A Project Report On

Dimensionality reduction with autoencoders

Project Report submitted in partial fulfillment of the requirements for the 6th Semester of Master of Computer Application (MCA) under University of Calcutta

Submitted By

Mainak Ghosh

Under the supervision of

Mr. Dibyendu Bikash Seal

Assistant Professor,

A.K. Choudhury School of Information Technology,

University of Calcutta

University of Calcutta, JD-2 Sector-III, Saltlake, Kolkata-700106

Contents

Abstract

Autoencoders play a fundamental role in unsupervised learning and in deep architectures for transfer learning and other tasks. In recent years, feature extraction becomes increasingly important as data grows high dimensional. Autoencoder as a neural network based feature extraction method can learn the structure of data adaptively and represent data efficiently. These properties make autoencoders achieve great success in generating abstract features of high dimensional data. The aim of this project is to focus on the dimensionality reduction ability of different kinds of autoencoders on different datasets. We also used autoencoders to visualize the high dimensional data in three dimensional space. Experiments are conducted on some real datasets, e.g. MNIST dataset and Forest cover type dataset and Wisconsin breast cancer dataset. The results suggest that convolutional autoencoders have lower construction loss on the image dataset, deep autoencoder and sparse autoencoder performs best on forest cover data and stacked autoencoder gives us better results for breast cancer data.
Keywords: Autoencoders, unsupervised learning, dimensionality reduction.

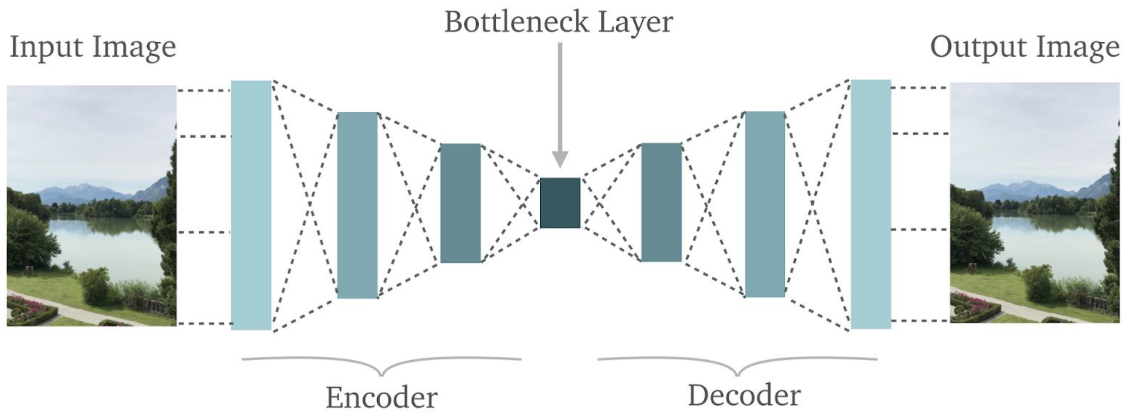
1. Introduction

An autoencoder is a type of artificial neural network that learns how to efficiently compress and encode data then learns how to reconstruct the data back from the reduced encoded representation to a representation that is as close to the original input as possible. The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore noise. Along with the reduction side, a reconstructing side is learnt, where the autoencoder tries to generate from the reduced encoding a representation as close as possible to its original input, hence its name.

1.1 Autoencoder Components:

1.1.1. Encoder : In which the model learns how to reduce the input dimensions and compress the input data into an encoded representation. Encoder can contain multiple hidden layers.

1.1.2 Bottleneck : This layer that contains the compressed representation of the input data. This is the lowest possible dimension of the input data.



1.1.3 Decoder : In which the model learns how to reconstruct the data from the encoded representation to be as close to the original input as possible. Decoder may contain multiple layers.

1.2 Different goals of Autoencoder

1.2.1. Dimensionality Reduction and feature Extraction: The number of input variables or features for a dataset is referred to as its dimensionality. Dimensionality reduction[1][2] refers to techniques that reduce the number of input variables in a dataset. Dimensionality reduction techniques are often used for data visualization. This technique can be used in applied machine learning to simplify a classification or regression dataset in order to better fit a predictive model. In a typical autoencoder, the encoder part contains multiple hidden layers, each of which contains less number of nodes than the previous one, thus reducing the dimension of data in each step. The bottleneck layer contains the lowest number of nodes. For feature extraction we simply collect the data from the bottleneck layer which gives us the required low dimensional data.

