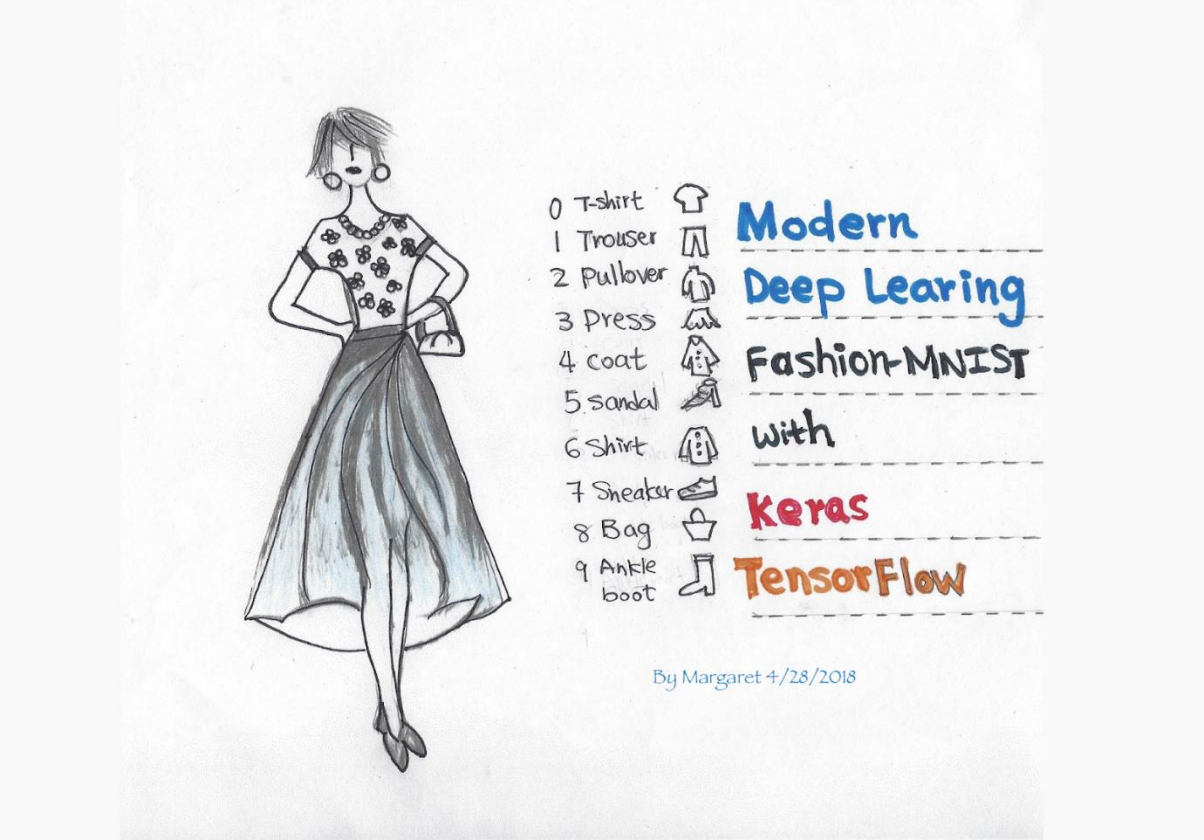
Computer Vision Group Project:

Image Denoising on Fashion MNIST Dataset



Source: TensorFlow Blog

Group Members:

Aryaman Gautam (J001)

Aayushmaan Jain (J022)

Pratyush Patro (J047)

Milindi Shah (J057)

Abstract

Noise removal or as people may say Image Denoising is one of the most important concerns in computer vision these days and is also an important topic of research these days which is also the topic for our group project. Noise may be added during any stage of image capturing, from human error while taking an image, a minor problem in the lens of the camera or while storing an image in various file formats, noise may be added to the image. Removing noise from the original image is still a challenging problem but there are many deep learning techniques to remove noise from the image. This project aims to demonstrate the use of autoencoders to perform Image Denoising

Introduction

Images have been an important role in our daily applications such as television, mobile phones, satellite imaging, etc. The images (datasets) collected by the sensors or lenses are generally contaminated by noise during the data collection process generally due to imperfect instruments, natural interference of light sources, or during the data storage or transmission process which generally requires compressing the image and sending it as an attachment in any social media application. Hence it is necessary to apply an efficient image denoising technique to compensate for such dataset contamination as noise may degrade the quality of data which is to be used further for other purposes.

