Vamsi Goriparthi

University at Albany

Course code: CINF 624

Faculty Name: Carol Cusano

Dec 10, 2021

Final Model Build for Anomaly Detection Using Encoder Decoder

Outlier detection models are classification models that classify outliers and normal data points in each dataset. Normally outliers in a dataset are very few, therefore, it is paramount to classify them correctly and to not misclassify normal data as outlier in the process. Neural networks models like Encoder-Decoder models that project the input data into a latent space and then recreate the original dataset from the latent space are widely used in outlier detection/ classification. The analysis undertaken in this project is to make an outlier detection/ classification model on Fashion-MNIST from Zolando’s article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. This is available at the following Github page:<https://github.com/zalandoresearch/fashion-mnist>

The labels are T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag and Ankle boot. This dataset contains equal distribution of each label. Eight of these labels have been retained as the parent dataset. The model is trained on the eight labels and post this, if any different image is fed to the model, the model will be able to classify the image as an outlier. The labels Trouser and Sneaker will be kept out of the training sample. These will be treated as outliers and mixed finally with the few other labels included in the training and the model will be required to predict which of the images are outliers.

## **Figure 1**

Images for Outlier detection

Text

Description automatically generated with low confidence

Note. Each image in this figure corresponds to a suite of Fashion-MNIST items, used to outlier detection/ classification model.

Each image in the dataset can be described as 28 pixels in height and 28 pixels in width, a total of 784 pixels. Each pixel has a value associated with it indicating the darkness of that pixel, higher numbers meaning darker. The values range from 0 to 255. The dataset has been specifically created to be used for benchmarking machine learning algorithms. Since all the images consist of 784 pixels with values from 0 to 255 associated with each pixel, they form 10 homogenous groups of different images. Once the model has been trained on the dataset, it will be tested on a test data containing outliers. The performance of the model will be tested based on the precision and recall of outlier detection. Ideally, both the metrics should be very high to justify the use of the model since the outliers will be only few in comparison to the normal data points.

## Goals

The Goal of the project is to build an outlier detection model on fashion dataset that can find out if an image is an outlier. The outcome of the project will be a model that can be trained on any kind of image dataset to predict outlier images.

## Evaluation

The evaluation will be based on the precision and recall of the model’s prediction of the outlier and non-outliers correctly. The deliverables are a code to source the FASHION MNIST data, and to convert it into an anomaly detection type training and test data; bias checks in the dataset; descriptive statistics; an encoder-decoder neural network-based model trained on the training data; the result of the optimized model on classifying the test data into anomaly and not anomaly.

The outcome of the project will be to gain an understanding of how neural networks and specifically encoder-decoder models can be used to learn latent dimensions and used in predictive modelling specifically to predict outliers from non-outliers.

# Data Overview and Collection

**Data-Generation Process**

The data used for the outlier detection analysis has been sampled from 8 classes of the training data of 60000 samples. The following steps have been taken to generate the data from the Parent Fashion MNIST dataset.

* From the eight classes excluding trouser and sneaker each containing 6000 samples, 4000 images has been randomly sampled giving a random seed for reproducibility. These 4000 samples has been kept for their respective labels to form the training data for the encoder decoder model.
* The remaining 2000 images for each of the eight labels have been chosen to be in the test data.
* From the remaining 2 labels i.e. trouser and sneaker, a sample of 500 each has been randomly sampled to form the sample of outliers.
* These outliers have been mixed with the 2000 images of the test data and that forms the testing data for the model.

Therefore, the training data consists of 32000 images whereas the test data consists of 17000 images. The random allocation of images from the already balanced dataset further ensures that there is no imbalance of any nature in the data i.e. each of the labels from the 8 chosen in the training data are equally represented. An artificial imbalance has been created in the test data by keeping less number of 2 types of images on which the model has not been trained. This has been done purposely because in real world scenarios, outliers are very few and they come mixed with the data under observation. The success of the model depends upon its precision to identify the outliers correctly and to not identify the non-outliers as outliers. Due to construction, the accuracy of the model can be very high if the model just predicts all the labels to be non-outliers. However, the task of the model is to identify outliers correctly and the performance will be judged based on this. The table below gives the frequency of data in the training and test data.

**Table 1**

Frequency of labels in the Sampled data

|  |  |  |
| --- | --- | --- |
| Label | Count Train | Count test |
| Ankleboot | 4000 | 2000 |
| Bag | 4000 | 2000 |
| Sneaker | 0 | 500 |
| Shirt | 4000 | 2000 |
| Sandal | 4000 | 2000 |
| Coat | 4000 | 2000 |
| Dress | 4000 | 2000 |
| Pullover | 4000 | 2000 |
| Trouser | 0 | 500 |
| T-shirt/top | 4000 | 2000 |

**Table 2**

summary of the datatype, missing values

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Label | Missing in X | Missing in Label | Datatype | mean | standard deviation |
| T-shirt/top | 0 | 0 | uint8 | 83.03 | 89.44 |
| Pullover | 0 | 0 | uint8 | 96.06 | 91.46 |
| Dress | 0 | 0 | uint8 | 66.02 | 90.33 |
| Coat | 0 | 0 | uint8 | 98.26 | 95.96 |
| Sandal | 0 | 0 | uint8 | 34.87 | 67.09 |
| Shirt | 0 | 0 | uint8 | 84.61 | 86.52 |
| Bag | 0 | 0 | uint8 | 90.16 | 93.14 |
| Ankle boot | 0 | 0 | uint8 | 76.81 | 94.49 |

Table 2 Summary of the training data

It can be observed from the above summary table that:

* there are no missing values.
* The data type is same across all the data in the training set
* The mean values are quite dissimilar across the labels while the standard deviation of sandal is only different from that of all others. The mean for the labels have been calculated by flattening the images into 1-dimensional arrays.

