Mini-Project 3: Image Understanding

Jack Wei
McGill University
yi.wei4@mail.mcgill.ca

Grey Yuan
McGill University
linwei.yuan@mail.mcgill.ca

Yu Wu
McGill University
yu.wu4@mail.mcgill.ca

Abstract

The main goal of this project is to predict the price of the article from the image data which each contain one article and several digits. This is a supervised image classification problem as the price of the article, which is the label of the image data, is based on the type of the article. Our proposed solution is to first apply unsupervised learning techniques of PCA and Gaussian Mixture model to extract article-only image patches. Then, we tested various convolutional neural network models, including a VGG-based and a custom designed CNN, on the extracted data. Lastly, we select the best model by evaluating and comparing their corresponding accuracy on the validation dataset. Our best achieved a validation accuracy of about 87.5%. This corresponds to our highest test accuracy of 87.8% on Kaggle. We have found that simple models with 3 convolutional layers works well on the extracted data, and batch normalization and dropout improves our model.

1 Introduction

In this project, we performed supervised image classification on image datasets modified by combining two benchmark datasets. In detail, each of the sample image data contains one Fashion-MNIST article image and at least one MNIST image. The task of the project is to predict the price of the fashion article in a sample image. In order to do so, we applied unsupervised learning techniques including Principal Component Analysis and Gaussian Mixture to extract the Fashion-MNIST image. Then, neural network models including a custom Visual Geometry Group (VGG) model and a simple custom convolution neural network (CNN) are trained on the extracted Fashion-MNIST image patches. Our best result is achieved by custom-designed CNN model and the test accuracy is 87.8% on Kaggle. During experimentations, we have discovered that complex models with significantly large set of parameters do not lead to better results. Our simple model of 3 convolutional layers and 2 fully-connected networks works well with dropout and batch normalization applied.

2 Dataset

The given training set contains 60000 modified images of MNIST + Fashion-MNIST with its associated labels ranging from 0 to 9. The test set contains 10000 unlabeled images. The MNIST dataset is a benchmark dataset of gray-scale images of the 10 handwritten numbers; and Fashion-MNIST is a dataset of fashion article images[1]. Each sample in the test and train set is a 28x28 gray-scale image, associated with a label from 10 classes. The given dataset combines these two datasets by combining 1 to 3 numbers from MNIST with a image of fashion article to make a 28x28 gray-scale image sample.

Preprint. Under review.

By examining the training dataset, we made the observation that there is no correlation between the MNIST number data and the class label. This means that labels are solely determined by the Fashion-MNIST part of a sample image. By this observation, we therefore come up with our proposed approach of extracting Fashion-MNIST data from the dataset discussed in Section 3.1. Analysis of the extracted data shows that the mean of pixel values is 0.285 the standard deviation is 0.316. With this value, we performed normalization on our input data by subtracting the mean and then divided by the standard deviation.

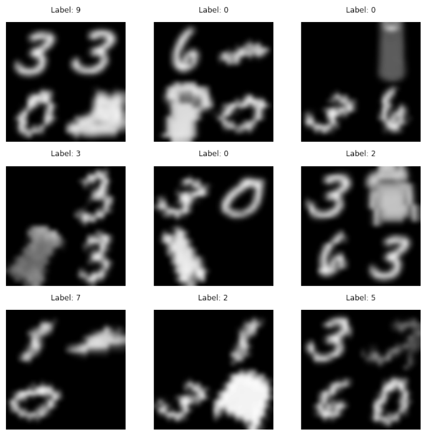


Figure 1: Sample Dataset Images

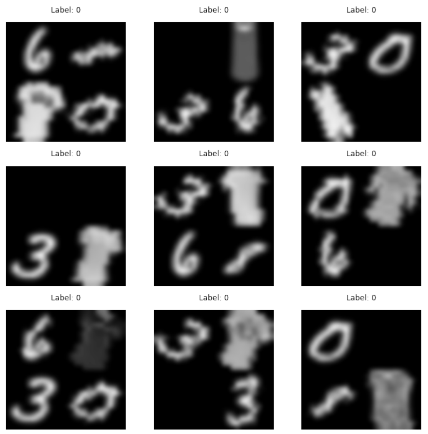


Figure 2: Sample Images with Label 0

3 Proposed Approach

3.1 Feature Extraction

As targets to the dataset are relevant only to the Fashion MNIST part of the sample, the hand-written numbers can thus be regarded as noises to the model. While neural networks, with decent amount of parameters, are capable of identifying relevant representations within the given inputs, the demand for such robustness to noisy data increases the need of models with higher complexity. In this case, models may run into additional problems of overfitting.

As a measure to reduce the load on our models associated with identifying relevant data, we have decided to preprocess and denoise our data by extracting the fashion MNIST image patches from the provided dataset with unsupervised learning techniques and train our models with the extracted patches only.

3.1.1 PCA + Gaussian Mixture

As fashion image patches correspond to any of the four quadrants of a sample, we first attempted to extract the four quadrants for each sample to form a new datasets with 240000 samples of 14x14 images patches. The apparent distinction of numbers and fashion images lead to the hypothesis that the latent space representations of the two dataset can form clear clusters. This leads to our proposed methodology of identifying fashion patches in a sample by using PCA dimension reduction and Gaussian Mixture clustering technique.