Dataset Details

The dataset used in this project is the Fashion MNIST dataset which was collected by **Zalando** which is a community from more than 140 different nations, representing a seemingly endless number of interests, hobbies, programming language preferences, personality and much more which is united by the purpose of delivering award winning and best in class shopping experiences to 46 million active customers and counting. The dataset contains 60,000 images in the training set and 10,000 images in the test set. The images are of 10 different classes of fashion equipment which are:

0) T-shirt/top
1) Trouser
2) Pullover
3) Dress
4) Coat
5) Sandal
6) Shirt
7) Sneaker
8) Bag
9) Ankle boot

The dataset is benchmarked where the images are taken against a clear background, then converted to grayscale orientation, then resized to a uniform size of 28 pixels by 28 pixels to ensure uniformity in the dataset and the images may have a minimal amount of noise (which may be introduced due to resizing the images to such a small size) and the dataset is suitable to train machine learning and deep learning models on it.

The dataset used from this project is sourced from Tensorflow datasets[1] which is an excellent data repository containing many datasets for training machine learning and deep learning models.

We will be using the Fashion MNIST dataset[2] for the purpose of this project

Loading the data:

We loaded the data using the load_data function from TensorFlow library in python

```
[3]  # Loading the data
     (X_train, y_train), (X_test,y_test) = tf.keras.datasets.fashion_mnist.load_data()

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
32768/29515 [==============================] - 0s 0us/step
40960/29515 [==========================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
26435584/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
16384/5148 [================================================================================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
4431872/4422102 [==============================] - 0s 0us/step
```

Checking the dimensions of the data:

```
# Seeing the dimensions of our data
print(f'Images for the train set -> {X_train.shape}')
print(f'Labels for the train set -> {y_train.shape}')
print(f'Images for the test set -> {X_test.shape}')
print(f'Labels for the test set -> {y_test.shape}')

Images for the train set -> (60000, 28, 28)
Labels for the train set -> (60000,)
Images for the test set -> (10000, 28, 28)
Labels for the test set -> (10000,)
```

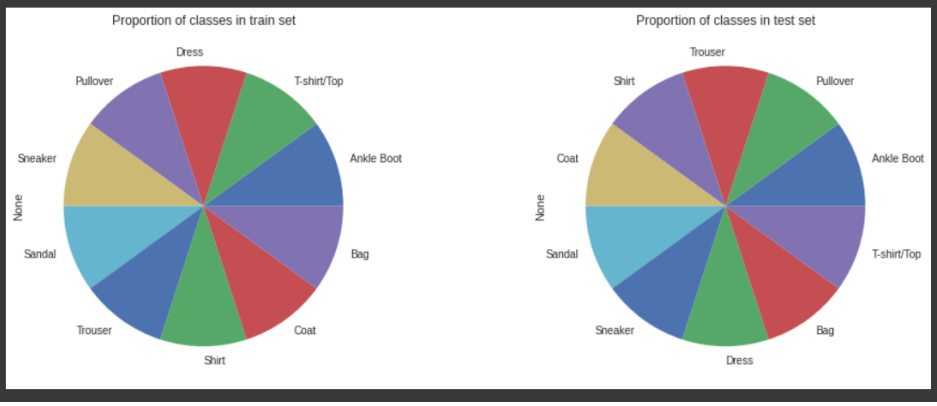
As said earlier, train set has 60,000 images with labels and test set has 10,000 images with labels

Since the labels are not in a desired format, we need to convert the labels to a desired format

```
# Initializing the labels as the labels in the tensorflow dataset are encoded, hence we need to decode them
# in order to make them readable for humans
labels = {
    0:'T-shirt/Top',
    1:'Trouser',
    2:'Pullover',
    3:'Dress',
    4:'Coat',
    5:'Sandal',
    6:'Shirt',
    7:'Sneaker',
    8:'Bag',
    9:'Ankle Boot'
}
y_train = pd.Series([labels[val] for val in y_train]) # decoding the labels for train set
y_test = pd.Series(labels[val] for val in y_test) # decoding the labels for test set
```