The test data also has similar characteristics. It has no missing data.

## Bias in the data

Balance in the data:

Since the data has equal number of instances in the training data, the training process will not be biased. Each label will be learnt with equal focus by the model. The test data is imbalanced due to construction. Two labels have been intentionally given lower frequency since they act as outliers.

Label bias:

There is no label bias in the data since all the type of labels have been equally weighted while sampling from the total population. Four thousand samples have been selected from 6000 samples for each label by random selection.

Sample bias:

From each of the label, the data has been sampled equal emphasis and randomly. The process for sample selection is same for each label.

# Exploring the Data

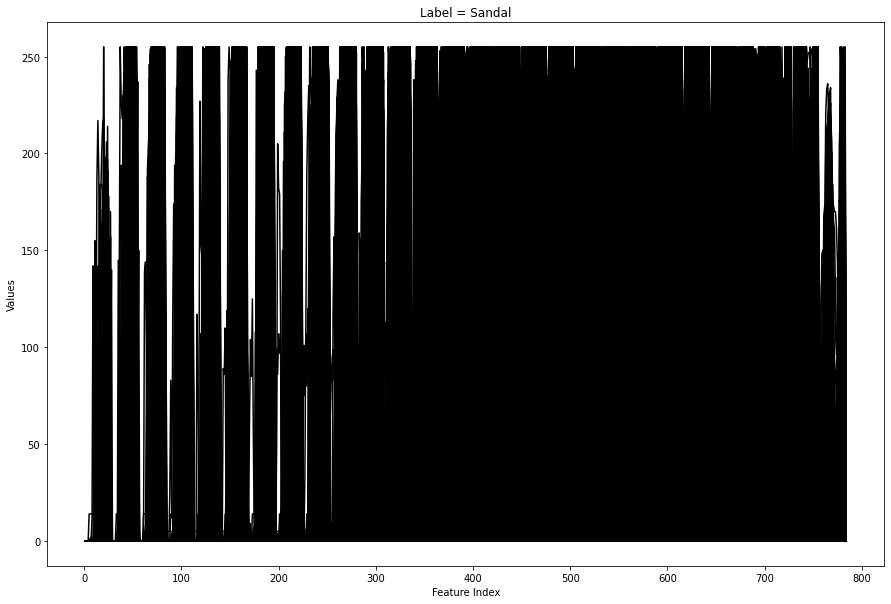
## Discoveries and insights

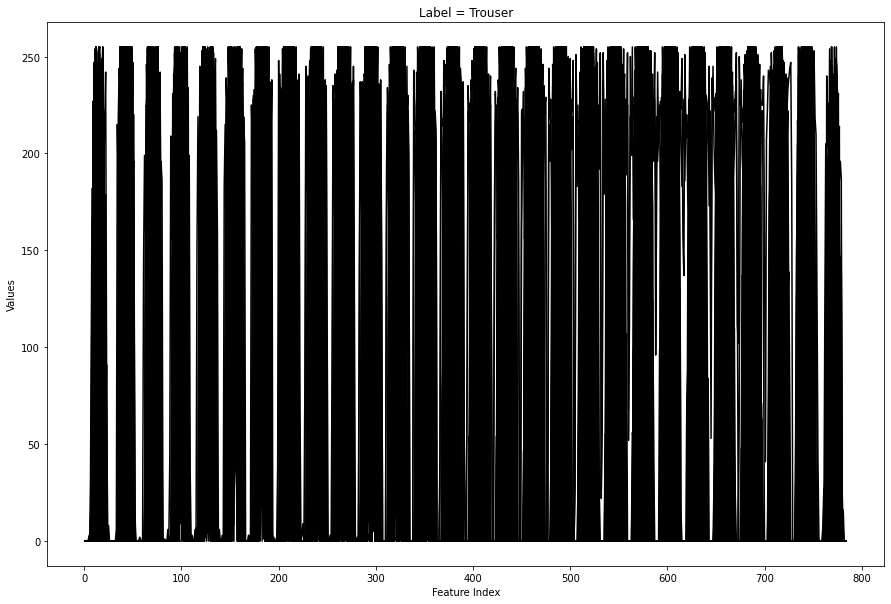
The dataset contains 10 different type of fashion related pictures of T-shirt/top, Trouser, Pullover etc. The pictures are 28\*28 grayscale images. Therefore, the shape of the training data is 8 x 4000 x 28 x 28. The test data has two different sets i.e. one of sneaker and trouser images on which the model will not be trained and they will be treated as outliers and the other set is that of other images which the model is familiar since it has been trained on other instances of same type of images.

While analyzing the pixel distributions in the datasets, it has been observed that different types of images have different types of distribution of dark and white areas. For example, given below is the distribution of pixel values for sandal and trouser.

**Figure 2**

distribution of pixel values for sandal and trouser.