Principle Component Analysis (PCA) is used to find representations of sample image patches in a lower dimension subspace. As image patches are sampled from different distributions, Gaussian mixture model (GMM) is suitable for modeling the probability which an image patch belongs to a cluster, i.e. whether it belongs to a particular distribution. As the image patches are either empty, sampled from MNIST hand-written numbers dataset, or sampled from fashion-MNIST, we fit the image patches with three clusters. The result of fitting the GMM model is surprisingly effective since the number of data labeled as belonging to the fashion clusters is the same as the size of the dataset. This suggests that the algorithm perfectly identifies all fashion data from each sample with no misclassification. This allows us to select the patches of fashion images as inputs to our model. The process is summarized in the flow chart in figure 3.

3.2 Models

Custom Convolutional Neural Network Convolutional neural network (CNN) is a feed-forward neural network. Its artificial neurons can respond to a part of the surrounding units in the coverage area, thus making it suitable for large-scale image processing. For this project, we have implemented a custom small scale CNN model as a base model for further experimentations.

VGG-Based Visual Geometry Group (VGG) is a standard deep Convolutional Neural Network (CNN) architecture with multiple layers which is popular for image recognition. It was first proposed by K. Simonyan and A. Zisserman in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition"[2]. The VGG network follows a $Conv - Pooling$ architecture standard for CNN models. We decide to implement a custom VGG-like model for classification of fashion-MNIST data.

3.3 Model Selection Pipeline

This section outlines the methodology of evaluating and comparing potential models for determining fashion article prices. We first start our experimentation with the dataset with a base model of different architecture. These are listed in the following section. We then train our model on 500000 training set samples and use the remaining 100000 samples as the validation set. The separation would create sufficient data for training and adequately large validation set to reflect the generalization ability of our models.

The base model is modified by tuning hyperparameters and model architecture. This creates a set of candidate models for comparison. The metrics for evaluating the models are average cross entropy loss and validation accuracy. The final proposed model is chosen by validation accuracy.

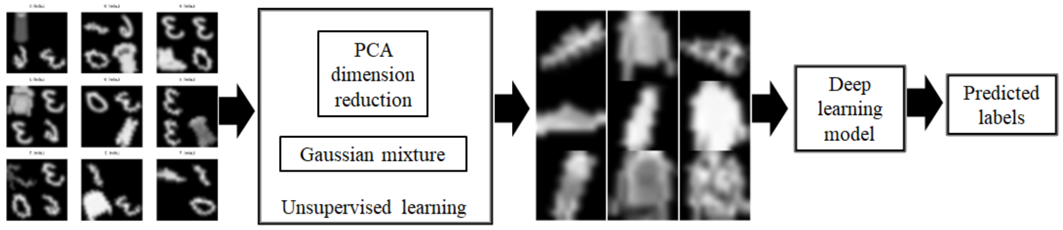


Figure 3: Feature extraction pipeline flow chart

4 Result

4.1 Custom VGG

We first implemented our custom VGG with the most basic structure as shown in the model summary below:

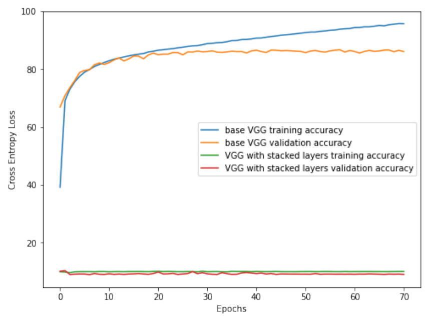


Figure 4: Comparison of adding convolution layers

Layer/Operation	Output Shape	Parameter Number
Conv2d-1	[-1, 32, 28, 28]	320
MaxPool2d-2	[-1, 32, 14, 14]	0
Conv2d-3	[-1, 64, 14, 14]	18,496
MaxPool2d-4	[-1, 64, 7, 7]	0
Conv2d-5	[-1, 128, 7, 7]	73,856
MaxPool2d-6	[-1, 128, 3, 3]	0
Conv2d-7	[-1, 256, 3, 3]	295,168
MaxPool2d-8	[-1, 256, 1, 1]	0
Linear-9	[-1, 128]	32,896
Linear-10	[-1, 10]	1,290

Table 1: Model summary of our proposed VGG model

With this basic VGG model, we achieved a validation accuracy of about 86%. As an attempt to improve the accuracy, we have modified the base VGG model by adding two additional layer in-between down-sampling maxPool layers to test whether additional layers can improve our model performance. Such a design is used in VGG-16 to encourage the model to learn more features before applying down-sampling. Surprisingly, adding convolution layers reduces the model to random guessing as evident in figure 4. This is likely caused by diminishing gradient due to model being too deep.

