60004170108 - Siddharth Sanghavi - TE COMPS B

ML Assignment - 1
Denoising Images using Autoencoders

Abstract:

One of the fundamental challenges in the field of Image Processing and Computer Vision is Image Denoising, where the underlying goal is to estimate the original image by suppressing noise from a noise-contaminated version of the image. Image noise may be caused by different intrinsic (i.e., sensor) and extrinsic (i.e., environment) conditions which are often not possible to avoid in practical situations.

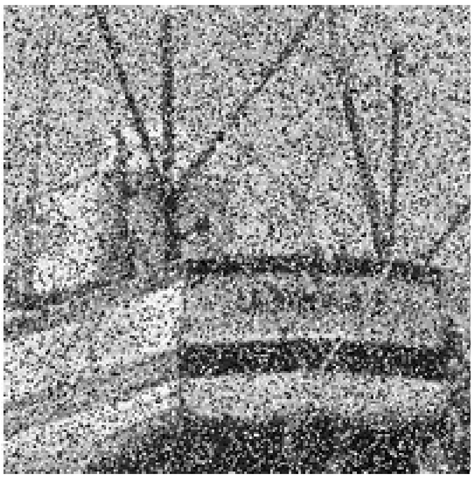


Fig. 1. Noisy Image

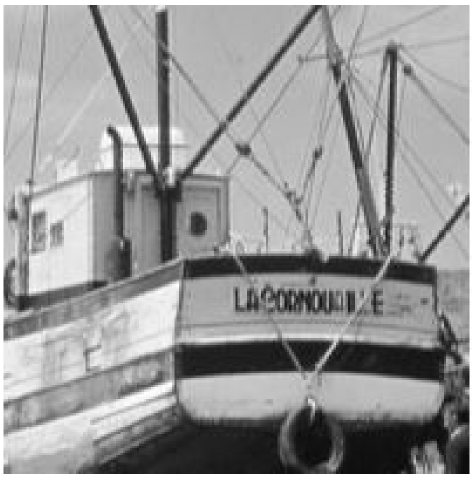


Fig. 2. Denoised Image

Therefore, Image Denoising plays an important role in a wide range of applications such as Image Restoration, Visual Tracking, Image Registration, Image Segmentation and Image Classification, where obtaining the original image content is crucial for strong performance. Here we examine an approach to solve the problem of image denoising based on Autoencoders using the Modified National Institute of Standards and Technology (MNIST) dataset.

Concept:

Autoencoders are self-supervised (a specific instance of supervised learning where the targets are generated from the input data) neural networks that seek to:

- Accept an input set of data.
- Internally compress the input data into a latent-space representation (i.e. a single vector that compresses and quantifies the input) which is substantially smaller than the input in terms of dimensionality.
- Reconstruct the input data from this latent representation.

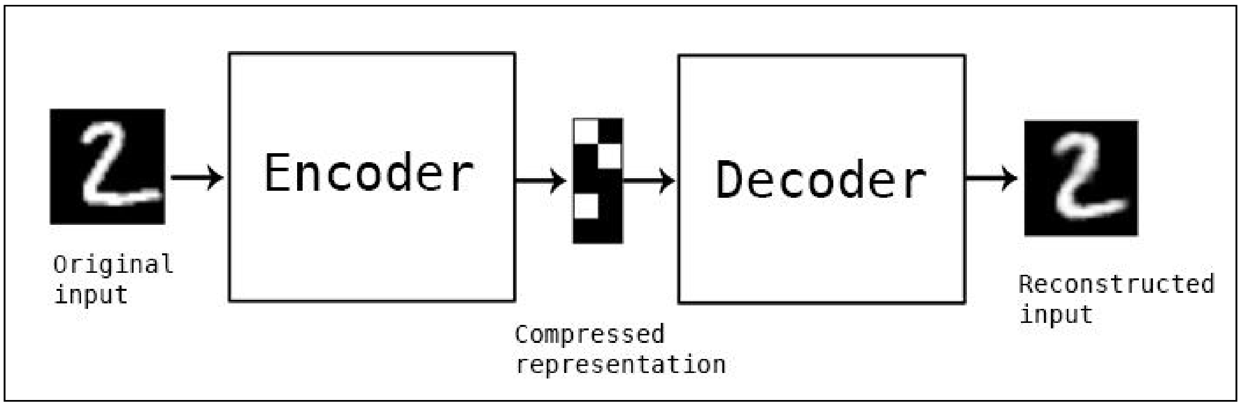


Fig. 3. Architecture of an Autoencoder

To build an autoencoder, we need three things: an Encoding function, a Decoding function, and a Distance function between the amount of information loss between the compressed representation of the input data and the decompressed representation (i.e. a Loss function).

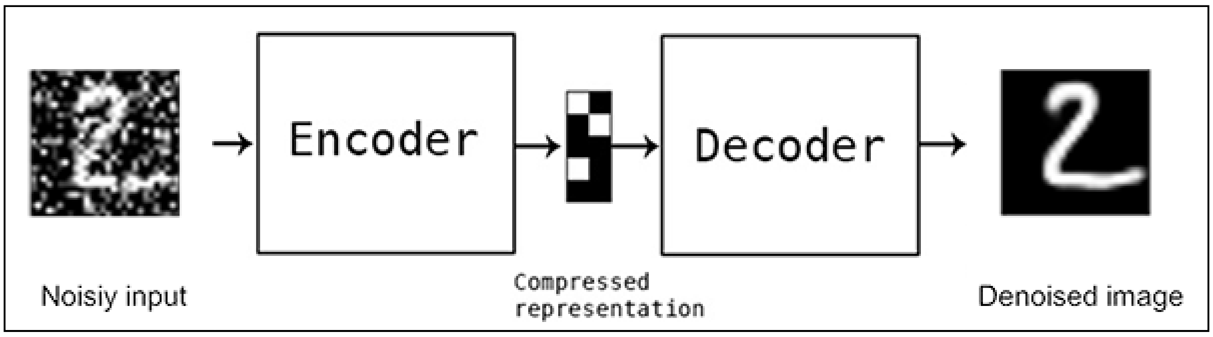


Fig. 4. Architecture of a Denoising Autoencoder

By training the Autoencoder to copy the input to the output, we force it to learn the latent-space representation which will take on the most salient and useful features of the training data. It’s possible to learn the Encoder and Decoder functions in a way so that noise from the input is removed. Hence, during training, we provide noisy images as input, and their corresponding noise-free images as targets. The Encoder and Decoder will together learn to remove the particular noise present in the predicted/output images.

Today, the two most interesting and practical applications of Autoencoders are Data Denoising and Dimensionality Reduction for Data Visualization. With appropriate dimensionality and sparsity constraints, Autoencoders can learn data projections that are more interesting than PCA or other basic techniques.

Dataset Description and Statistical Analysis:

In this case study, we use the Modified National Institute of Standards and Technology (MNIST) dataset, which is a large dataset of handwritten digits (from 0 to 9) that is commonly used for training and testing various Image Processing systems and models in Machine Learning. The MNIST dataset contains 60,000 training images and 10,000 testing images of resolution 28x28 pixels with a single grayscale channel.

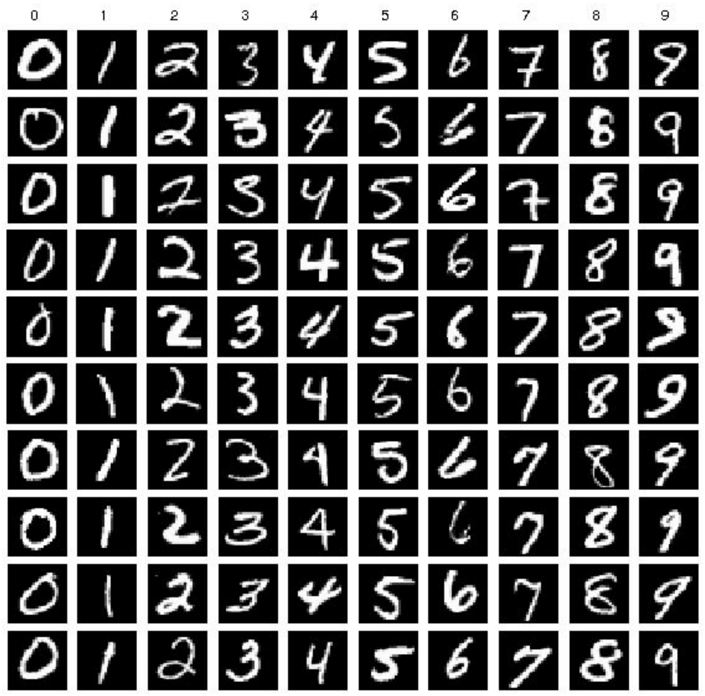
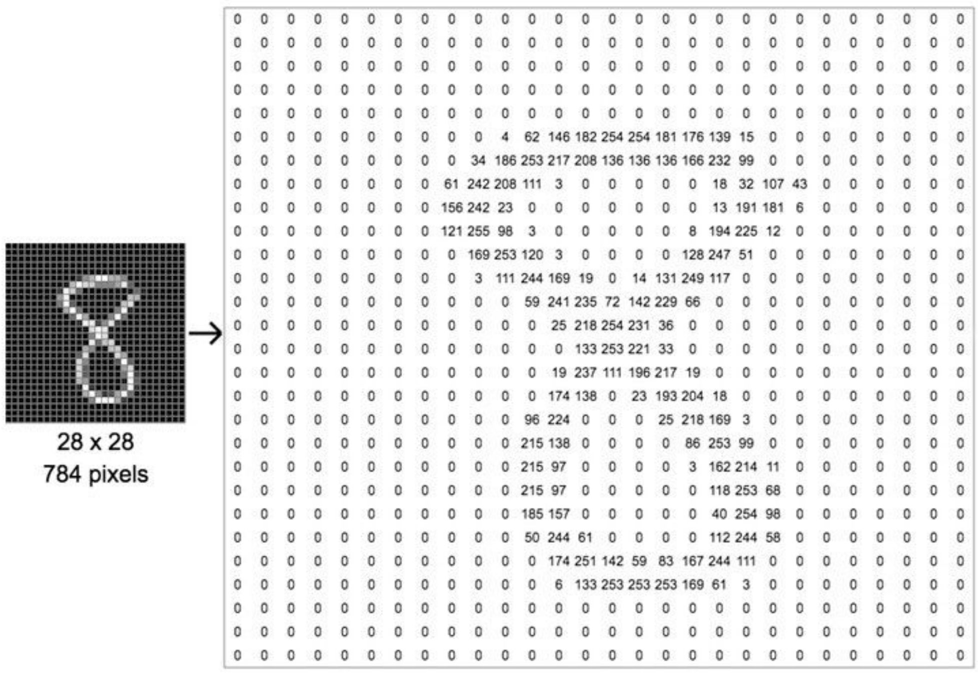


Fig. 5. The MNIST dataset with labels of each class

Each MNIST image can be thought of as a matrix of numbers describing how dark each pixel is (example shown below in Fig. 6.). Since each image has a resolution of 28x28 pixels, we get a 28x28 matrix. We can flatten the matrix into a 28x28 = 784 dimensional vector. Each component of the vector is a value between [0, 255] describing the intensity of the pixel where 0 stands for black and 255 stands for white. Thus, we generally think of MNIST as being a collection of 784-dimensional vectors.



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	4	62	146	182	254	254	181	176	139	15	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	34	186	253	217	208	136	136	136	166	232	99	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	61	242	208	111	3	0	0	0	0	0	18	32	107	43	0	0	0	0	0	0
0	0	0	0	0	0	0	0	156	242	23	0	0	0	0	0	0	13	191	181	6	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	121	255	98	3	0	0	0	0	0	8	194	225	12	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	169	253	120	3	0	0	0	0	128	247	51	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	3	111	244	169	19	0	14	131	249	117	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	59	241	235	72	142	229	66	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	25	218	254	231	36	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	133	253	221	33	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	19	237	111	196	217	19	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	174	138	0	23	193	204	18	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	96	224	0	0	0	25	218	169	3	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	215	138	0	0	0	0	86	253	99	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	215	97	0	0	0	0	3	162	214	11	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	215	97	0	0	0	0	0	118	253	68	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	185	157	0	0	0	0	0	40	254	98	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	50	244	61	0	0	0	0	112	244	58	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	174	251	142	59	83	167	244	111	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	6	133	253	253	253	169	61	3	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 6. Representation of digit 8 as a matrix of pixels

We can think of the MNIST data points as points suspended in a 784-dimensional cube. Each dimension of the cube corresponds to a particular pixel. The data points range from 0 to 255 according to the pixels intensity. On one side of the dimension, there are images where that pixel is white. On the other side of the dimension, there are images where it is black. In between, there are images where it is gray.

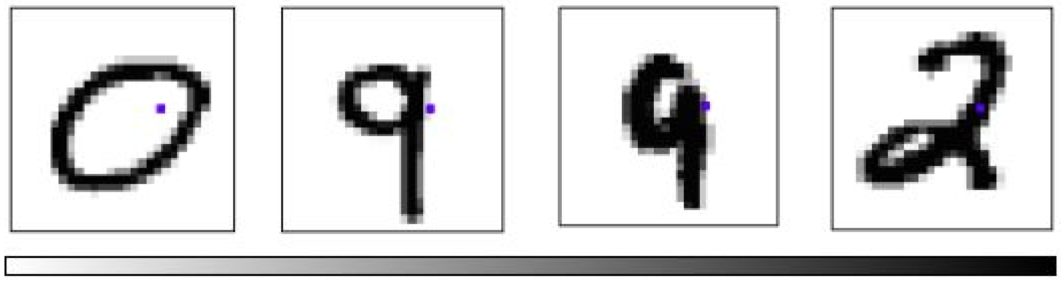


Fig. 7. Pixel intensity from left to right

- **Distribution of Classes:**
  The MNIST dataset consists of 10 classes, one for each of the digits from 0 to 9. We use a histogram to visualize the frequency distribution of training image labels and testing image labels by splitting them into 10 equal-sized bins and counting the number of points that fall into each bin. This histogram is constructed using Matplotlib’s histogram API as shown below in Fig. 8. The X-axis represents 10 bins each corresponding to a digit from 0 to 9. And the Y-axis represents the count/frequency of each digit from 60000 training images labels and 10000 testing image labels.