Since now we have the labels in a desired format, we can see the proportion of images of each class in the training set as well as the test set

```
# Plotting a pie chart of the proportions of each category of fashion apparel in train and test set
fig, ax = plt.subplots(figsize=(16,6),nrows=1, ncols=2)
y_train.value_counts().plot(kind='pie',ax=ax[0], title='Proportion of classes in train set')
y_test.value_counts().plot(kind='pie',ax=ax[1], title='Proportion of classes in test set')
plt.show()
```



Here, we can see that the classes are balanced in both train and test set, hence there is no imbalance in the data and hence it is perfect to work with

Data Exploration

For seeing some random examples from both train and test set, we have made a custom function which helps us to visualize the images from train and test set

Train Set

```
# Seeing some examples from train set
seeExamples(X_train, y_train)
```

Test Set

```
# Seeing some examples from the test set
seeExamples(X_test, y_test)
```

Methodology

Since the dataset was benchmarked, we had to add some noise to it for implementing denoising, hence we added some White Noise[3]

Here is the python code used to add the White Noise

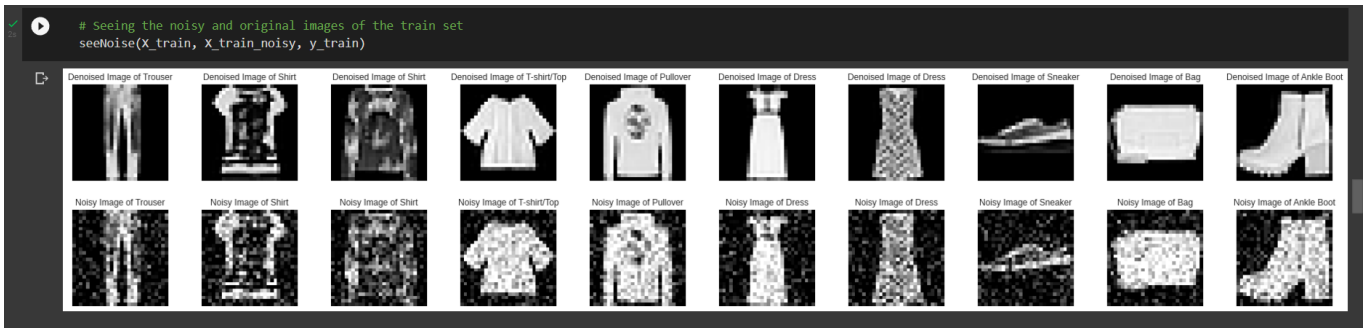
```
[13]  noise_factor = 0.2
      # Adding white Gaussian Noise
      X_train_noisy = X_train + noise_factor * tf.random.normal(shape=X_train.shape)
      X_test_noisy = X_test + noise_factor * tf.random.normal(shape=X_test.shape)

      X_train_noisy = tf.clip_by_value(X_train_noisy, clip_value_min=0., clip_value_max=1.)
      X_test_noisy = tf.clip_by_value(X_test_noisy, clip_value_min=0., clip_value_max=1.)
```