1.2.2. Anomaly Detection : Anomaly detection is a process that identifies data points, events, and/or observations that deviate from a dataset's normal behavior. Anomalous data can indicate critical incidents, such as a technical glitch, or potential opportunities, for instance a change in consumer behavior. Auto Encoders are progressively used to automate anomaly detection. E.g. Credit Card fraud detection. Anomaly detection using dimensionality reduction is based on the assumption that data has variables correlated with each other and can be embedded into a lower dimensional subspace in which normal samples and anomalous samples appear significantly different.[3]

1.2.3. Image Denoising : image denoising is a process where the 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. Therefore, image denoising plays an important role in a wide range of applications. To achieve this goal we train the autoencoder with a set of noisy images with respect to their corresponding clean images. When the model is trained, given a noisy image as input we get the clean image as output.[4][5]

1.2.4 Generative Models : Generative models can generate new data instances. A generative model includes the distribution of the data itself, and tells you how likely a given example is. For example, models that predict the next word in a sequence are typically generative models because they can assign a probability to a sequence of words. Variational autoencoder is one of the most popular approaches to learn complicated data distribution such as images using neural networks in an unsupervised fashion. It is a probabilistic graphical model rooted in Bayesian inference i.e., the model aims to learn the underlying probability distribution of the training data so that it could easily sample new data from that learned distribution.

1.3 Types of Autoencoder

Lets review the concept of autoencoders and some of its variants :

1.3.1 Simple Autoencoder : A Simple autoencoder is a one-hidden-layer neural network and its objective is to reconstruct the input using its hidden activations so that the reconstruction error is as small as possible. It takes the input and puts it through an encoding function to get the encoding of the input, and then it decodes the encodings through a decoding function to recover (an approximation of) the original input. More formally, let $x \in \mathbb{R}^d$ be the input, Then the encoded output is: $h = f_e(x) = s_e(W_e x + b_e)$ and the reconstruction of original input is: $x_r = f_d(h) = s_d(W_d h + b_d)$. Where $f_d : R^d \rightarrow R^h$ and $f_e : R^h \rightarrow R^d$ are encoding and decoding functions respectively, W_e and W_d are the weights of the encoding and decoding layers, and b_e and b_d are the biases for the two layers. s_e and s_d are elementwise nonlinear functions in general, and common choices are sigmoidal functions. For training, we want to find a set of parameters $\Theta = \{W_e, W_d, b_e, b_d\}$ that minimize the reconstruction error :

$$\sum_{x \in D} L\left(x, x_r\right)$$

Where $L : R^d \times R^d \rightarrow R$ is a loss function that measures the error between the reconstructed input x_r and the actual input x , and D denotes the training dataset. This model has a significant drawback in that if the number of hidden units h is greater than the dimensionality of the input data d, the model can perform very well during training but fail at test time because it trivially copied the input layer to the hidden one and then copied it back. As a work-around, one can set h < d to force the model to learn something meaningful, but the performance is still not very efficient.[2]6[]

1.3.2. Sparse Autoencoder : In a sparse autoencoder, a sparsity penalty $\Omega(h)$ is added on the code layer h. This involves constructing a loss function $L(x, x_r) + \Omega(h)$, where one term encourages our model to be sensitive to the inputs (ie. reconstruction loss $L(x, x_r)$ and an added regularize/sparsity penalty $\Omega(h)$ discourages memorization /overfitting. Sparse autoencoders may include more (rather than fewer) hidden units than inputs, but only a small number of the hidden units are allowed to be active at once. The penalty $\Omega(h)$ encourages the model to activate (i.e. output value close to 1) some specific areas of the network on the basis of the input data, while forcing all other neurons to be inactive (i.e. to have an output value close to 0).[2][5][6]

1.3.3. Denoising Autoencoder : The idea of denoising autoencoders is to take a partially corrupted input and train the autoencoder to recover the original undistorted input. In this way, the model is forced to learn representations that are useful since trivially copying the input will not optimize this denoising objective.

The training process of a denoising autoencoder works as follows:

1. The initial input x is corrupted into x_c via some corruption process.Formally, $x_c \sim q(x_c \mid x)$, where $q(\cdot \mid x)$ is some corruption process over the input x.
2. The corrupted input x_c then mapped to a hidden representation with the same process of the standard autoencoder,
$$h = f_e(x) = s_e(W_e x_c + b_e)$$
3. From the hidden representation the model reconstructs x_r following the same process as the standard autoencoder.

Denoising the inputs intentionally is particularly helpful in case of an overfitting problem of a model. Denoising autoencoder used for data denoising and image denoising.[1][2][6][9]

1.3.4. Deep Autoencoder : If an autoencoder has multiple hidden layers, they are called Deep autoencoders or stacked autoencoders. In general, an N-layer deep autoencoder with parameters $\Theta = \{\Theta^i \mid i \in \{1, 2, ...N\}\}$, where $\Theta^i = \{W_e^i, W_d^i, b_e^i, b_d^i\}$ can be formulated as follows:

$$h^i = f_e^i(h^{i-1}) = s_e^i(W_e^i h^{i-1} + b_e^i)$$
$$h_r^i = f_d^i(h_r^{i+1}) = s_d^i(W_d^i h_r^{i+1} + b_d^i)$$
$$h^0 = x$$

The deep autoencoder architecture therefore contains multiple encoding and decoding stages made up of a sequence of encoding layers followed by a stack of decoding layers. Therefore the deep autoencoder has a total of 2N layers.[6][7]