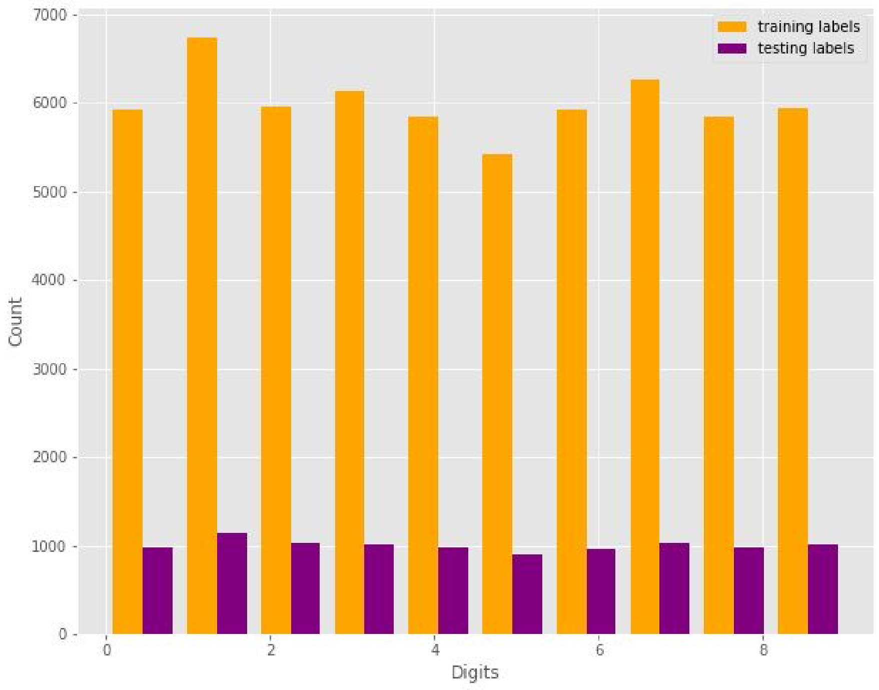


Fig. 8. Histogram Visualization of the MNIST dataset labels

- **t-SNE Visualization:**
  t-SNE (t-Distributed Stochastic Neighbor Embedding dimensionality reduction) algorithm is a nonlinear dimensionality reduction technique well-suited for embedding high-dimensional data for visualization in a low-dimensional space of two or three dimensions. Specifically, it models each high-dimensional object by a 2-dimensional or 3-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points with high probability.

  Here we use the Scikit-learn's t-SNE implementation to visualize 1797 training image labels. The visualization is constructed using Matplotlib's Scatter Plot API. The algorithm manages to project the 784-dimensional MNIST dataset onto a 2-dimensional space such that similar samples cluster together as shown below in Fig. 9.

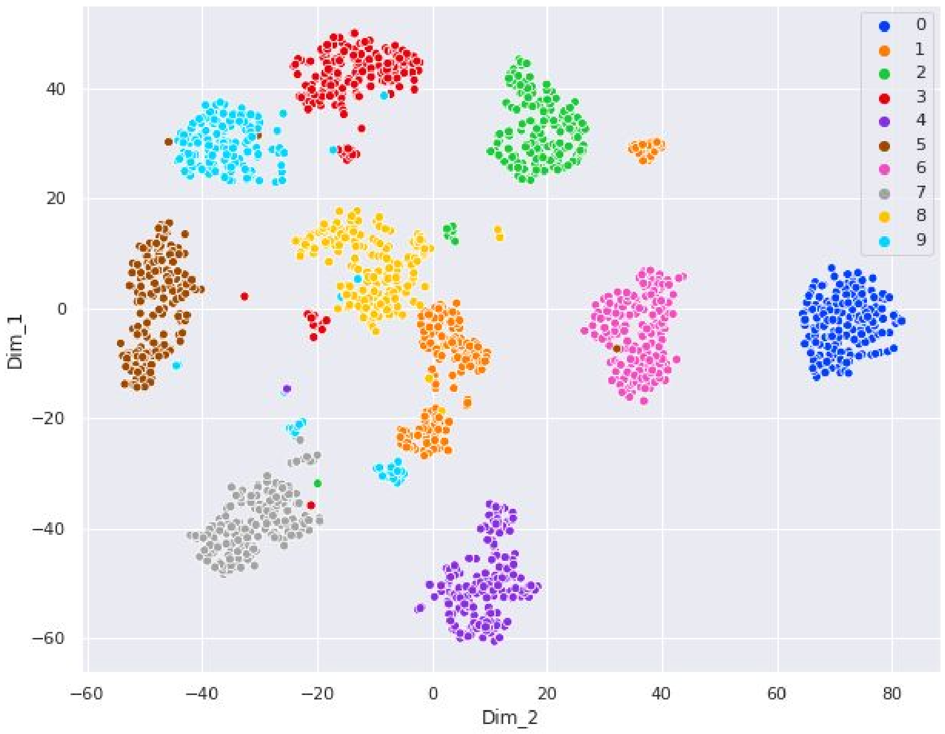


Fig. 9. t-SNE Visualization of the MNIST dataset training labels

- **PCA Visualization:**
  PCA (Principal Component Analysis) is an unsupervised linear dimensionality reduction technique that uses orthogonal transformations to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

  Here we use the Scikit-learn PCA (Principal Component Analysis) implementation to visualize 30000 training image labels. The visualization is constructed using Matplotlib's Scatter Plot API. The algorithm manages to project the high-dimensional MNIST dataset onto a 2-dimensional space using 2 Principal Components. There is a lot of overlap among classes as very few classes can be separated but most of them are mixed as shown below in Fig. 10.

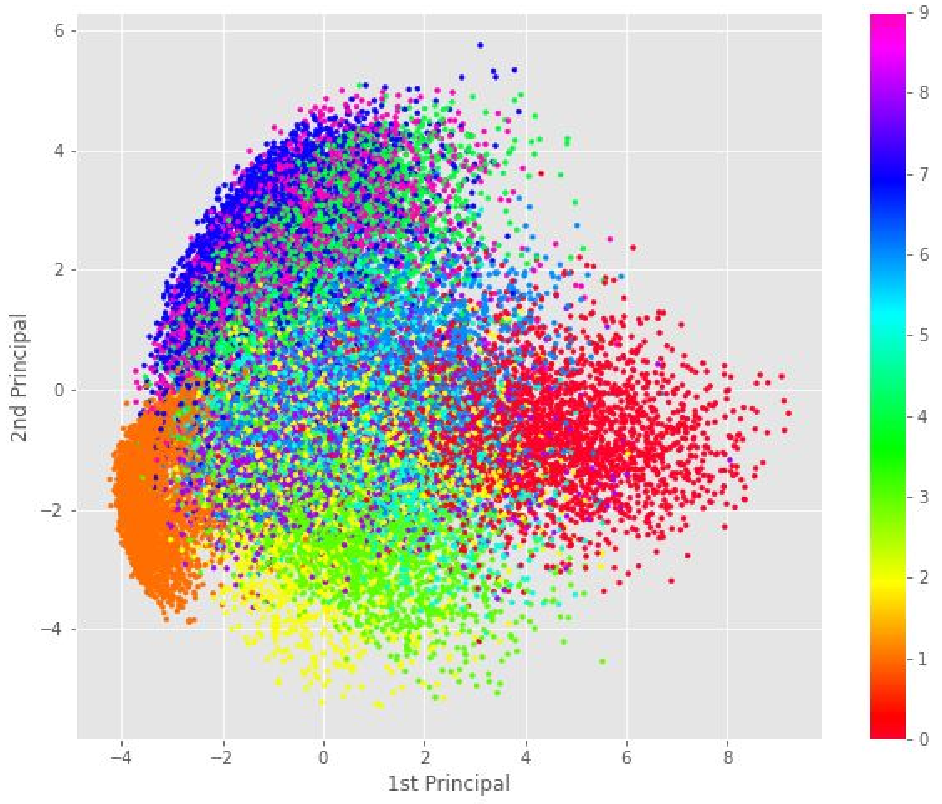


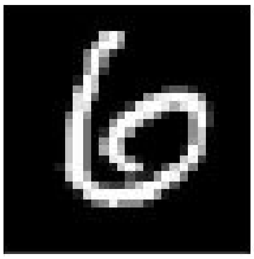
Fig. 10. PCA Visualization of the MNIST dataset training labels

- **Missing Values:**
  Since the MNIST dataset contains image data, it does not have any missing values which are generally found in numerical datasets.

- **Noise:**
  Since the MNIST dataset contains image data, it does not have any characteristic noise which should be removed during the dataset preprocessing stage. Instead, for this case study, we generate synthetic noisy digits by applying a Gaussian noise matrix to the dataset using NumPy's random normal distribution. This artificially generated noise in the dataset can be thought of as being present due to the following reasons:

  - Produced by a faulty or poor quality image sensor
  - Random variations in brightness or color
  - Quantization noise
  - Artifacts due to JPEG compression
  - Image perturbations produced by an image scanner or threshold post-processing
  - Poor paper quality (crinkles and folds) when trying to perform OC

- **Skewness and Kurtosis:**
  The MNIST dataset contains images. Hence finding out the Skewness and Kurtosis of the dataset as a whole is of no use in the Statistical Analysis. However skewness for each grayscale image can be visualized using an image histogram. An image histogram represents the distribution of colors in an image. It can be visualized as a graph (or plot) that gives a high-level intuition of the intensity (pixel value) distribution. Since the MNIST dataset contains grayscale images, each image will have pixel values in the range between [0 to 255]. When plotting the histogram for the digit 6, as shown below in Fig. 11., the X-axis serves as our "bins". If we construct a histogram with 256 bins, then we are effectively counting the number of times each pixel value occurs. The number of pixels binned to the X-axis value is then plotted on the Y-axis. The visualization is constructed using Matplotlib's Histogram API.



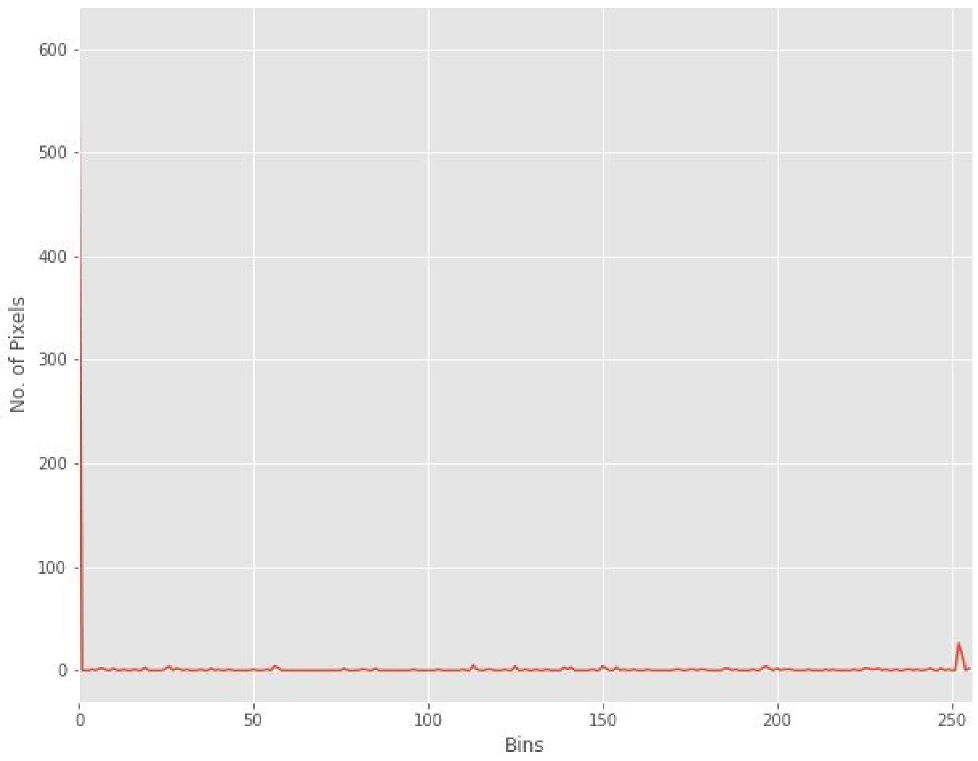


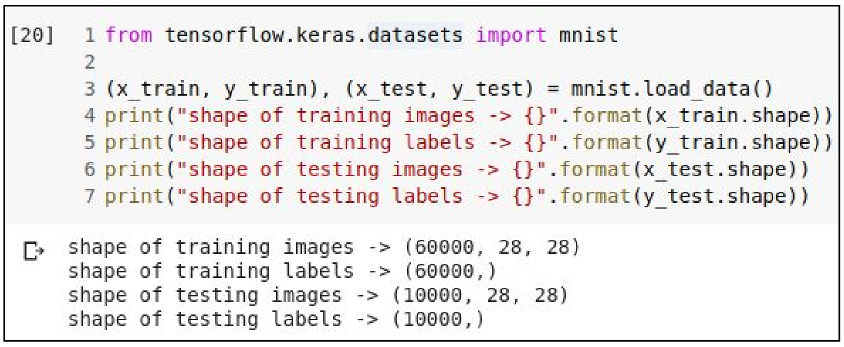
Fig. 11. Image histogram for the digit 6

- **Correlations in the attributes:**
  Since the MNIST dataset contains image data, there is no correlation between different image attributes.