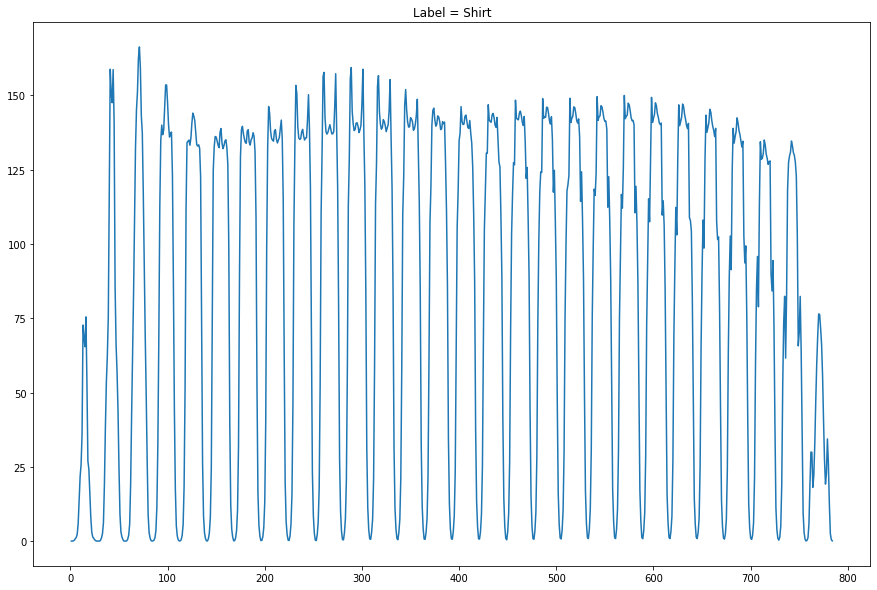


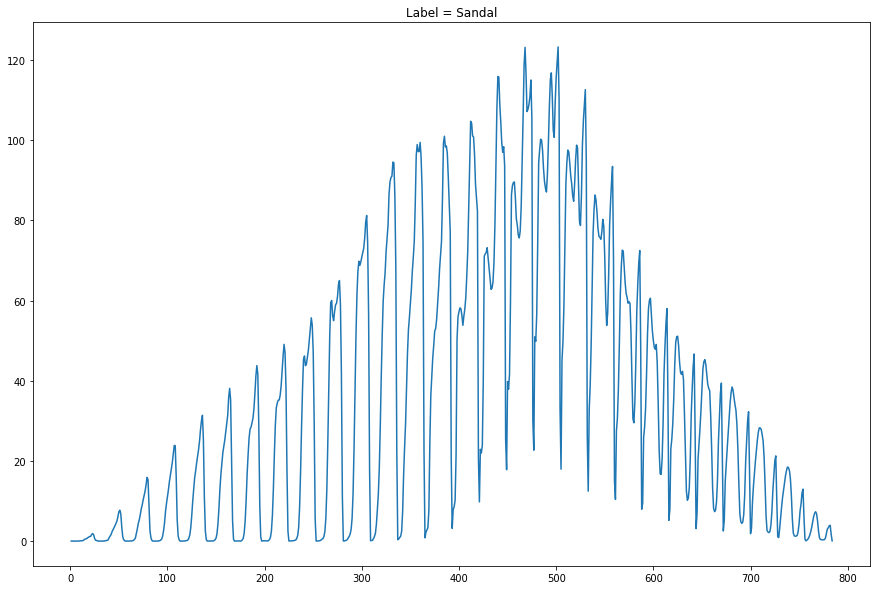
Note: The above distributions show an inherent difference in distributions for each type of images. Thus, models can learn from the differences and understand outliers.

It can be observed that the beyond the index 300, the values of pixels for sandals are quite large and show no periodic increase and decrease of pixel values like that in trouser. It has been observed that there is an intrinsic variation in the distribution and spread of pixel values for various types of images. For example, the maxima of pixel values occurs near the 500th pixel for sandals as shown in the images below and that for shirt occurs near the 100th pixel for shirt.

**Figure 3**

distribution of mean values for shirt and sandal.