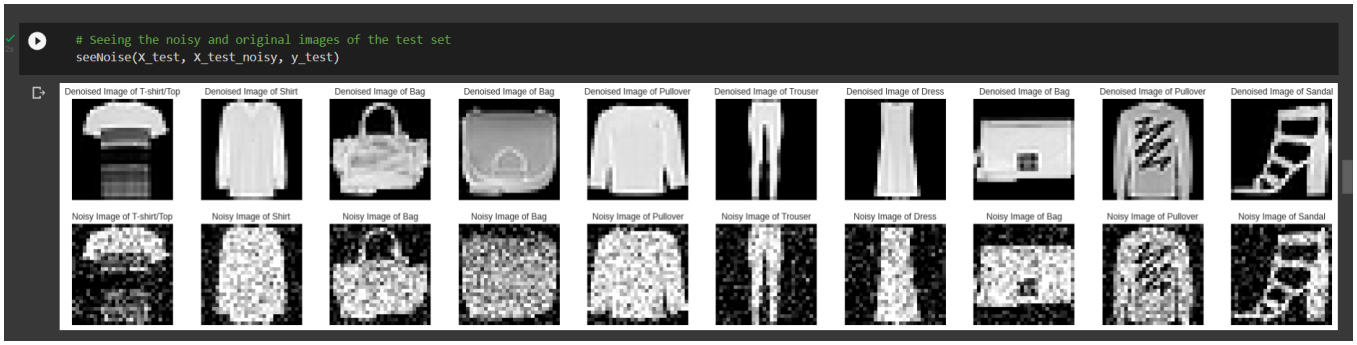
Visualizing denoised and noised images

For visualizing the effect of adding the noise, we have made a custom function to visualize the images before and after adding the noise

Train set:



Test set:



Model used:

The model that we are using is an Autoencoder[4] model. Autoencoder is an unsupervised model which learns the data encodings in an unsupervised manner.

The autoencoder architecture includes three components:

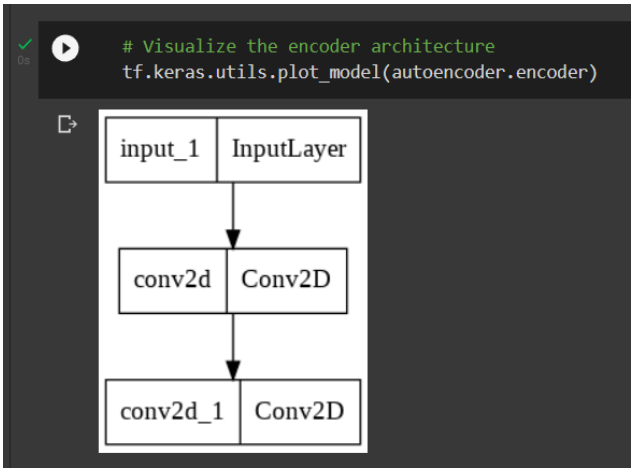
- ➔ Encoder – The part which performs the feature extraction of the input data, and while doing so, it compresses the input data into a lower dimension.
- ➔ Bottleneck – The part which contains the compressed features, which makes it the most important part of the auto encoder
- ➔ Decoder – The part which decompresses the features and reconstructs the data back to it's original form

This architecture makes the autoencoder model a perfect model to use, hence we will use autoencoders to perform image denoising

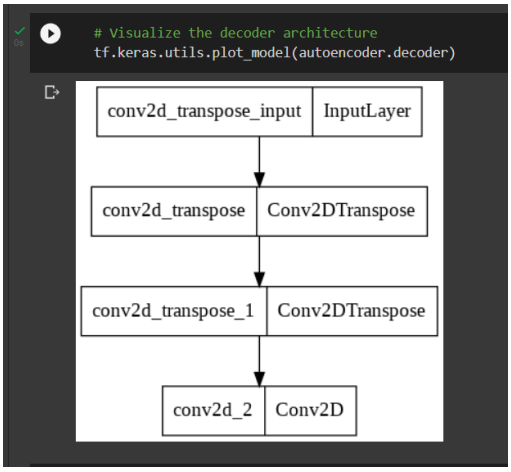
The encoder part is composed of two Convolutional layers, the first one containing 16 neurons and the second one containing 8 neurons.

The decoder part is composed of two Transpose Convolutional layers, the first one containing 8 neurons and the second one containing 16 neurons.