1.3.5. Convolutional Autoencoder : The Convolutional Autoencoder is extended from autoencoder by instantiating encoder function and decoder function with convolutional neural networks (CNN). The basic building blocks of CNN are convolutional layers and pooling layers, a convolutional layer consists of multiple convolutional nodes whose inputs are 2-dimensional feature maps, the learning parameters are the elements of filter matrices.A pooling layer is also called sampling layer, the nodes of a pooling layer are obtained by sampling the corresponding nodes of convolutional layer. Consequently, if the number of nodes of the convolutional layer is m, then the number of nodes of the pooling layer is also m. Because a CNN is a feedforward neural network, the CAE can be trained with gradient descent algorithm or stochastic gradient descent algorithm. [4][6]

1.3.6. Stacked Autoencoder : the structure of SAEs is stacking autoencoders into hidden layers by an unsupervised layer-wise learning algorithm and then fine-tuned by a supervised method.[6][8][9][10] So the SAEs based method can be divided into three steps:

1. Train the first autoencoder by input data and obtain the learned feature vector;
2. The feature vector of the former layer is used as the input for the next layer, and this procedure is repeated until the training completes.
3. After all the hidden layers are trained, a backpropagation algorithm (BP) is used to minimize the cost function and update the weights with labeled training set to achieve fine-tuning.

2. Problem Statement

The objective of this project is three fold.

1. Our first objective is to build different types of autoencoders.
2. Our second objective is to compare the dimensionality reduction ability of those different kinds of autoencoders on different datasets.
3. Our third objective is to compare the results with other state-of-the-art dimensionality reduction techniques.

3. Materials and Methods

3.1 Data Acquisition

We have used three different datasets for this project. The first dataset we have collected is the Forest cover type dataset.. This dataset contains tree observations from four areas of the Roosevelt National Forest in Colorado. Second dataset is a database of handwritten digits. This image database contains a training set of 60,000 examples, and a test set of 10,000 examples. Each row consists of 785 values: the first value is the label (a number from 0 to 9) and the remaining 784 values are the pixel values (a number from 0 to 255) and stored as a csv file. Third dataset is Breast Cancer Wisconsin (Diagnostic) Data Set. Here features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. The target label has two distinct variables either malignant or benign. Number of instances, features and the source of these datasets are shown in Table 1.

Table 1 : Datasets

No	Name	Instances	Features	Source
1.	Forest cover type dataset	581012	54	https://www.kaggle.com/uciml/forest-cover-type-dataset
2.	Mnist handwritten digits database	70000	784	https://www.kaggle.com/oddrationale/mnist-in-csv
3.	Breast Cancer Wisconsin (Diagnostic) Dataset	569	30	https://www.kaggle.com/uciml/breast-cancer-wisconsin-data

3.2 Data Preprocessing

3.2.1 Forest cover type dataset

After reading the data file using pandas library, we checked for null values in the dataset, there was none. Then as it's a categorical data we checked the distribution of the data for each class i.e. for each cover type. As shown in Figure 1 the distribution is highly imbalanced. Cover type 1 and cover type 2 has the absolute majority over the other cover types.

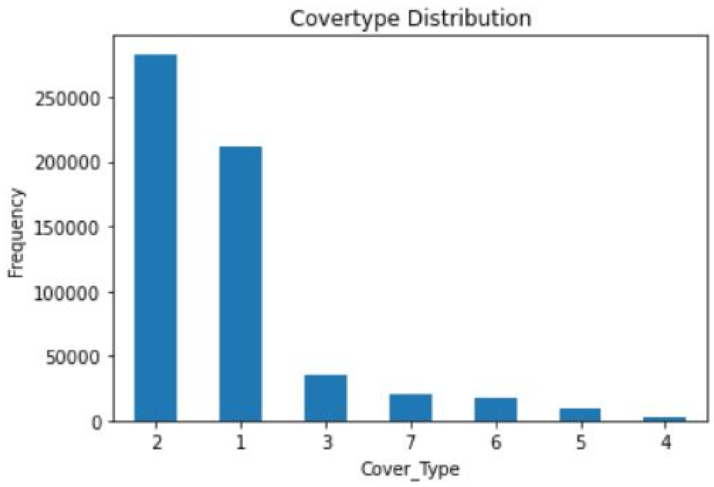


Figure 1 : Covertype distribution

So to handle the imbalanced dataset we used an undersampling method with the help of imbalanced-learn API. After undersampling we have 19229 instances to work on. To train the autoencoders we do not need the target label “cover_type” so we dropped the target label “cover_type” from the train dataset and stored it separately to use later on in classification. Now as the range of values of different features in the data vary widely, we used min-max scaling to normalize the data. Finally the data is divided into three parts for training, validation and testing.