4.2 Convolutional Neural Network

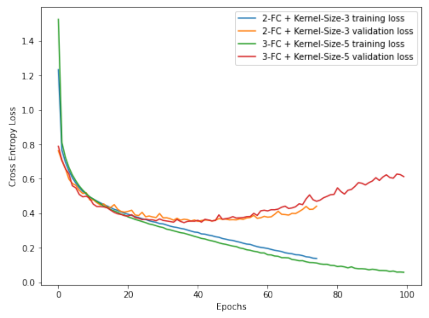
In our first custom CNN model, we developed a model with 3 convolution layers and two fully connected layers without any extra modification as shown in the table below. This model exceeds a validation accuracy of about 87%. Then we further developed 3 versions to investigate the impact of the number of fully connected networks, kernel size in convolutional layers, batch normalization and dropout. The results are shown in figure 4, 5 and 6.

Layer/Operation	Output Shape	Parameter Number
Conv2d-1	[-1, 32, 28, 28]	320
MaxPool2d-2	[-1, 32, 14, 14]	0
Conv2d-3	[-1, 64, 14, 14]	18,496
MaxPool2d-4	[-1, 64, 7, 7]	0
Conv2d-5	[-1, 128, 7, 7]	8,320
MaxPool2d-6	[-1, 128, 4, 4]	0
Linear-7	[-1, 1200]	2,458,800
Linear-8	[-1, 10]	12,010

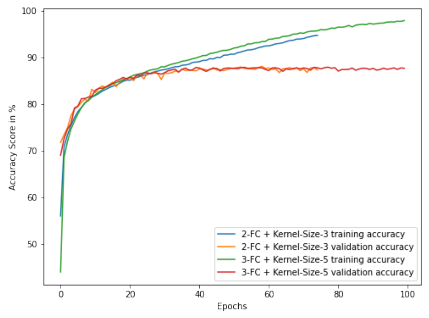
Table 2: Model summary of our proposed CNN model

4.3 Experimentations on custom CNN

As can be seen in the figures, dropout effectively prevents the model from over-fitting the training data, hence this modification is preferred. On the other hand, batch normalization encourages faster learning and earlier convergence of validation accuracy. The validation accuracy is also slightly improved. Lastly, adding additional fully connected layers does not improve accuracy and results in overfitting. Hence, the optimal candidate model applies drop-out, and batch-normalization and uses two fully connected layers. This consistently produce validation accuracy of over 87.5% which is higher than that of the VGG-based model. Hence, the model is selected as our proposed model and achieves a test accuracy of 87.7% on the Kaggle competition.

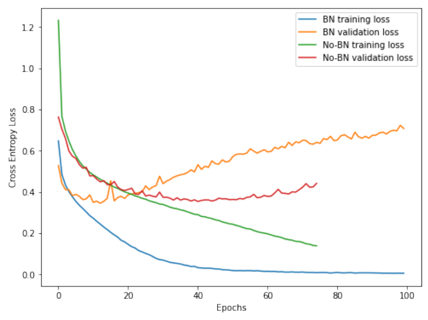


(a) Fully connected neurons and kernel size vs. average entropy loss

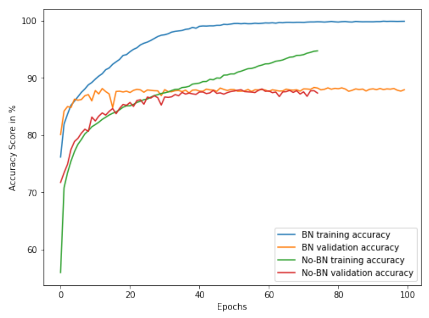


(b) Fully connected neurons and kernel size vs. validation accuracy

Figure 5: Comparison of fully connected neurons and kernel size

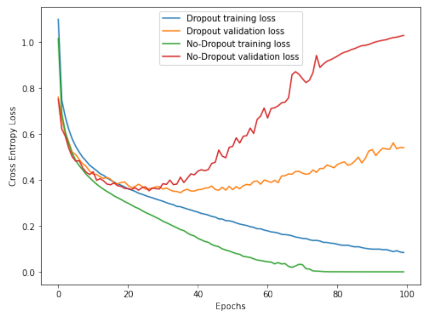


(a) Batch normalization vs. average entropy loss

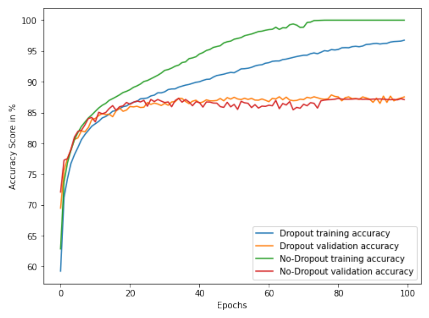


(b) Batch normalization vs. validation accuracy

Figure 6: Comparison of batch normalization



(a) Dropout vs. average entropy loss



(b) Dropout vs. validation accuracy

Figure 7: Comparison of adding adding dropout

5 Discussion and Conclusion

5.1 Major Findings

The largest finding of our project is that we should apply appropriate CNN models for image classification and modify it according to dataset. For example, if the given dataset is a grey-scale Fashion-MNIST, we should not consider an overly complicated model like AlexNet or VGG16 because this generally costs lots of time to train and may result in diminishing gradients.

Also, we realized that preprocessing is an important part of image classification. By extracting relevant images, we were able to achieve high accuracy with relatively simple models. The effectiveness of unsupervised learning technique in the prepossessing step also highlights the importance of performing data analysis before applying machine learning models.