Dataset Preprocessing:

Digital image processing is the use of computer algorithms to perform image processing on digital images.The acquired image data is usually messy and comes from different sources. To feed them to the ML model (or neural network), it needs to be standardized and cleaned up. More often than not, preprocessing is used to conduct steps that reduce the complexity and increase the accuracy of the applied algorithm. We can't write a unique algorithm for each of the condition in which an image is taken, thus, when we acquire an image, we tend to convert it into a form that allows a general algorithm to solve it.The image preprocessing pipeline on the MNIST dataset involves the following steps:

1. **Loading the MNIST dataset:**
   We first import the MNIST dataset from the tensorflow (free and open-source software library for dataflow and differentiable programming) and then load it into the memory as shown below in Fig. 12. The load_data( ) function returns 2 tuples containing the 60000 training and 10000 testing images with their corresponding labels. The images in each tuple is a 3-dimensional uint8 NumPy array. The 3 dimensions correspond to (no_of_images, height, width). Each image has a width and height of 28 pixels. The labels in each tuple is a NumPy vector of the format uint8.



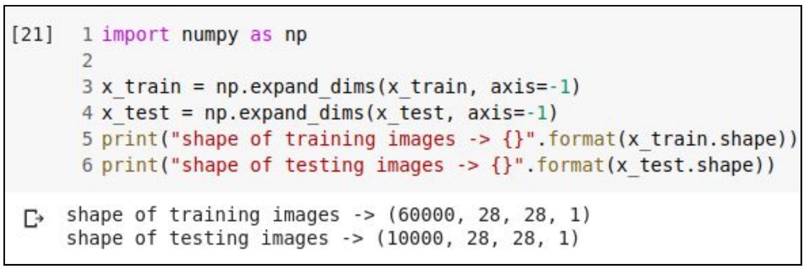
```
[20]
from tensorflow.keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
print("shape of training images -> {}".format(x_train.shape))
print("shape of training labels -> {}".format(y_train.shape))
print("shape of testing images -> {}".format(x_test.shape))
print("shape of testing labels -> {}".format(y_test.shape))
```

```
shape of training images -> (60000, 28, 28)
shape of training labels -> (60000,)
shape of testing images -> (10000, 28, 28)
shape of testing labels -> (10000,)
```

Fig. 12. Loading the MNIST dataset

2. **Adding a Channel Dimension:**
   Since a neural network takes input data in batches, we add a superfluous channel dimension to the training and testing images. Hence the new shape of the images now correspond to (no_of_images, height, width, no_of_channels). Here, the channel dimension has a value of 1 as the images in the MNIST dataset are grayscale.



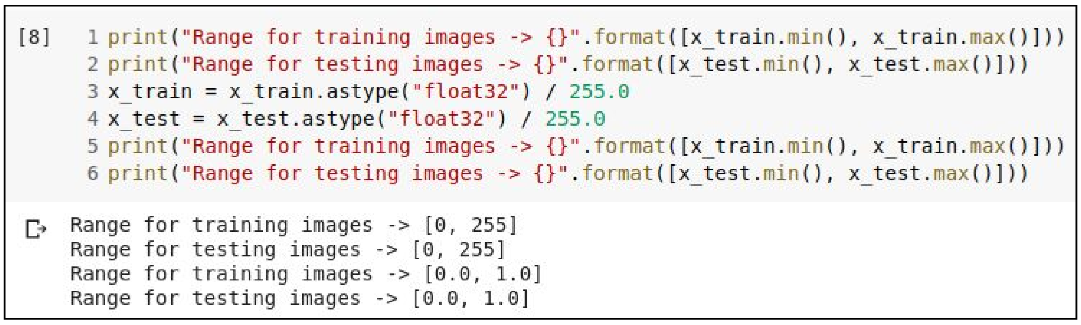
```
[21]
import numpy as np

x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)
print("shape of training images -> {}".format(x_train.shape))
print("shape of testing images -> {}".format(x_test.shape))
```

```
shape of training images -> (60000, 28, 28, 1)
shape of testing images -> (10000, 28, 28, 1)
```

Fig. 13. Adding a channel dimension

3. **Image Normalization:**
   Image normalization is an important step which ensures that each input parameter (i.e, pixel in this case) has a similar data distribution. This makes convergence faster while training the neural network. So we first convert each image from uint8 to float32 format. And then we normalize the pixel intensities of each image by dividing it by 255.0 so that it lies between the range [0.0, 1.0].

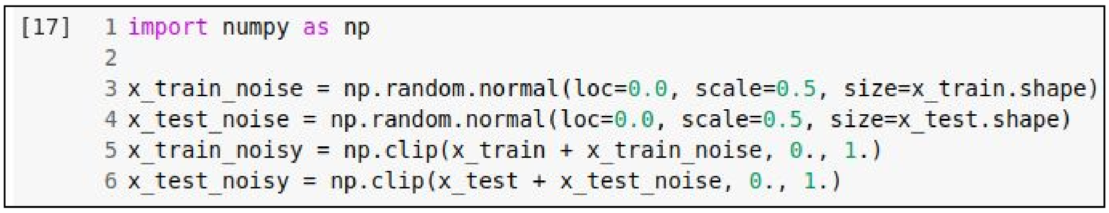


```
[8]
print("Range for training images -> {}".format([x_train.min(), x_train.max()]))
print("Range for testing images -> {}".format([x_test.min(), x_test.max()]))
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
print("Range for training images -> {}".format([x_train.min(), x_train.max()]))
print("Range for testing images -> {}".format([x_test.min(), x_test.max()]))
```

```
Range for training images -> [0, 255]
Range for testing images -> [0, 255]
Range for training images -> [0.0, 1.0]
Range for testing images -> [0.0, 1.0]
```

Fig. 14. Normalizing the MNIST dataset

4. **Adding Synthetic Noise to the Images:**
In this step, we generate synthetic noisy digits by applying a Gaussian noise matrix and clipping the images between 0 and 1. To add random noise to the MNIST images, we use NumPy's random normal distribution with a mean of 0.0 and a standard deviation of 0.5 as shown below in Fig. 15.



```python
import numpy as np

x_train_noise = np.random.normal(loc=0.0, scale=0.5, size=x_train.shape)
x_test_noise = np.random.normal(loc=0.0, scale=0.5, size=x_test.shape)
x_train_noisy = np.clip(x_train + x_train_noise, 0., 1.)
x_test_noisy = np.clip(x_test + x_test_noise, 0., 1.)
```

Fig. 15. Adding synthetic noise to the MNIST dataset

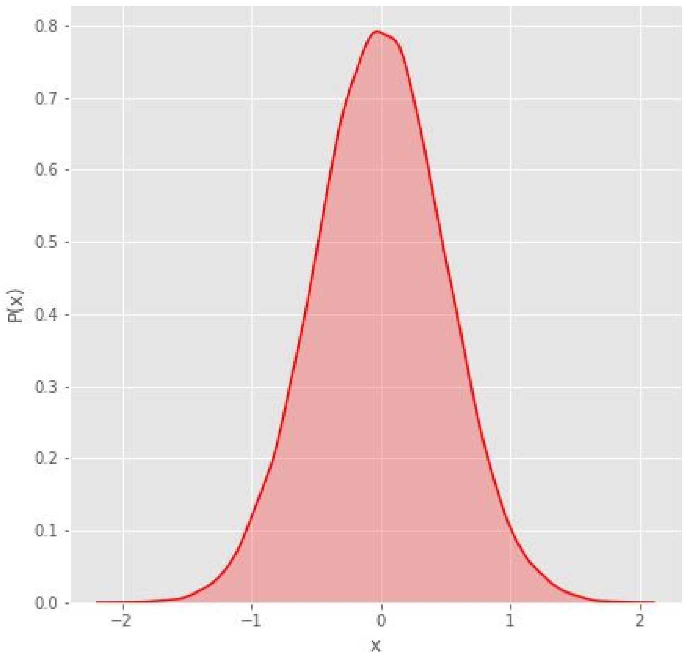


Fig. 16. Normal distribution N(0, 0.5)

The following figure shows an example of how the MNIST images look before (first row) and after (second row) adding synthetic noise sampled from a Normal Distribution N(0, 0.5).

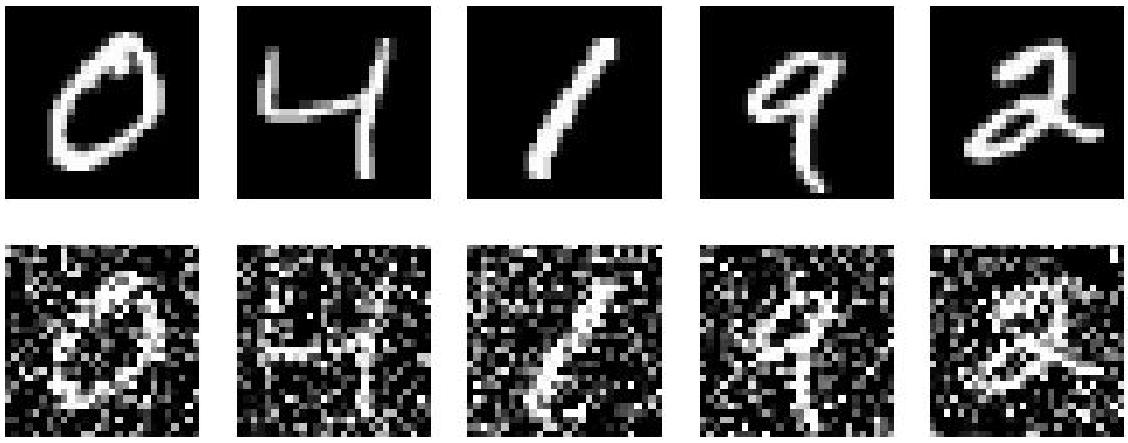


Fig. 17. Images before (first row) and after (second row) addition of synthetic noise

Conclusion:

This completes the Data Preprocessing stage for this case study. This preprocessed dataset is then used for training, validating and testing the model which is explained in Assignment - 2.

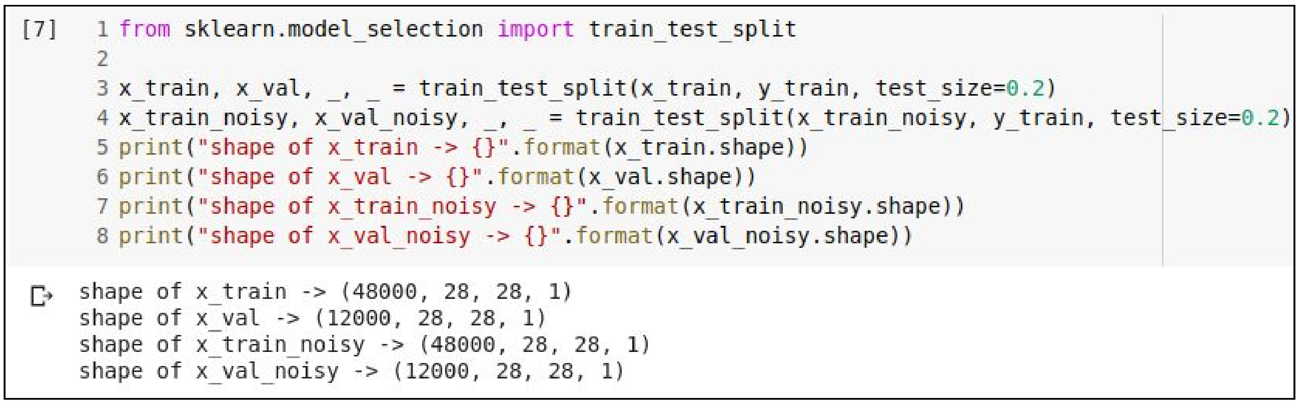
60004170108 - Siddharth Sanghavi - TE COMPS B

ML Assignment - 2

Denoising Images using Autoencoders

Dataset Training, Validation and Test Split:

Since the original MNIST dataset has 60000 training images and 10000 testing images, there is no need to split the training data into training and testing images. Instead we split the training images into 48000 training images and 12000 validation images using Scikit-learn's train_test_split( ) function. We similarly split the preprocessed (noisy) MNIST dataset as shown below in Fig. 1.



```
[7]
from sklearn.model_selection import train_test_split

x_train, x_val, _, _ = train_test_split(x_train, y_train, test_size=0.2)
x_train_noisy, x_val_noisy, _, _ = train_test_split(x_train_noisy, y_train, test_size=0.2)
print("shape of x_train -> {}".format(x_train.shape))
print("shape of x_val -> {}".format(x_val.shape))
print("shape of x_train_noisy -> {}".format(x_train_noisy.shape))
print("shape of x_val_noisy -> {}".format(x_val_noisy.shape))
```

```
shape of x_train -> (48000, 28, 28, 1)
shape of x_val -> (12000, 28, 28, 1)
shape of x_train_noisy -> (48000, 28, 28, 1)
shape of x_val_noisy -> (12000, 28, 28, 1)
```

Fig. 1. Training and Validation split for the MNIST dataset

Modelling Technique:

The agenda for the field of Computer Vision is to enable machines to view the world as humans do, perceive it in a similar manner and even use the knowledge for a multitude of tasks such as Image and Video recognition, Image Analysis & Classification, Media Recreation, etc. The advancements in Computer Vision with Deep Learning has been constructed and perfected with time, primarily over one particular algorithm - a Convolutional Neural Network.

