
  # print(len(predi[0]))
  # print(len(predi)) # this should

  test_loss /= len(test_loader.dataset)
  test_losses.append(test_loss)
  print('\nTest set: Avg. loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.
   format(
    test_loss, correct, len(test_loader.dataset),
    100. * correct / len(test_loader.dataset)))
  return pred, predi

def test_val():
  network.eval()
  test_loss = 0
  correct = 0
```

```
  predi = []
  with torch.no_grad():
    for data in test_loader:

      #target = target.to(device)
      data = data.to(device)
      output = network(data)
      # test_loss += F.nll_loss(output, target, size_average=False).item()
      pred = output.data.max(1, keepdim=True)[1]
      #correct += pred.eq(target.data.view_as(pred)).sum()
      predi.append(pred)
      #print(pred)
  # print(len(predi[0]))
  # print(len(predi)) # this should

  # test_loss /= len(test_loader.dataset)
  # test_losses.append(test_loss)
  # print('\nTest set: Avg. loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
  #   test_loss, correct, len(test_loader.dataset),
  #   100. * correct / len(test_loader.dataset)))
  return pred, predi

def load_network():
  network.load_state_dict(torch.load('./models/model.pth'))
  optimizer.load_state_dict(torch.load('./models/optimizer.pth'))

if training_phase:
  if re_train:
    for epoch in range(1, n_epochs+1):
      train(epoch)
      pred, predi = test()
  else:
    load_network()
    pred, predi = test()
else:
  if re_train:
    for epoch in range(1, n_epochs+1):
      train(epoch)
    pred, predi = test_val()
  else:
    load_network()
    pred, predi = test_val()

"""# Visualize result"""

if re_train:
  fig, ax1 = plt.subplots(figsize=(14, 10))

  ax1.set_xlabel('Number of training examples seen', color='blue')
  ax1.set_ylabel('negative log likelihood loss')
  lns1 = ax1.plot(train_counter, train_losses, color='blue', label='Train Loss')
  ax1.tick_params(axis='x', labelcolor='blue')

  plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
  plt.savefaph/train_loss.png')

  if training_phase:
    ax2 = ax1.twiny()
    lns2 = ax2.plot(test_counter, test_losses, color='red', label='Test Loss')
    ax2.set_xlabel('Nig(path_lib + 'grumber of training epochs', color='red')
    ax2.tick_params(axis='x', labelcolor='red')
```

```
    lns = lns1+lns2
    labs = [l.get_label() for l in lns]

    ax2.legend(lns, labs, loc='upper right')
    plt.show

    plt.savefig(path_lib + 'graph/train_test_loss.png')

file = open('./result/lr0001.pkl', 'wb')
pickle.dump([train_counter, train_losses, test_losses], file)
file.close()

file = open('./result/lr0001.pkl', 'rb')
train_counter2, train_losses2, test_losses2 = pickle.load(file)
plt.plot(train_counter2, train_losses2)

"""
This creates a single list out of the Predi variable.
Since Predi is a nested list, each elements are appended to the new
    list predictions
Since the batch size doesnt exactly count for the total number of
    values,
this deletes all values above the size of the test_dataset
"""
batch_size = len(predi[0])
test_size = len(test_dataset)

predictions = []
for list in predi:
    for item in list:
      predictions.append(item.item() + min_target)
#print(len(predictions))

del predictions[test_size:]
#print(len(predictions))
#print(predictions)

"""
Create a Pandas Datafram to see the data
It is also easier to export
"""

result=pd.DataFrame(predictions)
result.reset_index(level=0, inplace=True)
result.columns = ["id","class"]

if not training_phase:
  if re_train:
    result.to_csv("./result/ECSE551_Group20.csv",index=False)
  else:
    result.to_csv("./result/ECSE551_Group20_.csv",index=False)

# files.download("ECSE551_Group20_.csv")

#print(predi[0][0].item())

fig = plt.figure()
result['class'].value_counts().plot.bar()
plt.ylabel("Observation count")
plt.show()

if training_phase:
  # https://towardsdatascience.com/multi-class-text-classification-
    with-scikit-learn-12f1e60e0a9f
  """
```

```
Let's build a confusion matrix
"""
import seaborn as sns
from sklearn.metrics import confusion_matrix

# conf_mat = confusion_matrix(test_dataset.targets + min_target, predictions)
conf_mat = confusion_matrix(test_loader.dataset.targets + min_target, predictions)
fig, ax = plt.subplots(figsize=(10,10))

unique_price = np.unique(test_loader.dataset.targets + min_target).astype('int32')
sns.heatmap(conf_mat, annot=True, fmt='d', xticklabels=unique_price, yticklabels=unique_price)
plt.savefig(path_lib + 'graph/heatmap.png')

# sns.heatmap(conf_mat, annot=True, fmt='d',
#             xticklabels=test_dataset.targets, yticklabels=test_dataset.targets)

plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```