Lastly, we learned about the effects of various modifications on CNN model on model performance as explained in section 4.3.

5.2 Future Investigation

As suggestions for future investigation, we have found certain algorithms from literature that can reach over 90% test accuracy in predicting labels for Fashion-MNIST[3]. Since we have simplified the problem with feature extraction to a Fashion-MNIST prediction problem, we can test models and techniques such as ResNet, MobileNet, and data augmentations techniques that may improve our validation accuracy to above 90%. Moreover, we can test different CNN model architectures and perform grid search on various hyper-parameters for hyper-parameter tuning of our proposed model.

6 Statement of Contribution

Jack Wei: Implementation of algorithms, optimized models, feature preprocessing, report
Grey Yuan: draft programming, feature preprocessing, graphs, tables, report
Yu Wu: literature review, tables, report and report organization
This project is completed as a team of 3, and each member worked actively and made significant contributions to the project.

References

[1] Zalando Research, https://www.kaggle.com/zalando-research/fashionmnist.

[2] Simonyan, K., Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. CoRR, abs/1409.1556.

[3] M. S. Tanveer, M. U. Karim Khan and C. -M. Kyung, "Fine-Tuning DARTS for Image Classification," 2020 25th International Conference on Pattern Recognition (ICPR), 2021, pp. 4789-4796, doi: 10.1109/ICPR48806.2021.9412221.