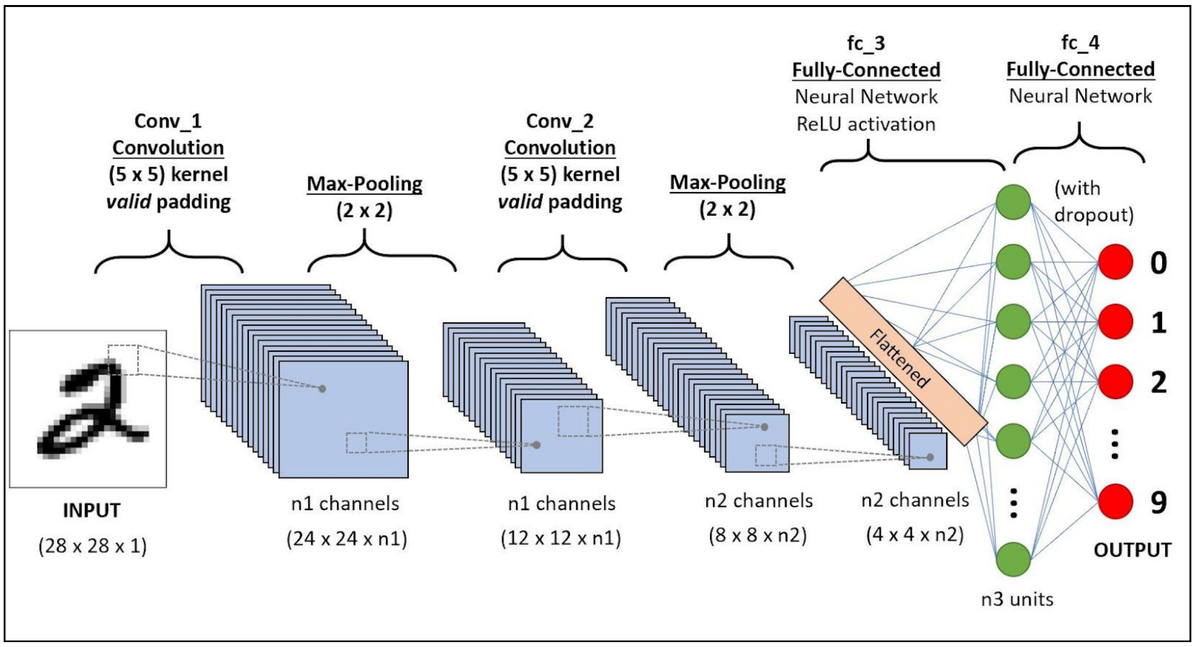


Fig. 2. Architecture of a Convolutional Neural Network

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The preprocessing required in a CNN is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, CNNs have the ability to learn these filters/characteristics.

CNNs take biological inspiration from the visual cortex in the Human Brain. The visual cortex has small regions of cells that are sensitive to specific regions of the visual field. This idea was expanded upon by a fascinating experiment by Hubel and Wiesel in 1962 where they showed that some individual neuronal cells in the brain responded (or fired) only in the presence of edges of a certain orientation. For example, some neurons fired when exposed to vertical edges and some when shown horizontal or diagonal edges. Hubel and Wiesel found out that all of these neurons were organized in a columnar architecture and that together, they were able to produce visual perception. This idea of specialized components inside of a system having specific tasks (the neuronal cells in the visual cortex looking for specific characteristics) is one that machines use as well, and is the basis behind CNNs.

Justification for Choosing Modelling Technique:

The main motivation behind the emergence of Convolutional Neural Networks (CNNs) in Deep Learning scenarios has been to address many of the limitations that traditional neural networks faced when applied to those problems. Some of which are explained below:

- In the past, Traditional Multilayer Perceptron (MLP) models have been used for image recognition. However, due to the full connectivity between nodes, they suffered from the curse of dimensionality, and did not scale well with higher resolution images. A 1000×1000 pixel image with RGB color channels has 3 million weights, which is too high to feasibly process efficiently at scale with full connectivity.

- Also, such network architecture does not take into account the spatial structure of data, treating input pixels which are far apart in the same way as pixels that are close together. This ignores locality of reference in image data, both computationally and semantically. Thus, full connectivity of neurons is wasteful for purposes such as image recognition that are dominated by spatially local input patterns.

CNNs mitigate the challenges posed by the MLP architecture by exploiting the strong spatially local correlation present in natural images. As opposed to MLPs, CNNs have the following distinguishing features:

- **3D volumes of neurons:** The layers of a CNN have neurons arranged in 3 dimensions: width, height and depth where each neuron inside a convolutional layer is connected to only a small region of the layer before it, called a receptive field. Distinct types of layers, both locally and completely connected, are stacked to form a CNN architecture.

- **Local connectivity:** CNNs exploit spatial locality by enforcing a local connectivity pattern between neurons of adjacent layers. The architecture thus ensures that the learned "filters" produce the strongest response to a spatially local input pattern. Stacking many such layers leads to non-linear filters that become increasingly global (i.e. responsive to a larger region of pixel space) so that the network first creates representations of small parts of the input, then from them assembles representations of larger areas.

- **Parameter Sharing:** In CNNs, each filter is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and form a feature map. This means that all the neurons in a given convolutional layer respond to the same feature within their specific response field. Replicating units in this way allows for the resulting feature map to be equivariant under changes in the locations of input features in the visual field, i.e. they grant translational equivariance. Weight sharing dramatically reduces the number of free parameters learned, thus lowering the memory requirements for running the network and allowing the training of larger, more powerful networks.

- **Pooling:** In a CNN's pooling layers, feature maps are divided into rectangular sub-regions, and the features in each rectangle are independently down-sampled to a single value, commonly by taking their average or maximum value. In addition to reducing the sizes of feature maps, the pooling operation grants a degree of translational invariance to the features contained therein, allowing the CNN to be more robust to variations in their positions.

- **Sparse Representations:** A CNN is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. The “filters” in CNNs tend to be drastically smaller than the input which simplifies the number of computations required to train the model or to make predictions. This results in reducing the computational operations by a huge factor and leads to efficient training.

Together, these properties allow CNNs to achieve better generalization on Computer Vision problems. **Hence, for this case study, we choose to use a CNN-based architecture to construct the Autoencoder.**

Model Architecture:

Typically, we think of an autoencoder having two components/subnetworks:

- **Encoder:** Accepts the input data and compresses it into the latent-space. If we denote the input data as **x** and the encoder function as **E**, then the output latent-space representation, s, would be given by: **s = E(x)**

- **Decoder:** Accepts the latent-space representation **s** and then reconstructs the original input. If we denote the decoder function as **D** and the output of the decoder as **o**, then we can represent the decoder as: **o = D(s)**

Combining the aforementioned mathematical notations, the entire training process of the autoencoder can be represented as: **o = D(E(x))**

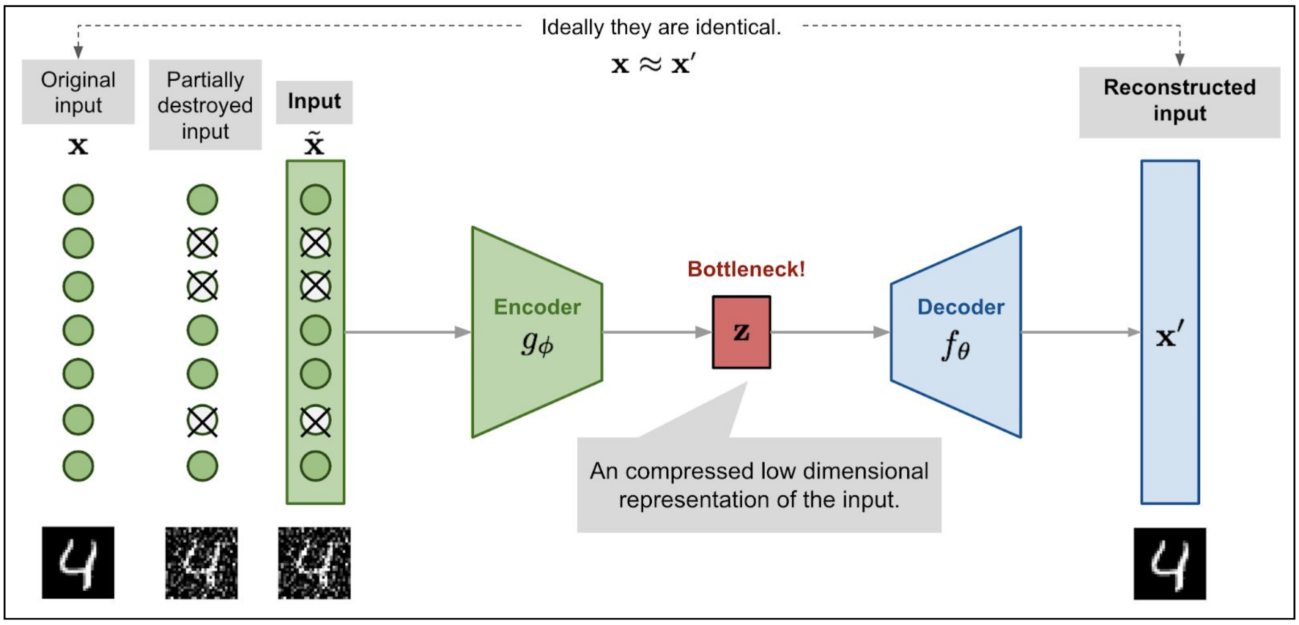
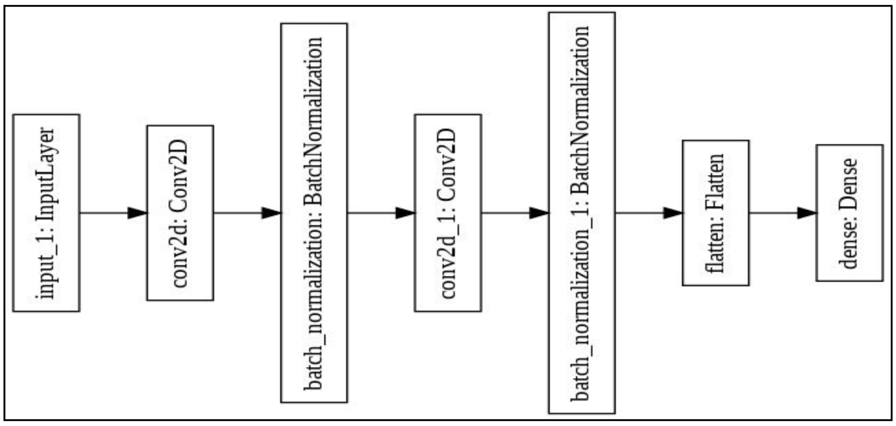
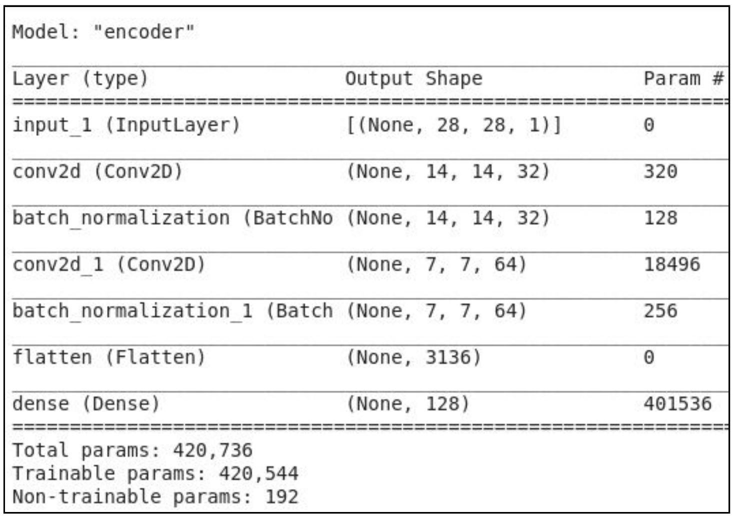

Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Original input
$\mathbf{x}$
Partially destroyed input
Input
$\tilde{\mathbf{x}}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
Reconstructed input
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Fig. 3. Architecture of a Denoising Autoencoder

- **Encoder:**
  The encoder in this architecture consists of a CNN without the last Fully Connected layers. It takes a batch of noisy input images as a 4-dimensional NumPy array of shape (batch_size, 28, 28, 1). It then downsamples/encodes this input and outputs a 2-dimensional NumPy array of shape (batch_size, 128). Each row in the output array represents the 128-dimensional latent-space representation of an input image in that particular batch.


input_1: InputLayer
conv2d: Conv2D
batch_normalization: BatchNormalization
conv2d_1: Conv2D
batch_normalization_1: BatchNormalization
flatten: Flatten
dense: Dense



```
Model: "encoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 14, 14, 32)        320
_________________________________________________________________
batch_normalization (BatchNo (None, 14, 14, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 7, 7, 64)          18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 7, 7, 64)          256
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 128)               401536
=================================================================
Total params: 420,736
Trainable params: 420,544
Non-trainable params: 192
```

Fig. 4. Architecture of the Encoder

The encoder uses stacks of Convolutional layers as shown above in Fig. 4. to achieve this downsampled representation of the input image. Each Convolutional layer performs the Convolution operation as explained after Fig. 5.