3.2.2 Mnist handwritten dataset

The dataset contains two files, one for training and one for testing. After reading the files using pandas library for both train and test dataset, we extracted the “values” column into a separate variable and then dropped the “values” column to only retain the columns with the pixel values. After that those DataFrames are converted into numpy arrays. Each array contains 784 columns. Since the pixel values varied from 0 to 255, to normalize the arrays between 0 and 1, both the arrays are divided by 255. In Figure 2 we showed the first 10 sample data after normalization. Finally we divided the train data into three parts, training set, validation set, test set.

Figure 2 : Sample images

3.2.3 Breast Cancer diagnostic dataset

After reading the data file using pandas library, we checked for null values in the dataset, there was none. Then as it's a categorical data we checked the distribution of the data for each class. As shown in Figure 3 the distribution of diagnosis is fairly balanced.

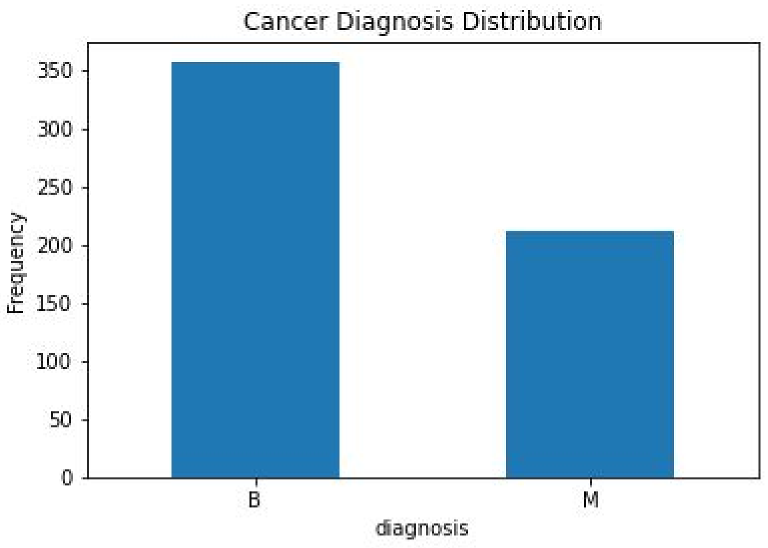


Figure 3 : Breast cancer diagnosis distribution

To train the autoencoders we do not need the target label "diagnosis" so we dropped the target label "diagnosis" from the train dataset and stored it separately to use later on in classification. Now as the range of values of different features in the data vary widely, we used min-max scaling to normalize the data. Finally the data is divided into three parts for training, validation and testing.

3.3 Data Analysis

3.3.1 Simple Autoencoder

Steps:

- First we created an Input layer to receive the input data.
- Then we encoder takes the input data from the input layer and encodes the data in preferred dimensions. Since all the values are normalized between 0 and 1 for the encoder we used 'Relu' as an activation function. This layer is the hidden layer of the simple autoencoder.
- Finally, the decoder takes the encoded data of reduced dimensions and decodes it back to its original dimensions. For the decoder we used 'sigmoid' as an activation function.
- We now create the autoencoder model with input as the input layer and output as the last decoder layer.

- To compile the data we use ‘MSE’ as the loss function and optimizer function based on the dataset.
- Now for training we will use input data as the target output and train the data for a certain amount of epochs, which depends upon the dataset we use.

3.3.1.1 Forest cover type data : Input dimension of our forest cover type data is 54, we will encode it to 3 and then decode it back to the original dimension using the. We compiled the model using ‘Adam’ as the optimizer and ‘MSE’ as the loss function and trained the model for 200 epochs with batch size of 512. We also shuffled the training data during the training process.

3.3.1.2 Mnist handwritten digits data : Our input image has a dimension of 784, we will encode it to 32 and then decode it back to the original dimension of 784. So first we take the input layer with 784 nodes. Then we compile the autoencoder model with adadelta optimizer. We used Binary Cross-Entropy as the loss function. We now train the autoencoder using the training data with 100 epochs and batch size of 256. We also shuffle the training data.

3.3.1.3 Breast cancer data : Input dimension of our forest cover type data is 30, we will encode it to 3 and then decode it back to the original dimension using the .
We compile the model using ‘Adam’ and use ‘MSE’ as the loss function and trained the model for 500 epochs. We also shuffled the training data during the training process.

3.3.2 Sparse Autoencoder

In sparse autoencoder to achieve the sparsity we added a regularization term in the encoder part of the simple autoencoder. Rest of the process is similar to the simple autoencoder model.

3.3.2.1 Forest cover type data : For this data we added L1 regularization of 1e-6 in the encoding layer.

3.3.2.2 Mnist handwritten digits data : For the mnist data we have one extra layer in the encoder and decoder. First encoding layer takes the input data and reduces the dimension to 128. The second layer enodes the 128 dimensional data into 32 dimensions. Here in the second layer we added the L1 regularization of 1e-6. Then decoder first decodes the encoded representation to 128 dimensions and then decoders back to the original input. Rest of the process is the same.