7 Appendix

```
# -*- coding: utf-8 -*-
"""ECSE551_Assignment3.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1SmpxahaXFWzE9luBZAWjxIulfm0iOD4-

# ECSE 551 Assignment 3
"""

from google.colab import drive

# General libraries
import pickle
import numpy as np
import cv2 as cv
import random as rd
import pandas as pd
from PIL import Image
import matplotlib.pyplot as plt
import skimage.transform as st
from google.colab import files

# Torch libraries
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torchvision import transforms
from torch.utils.data import Dataset
from torch.utils.data import DataLoader, TensorDataset

# Scikit Learn
from sklearn.model_selection import train_test_split
from sklearn.mixture import GaussianMixture

from sklearn.decomposition import PCA

drive.mount('/content/gdrive')
data_path = './gdrive/MyDrive/ECSE_551_Assignment_3/Data/'
model_path = './gdrive/MyDrive/ECSE_551_Assignment_3/Models/'

DEVICE = ("cuda" if torch.cuda.is_available() else "cpu")
print(DEVICE)

"""## Loading Data"""

train_data = pickle.load(open(data_path + 'Train.pkl', 'rb'))
test_data = pickle.load(open(data_path + 'Test.pkl', 'rb'))
train_targets = np.genfromtxt(data_path + 'Train_labels.csv', delimiter=',')
train_targets = train_targets[1:60001,1]
# data = np.squeeze(data, axis=1)

print("Number␣of␣train␣data:␣" + str(train_data.shape[0]))
print("Number␣of␣train␣targets:␣" + str(train_targets.shape[0]))
print("Number␣of␣test␣data:␣" + str(test_data.shape[0]))
print("Dimensions␣of␣a␣sample:", train_data.shape[1:])
print("Type␣of␣the␣dataset:", type(train_data))
print("Atomic␣datatype:", train_data.dtype)

"""## Dataset"""

def plt_grid(train_data, indices, label_on = True):
  fig, axarr = plt.subplots(3,3, figsize=(12,12))
  index = 0
  for ax in axarr.flat:
      ax.matshow(train_data[indices[index]][0], cmap='gray', interpolation="bicubic")
      ax.axis('off')
      if label_on:
        ax.set_title('Label:␣{}'.format(int(train_targets[indices[index]])))
      index += 1
  if not label_on:
    plt.subplots_adjust(wspace=0, hspace=0)
  plt.show()

print("Sample␣training␣images:")
plt_grid(train_data, range(0,9))

label = 0
indices = np.argwhere(train_targets == label)[:9][:,0]

print("Sample␣training␣images␣where␣the␣label␣is", label, ":")
plt_grid(train_data, indices)

print("Minimum␣Pixel␣Intensity:", train_data.min())
print("Maximum␣Pixel␣Intensity:", train_data.max())
print("Mean␣Pixel␣Intensity:", train_data.mean())
print("Pixel␣Intensity␣Std:", train_data.std())

sample_resized_alexnet = np.array([cv.resize(img,(227,227)) for img in train_data[:30]]) # default bilinear
print("Minimum␣Pixel␣Intensity:", sample_resized_alexnet.min())
print("Maximum␣Pixel␣Intensity:", sample_resized_alexnet.max())
print("Mean␣Pixel␣Intensity:", sample_resized_alexnet.mean())
print("Pixel␣Intensity␣Std:", sample_resized_alexnet.std())

"""## Preprocessing

### Cropping Quadrants
"""

qd1 = train_data[:,:,:14,14:].copy()
qd2 = train_data[:,:,14:,14:].copy()
qd3 = train_data[:,:,14:,:14].copy()
qd4 = train_data[:,:,:14,:14].copy()

qd1_t = test_data[:,:,:14,14:].copy()
qd2_t = test_data[:,:,14:,14:].copy()
qd3_t = test_data[:,:,14:,:14].copy()
qd4_t = test_data[:,:,:14,:14].copy()

qds = np.concatenate([qd1,qd2,qd3,qd4])
qds_t = np.concatenate([qd1_t,qd2_t,qd3_t,qd4_t])

plt_grid(qd1, range(0,9), False)

plt_grid(qd1_t, range(0,9), False)

"""## PCA Dimention Reduction"""

def normalize(data):
  data = data - data.mean()
  return (data/data.std())

# Flatten Training
flatten_qds = qds[:,0,:,:].reshape(len(qds), -1) * 255
flatten_qds = normalize(flatten_qds)

# Flatten Test
flatten_qds_t = qds_t[:,0,:,:].reshape(len(qds_t), -1) * 255
flatten_qds_t = normalize(flatten_qds_t)

# Dimention Reduction
pca = PCA(0.95)
redu_qds = pca.fit_transform(flatten_qds)
pca = PCA(0.95)
redu_qds_t = pca.fit_transform(flatten_qds_t)

gm = GaussianMixture(n_components=3, tol=1e-3, random_state=0)
gm.fit(redu_qds)
qds_gm_labels = gm.predict(redu_qds)

gm = GaussianMixture(n_components=3, tol=1e-3, random_state=0)
gm.fit(redu_qds_t)
qds_gm_t_labels = gm.predict(redu_qds_t)

qd1_gm_labels = qds_gm_labels[0:60000].copy()
qd2_gm_labels = qds_gm_labels[60000:120000].copy()
qd3_gm_labels = qds_gm_labels[120000:180000].copy()
qd4_gm_labels = qds_gm_labels[180000:240000].copy()
```

```
qd1_t_gm_labels = qds_gm_t_labels[0:10000].copy()
qd2_t_gm_labels = qds_gm_t_labels[10000:20000].copy()
qd3_t_gm_labels = qds_gm_t_labels[20000:30000].copy()
qd4_t_gm_labels = qds_gm_t_labels[30000:40000].copy()

np.bincount(qds_gm_labels)

np.bincount(qds_gm_t_labels)

fashion_imgs = []
digits_imgs = []
conflict_count = 0
for i in range(60000):
  fashion_index = qd1_gm_labels[2]
  labels = np.array([qd1_gm_labels[i],
                     qd2_gm_labels[i],
                     qd3_gm_labels[i],
                     qd4_gm_labels[i]])
  qds = [qd1[i], qd2[i], qd3[i], qd4[i]]

  indices = np.argwhere(labels == fashion_index)
  if(indices.size == 1):
    fashion_imgs.append(qds[indices[0,0]])
  else:
    confict_count += 1
    fashion_imgs.append(qds[indices[0,0]])

print("Number␣of␣conflict␣images␣(hard␣to␣classify):␣", conflict_count)

fashion_imgs = np.array(fashion_imgs)