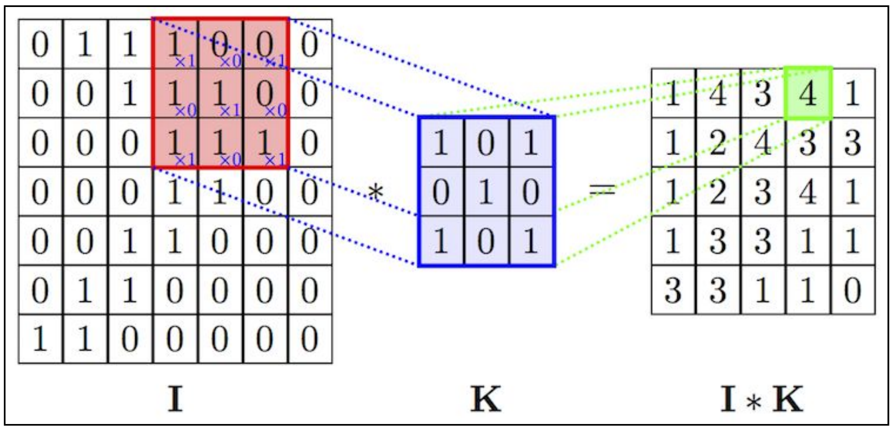
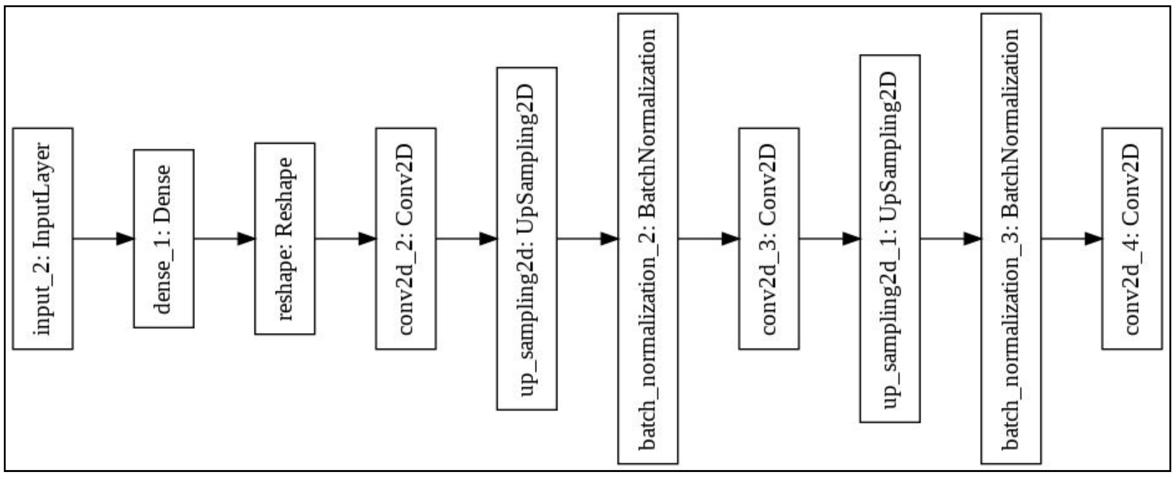


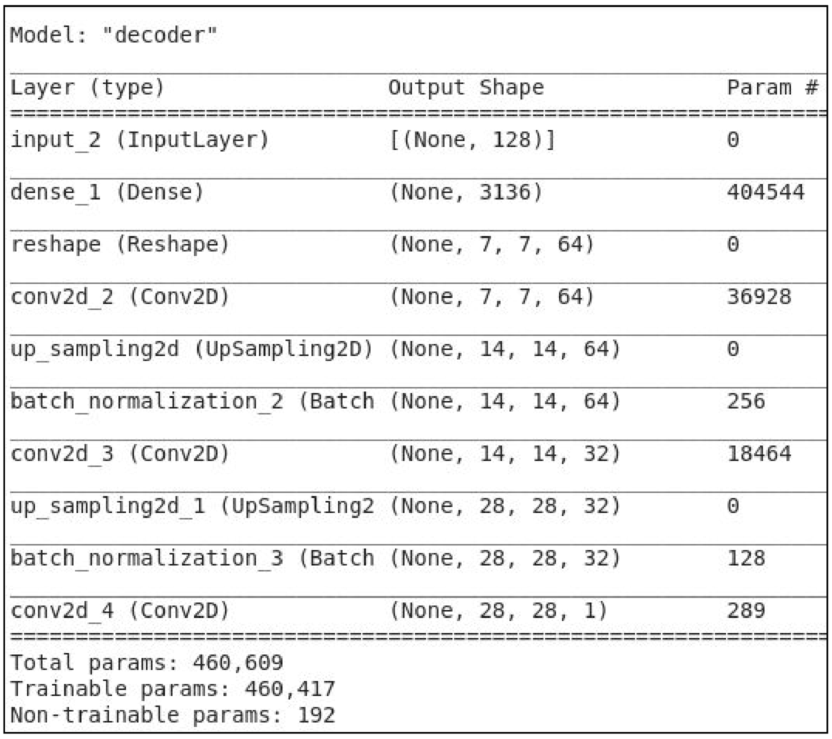
Fig. 5. The Convolution operation

Convolutional Layer: The main building block of the encoder is the Convolutional Layer. The primary purpose of Convolution is to extract features from the input image. It preserves the spatial relationship between pixels by learning image features using small squares of input data. The type of convolutions we are interested in are 2-dimensional discrete convolutions, which act like a weighted sliding sum over an area of pixels.

For instance, a 3x3 matrix called a kernel/filter (K in Fig. 5.) slides across the pixels in an image. At each point, it calculates the weighted sum of the kernels' values and every pixel in the 3x3 chunk of the image. The sum is then put in the first value of the output image called the feature map (I*K in Fig. 5.). The kernel then slides over one pixel and repeats the process for every pixel in the image.The red area shown above in Fig. 5. where the convolution operation takes place is called the receptive field.

- **Decoder:** The decoder in this architecture consists of another CNN without the last Fully Connected layers. It takes the output of the encoder as input in the form of a 2-dimensional NumPy array of shape (batch_size, 128). It then upsamples/decodes this input and outputs a 4-dimensional Numpy array of shape (batch_size, 28, 28, 1). Each row in the batch_size dimension represents the denoised image from that particular batch. The decoder uses stacks of Upsampling layers and Convolutional layers as shown below in Fig. 6. to achieve this downsampled representation of the input image.


input_2: InputLayer
dense_1: Dense
reshape: Reshape
conv2d_2: Conv2D
up_sampling2d: UpSampling2D
batch_normalization_2: BatchNormalization
conv2d_3: Conv2D
up_sampling2d_1: UpSampling2D
batch_normalization_3: BatchNormalization
conv2d_4: Conv2D



```
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 128)]             0
_________________________________________________________________
dense_1 (Dense)              (None, 3136)              404544
_________________________________________________________________
reshape (Reshape)            (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 64)          36928
_________________________________________________________________
up_sampling2d (UpSampling2D) (None, 14, 14, 64)        0
_________________________________________________________________
batch_normalization_2 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 32)        18464
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 28, 28, 32)        0
_________________________________________________________________
batch_normalization_3 (Batch (None, 28, 28, 32)        128
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 28, 28, 1)         289
=================================================================
Total params: 460,609
Trainable params: 460,417
Non-trainable params: 192
```

Fig. 6. Architecture of the Decoder

Each block in the decoder consists of a Convolutional layer followed by an Upsampling layer. Many such blocks are arranged in stacks to build the decoder. Each block performs the Convolution operation which is the same as that explained above for the encoder. The Convolution operation is followed by an Upsampling operation which is explained as follows.

UpSampling Layer: Upsampling cannot reconstruct any lost information. Its role is to bring back the resolution to the resolution of the previous layer. The sole purpose of Upsampling layers is to reduce computations in each layer, while keeping the dimension of input/output the same as before. The ways to upsample the compressed image is by Unpooling (the reverse of pooling) using techniques such as:

- **Nearest Neighbor Interpolation:** It copies the value from the nearest pixel.
- **Bilinear Interpolation:** It uses all nearby pixels to calculate the pixel's value, using linear interpolations.
- **Bicubic Interpolation:** Again uses all nearby pixels to calculate the pixel's values, through polynomial interpolations. Usually produces a smoother surface than the previous techniques, but it is harder to compute.

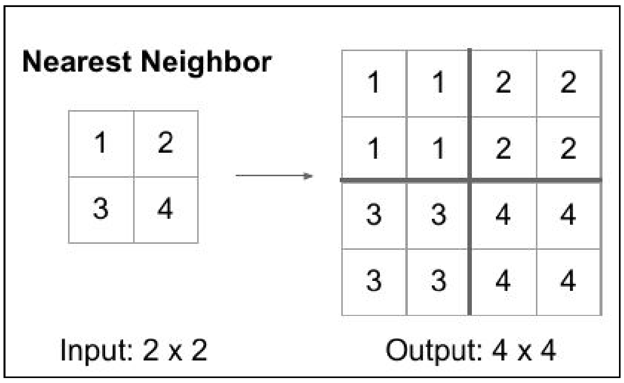


Fig. 7. Nearest Neighbor Upsampling operation

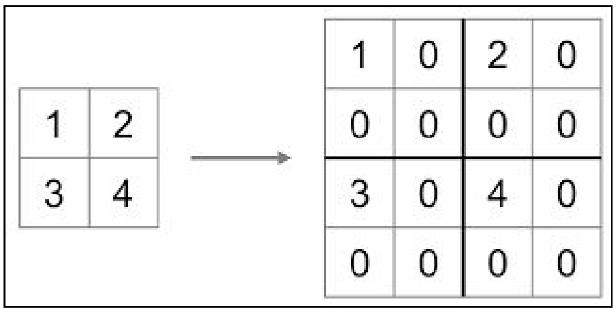


Fig. 8. Bilinear Upsampling operation

- **Autoencoder:**
The Autoencoder is then constructed by concatenating the output of the encoder with the input of the decoder as shown below in Fig. 9.

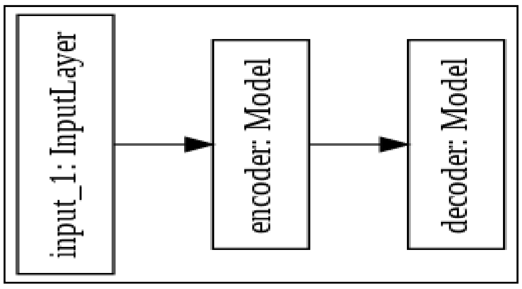
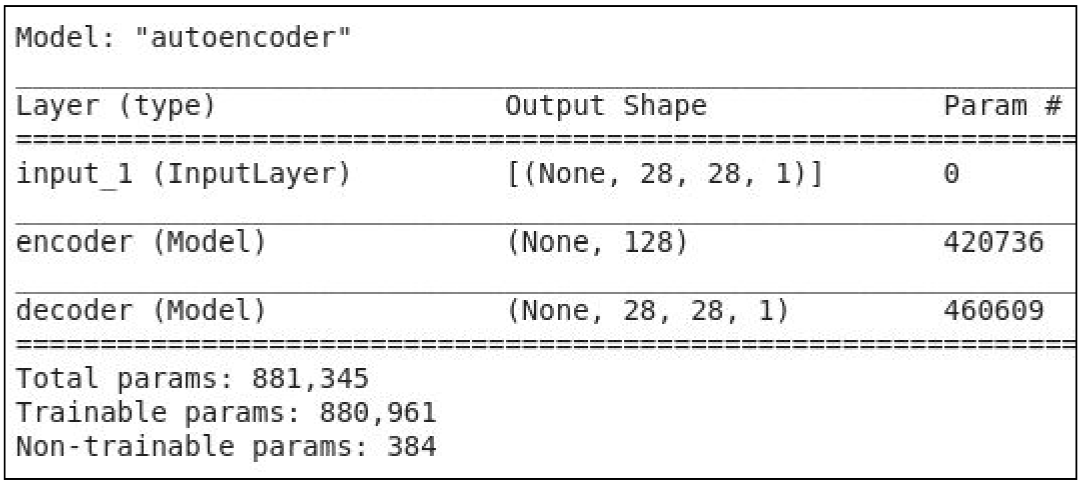


Fig. 9. Concatenating the encoder output with the decoder input



```
Model: "autoencoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
encoder (Model)              (None, 128)               420736
_________________________________________________________________
decoder (Model)              (None, 28, 28, 1)         460609
=================================================================
Total params: 881,345
Trainable params: 880,961
Non-trainable params: 384
```

Fig. 10. Architecture of the Autoencoder

Learning Methodology and Parameter Tuning for Model Training:

The encoder and decoder are chosen to be parametric functions and to be differentiable with respect to the distance function, so the parameters of the encoding/decoding functions can be optimized to minimize the reconstruction loss, using the **Mini-batch Gradient Descent Algorithm.**

Gradient Descent Algorithm:

- Gradient Descent is a first-order optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to weights **w** and biases **b** in our neural network.

- Optimization refers to the task of minimizing/maximizing an objective function **f(x)** parameterized by **x**. In machine learning terminology, it is the task of minimizing the cost/loss/objective function **J(w)** parameterized by the model's parameters **w, b ∈ R^d**.

- Consider the 3-dimensional graph below (Fig. 11.) in the context of a cost function. Our goal is to move from the mountain in the top right corner (high cost) to the dark blue sea in the bottom left (low cost). The arrows represent the direction of steepest descent (negative gradient) from any given point–the direction that decreases the cost function as quickly as possible.
- Starting at the top of the mountain, we take our first step downhill in the direction specified by the negative gradient. Next we recalculate the negative gradient (passing in the coordinates of our new point) and take another step in the direction it specifies. We continue this process iteratively until we get to the bottom of our graph, or to a point where we can no longer move downhill–a local minimum.

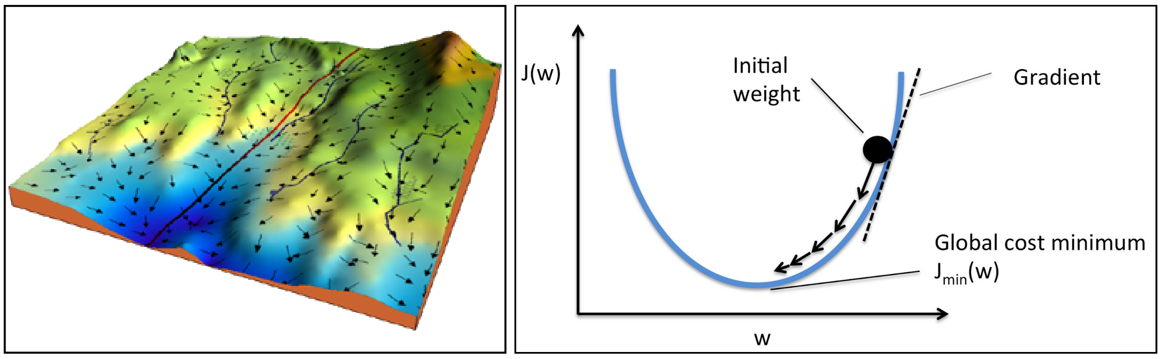


Fig. 11. Gradient Descent algorithm objective

- **Learning Rate:** The size of these steps is called the learning rate. With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing. A low learning rate is more precise, but calculating the gradient is time-consuming, so it will take us a very long time to get to the bottom.