3.3.2.3 Breast cancer data : For breast cancer data we added L1 regularization of 1e-6 in the encoding layer. Rest of the process is the same as the simple autoencoders.

3.3.3 Deep Autoencoder

3.3.3.1 Forest cover type data : For deep autoencoders we used six hidden layers, three encoder layers and 3 decoder layers. The encoded representation has the dimension of 3.
First we Create the Input layer which takes original data with 54 dimensions and converts into an input tensor. This input tensor is the input to the first layer of the Encoder. Output of the first encoder layer is 27 dimensional data and output of the first layer is the input of the second layer. Output of the second encoder layer is 9 dimensional data. Similarly output of the third encoder layer is 3 dimensional data. Which is the latent representation of the Original input data. Each of the encoded layers is a dense layer and uses relu as an activation function. We also used batch normalization layers between each layer for better results. Then the decoder takes the encoded data and creates the reconstruction of the original input via three layers which have output dimensions of 9, 27, 54 respectively. Batch normalization layers also have been used between each decoder layer to normalize the data. Finally we compile the model using the 'MSE' as the loss function and 'Adam' as the optimizer. After that we train the model similarly as a simple autoencoder.

3.3.3.2 Mnist handwritten digits data : First we create the input layer with 784 nodes. From the Input layer Input image will be Encoded to a dimension of 32. In the Encoding layer we used two dense layers. First dense layer has 128 nodes and the second dense layer has 64 nodes. Both of these layers use 'Relu' as an activation function. For the code layer we used a dense layer of dimension 32, which uses Relu as an activation function.For the Decoding layer we used two dense layers. As shown in Figure 10 , the first layer has 64 nodes and the second layer has 128 nodes. Both of these layers used 'Relu' as an activation function. Finally the output layer takes the 128 dimensional data and reconstructs the original input as output. In the output layer we used sigmoid as the activation function.For compiling and training we will use the same steps as the simple autoencoder.

3.3.3.3 Breast cancer data : For this dataset we have used the four encoder layers each of which has the output dimension of 25, 20, 10, 3 respectively and four decoder layers with output dimension of 10, 20, 25, 30 respectively. For the output layer we used sigmoid as the activation function and the relu for the rest of the layers. For model compilation we used Adam as the optimizer and MSE as the loss function. And to train the data we followed the same steps as the simple autoencoder.

3.3.4 Denoising Autoencoder

3.3.4.1 Forest cover type data : In this method we deliberately introduce noise to the train dataset. To create noise we create an array of random numbers with mean of zero and the variance of 0.1 and with the shape of the training data. Then we add the noise to the original data to create the noisy train data. For the model creation and compilation we followed the same steps as the simple autoencoder. Finally when training the model here we use the noisy data as the input and clean data as the target output.

3.3.4.2 Mnist handwritten digits data :

Firstly we added noise to the dataset. To create noise we create an array of random numbers with mean of zero and the variance of 1 and with the shape of the training data. Then we multiplied the array with 0.4. Then we add the noise to the original data to create the noisy train data. Then build the autoencoder model similar to the deep autoencoder model. In the training process, we train the model with noisy images as input and clean images as target output allowing the model to learn essential features of the given data. In Figure 4 we displayed the sample images after adding noise to the Mnist data.

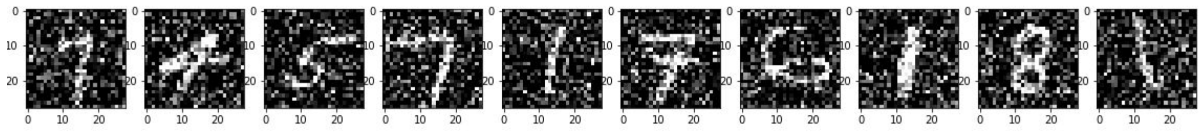


Figure 4 : Denoising Autoencoder noisy images.

3.3.4.3 Breast cancer data :

To introduce noise we create an array of random numbers with mean of zero and the variance of 0.1 and with the shape of the training data. Then we add the noise to the original data to create the noisy train data. For the model creation and compilation we followed the same steps as the deep autoencoder model. Finally when training the model here we use the noisy data as the input and clean data as the target output.

3.3.5 Convolutional Autoencoder

3.3.5.1 Forest cover type data :

- The dimension of the input image is 54. So to use convolutional neural network we reshaped the dimension of the data into 54 * 1.
- For the encoder part we used convolutional layer 1d and max pooling 1d layer. Each convolutional layer has kernel size 3 (or 2 in some occasions) and padding is set to the same as the input dimension. For each convolutional layer there is a max pooling 1d layer to reduce the dimension of the tensor. After each convolutional layer we used a batch normalization layer to normalize the data for better results. For each convolutional layer we used Relu as the activation function.
- Finally the output of the encoder is the latent representation of dimension 3*1 i.e 3.
- Similarly in the decoder part we used convolutional layer 1d and up sampling 1d layer. Each convolutional layer has kernel size 3 and padding is set to the same input dimension. For each convolutional layer there is an up sampling 1d layer to upscale the dimension of the tensor. For each convolutional layer we used Relu as the activation function.
- Lastly we compiled the model using adam as the optimizer and MSE as the loss function.