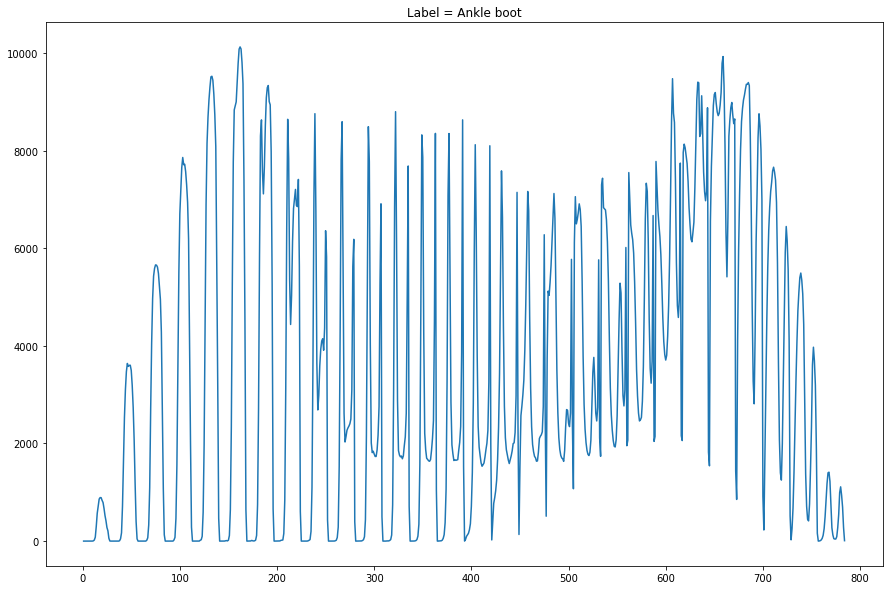


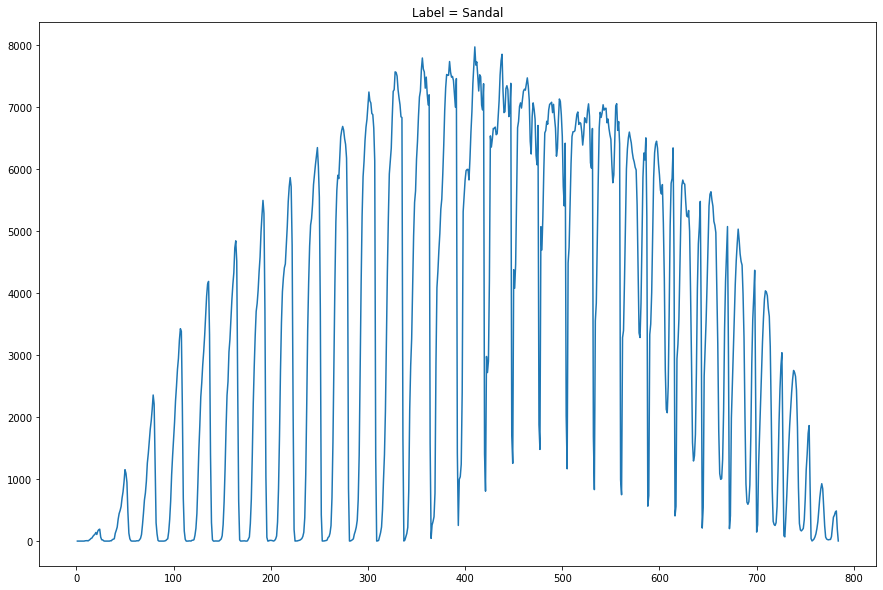
Note: The above distribution of mean values shows the differences in concentration of pixel values which can help detect an image from the other.

This distributional difference in pixels will be captured by the encoder-decoder models. Convolutional neural networks(CNN) that are adapted for image recognition will be used to compress the images to latent dimensions and regenerate them to the original dimensions.

**Figure 4**

variance of values for different labels

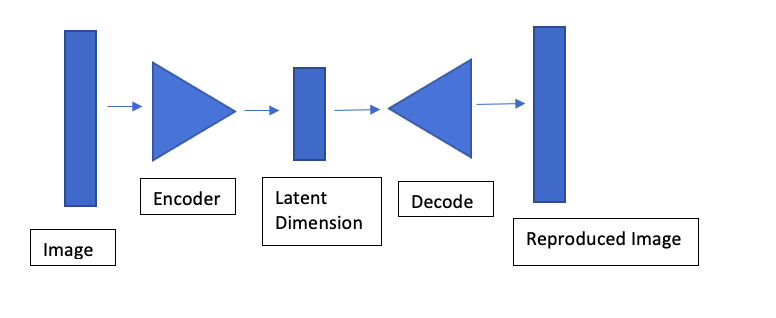




It can be observed from the above plots that the distribution of variance across pixels is different for each type of labels. For example, an ankle boot has higher variance towards 100-250 indices and 600–700-pixel indices while a sandal has higher variance in 400-500 index region.

# Modelling the data

The data has been modelled using an encoder decoder model. The image is fed to the encoder, the encoder projects it into a latent dimension after eliminating noise. The decoder process the output from the latent dimension and reproduces the input image. The flowchart of the process is given below.



The latent dimensions contain information about features that are not very specific to some given image. Rather, they contain information that can be generalized across a class of image fed to it. The encoder and the decoder are exact replicas of each other, and the image passes in a sequential manner. The reproduced image and the input image are compared to each other once the model has been trained. This model has been trained using “Keras” with Tensorflow backend.

Given below are the details of the encoder-decoder model:

* Input layer with shape (28,28,1)
* Two-dimensional convolutional (CNN) layer with filter size 64, kernel size 3 with same padding
* Batch Normalization on the previous layer outcome
* Leaky Rectified linear unit (ReLU) layer post the batch normalization output
* Two-dimensional convolutional layer with 32 filters, kernel size of 3, 2 strides, same padding post which batch normalization and Leaky ReLU are applied.
* Deconvolution layer with 32 filters, kernel size of 3, 2 strides and same padding post which batch normalization and Leaky ReLU are applied
* Deconvolution layer with 64 filters, kernel size of 3, 1 strides and same padding post which batch normalization and Leaky ReLU are applied
* Output Two-dimensional convolutional layer with 1 filter, kernel size 3, same padding and sigmoid activation.

Kernel size of 32 and 64 are standard in CNN architecture. Kernel size of 3 implies the 2D size of the convolution window of 3x3. Batch normalization refers to the process of standardizing each input in each batch input to a neural network. It stabilizes the learning process. Leaky ReLU is an activation function widely used in neural networks. Stride refers to the movement size of CNN window, e.g. for a stride of 1, the window models 1 pixel at a time. The ‘same’ padding refers to padding an image such that for a stride of 1, the output of the image passing through a CNN comes out to be of the same size as the input. (TensorFlow Core v2.7.0, n.d.)

The above model was compiled with an optimizer ‘Adam’ in tensorflow and the model was compiled using structural similarity index measure between input and output as the loss function.Optimizer implies the algorithm which is used to minimize the loss function with respect to the model’s parameters. Given below are the hyper-parameters of the model.

- Epochs = 5( the number of passes through which the model is trained )

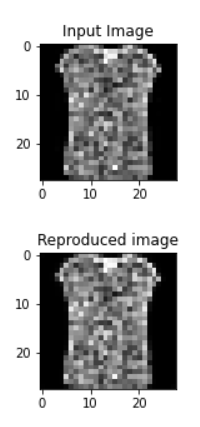
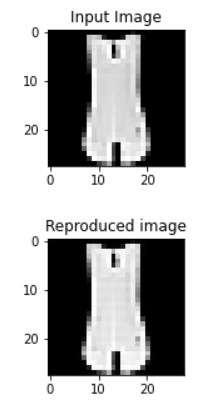
- Batch size = 128 ( pretty standard for CNN optimization )

- Shuffle = True ( the shuffling of data )

After training the model, the structural similarity index measure(SSIM ) for the training data is 0.0092.

**Figure 6**

plots showing how the reproduced images look in comparison to the original images



It can be observed from the above plots that the image reconstruction is quite good on the training data. Using the same model, predictions were made on the test data as well.