plt_grid(fashion_imgs, range(9,18), False)

data = torch.from_numpy(fashion_imgs)
targets = torch.from_numpy(train_targets)

dataset = TensorDataset(data, targets)

print("Minimum␣Pixel␣Intensity:", fashion_imgs.min())
print("Maximum␣Pixel␣Intensity:", fashion_imgs.max())
print("Mean␣Pixel␣Intensity:", fashion_imgs.mean())
print("Pixel␣Intensity␣Std:", fashion_imgs.std())

np.bincount(qds_gm_t_labels)

fashion_imgs_test = []
conflict_count = 0
for i in range(10000):
  fashion_index = qd1_t_gm_labels[2]
  labels = np.array([qd1_t_gm_labels[i],
                     qd2_t_gm_labels[i],
                     qd3_t_gm_labels[i],
                     qd4_t_gm_labels[i]])
  qds_t = [qd1_t[i], qd2_t[i], qd3_t[i], qd4_t[i]]

  indices = np.argwhere(labels == fashion_index)
  if(indices.size == 1):
    fashion_imgs_test.append(qds_t[indices[0,0]])
  else:
    confict_count += 1
    fashion_imgs.append(qds_t[indices[0,0]])

print("Number␣of␣conflict␣images␣(hard␣to␣classify):␣", conflict_count)

fashion_imgs_test = np.array(fashion_imgs_test)
plt_grid(fashion_imgs_test, range(9,18), False)

MEAN = 0.28505665
STD = 0.31596282

"""## Model"""

class Dataset():
    def __init__(self, my_dataset, transform=None, is_test = False):
        self.dataset = my_dataset
        self.transform = transform
        self.is_test = is_test

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

    def __getitem__(self, idx):
        image = self.dataset[idx][0]
        if(self.is_test != True):
           label = self.dataset[idx][1]
        else:
           label = torch.empty((1,1), dtype=torch.int32)

        if self.transform is not None:
            # transfrom the numpy array to PIL image before the transform function
            image = self.transform(image)
        return image, label

SimpleNetTransform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((28, 28)),
    transforms.ToTensor(),
    transforms.Normalize((MEAN,), (STD,))
])

transform = SimpleNetTransform
batch_size = 50 #feel free to change it

train_dataset, val_dataset = torch.utils.data.random_split(dataset, [55000, 5000])

train_dataloader = DataLoader(Dataset(train_dataset, transform=SimpleNetTransform),batch_size=batch_size, shuffle=True)
val_dataloader = DataLoader(Dataset(val_dataset, transform=transform),batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(Dataset(fashion_imgs_test, transform=transform, is_test = True),batch_size=1, shuffle=False)

# Read a batch of data and their labels and display them

[imgs, labels] = (next(iter(train_dataloader)))
# imgs = np.squeeze(imgs)
# plt.imshow(imgs[50].numpy(),cmap='gray',interpolation="bicubic") #.transpose()
img_grid = torchvision.utils.make_grid((imgs + MEAN)*STD)

plt.figure(figsize=(20,15))
plt.imshow(img_grid.permute(1, 2, 0), cmap='gray')
plt.show()

print(labels)

# training batches of our network
EPOCHS = 100

class VGG_Basic(nn.Module):
    def __init__(self):
        super(VGG_Basic, self).__init__()

        self.conv1 = nn.Conv2d(1, 32, kernel_size = 3, padding = 'same')
        self.conv2 = nn.Conv2d(32, 64, kernel_size = 3, padding = 'same')
        self.conv3 = nn.Conv2d(64, 128, kernel_size = 3, padding = 'same')
        self.conv4 = nn.Conv2d(128, 256, kernel_size = 3, padding = 'same')

        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(256, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.maxpool(F.relu(self.conv1(x)))
        x = self.maxpool(F.relu(self.conv2(x)))
        x = self.maxpool(F.relu(self.conv3(x)))
        x = self.maxpool(F.relu(self.conv4(x)))

        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, 0.5)
        x = F.relu(self.fc2(x))
        return x

class VGG_Ver2(nn.Module):
    def __init__(self):
        super(VGG_Ver2, self).__init__()

        self.conv1 = nn.Conv2d(1, 32, kernel_size = 3, padding = 'same')
        self.conv2 = nn.Conv2d(32, 32, kernel_size = 3, padding = 'same')
        self.conv3 = nn.Conv2d(32, 64, kernel_size = 3, padding = 'same')
        self.conv4 = nn.Conv2d(64, 64, kernel_size = 3, padding = 'same')
        self.conv4 = nn.Conv2d(64, 64, kernel_size = 3, padding = 'same')
        self.conv5 = nn.Conv2d(64, 128, kernel_size = 3, padding = 'same')
        self.conv6 = nn.Conv2d(128, 128, kernel_size = 3, padding = 'same')
        self.conv6 = nn.Conv2d(128, 128, kernel_size = 3, padding = 'same')
        self.conv7 = nn.Conv2d(128, 256, kernel_size = 3, padding = 'same')
        self.conv8 = nn.Conv2d(256, 256, kernel_size = 3, padding = 'same')
        self.conv8 = nn.Conv2d(256, 256, kernel_size = 3, padding = 'same')

        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(256, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.maxpool(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.maxpool(F.relu(self.conv4(x)))
        x = F.relu(self.conv5(x))
        x = F.relu(self.conv6(x))
        x = self.maxpool(F.relu(self.conv6(x)))
        x = F.relu(self.conv7(x))
        x = F.relu(self.conv8(x))
        x = self.maxpool(F.relu(self.conv8(x)))

        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, 0.5)
        x = F.relu(self.fc2(x))
        return x

class VGG_Ver3(nn.Module):
    def __init__(self):