3.3.5.2 Mnist handwritten digits data :

- The dimension of the input image is 784. So to use convolutional neural network we reshaped the dimension of the data into 28 * 28 * 1.
- For the encoder part we used convolutional layer 2d and max pooling 2d layer. Each convolutional layer has kernel size 3*3 and padding is set to the same input dimension. For each convolutional layer there is a max pooling 2d layer to reduce the dimension of the tensor. For each convolutional layer we used Relu as the activation function.
- Finally the output of the encoder is the latent representation of dimension 4*4*8 i.e 128.
- Similarly in the decoder part we used convolutional layer 2d and up sampling 2d layer. Each convolutional layer has kernel size 3*3 and padding is set to the same input dimension. For each convolutional layer there is an up sampling 2d layer to upscale the dimension of the tensor. For each convolutional layer we used Relu as the activation function. In figure 14 the structure is given.
- Lastly we compiled the model using adam as the optimizer and MSE as the loss function.

3.3.5.3 Breast cancer data :

- The dimension of the input image is 30. So to use convolutional neural network we reshaped the dimension of the data into 30 * 1.
- For the encoder part we used convolutional layer 1d and max pooling 1d layer. Each convolutional layer has kernel size 3 and padding is set to the same as input dimension. For each convolutional layer there is an average pooling layer 1d layer to reduce the dimension of the tensor. After each convolutional layer we used a batch normalization layer to normalize the data for better results. For each convolutional layer we used Relu as the activation function.
- Finally the output of the encoder is the latent representation of dimension 3*1 i.e 3.
- Similarly in the decoder part we used convolutional layer 1d and up sampling 1d layer. Each convolutional layer has kernel size 3 and padding is set to the same input dimension. For each convolutional layer there is an up sampling 1d layer to upscale the dimension of the tensor. For each convolutional layer we used Relu as the activation function.
- Lastly we compiled the model using adam as the optimizer and MSE as the loss function.

3.3.6 Stacked Autoencoder

Stacked autoencoder model is very similar to the deep autoencoder model, the difference is that in stacked autoencoders first we train multiple simple autoencoders then import those pretrained weights into the final deep model.

3.3.6.1 Forest cover type data : We used three simple autoencoders for pre training of the layers. First autoencoder has the model structure of <54,27,54>, which means the input layer accepts 54 dimensional data and passes the input tensor to the encoder which encodes the 54 dimensional data to 27 dimensions. Then the decoder takes the encoded data to reconstruct the 54 dimensional data. In the encoded layer of the first autoencoder we used relu as an activation function and for the decoder layer we used sigmoid as the activation function. Similarly the second autoencoder has the model structure of <27,9,27> and here the input data is the encoded data from the first autoencoder model. The third autoencoder model has the structure of <9,3,9> and takes the encoded data of the second autoencoder model as the input. In the second and third autoencoder we used relu as the activation function in each layer. Each autoencoder model has been compiled using MSE as the loss function and Adam as the activation function. Now we train each of the autoencoders one by one then import the layers to the final deep model which has the model structure of <54,27,9,3,9,27,54>.

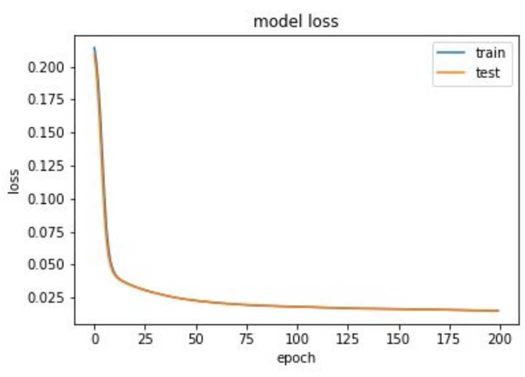
3.3.6.2 Mnist handwritten digits data : Similar to the previous dataset we used three simple autoencoders for pre training of the layers. First autoencoder has the model structure of <784,128,784>. The second autoencoder has the model structure of <128,64,128> and here the input data is the encoded data from the first autoencoder model. The third autoencoder model has the structure of <64,32,64> and takes the encoded data of the second autoencoder model as the input. Each of these autoencoder uses Relu as the activation function in the encoder layer and Sigmoid as the activation function in the decoder layer Now we train each of the autoencoders using Binary Cross-Entropy as the loss function and Adadelta as the optimizer one by one then import the layers to the final deep model which has the model structure of <784,128,64,32,64,128,784>.