**Figure 7**

reconstructed images from the test set

Chart

Description automatically generatedGraphical user interface, chart

Description automatically generated

It can be observed from the above figures that the unseen images have also been reconstructed accurately by the model. This might create problems in outlier detection since theoutlier images are also getting reconstructed well so the SSIM loss will be small for the outliers as well. The distribution of SSIM loss has been plotted below for the test set.

**Figure 8**

Ssim error plot



From the above histogram, it appears that a threshold of 0.02 should be used to identify the outlier from a normal image based on SSIM error. Using this threshold, the following results have been obtained.

|  |  |  |
| --- | --- | --- |
| labels | count | Full count |
| T-shirt/top | 38 | 2000 |
| Trouser | 1 | 500 |
| Pullover | 17 | 2000 |
| Dress | 11 | 2000 |
| Coat | 6 | 2000 |
| Sandal | 11 | 2000 |
| Shirt | 33 | 2000 |
| Sneaker | 5 | 500 |
| Bag | 18 | 2000 |
| Ankle boot | 7 | 2000 |

The above table implies that the outliers have not been detected well. Therefore, the model is regenerating the outlier images too well. The code for the model is given in the appendix.

## Validating the model

The model has been tested on a dataset of 17000 images that contains 1000 outlier images. The model was not trained on this dataset. The efficiency of the model has been tested based on the true positive on the normal and outliers in this set. Given below is the definition of true positives

* True Positive = Number of Outliers from all the outliers that are identified correctly.

Given below is the performance of outlier detection model.

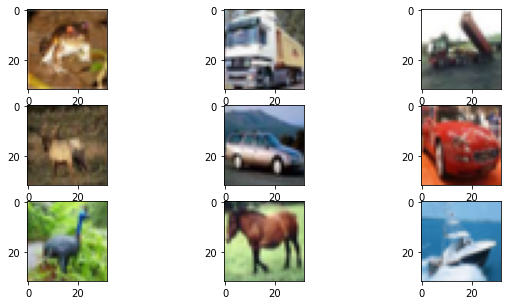
Performance on Trouser = 1/500 = 0.2%

Performance on Sneaker = 5/500 = 1%

## Subsequent Model Build

Due to low accuracy of the model build previously on the Fashion MNIST data, it has been concluded that the model used to fit to the data is very powerful while the data is too simple. Therefore, the model reproduces the given images correctly. So, the complexity of the data has been increased while the model’s complexity has been decreased. The new dataset in use is the CIFAR-10 dataset that contains colored images of various things like airplanes, automobiles etc. The dataset is also available with Tensorflow in Python. The images in this dataset have a shape (32, 32, 3) i.e. there are more pixels in each image as compared to the images in the Fashion MNIST data.

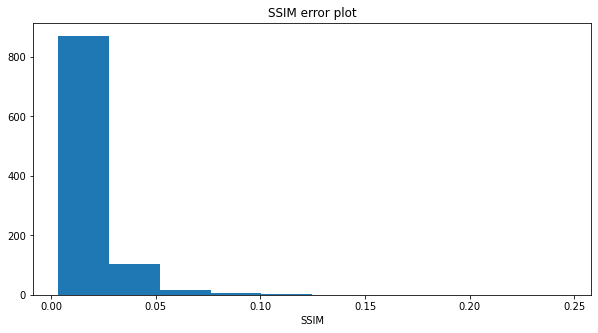
Given below are example images :



The following method has been used to create the training and test dataset and to build the model in it.

* There are a total of 60000 images in the CIFAR-10 data. From these 60000 images, only 9000 images have been selected for the model. From these, 8000 images form the training set while 1000 images form the test set. The outliers contain images of “automobiles” and “horses”. Since this dataset has colored images ( i.e. 3 channels ), therefore it is more complex as compared to the MNIST or Fashion MNIST data.
* The batch normalization has been removed from the model structure. Therefore, the model complexity has been reduced since batch normalization produces better results on average.

The model has been trained with 10 epochs and with a batch size of 128. The SSIM loss has been again calculated on the test set and given below is the histogram of SSIM loss.



Based on the above histogram, the threshold for outlier calculation has been taken as 0.035. There are 100 images of 10 labels therefore, there are also 100 images each of horses and automobiles. Given below are the results of outlier detection:

|  |  |  |
| --- | --- | --- |
| labels | count | Accuracy |
| airplane | 8 | 92% |
| automobile | 16 | 16% |
| bird | 8 | 92% |
| cat | 3 | 97% |
| deer | 12 | 88% |
| dog | 1 | 99% |
| frog | 8 | 92% |
| horse | 2 | 2% |
| ship | 8 | 92% |
| truck | 13 | 87% |

From the above table, the non-outliers are detected with very good accuracy while the outlier automobile is detected with 16% accuracy and the horses are detected only with 2% accuracy.

## Conclusion

The results above show that the model regenerates images quite well but performance of the model in terms of outlier detection on the test data created from the Fashion MNIST data is extremely poor. Therefore, the dataset for testing has been altered and the CIFAR10 dataset which is more complex than the Fashion MNIST data has been used. The complexity of the model has also been reduced. This way, the new model on the new data is better able to detect one of the outlier labels i.e. automobiles while the detection of other label i.e. “horses” is bad. However, the new less complex model on the new more complex data is performing better as compared to the previous model on the Fashion MNIST data.