- **Loss Function:** It tells us “how good” our model is at making predictions for a given set of parameters. The cost function has its own curve and its own gradients. The slope of this curve tells us how to update our parameters to make the model more accurate.

Cost Function

$$J(\Theta_0, \Theta_1) = \frac{1}{2m}\sum_{i=1}^{m}[\underset{\substack{\uparrow \\ \text{Predicted Value}}}{h_\Theta(x_i)} - \underset{\substack{\uparrow \\ \text{True Value}}}{y_i}]^2$$

Fig. 12. Loss function

- **Algorithm:**
  Repeat until convergence {

$$\begin{aligned} \frac{\partial}{\partial\Theta}J_\Theta &= \frac{\partial}{\partial\Theta}\frac{1}{2m}\sum_{i=1}^{m}[h_\Theta(x_i) - y]^2 \\ &= \frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x_i) - y)\frac{\partial}{\partial\Theta_j}(\Theta x_i - y) \\ &= \frac{1}{m}(h_\Theta(x_i) - y)x_i \end{aligned}$$

Therefore,

$$\boxed{\Theta_j := \Theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}[(h_\Theta(x_i) - y)x_i]}$$

  }

The weights and biases are updated using the **Backpropagation Algorithm.** Here we use the **Mean Squared Error** between the predicted denoised images (yhat in Fig. 13.) and the original images (y in Fig. 13.) as the loss/objective function. This loss is then minimized using the **Mini-batch Gradient Descent Algorithm.**

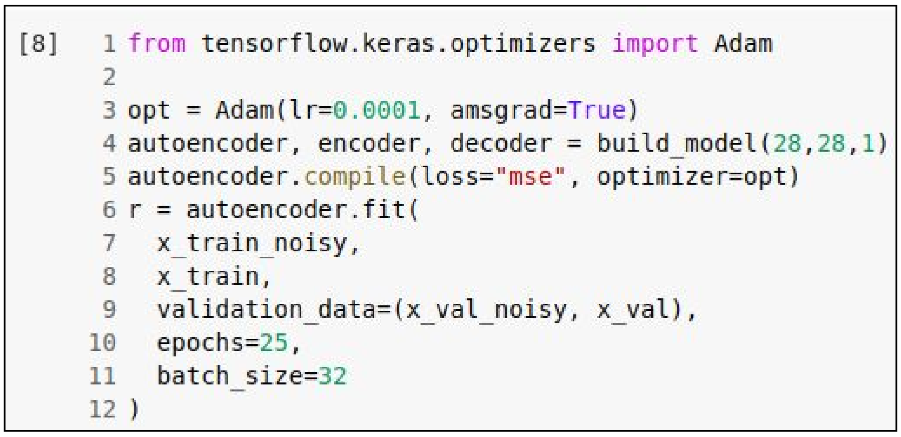
$$MSE = \frac{1}{N}\sum_{i=0}^{N}(\hat{y}_i - y_i)^2$$

Fig. 13. Mean Squared Error

The use of Mini-batch Gradient Descent in the neural network setting is motivated by the high cost of running back propagation over the full training set. It can overcome this cost and lead to fast convergence. Generally each parameter update in Mini-batch Gradient Descent is computed w.r.t a few training examples or a mini-batch as opposed to a single example. The reason for this is twofold: first this reduces the variance in the parameter update and can lead to more stable convergence, second this allows the computation to take advantage of highly optimized matrix operations that should be used in a well vectorized computation of the cost and gradient. Hence we use a **mini-batch size of 32 images for 25 epochs.**

Hence, during training, we provide batches of noisy images as input, and their corresponding batches of noise-free images as targets, and the encoder and decoder will together learn to remove the particular noise present in the predicted/output images.

We use the **Adam Optimizer with an initial learning rate (alpha) of 0.0001** along with Mini-batch Gradient Descent. Adaptive Moment Estimation (Adam) is an adaptive learning rate method that computes adaptive learning rates for each parameter. Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with Momentum.



```
[8]
from tensorflow.keras.optimizers import Adam

opt = Adam(lr=0.0001, amsgrad=True)
autoencoder, encoder, decoder = build_model(28,28,1)
autoencoder.compile(loss="mse", optimizer=opt)
r = autoencoder.fit(
  x_train_noisy,
  x_train,
  validation_data=(x_val_noisy, x_val),
  epochs=25,
  batch_size=32
)
```

Fig. 14. Model training

Conclusion:

The trained and validated model will then be used for the testing stage in which noisy images will be denoised by the Autoencoder which is explained in Assignment - 3.

60004170108 - Siddharth Sanghavi - TE COMPS B

ML Assignment - 3

Denoising Images using Autoencoders

Fit Statistics:

The model is trained on 60000 noisy images and validated on 10000 noisy images in mini-batches of 32 over 25 epochs. Since this is a Regression problem, the metrics of Accuracy, Precision, Recall and ROC are not applicable here. Instead, the fit metrics of this problem explained with respect to Regression as follows:

- **Mean Squared Error (MSE):**
  In statistics, the Mean Squared Error (MSE) of an estimator (of a procedure for estimating an unobserved quantity) measures the average of the squares of the errors - i.e., the average squared difference between the estimated values and what is estimated.

$$\boxed{\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

Fig. 1. Mean Squared Error

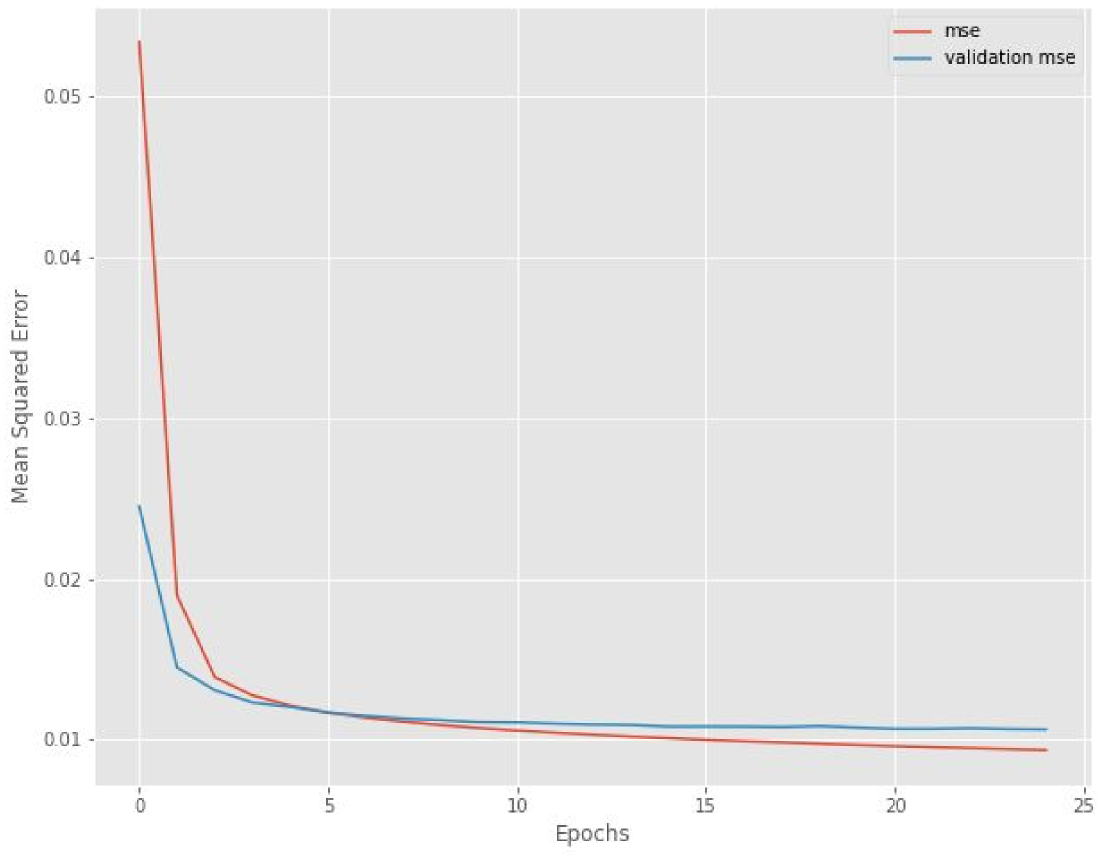


Fig. 2. MSE vs. Epochs

  - The fact that MSE is almost always strictly positive (and not zero) is because of randomness or because the estimator/model does not account for information that could produce a more accurate estimate.
  - MSE has the disadvantage of heavily weighting outliers. This is a result of the squaring of each term, which effectively weights large errors more heavily than small ones. This property, undesirable in many applications, has led researchers to use alternatives such as the mean absolute error or root mean squared error.

- As shown in Fig. 2., the model is able to minimize the MSE starting from Epoch 0 to Epoch 25. However, the model slightly overfits the dataset from epoch 15 as the training MSE is slightly less than validation MSE in that region.

- **Mean Absolute Error (MAE):**
  In statistics, the Mean Absolute Error (MAE) of an estimator (of a procedure for estimating an unobserved quantity) measures the average of the absolute of the errors - i.e., the average absolute difference between the estimated values and what is estimated.

$$\boxed{\mathrm{MAE} = \frac{1}{n}\sum_{j=1}^{n} |y_j - \hat{y}_j|}$$

Fig. 3. Mean Absolute Error

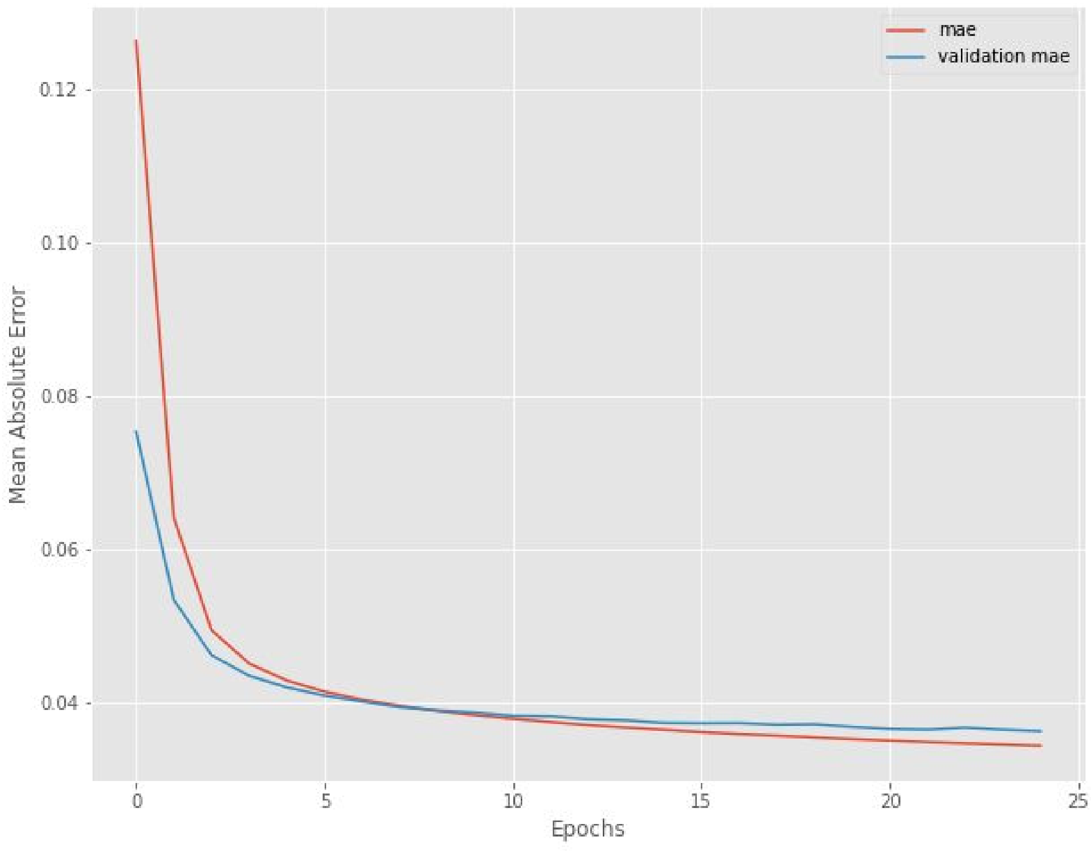


Fig. 4. MAE vs. Epochs

  - MAE measures the average magnitude of the errors in a set of predictions, without considering their direction. All individual differences in MAE have equal weight. Hence it is less sensitive to the occasional large errors because it does not square the errors in the calculation. This means that outliers in our data will contribute to much higher total error in the MSE than they would the MAE.

  - As shown in Fig. 4., the model is able to minimize the MAE starting from Epoch 0 to Epoch 25. However, the model slightly overfits the dataset from epoch 20 as the training MAE is slightly less than validation MAE in that region.