3.3.6.3 Breast cancer data : Here we also used three simple autoencoders for pre training of the layers. First autoencoder has the model structure of <30,20,30>, which means the input layer accepts 30 dimensional data and passes the input tensor to the encoder which encodes the 30 dimensional data to 20 dimensions. Then the decoder takes the encoded data to reconstruct the 30 dimensional data. Similarly the second autoencoder has the model structure of <20,10,20> and here the input data is the encoded data from the first autoencoder model. The third autoencoder model has the structure of <10,3,10> and takes the encoded data of the second autoencoder model as the input. Each autoencoder model has been compiled using MSE as the loss function and Adam as the activation function. Now we train each of the autoencoders one by one then import the layers to the final deep model which has the model structure of <30,20,10,3,10,20,30>.

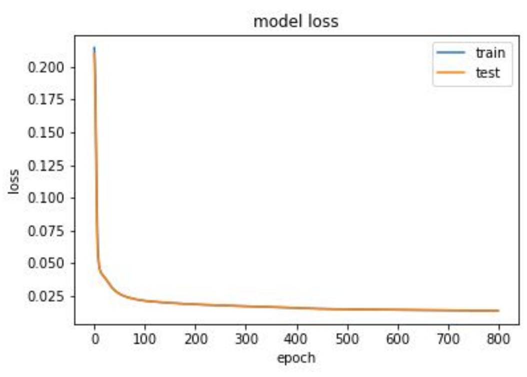
4. Results and Discussion

4.1 Forest covertype Dataset

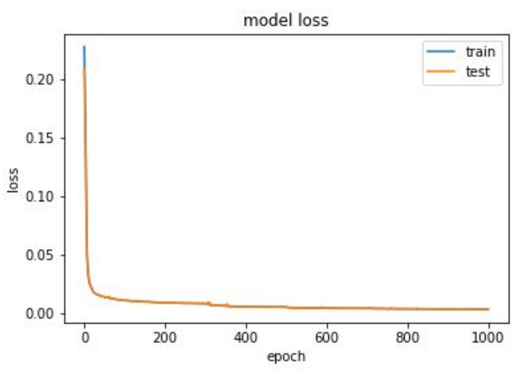
The training loss and the validation loss tells us how much the reconstructed output from the autoencoder model varied from the original input. We used MSE as the loss function for this dataset. In Figure 5 For each autoencoder model we plotted the training loss and the validation loss.



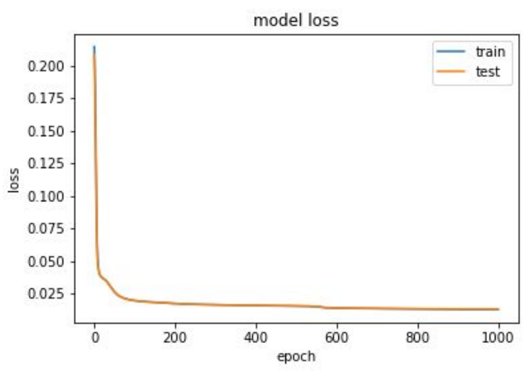
Simple AE



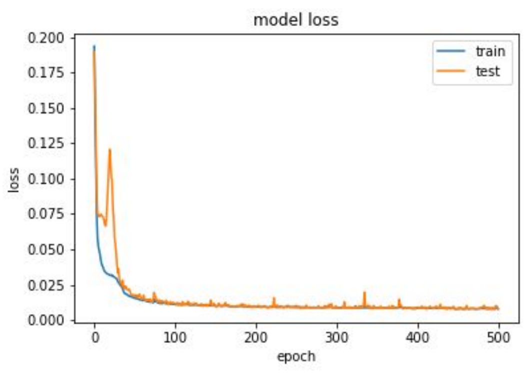
Sparse AE



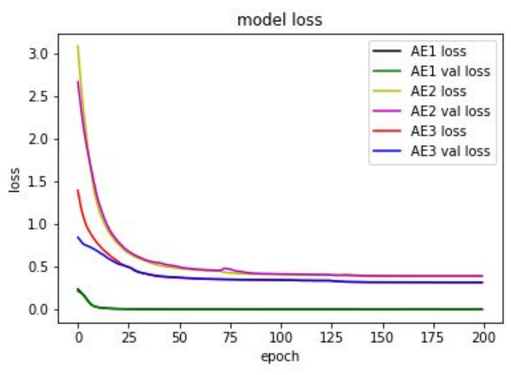
Deep AE



Denoising AE



Convolutional AE



Stacked AE

Figure 5 : Comparison between model loss of the Forest cover data.

From Figure 5 we can see that all the models are fitting quite perfectly. The mse value of the traint and validation is close to 0. After the training, we took the encoded representation of the test set to visualize the data in 3 dimensions and the results are shown in Figure 6.