# Bibliography

*TensorFlow Core v2.7.0*. (n.d.). Retrieved from Tensorflow: https://www.tensorflow.org/api\_docs/python/tf/keras/layers/Conv2DTranspose

**Appendix**

**######### OLD Model code ################**

**# Step 1. Importing libraries**

import tensorflow as tf

import numpy as np

import pandas as pd

from matplotlib import pyplot

import matplotlib.pyplot as plt

from keras.datasets import fashion\_mnist, mnist

# load dataset

(trainX, trainy), (testX, testy) = fashion\_mnist.load\_data()

# summarize loaded dataset

print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))

print('Test: X=%s, y=%s' % (testX.shape, testy.shape))

# plot first few images

for i in range(9):

# define subplot

pyplot.subplot(330 + 1 + i)

# plot raw pixel data

pyplot.imshow(trainX[i], cmap=pyplot.get\_cmap('gray'))

# show the figure

pyplot.show()

**# Step 2. Generating training and test data**

np.random.seed(1)

full\_data = []

full\_data\_y = []

full\_data\_test = []

full\_data\_test\_y = []

for j in list(set(trainy)):

if j not in [1,7]:

temp\_data = trainX[trainy==j,]

temp\_y = trainy[trainy==j]

print(j, sum(sum(sum(np.isnan(temp\_data)))), sum(np.isnan(temp\_y)),temp\_data.dtype, temp\_y.dtype, temp\_data.flatten().mean(),temp\_data.flatten().std())

index = np.random.choice(temp\_data.shape[0], 4000, replace=False)

left\_index = [k for k in range(temp\_data.shape[0]) if k not in index]

full\_data\_test.append(temp\_data[left\_index,])

full\_data.append(temp\_data[index,])

full\_data\_y.append(temp\_y[index,])

full\_data\_test\_y.append(temp\_y[left\_index,])

else:

temp\_data = trainX[trainy==j,]

temp\_y = trainy[trainy==j]

index = np.random.choice(temp\_data.shape[0], 500, replace=False)

full\_data\_test.append(temp\_data[index,])

full\_data\_test\_y.append(temp\_y[index,])

full\_data = np.array(full\_data)

full\_data\_test = np.array(full\_data\_test)

full\_data\_y = np.array(full\_data\_y)

full\_data\_test\_y = np.array(full\_data\_test\_y)

for k in full\_data\_test:

print(k.shape)

label\_desc={0:"T-shirt/top",1:"Trouser",2:"Pullover",3:"Dress",

4:"Coat",5:"Sandal",6:"Shirt",7:"Sneaker",8:"Bag",9:"Ankle boot"}

**# Step 3. Plotting the data**

plt.rcParams['figure.figsize']=(15,10)

for i,k in enumerate(full\_data):

flat\_data = k.reshape(-1,784)

plt.plot(np.linspace(1,784,784),flat\_data.T,color="black")

plt.title("Label = {}".format(label\_desc[list(set(full\_data\_y[i]))[0]]))

plt.xlabel("Feature Index")

plt.ylabel("Values")

plt.show()

for i,k in enumerate(full\_data\_test):

if list(set(full\_data\_test\_y[i]))[0] in [1,7]:

flat\_data = k.reshape(-1,784)

plt.plot(np.linspace(1,784,784),flat\_data.T,color="black")

plt.title("Label = {}".format(label\_desc[list(set(full\_data\_test\_y[i]))[0]]))

plt.xlabel("Feature Index")

plt.ylabel("Values")

plt.show()

all\_means = []

plt.rcParams['figure.figsize']=(15,10)

for i,k in enumerate(full\_data):

flat\_data = k.reshape(-1,784)

mean\_vals = flat\_data.mean(axis=0)

all\_means.append(mean\_vals)

plt.plot(np.linspace(1,784,784), mean\_vals)

plt.title("Label = {}".format(label\_desc[list(set(full\_data\_y[i]))[0]]))

plt.show()

all\_vars = []

for i,k in enumerate(full\_data):

flat\_data = k.reshape(-1,784)

var\_temp = flat\_data.var(axis=0)

plt.plot(np.linspace(1,784,784), var\_temp)

plt.title("Label = {}".format(label\_desc[list(set(full\_data\_y[i]))[0]]))

plt.show()

**# Step 4. Modelling the data**

# concatenating all the training data together

train\_data\_x = full\_data[0]

train\_data\_y = full\_data\_y[0]

for j in range(1,full\_data.shape[0]):

train\_data\_x = np.vstack((train\_data\_x, full\_data[j]))

train\_data\_y = np.hstack((train\_data\_y, full\_data\_y[j]))

# concatenating test data together

test\_data\_x = full\_data\_test[0]

test\_data\_y = full\_data\_test\_y[0]

for j in range(1,full\_data\_test.shape[0]):

test\_data\_x = np.vstack((test\_data\_x, full\_data\_test[j]))

test\_data\_y = np.hstack((test\_data\_y, full\_data\_test\_y[j]))

# normalizing the data to 0-1

train\_data\_x = train\_data\_x.astype('float32') / 255.

test\_data\_x = test\_data\_x.astype('float32') / 255.

train\_data\_x = np.reshape(train\_data\_x, (len(train\_data\_x), 28, 28, 1))

test\_data\_x = np.reshape(test\_data\_x, (len(test\_data\_x), 28, 28, 1))

def SSIMLoss(y\_true, y\_pred):

return 1 - tf.reduce\_mean(tf.image.ssim(y\_true, y\_pred,1.0))

# creating the input layer

Input\_layer = tf.keras.Input(shape=(28, 28, 1), name='input\_layer')

# Encoder Layers

Encoder = tf.keras.layers.Conv2D(64, kernel\_size=3, padding='same', name='ConvLayer1')(Input\_layer)

Encoder = tf.keras.layers.BatchNormalization(name='BatchNorm1')(Encoder)

Encoder = tf.keras.layers.LeakyReLU(name='LeakyRelu1')(Encoder)