        super(VGG_Ver3, self).__init__()

        self.conv1 = nn.Conv2d(1, 32, kernel_size = 3, padding = 'same')
        self.conv2 = nn.Conv2d(32, 64, kernel_size = 3, padding = 'same')
        self.conv3 = nn.Conv2d(64, 64, kernel_size = 3, padding = 'same')
        self.conv4 = nn.Conv2d(64, 128, kernel_size = 3, padding = 'same')
        self.conv5 = nn.Conv2d(128, 128, kernel_size = 3, padding = 'same')
        self.conv6 = nn.Conv2d(128, 256, kernel_size = 3, padding = 'same')
        self.conv7 = nn.Conv2d(256, 256, kernel_size = 3, padding = 'same')

        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(256, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.maxpool(F.relu(self.conv1(x)))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = self.maxpool(F.relu(self.conv4(x)))
        x = F.relu(self.conv5(x))
        x = self.maxpool(F.relu(self.conv6(x)))
        x = F.relu(self.conv7(x))
        x = self.maxpool(F.relu(self.conv8(x)))

        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, 0.5)
        x = F.relu(self.fc2(x))
        return x

class MyConvNet1(nn.Module):
    def __init__(self):
        super(MyConvNet1, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, padding='same')
        self.pool = nn.MaxPool2d(2, padding=0)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, padding='same')
        self.conv3 = nn.Conv2d(64, 128, kernel_size=1, padding='same')
        self.fc1 = nn.Linear(1152, 512)
        self.fc2 = nn.Linear(512, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        # x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)

class MyConvNet2(nn.Module):
    def __init__(self):
        super(MyConvNet2, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, padding='same')
        self.pool = nn.MaxPool2d(2, padding=0, ceil_mode=True)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, padding='same')
        self.conv3 = nn.Conv2d(64, 128, kernel_size=1, padding='same')
        self.fc1 = nn.Linear(2048, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = F.relu(self.fc2(x))
        x = F.dropout(x, training=self.training)
        x = F.relu(self.fc3(x))
        return F.log_softmax(x, dim=1)

class MyConvNet3(nn.Module):
    def __init__(self):
        super(MyConvNet3, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding='same')
        self.pool = nn.MaxPool2d(2, padding=0, ceil_mode=True)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding='same')
        self.conv3 = nn.Conv2d(64, 128, kernel_size=1, padding='same')
        self.fc1 = nn.Linear(2048, 1000)
        self.fc2 = nn.Linear(1000, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = F.relu(self.fc2(x))
        return F.log_softmax(x, dim=1)

class MyConvNet4(nn.Module):
    def __init__(self):
        super(MyConvNet4, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding='same')
        self.norm1 = nn.BatchNorm2d(32)
        self.pool = nn.MaxPool2d(2, padding=0, ceil_mode=True)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding='same')
        self.norm2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=1, padding='same')
        self.norm3 = nn.BatchNorm2d(128)
        self.fc1 = nn.Linear(2048, 1200)
        self.fc2 = nn.Linear(1200, 10)

    def forward(self, x):
        x = self.pool(self.norm1(F.relu(self.conv1(x))))
        x = self.pool(self.norm2(F.relu(self.conv2(x))))
        x = self.pool(self.norm3(F.relu(self.conv3(x))))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = F.relu(self.fc2(x))
        return F.log_softmax(x, dim=1)

from torchsummary import summary
model = VGG_Ver2().to(DEVICE)
summary(model, (1, 28, 28))

model = MyConvNet4().to(DEVICE)
summary(model, (1, 28, 28))

# Select model for training here
model = VGG_Ver2().to(DEVICE)
criterion_cel = nn.CrossEntropyLoss()
criterion_nll = nn.NLLLoss()
adam_optimizer = optim.Adam(model.parameters())
sgd_optimizer = optim.SGD(model.parameters(), lr=0.01,
                          momentum=0.5)

def train(model, epoch, train_loader, optimizer, criterion):

    correct_train_pred = 0
    train_loss = 0

    model.train()
    for train_batch_idx, (data, target) in enumerate(train_loader):
        target = target.type(torch.LongTensor)
        data, target = data.to(DEVICE), target.to(DEVICE)
        optimizer.zero_grad() # Clear the gradients
        torch.cuda.empty_cache()
        output = model(data)  # Forward pass
        loss = criterion(output, target) # Compute loss
        loss.backward()       # Backward propagation
        optimizer.step()      # Update weights

        _, pred = torch.max(output, 1)
        correct_train_pred += (pred == target).sum()

        train_loss += loss.item();

        if (train_batch_idx + 1) % 30 == 0:
          print("Epoch:{}␣[{}/{}␣({:.0f}%)]\tTrain␣Loss:␣{:.6f}\tTrain␣accuracy:␣{:.2f}%".format(
                epoch, train_batch_idx * len(data), len(train_loader.dataset),
                100. * train_batch_idx / len(train_loader), loss.item(),
                correct_train_pred.item() / (len(data) * (train_batch_idx + 1)) * 100))

    return train_loss / len(train_loader), correct_train_pred.item() / (len(data) * (train_batch_idx + 1)) * 100