Encoder



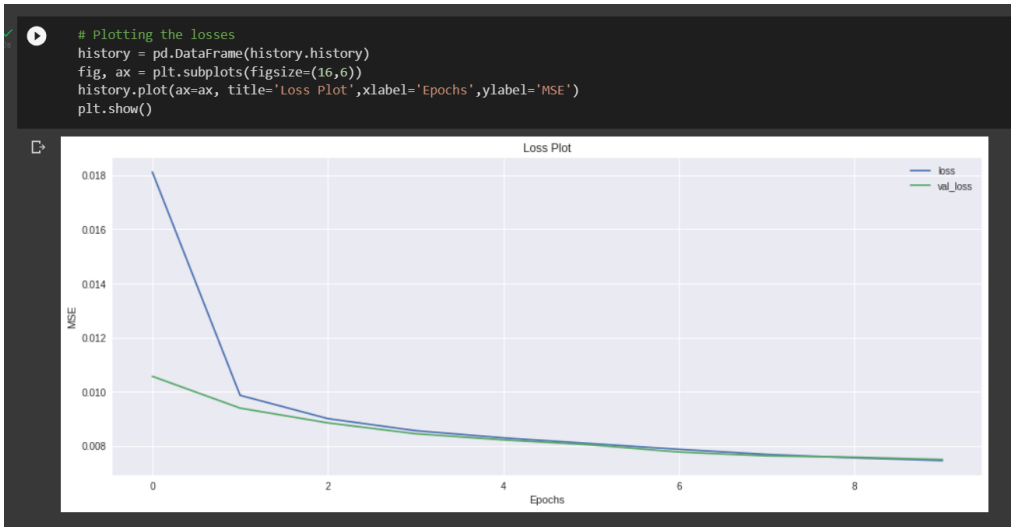
Decoder



Parameters of the model

- ➔ Loss – Mean Squared Error
- ➔ Optimizer – Adam
- ➔ Epochs – 10

After training the model for 10 epochs, we observe that the model converges quite well and the loss is also close to zero with almost no overfitting, hence this model is suitable for further use.



Analysis

Since the model is performing well, we will evaluate it by using the K-fold[5] evaluation technique which is very useful in checking the robustness of the architecture. In this technique, we will split the data in five equal parts, where 4 of them will be for training and 1 will be for the evaluation, then we will keep on changing the parts until all the parts have been used for the test set at least once.

From the results of the five folds, we can see that the losses are very close to each other which signifies that the model architecture is robust and can perform well on a well generalized set of unseen data.

```
==================================================Fold 1==================================================
MSE on test set for fold 1 -> 0.007183538284152746


==================================================Fold 2==================================================
MSE on test set for fold 2 -> 0.00778339896351099


==================================================Fold 3==================================================
MSE on test set for fold 3 -> 0.0068662757985293865


==================================================Fold 4==================================================
MSE on test set for fold 4 -> 0.007017388008534908


==================================================Fold 5==================================================
MSE on test set for fold 5 -> 0.007505131419748068
```

Results

Now, we can observe that the loss is consistent for all the folds, we can use this model for further predictions.

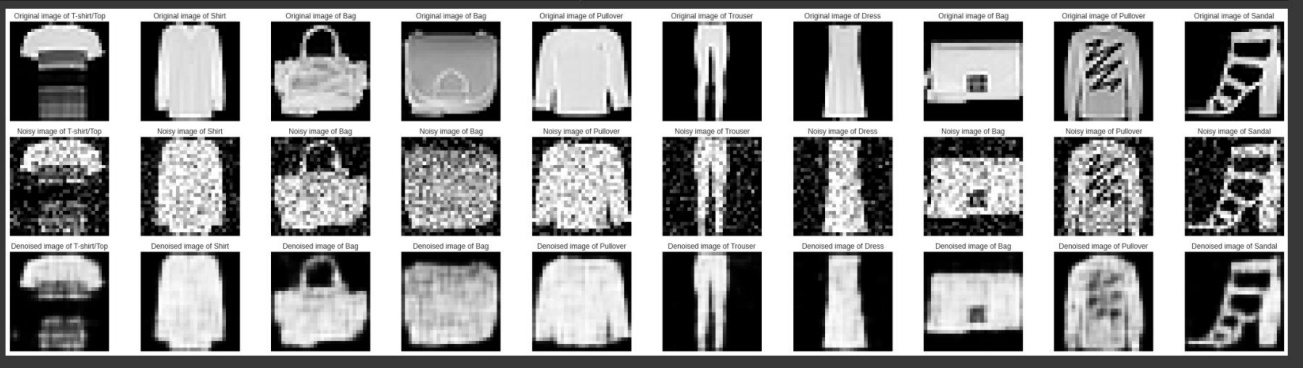
We have implemented a custom function which shows the original image, the noisy image and the denoised image.