Encoder = tf.keras.layers.Conv2D(32, kernel\_size=3, strides=2, padding='same', name='ConvLayer2')(Encoder)

Encoder = tf.keras.layers.BatchNormalization(name='BatchNorm2')(Encoder)

Encoder = tf.keras.layers.LeakyReLU(name='LeakyRelu2')(Encoder)

# Decoder Layers

Decoder = tf.keras.layers.Conv2DTranspose(32, kernel\_size=3, strides= 2, padding='same',name='ConvLayertranspose2')(Encoder)

Decoder = tf.keras.layers.BatchNormalization(name='BatchNorm5')(Decoder)

Decoder = tf.keras.layers.LeakyReLU(name='LeakyRelu5')(Decoder)

Decoder = tf.keras.layers.Conv2DTranspose(64, kernel\_size=3, padding='same', name='ConvLayertranspose3')(Decoder)

Decoder = tf.keras.layers.BatchNormalization(name='BatchNorm6')(Decoder)

Decoder = tf.keras.layers.LeakyReLU(name='LeakyRelu6')(Decoder)

# output

Output\_layer = tf.keras.layers.Conv2DTranspose(1, kernel\_size=3, strides=1,padding='same', activation='sigmoid', name='ConvLayertranspose4')(Decoder)

# declaring the autoencoder model

autoencoder = tf.keras.Model(Input\_layer, Output\_layer)

# declaring Adam optimizer with learning rate 0.0005

optimizer = tf.keras.optimizers.Adam(lr = 0.0005)

# compiling the model with mean squared error loss

autoencoder.compile(optimizer=optimizer, loss=SSIMLoss)

autoencoder.summary()

# running the model with 5 epochs, 128 batch size

hist=autoencoder.fit(train\_data\_x, train\_data\_x,

epochs=5,

batch\_size=128,

shuffle=True

)

# making predictions on the

decoded\_images = autoencoder.predict(test\_data\_x)

**# Step 5. Plotting reconstructed images and generating validation metrics**

# plotting the reconstructed images

plt.rcParams['figure.figsize']=(2,2)

for i in range(1,3):

# Input image

plt.imshow(test\_data\_x[i].reshape(28, 28))

plt.title("Input Image")

plt.gray()

plt.show()

# The reproduced image

plt.imshow(decoded\_images[i].reshape(28, 28))

plt.title("Reproduced image")

plt.gray()

plt.show()

arg\_1 = np.argmax(test\_data\_y==1)

arg\_7 = np.argmax(test\_data\_y==7)

# plotting data on which the model was not trained

for j in [arg\_1,arg\_7]:

# Input image

plt.imshow(test\_data\_x[j].reshape(28, 28))

plt.title("Input Image")

plt.gray()

plt.show()

# The reproduced image

plt.imshow(decoded\_images[j].reshape(28, 28))

plt.title("Reproduced image")

plt.gray()

plt.show()

# getting the mean squared errors

ssi\_vals = [SSIMLoss(test\_data\_x[j],decoded\_images[j]).numpy() for j in range(len(decoded\_images))]

ssi\_vals = np.array(ssi\_vals)

ssi\_vals

plt.rcParams['figure.figsize']=10,5

plt.hist(ssi\_vals)

plt.title("SSIM error plot")

plt.xlabel("SSIM")

plt.show()

# keeping the threshold at 0.02

outcome = pd.DataFrame(np.unique(test\_data\_y[ssi\_vals>0.02],return\_counts=True)).T

outcome.columns = ['labels','count']

for j in range(outcome.shape[0]):

outcome.loc[j,'labels'] = label\_desc[outcome.loc[j,'labels']]

outcome

**############ NEW MODEL CODE #########**

import tensorflow as tf

import numpy as np

import pandas as pd

from matplotlib import pyplot

import matplotlib.pyplot as plt

from keras.datasets import fashion\_mnist, mnist,cifar10

# load dataset

(trainX, trainy), (testX, testy) = cifar10.load\_data()

# summarize loaded dataset

print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))

print('Test: X=%s, y=%s' % (testX.shape, testy.shape))

# plot first few images

for i in range(9):

# define subplot

pyplot.subplot(330 + 1 + i)

# plot raw pixel data

pyplot.imshow(trainX[i], cmap=pyplot.get\_cmap('gray'))

# show the figure

pyplot.show()

np.random.seed(1)

full\_data = []

full\_data\_y = []

full\_data\_test = []

full\_data\_test\_y = []

for j in list(set(trainy.squeeze().tolist())):

if j not in [1,7]:

temp\_data = trainX[trainy.squeeze()==j]

temp\_y = trainy.squeeze()[trainy.squeeze()==j]

print(j, sum(sum(sum(sum(np.isnan(temp\_data))))), sum(np.isnan(temp\_y)),temp\_data.dtype, temp\_y.dtype, temp\_data.flatten().mean(),temp\_data.flatten().std())

index = np.random.choice(temp\_data.shape[0], 1000, replace=False)

left\_index = [k for k in range(temp\_data.shape[0]) if k not in index]

full\_data\_test.append(temp\_data[left\_index[0:100],])

full\_data.append(temp\_data[index,])

full\_data\_y.extend(temp\_y[index,])

full\_data\_test\_y.extend(temp\_y[left\_index[0:100],])

else:

temp\_data = trainX[trainy.squeeze()==j,]

temp\_y = trainy.squeeze()[trainy.squeeze()==j]

index = np.random.choice(temp\_data.shape[0], 100, replace=False)

full\_data\_test.append(temp\_data[index,])

full\_data\_test\_y.extend(temp\_y[index,])

full\_data = np.array(full\_data)

full\_data\_test = np.array(full\_data\_test)

full\_data\_y = np.array(full\_data\_y)

full\_data\_test\_y = np.array(full\_data\_test\_y)

for k in full\_data\_test:

print(k.shape)

label\_desc={0:"airplane",1:"automobile",2:"bird",3:"cat",

4:"deer",5:"dog",6:"frog",7:"horse",8:"ship",9:"truck"}

# concatenating all the training data together

train\_data\_x = full\_data[0]

train\_data\_y = full\_data\_y[0]

for j in range(1,full\_data.shape[0]):

train\_data\_x = np.vstack((train\_data\_x, full\_data[j]))