- **Root Mean Squared Error (RMSE):**
  In statistics, the Root Mean Squared Error (RMSE) of an estimator (of a procedure for estimating an unobserved quantity) measures the square root of the average of the squares

of the errors - i.e., the square root of average squared difference between the estimated values and what is estimated.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2}$$

Fig. 5. Root Mean Squared Error

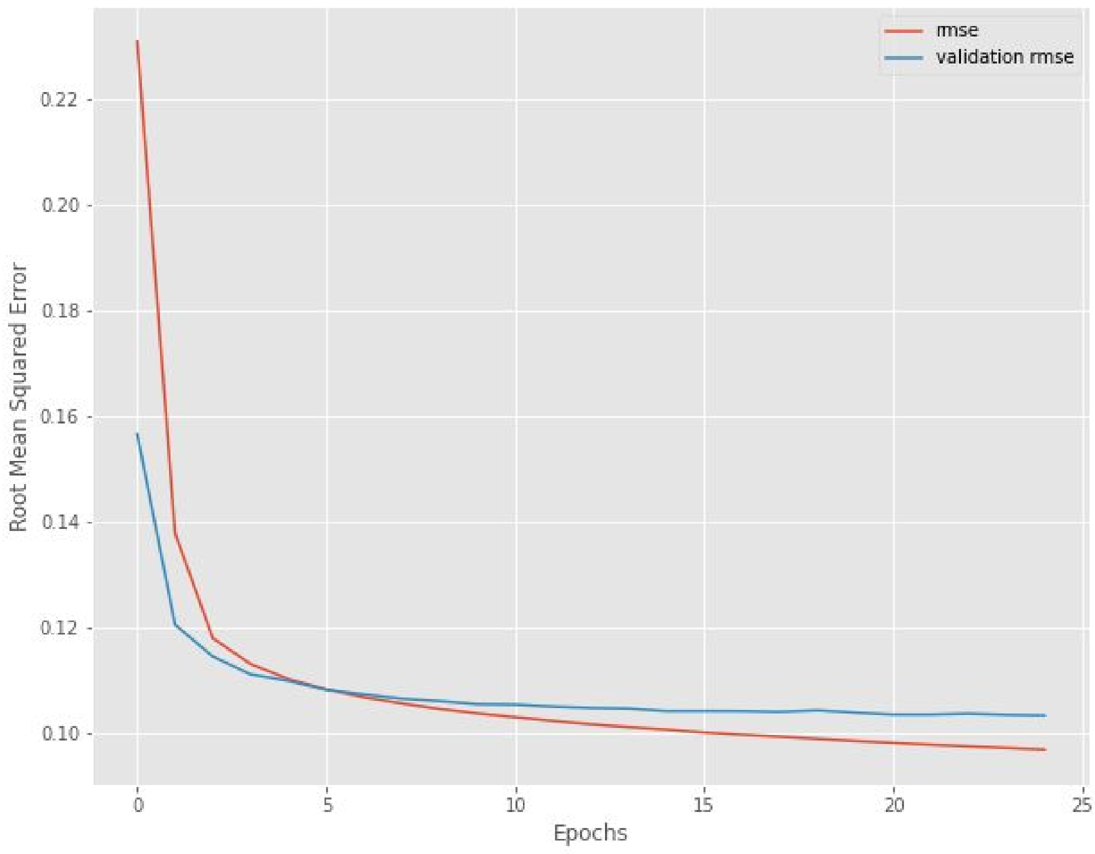


Fig. 6. RMSE vs. Epochs

- RMSE is a quadratic scoring rule that also measures the average magnitude of the error. However, it is more sensitive than other measures to the occasional large error. Taking the square root of the average squared errors has some interesting implications for RMSE. Since the errors are squared before they are averaged, the RMSE gives a relatively high weight to large errors. This means the RMSE is more useful when large errors are particularly undesirable.
- As shown in Fig. 6., the model is able to minimize the RMSE starting from Epoch 0 to Epoch 25. However, the model slightly overfits the dataset after epoch 20 as the training RMSE is slightly less than validation RMSE in that region.

- **Soft Accuracy:**
  The soft accuracy is measured by comparing the similarity between the input target images and the output denoised images. We check the element-wise equality between the original image array and denoised image array for each image in a mini-batch This returns batch_size no. of boolean arrays. Then we take the reduced mean of this mini-batch and return a scalar value in the range [0, 1] which represents the soft accuracy metric. As shown below in Fig. 7., the model manages to obtain a training accuracy of 0.9722 and validation accuracy of .9697. This means that the model slightly overfits the datasets.

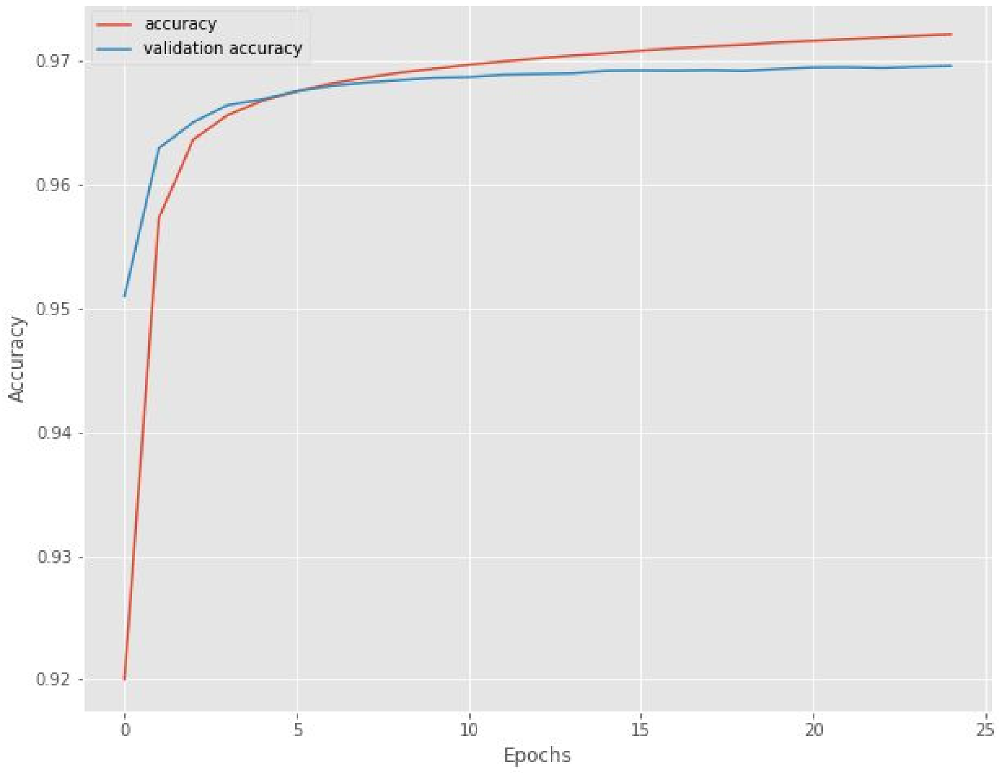


Fig. 7. Accuracy vs. Epochs

- **R2 Score:** R-squared (R2) or Coefficient of Determination is a statistical measure that represents the proportion of the variance for a dependent variable that's explained by an independent variable or variables in a regression model.

$$\boxed{R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \overline{y})^2}}$$

Fig. 8. R2 Score

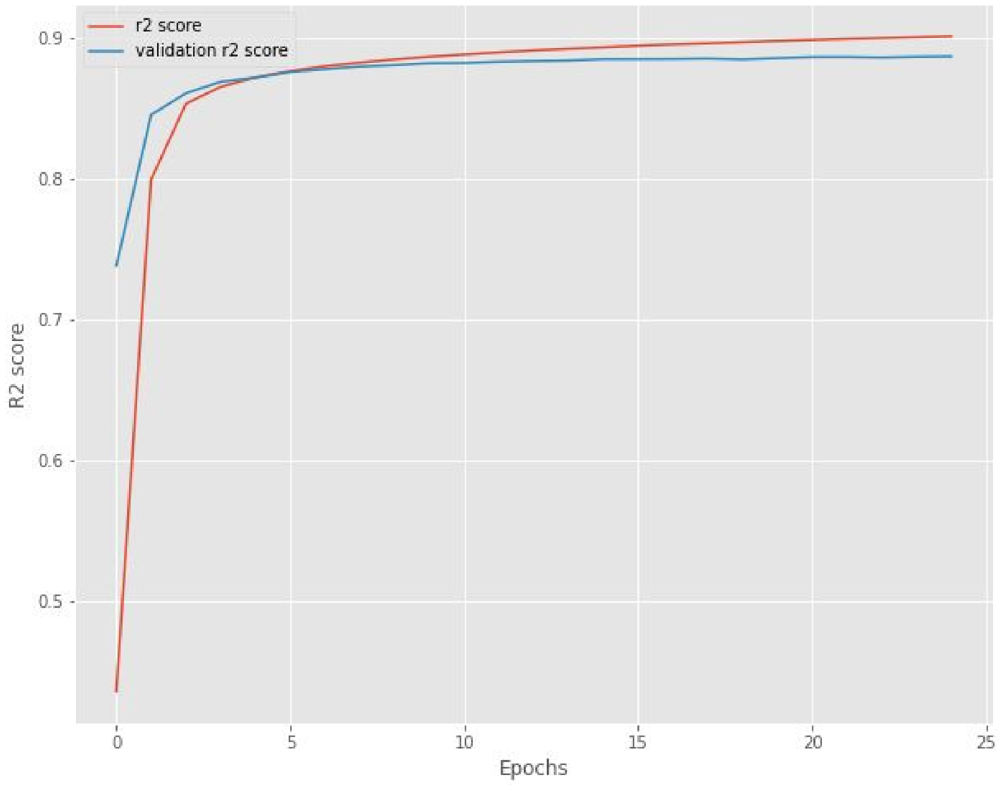


Fig. 9. R2 Score vs. Epochs

- It provides a measure of how well observed outcomes are replicated by the model, based on the proportion of total variation of outcomes explained by the model. R2 Score gives the statistical measure of how well the regression predictions approximate the real data points. An R2 Score of 1 indicates that the regression predictions perfectly fit the data.

- As shown in Fig. 9., the model is able to maximize the R2 Score starting from Epoch 0 to Epoch 25. The R2 Score at Epoch 25 for training has a value of 0.9012. Whereas for validation, the value of R2 Score is 0.8869. This means that the model is able to near perfectly fit the original data.

Latent-space Representation:

The latent-space is simply a representation of compressed data in which similar data points are closer together in space. Latent-space is useful for learning data features and for finding simpler representations of data for analysis. We can understand patterns or structural similarities between data points by analyzing data in the latent-space, be it through manifolds, clustering, etc. In this case study, the output of the Encoder is a 2-dimensional NumPy array of shape (batch_size, 128). Each row in this array represents a 128-dimensional latent-space vector.

Latent-space Encodings for Training Images:

- **PCA Visualization:**
  Here we use the Scikit-learn's PCA (Principal Component Analysis) implementation to visualize 30000 encoded training images. The algorithm manages to project the 128-dimensional latent-space vector for each image onto a 2-dimensional space using 2 Principal Components as shown below in Fig. 10.

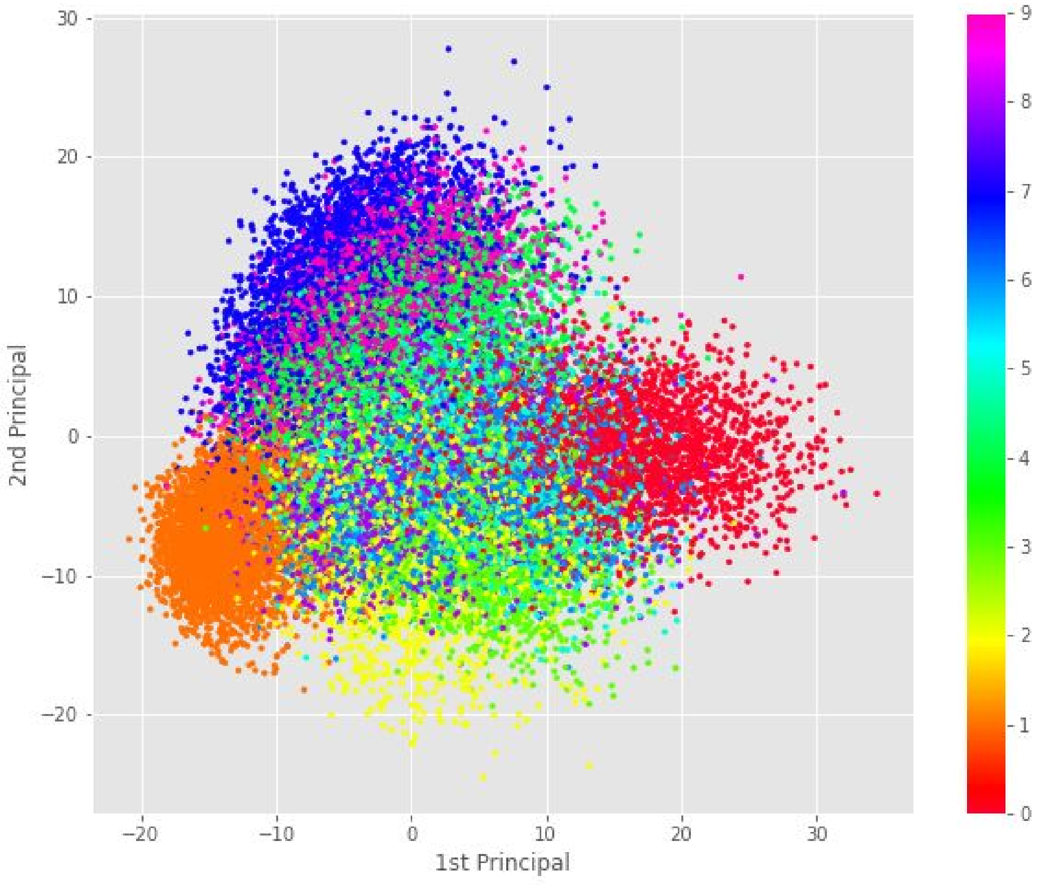


Fig. 10. PCA Visualization of the latent-space for training images

- **Latent-space Image Visualization:**
  These 128-dimensional latent-space representations of training images are of the shape 8x4x4, so we reshape them to 4x32 in order to be able to display them as images. Here we visualize the latent-space encodings for 5 training images as shown below in Fig. 11.