Train set:

```
[28]  # Seeing predictions on train data
      seePredictions(data=X_train, noisyData=X_train_noisy,labels=y_train,model=autoencoder)
```

Test set:

```
[29]  # Seeing predictions on test data
      seePredictions(data=X_test, noisyData=X_test_noisy,labels=y_test,model=autoencoder)
```



Benchmarking the dataset

For benchmarking the dataset, we will implement data augmentation which helps us to achieve the purpose

Augmentations implemented

- ➔ Shearing – to account for little shaking which might occur when the picture is taken
- ➔ Zoom – to account for the varying distances while taking the picture
- ➔ Horizontal flipping – to account for the mirror image if the picture of the object is taken from the other side
- ➔ Rescaling– rescaling the image for faster convergence

We will be using the ImageDataGenerator[6] from TensorFlow to implement these augmentations.

Given below, is a screenshot of the code for implementing data augmentation

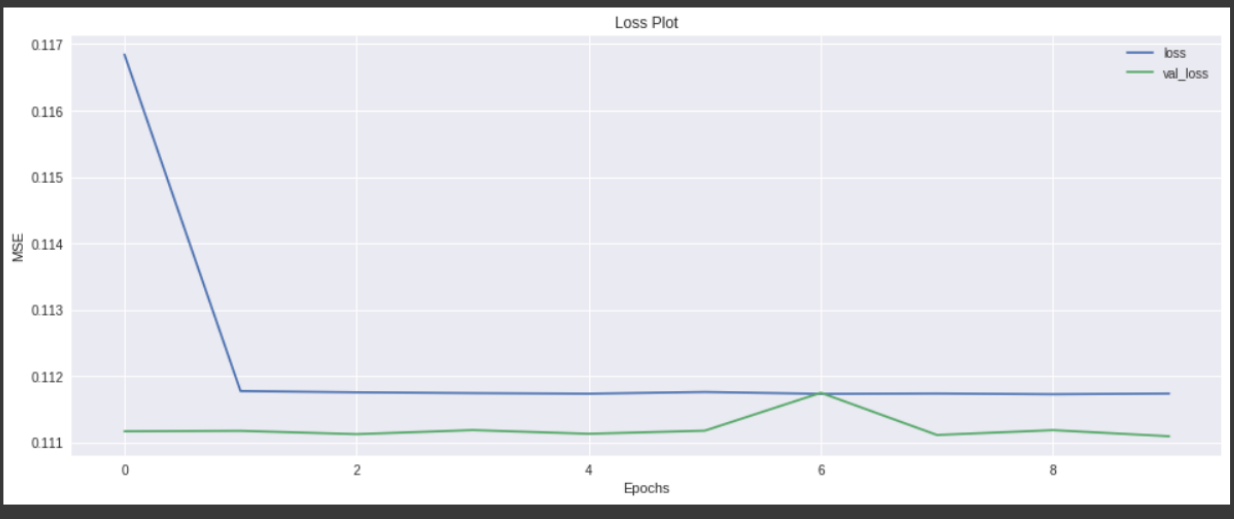
```
[28]  # Data Augmentation
      train_generator = tf.keras.preprocessing.image.ImageDataGenerator(
          zoom_range=0.2,
          shear_range=0.2,
          horizontal_flip=True,
          rescale=1/255.0
      )

      test_generator = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1/255.0)
```

```
[29]  train_set = train_generator.flow(X_train_noisy.numpy(), X_train)
      test_set = test_generator.flow(X_test_noisy.numpy(), X_test)
```

After implementing the augmentations, we fit the model again

```
[32]  # Plotting the losses
      history = pd.DataFrame(history.history)
      fig, ax = plt.subplots(figsize=(16,6))
      history.plot(ax=ax, title='Loss Plot',xlabel='Epochs',ylabel='MSE')
      plt.show()
```



Here we observe that the model fitting is not that consistent and the loss is increasing suddenly at certain epochs.

References

[1] TensorFlow datasets link

[2] Fashion MNIST dataset link

[3] White Noise Wikipedia page

[4] Autoencoder Wikipedia page

[5] K Fold documentation from scikit-learn page

[6] ImageDataGenerator documentation

Source Code