#train\_data\_y = np.hstack((train\_data\_y, full\_data\_y[j]))

# concatenating test data together

test\_data\_x = full\_data\_test[0]

test\_data\_y = full\_data\_test\_y[0]

for j in range(1,full\_data\_test.shape[0]):

test\_data\_x = np.vstack((test\_data\_x, full\_data\_test[j]))

#test\_data\_y = np.hstack((test\_data\_y, full\_data\_test\_y[j]))

test\_data\_y = full\_data\_test\_y

train\_data\_y = full\_data\_y

# printing the shape of both datasets

print(train\_data\_x.shape, test\_data\_x.shape)

# normalizing the data to 0-1

train\_data\_x = train\_data\_x.astype('float32') / 255.

test\_data\_x = test\_data\_x.astype('float32') / 255.

train\_data\_x = np.reshape(train\_data\_x, (len(train\_data\_x), 32, 32, 3))

test\_data\_x = np.reshape(test\_data\_x, (len(test\_data\_x), 32, 32, 3))

def SSIMLoss(y\_true, y\_pred):

return 1 - tf.reduce\_mean(tf.image.ssim(y\_true, y\_pred,1.0))

# creating the input layer

Input\_layer = tf.keras.Input(shape=(32, 32, 3), name='input\_layer')

# Encoder Layers

Encoder = tf.keras.layers.Conv2D(64, kernel\_size=3, padding='same', name='ConvLayer1')(Input\_layer)

#Encoder = tf.keras.layers.BatchNormalization(name='BatchNorm1')(Encoder)

Encoder = tf.keras.layers.LeakyReLU(name='LeakyRelu1')(Encoder)

Encoder = tf.keras.layers.Conv2D(32, kernel\_size=3, strides=2, padding='same', name='ConvLayer2')(Encoder)

#Encoder = tf.keras.layers.BatchNormalization(name='BatchNorm2')(Encoder)

Encoder = tf.keras.layers.LeakyReLU(name='LeakyRelu2')(Encoder)

# Decoder Layers

Decoder = tf.keras.layers.Conv2DTranspose(32, kernel\_size=3, strides= 2, padding='same',name='ConvLayertranspose2')(Encoder)

#Decoder = tf.keras.layers.BatchNormalization(name='BatchNorm5')(Decoder)

Decoder = tf.keras.layers.LeakyReLU(name='LeakyRelu5')(Decoder)

Decoder = tf.keras.layers.Conv2DTranspose(64, kernel\_size=3, padding='same', name='ConvLayertranspose3')(Decoder)

#Decoder = tf.keras.layers.BatchNormalization(name='BatchNorm6')(Decoder)

Decoder = tf.keras.layers.LeakyReLU(name='LeakyRelu6')(Decoder)

# output

Output\_layer = tf.keras.layers.Conv2DTranspose(3, kernel\_size=3, strides=1,padding='same', activation='sigmoid', name='ConvLayertranspose4')(Decoder)

# declaring the autoencoder model

autoencoder = tf.keras.Model(Input\_layer, Output\_layer)

# declaring Adam optimizer with learning rate 0.0005

optimizer = tf.keras.optimizers.Adam(lr = 0.0005)

# compiling the model with mean squared error loss

autoencoder.compile(optimizer=optimizer, loss=SSIMLoss)

autoencoder.summary()

# running the model with 10 epochs, 128 batch size

hist = autoencoder.fit(train\_data\_x, train\_data\_x,

epochs=10,

batch\_size=128,

shuffle=True

)

# making predictions on the test data

decoded\_images = autoencoder.predict(test\_data\_x)

# plotting the reconstructed images

plt.rcParams['figure.figsize']=(2,2)

for i in range(1,3):

# Input image

plt.imshow(test\_data\_x[i],cmap=pyplot.get\_cmap('gray'))

plt.title("Input Image")

plt.gray()

plt.show()

# The reproduced image

plt.imshow(decoded\_images[i],cmap=pyplot.get\_cmap('gray'))

plt.title("Reproduced image")

plt.gray()

plt.show()

arg\_1 = np.argmax(test\_data\_y==1)

arg\_7 = np.argmax(test\_data\_y==7)

# plotting data on which the model was not trained

for j in [arg\_1,arg\_7]:

# Input image

plt.imshow(test\_data\_x[j])

plt.title("Input Image")

plt.gray()

plt.show()

# The reproduced image

plt.imshow(decoded\_images[j])

plt.title("Reproduced image")

plt.gray()

plt.show()

# getting the mean squared errors

ssi\_vals = [SSIMLoss(test\_data\_x[j],decoded\_images[j]).numpy() for j in range(len(decoded\_images))]

ssi\_vals = np.array(ssi\_vals)

plt.rcParams['figure.figsize']=10,5

plt.hist(ssi\_vals)

plt.title("SSIM error plot")

plt.xlabel("SSIM")

plt.show()

# keeping the threshold at 0.07

outcome = pd.DataFrame(np.unique(test\_data\_y[ssi\_vals>0.035],return\_counts=True)).T

outcome.columns = ['labels','count']

for j in range(outcome.shape[0]):

outcome.loc[j,'labels'] = label\_desc[outcome.loc[j,'labels']]

outcome