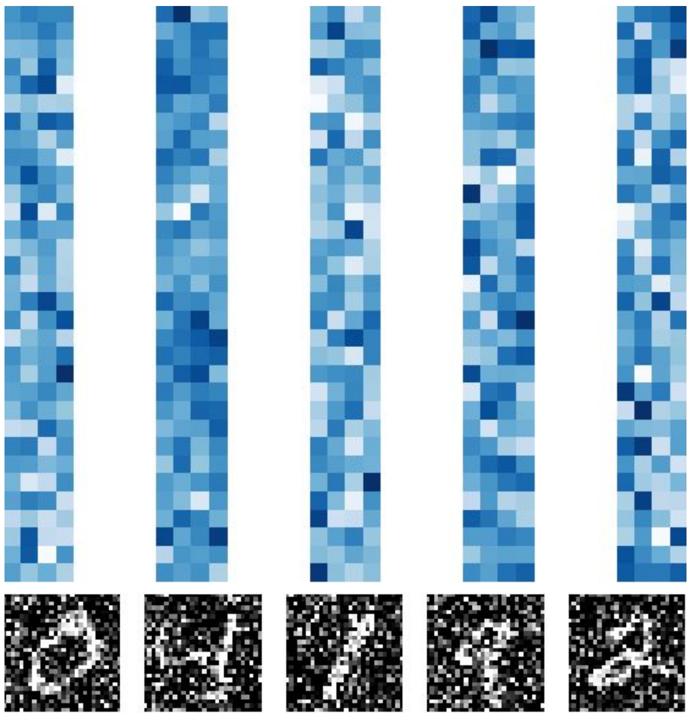


Fig. 11. Latent-space encodings for corresponding noisy training images

Model Testing:

In this stage, we provide the 10000 noisy testing images as input to the Autoencoder that has been previously trained and validated. The Autoencoder outputs a 4-dimensional NumPy array of shape (10000, 28, 28, 1). Each row in this array represents the images that are denoised by the Autoencoder. These denoised images are then visualized with original, noisy testing images as shown below in Fig. 12. and Fig. 13.

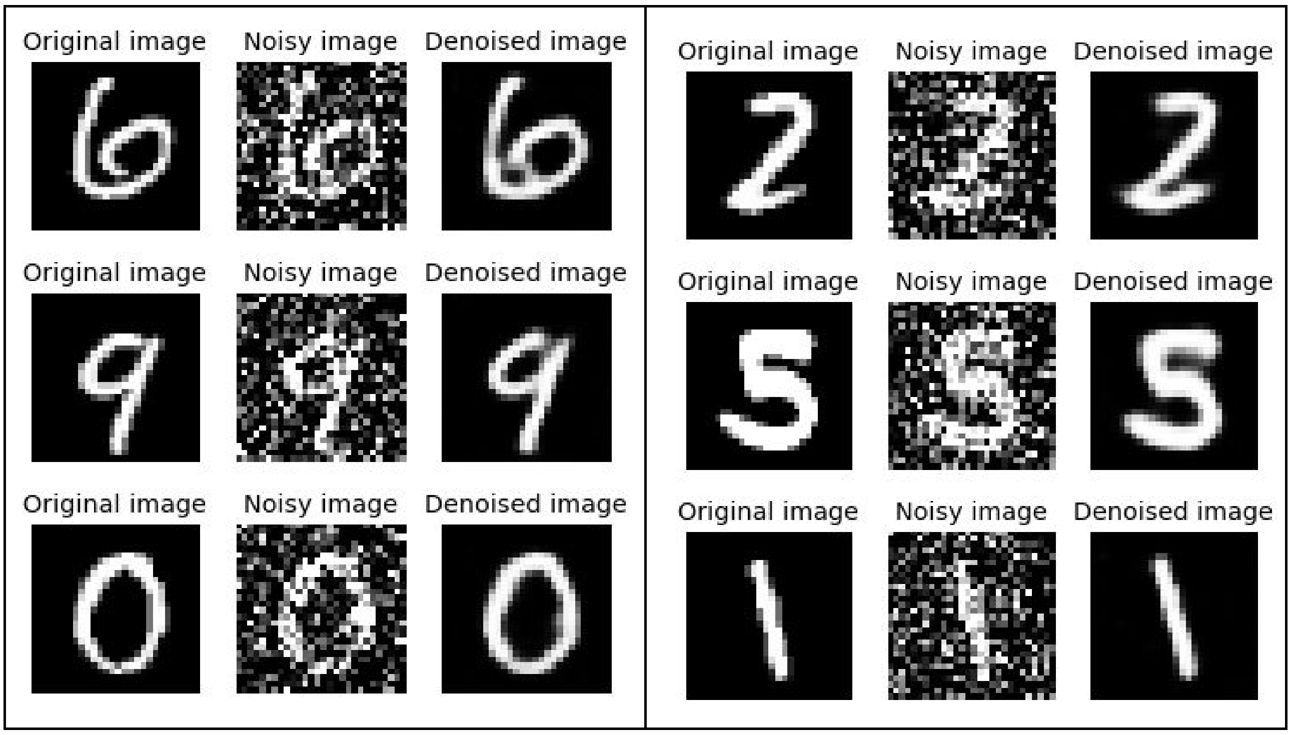


Fig. 12. Comparison of Original, Noisy and Denoised Images

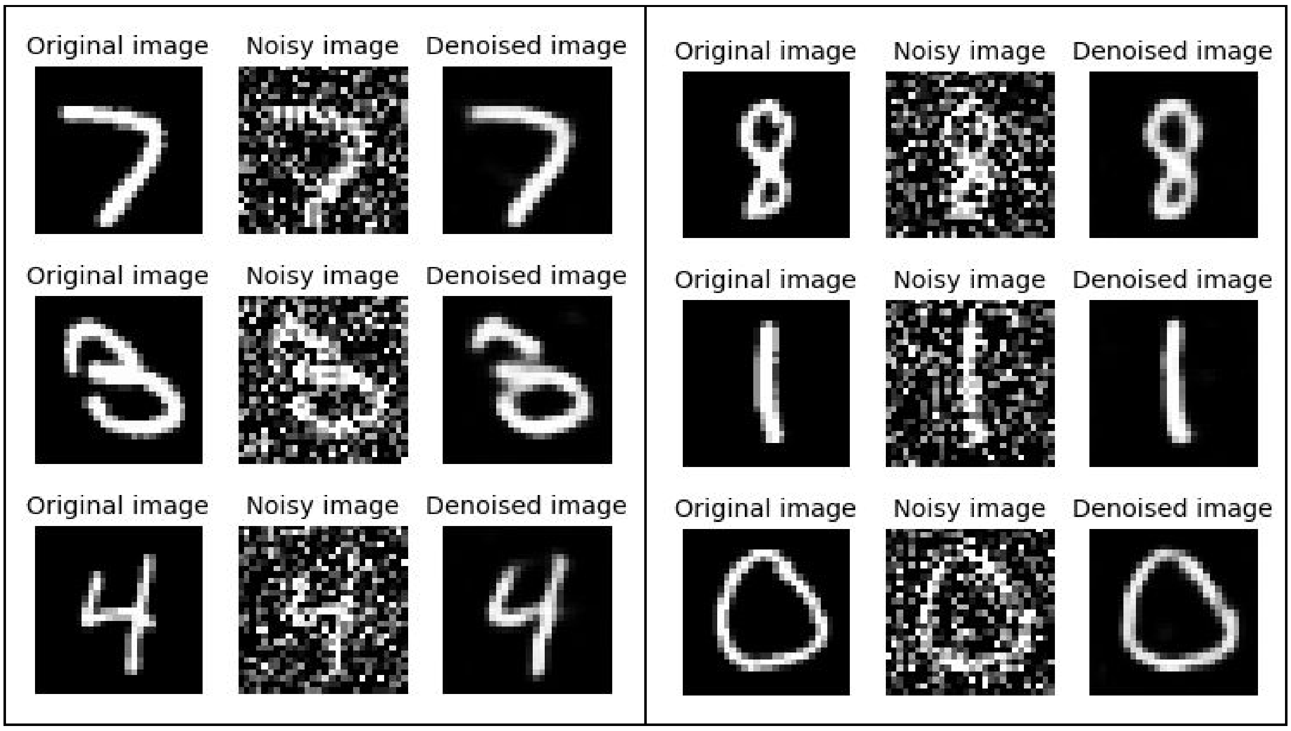


Fig. 13. Comparison of Original, Noisy and Denoised Images

Latent-space Encodings for Testing Images:

- **PCA Visualization:**
  Here we use the Scikit-learn's PCA (Principal Component Analysis) implementation to visualize 10000 encoded testing images. The algorithm manages to project the 128-dimensional latent-space vector for each image onto a 2-dimensional space using 2 Principal Components as shown below in Fig. 14.

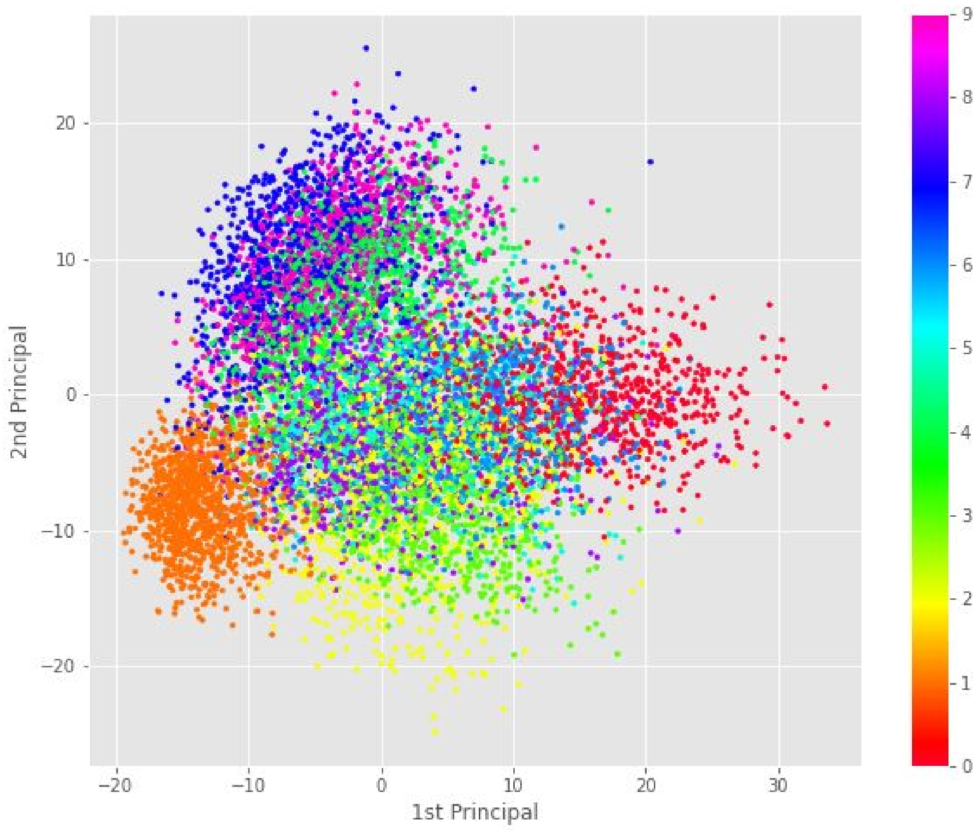


Fig. 14. PCA Visualization of the latent-space for testing images

- **Latent-space Image Visualization:**
  These 128-dimensional latent-space representations of testing images are of the shape 8x4x4, so we reshape them to 4x32 in order to be able to display them as images. Here we visualize the latent-space encodings for 5 training images as shown below in Fig. 15.

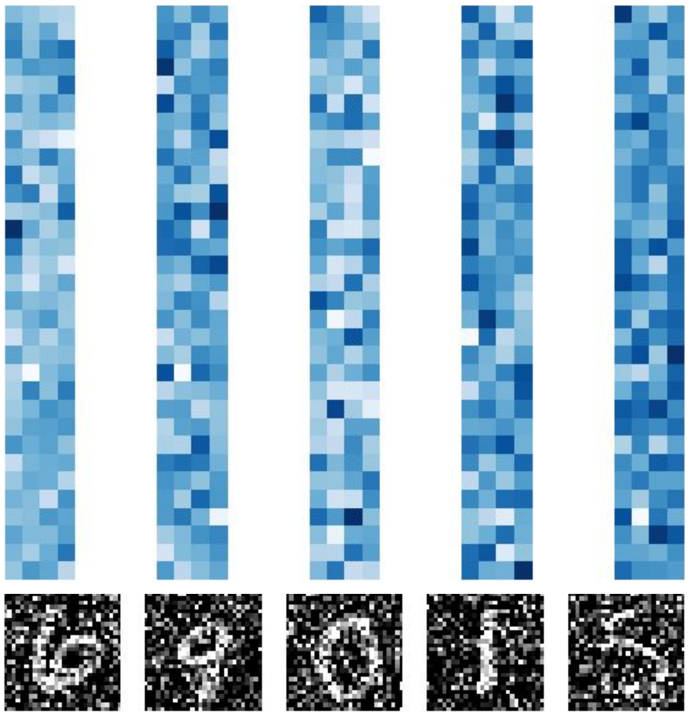


Fig. 15. Latent-space encodings for corresponding noisy testing images

Conclusion:

One of the main application areas for autoencoders is Noise Reduction also called Denoising. As evident from the above results, one can find a few key principles that tell us why Autoencoders are so suitable for removing noise from signals or images. These principles are elaborated as follows:

- **Autoencoders are learned automatically from data examples**, which is a useful property: it means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It doesn't require any new engineering, just appropriate training data.

- **Autoencoders are lossy,** which means that the decompressed outputs will be degraded compared to the original inputs (similar to MP3 or JPEG compression). Because the Autoencoder learns to convert high-dimensional data (e.g., an image) into lower-dimensional format (i.e., the encoded state/latent-space), data must be dropped in order to maximize the relationships between image and encoded state. Additionally, going from latent state to output also incurs information loss.

- **Autoencoders are data-specific,** which means that they will only be able to compress data similar to what they have been trained on. An Autoencoder trained on pictures of faces would do a rather poor job of compressing pictures of trees, because the features it would learn would be face-specific. This behavior emerges because the features the Autoencoder is used on have never been used for learning, and are therefore not present in the latent-space.