def validate(model, val_loader, criterion):
  correct_val_pred = 0
  val_loss = 0

  model.eval()
  with torch.no_grad():
      for val_batch_index, (data, target) in enumerate(val_loader):
          target = target.type(torch.LongTensor).to(DEVICE)
          data, target = data.to(DEVICE), target.to(DEVICE)
          output = model(data)
          val_loss += criterion(output, target).item()
          _, pred = torch.max(output, 1)
          correct_val_pred += (pred == target).sum()
  accuracy = 100. * correct_val_pred.item() / (len(data) * len(val_loader))
  print('Test:␣Avg.␣loss:␣{:.6f},␣Accuracy:␣{:.2f}%\n'.format(val_loss/len(val_loader.dataset), accuracy))

  return val_loss/len(val_loader), accuracy

train_loss_arr = []
train_acc_arr = []
val_loss_arr = []
val_acc_arr = []
for epoch in range(1, EPOCHS+1):
  train_loss, train_acc = train(model,epoch,train_dataloader,sgd_optimizer,criterion_cel)
  val_loss, val_acc = validate(model, val_dataloader,criterion_cel)
  train_loss_arr.append(train_loss)
  train_acc_arr.append(train_acc)
  val_loss_arr.append(val_loss)
  val_acc_arr.append(val_acc)

np.save(model_path + 'No_dropout_MyConvNet1_train_loss.npy', np.array(train_loss_arr))
np.save(model_path + 'No_dropout_MyConvNet1_val_loss.npy', np.array(val_loss_arr))
np.save(model_path + 'No_dropout_MyConvNet1_train_acc.npy', np.array(train_acc_arr))
np.save(model_path + 'No_dropout_MyConvNet1_val_acc.npy', np.array(val_acc_arr))

torch.save(model.state_dict(), model_path + 'MyConvNet4_97_train_89_val.pt')

model = Net().to(DEVICE)
model.load_state_dict(torch.load(model_path + 'net_71.pt'))

pred_arr = []
model.eval()
with torch.no_grad():
  for data, target in test_dataloader:
    data, target = data.to(DEVICE), target.to(DEVICE)
    output = model(data)
    pred = output.data.max(1, keepdim=True)[1]
    pred_arr.append(pred[0,0])

for i in range(len(pred_arr)):
  pred_arr[i] = pred_arr[i].cpu().item()

def download_csv(pred):
  output_df = pd.DataFrame(pred_arr, columns = ['class'])
  output_df.insert(0, 'id', output_df.index)
  output_df.to_csv("submission.csv", index=None)
  files.download('submission.csv')

download_csv(pred_arr)

dropout_train_acc =np.load(model_path + 'MyConvNet1_train_acc_final.npy')
dropout_val_acc = np.load(model_path + 'MyConvNet1_val_acc_final.npy')
dropout_train_loss = np.load(model_path + 'MyConvNet1_train_loss_final.npy')
dropout_val_loss = np.load(model_path + 'MyConvNet1_val_loss_final.npy')

no_dropout_train_acc =np.load(model_path + 'No_dropout_MyConvNet1_train_acc.npy')
no_dropout_val_acc = np.load(model_path + 'No_dropout_MyConvNet1_val_acc.npy')
no_dropout_train_loss = np.load(model_path + 'No_dropout_MyConvNet1_train_loss.npy')
no_dropout_val_loss = np.load(model_path + 'No_dropout_MyConvNet1_val_loss.npy')

plt.figure(figsize=(8,6))

plt1 = plt.plot(dropout_train_acc)
plt1 = plt.plot(dropout_val_acc)

plt1 = plt.plot(no_dropout_train_acc)
plt1 = plt.plot(no_dropout_val_acc)

plt.xlabel('Epochs')
plt.ylabel('Accuracy␣Score␣in␣%')

plt.legend(['Dropout␣training␣accuracy','Dropout␣validation␣accuracy','No-Dropout␣training␣accuracy', 'No-Dropout␣validation␣accuracy'])

plt.show()

plt.figure(figsize=(8,6))

plt1 = plt.plot(dropout_train_loss)
plt1 = plt.plot(dropout_val_loss)

plt1 = plt.plot(no_dropout_train_loss)
plt1 = plt.plot(no_dropout_val_loss)

plt.xlabel('Epochs')
plt.ylabel('Cross␣Entropy␣Loss')

plt.legend(['Dropout␣training␣loss','Dropout␣validation␣loss','No-Dropout␣training␣loss', 'No-Dropout␣validation␣loss'])

plt.show()

ver3_train_acc =np.load(model_path + 'MyConvNet3_train_acc.npy')
ver3_val_acc = np.load(model_path + 'MyConvNet3_val_acc.npy')
ver3_train_loss = np.load(model_path + 'MyConvNet3_train_loss.npy')
ver3_val_loss = np.load(model_path + 'MyConvNet3_val_loss.npy')
```

```
ver2_train_acc =np.load(model_path + 'myconv2_train_acc.npy')
ver2_val_acc = np.load(model_path + 'myconv2_val_acc.npy')
ver2_train_loss = np.load(model_path + 'myconv2_train_loss.npy')
ver2_val_loss = np.load(model_path + 'myconv2_val_loss.npy')

plt.figure(figsize=(8,6))

plt1 = plt.plot(ver3_train_acc)
plt1 = plt.plot(ver3_val_acc)

plt1 = plt.plot(ver2_train_acc)
plt1 = plt.plot(ver2_val_acc)

plt.xlabel('Epochs')
plt.ylabel('Accuracy␣Score␣in␣%')

plt.legend(['2-FC␣+␣Kernel-Size-3␣training␣accuracy','2-FC␣+␣Kernel-Size-3␣validation␣accuracy','3-FC␣+␣Kernel-Size-5␣training␣accuracy', '3-

plt.show()
# plt2 = plt.plot(batch_train_loss)
# plt2 = plt.plot(batch_val_loss)

plt.figure(figsize=(8,6))

plt1 = plt.plot(ver3_train_loss)
plt1 = plt.plot(ver3_val_loss)

plt1 = plt.plot(ver2_train_loss)
plt1 = plt.plot(ver2_val_loss)

plt.xlabel('Epochs')
plt.ylabel('Cross␣Entropy␣Loss')

plt.legend(['2-FC␣+␣Kernel-Size-3␣training␣loss','2-FC␣+␣Kernel-Size-3␣validation␣loss','3-FC␣+␣Kernel-Size-5␣training␣loss', '3-FC␣+␣Kernel-

plt.show()

plt.figure(figsize=(8,6))

plt1 = plt.plot(vgg1_train_acc_arr)
plt1 = plt.plot(vgg1_val_acc_arr)

plt1 = plt.plot(vgg2_train_acc_arr)
plt1 = plt.plot(vgg2_val_acc_arr)

plt.xlabel('Epochs')
plt.ylabel('Cross␣Entropy␣Loss')

plt.legend(['base␣VGG␣training␣accuracy','base␣VGG␣validation␣accuracy','VGG␣with␣stacked␣layers␣training␣accuracy', 'VGG␣with␣stacked␣layers

plt.show()
```