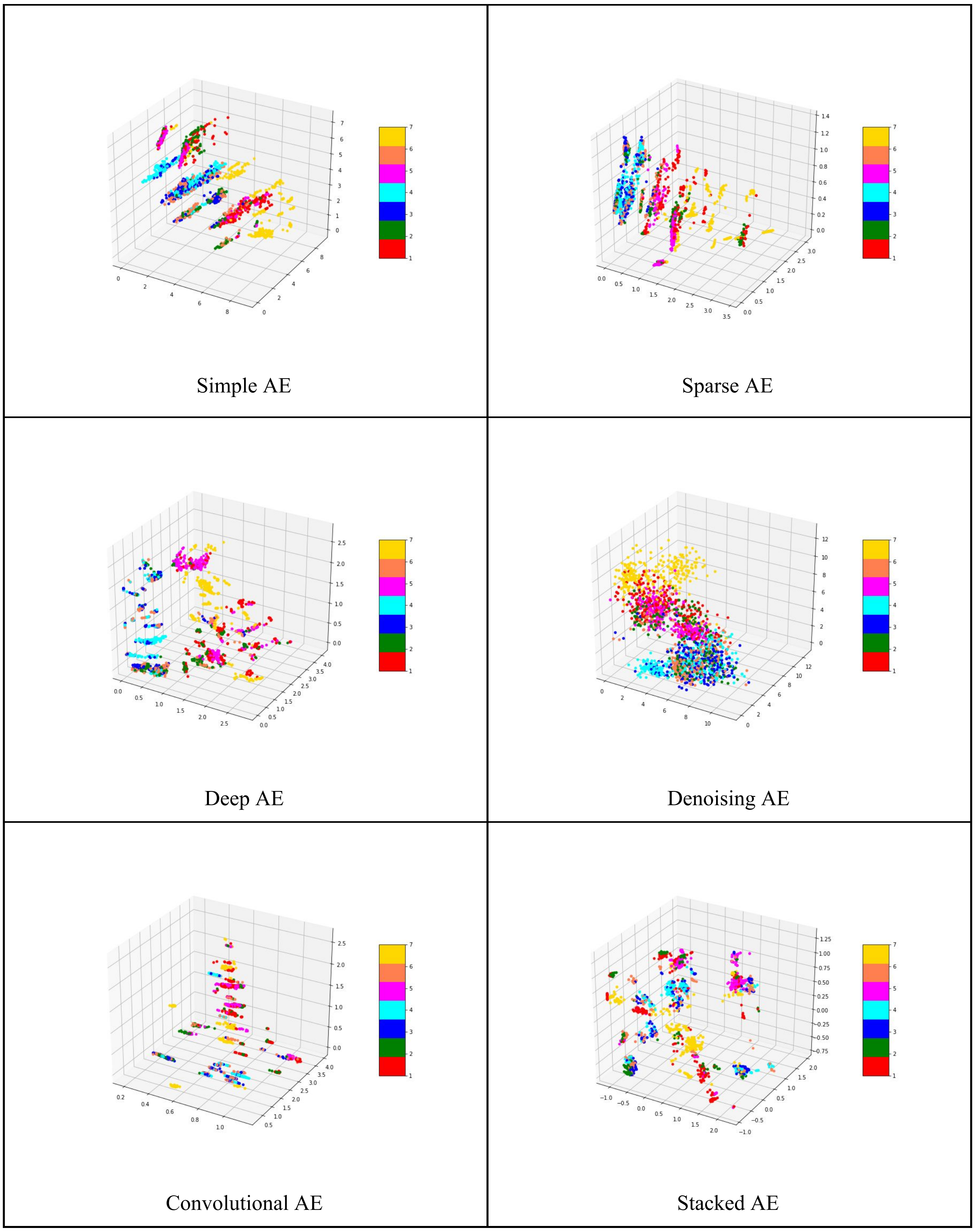


Figure 6 : Visualization of the encoded representation of the Forest cover data.

Next we feed the encoded representation of the test dataset into the KNN model to evaluate the classification accuracy of the encoded representation of the autoencoder models. Now the classification accuracy evaluates the dimensionality reduction performance of each auto encoder model i.e. we can see how effectively the autoencoder can retain information in encoded representation. Here's our result:

Table 2: Comparison between model loss and KNN score of the Forest cover data.

	No. of features (Original)	No. of features (Reduced)	Training Loss	Validation Loss	Test Loss	KNN Score
Simple AE	54	3	0.150	0.0150	0.0147	0.7779
Sparse AE	54	3	0.0133	0.0133	0.0120	0.7847
Deep AE	54	3	0.0030	0.0028	0.0028	0.8013
Convolutional AE	54	3	0.0079	0.0078	0.0077	0.7108
Denoising AE	54	3	0.0131	0.0134	0.0130	0.5605
Stacked AE	54	3	8.0239e-04 0.1104 0.8801	0.0010 0.1131 0.9174	0.02917	0.8169
PCA	54	3	-	-	-	0.8263

From Table 2 we can see that the deep Autoencoder has the lowest loss value throughout the training and the testing. The test loss of the stacked Autoencoder is greater than the other models but from the KNN score we can see that the stacked autoencoder model performs better than the other models. For comparison we also applied PCA transformation to the data and computed the KNN score which gives us slightly better results from what we got with the stacked autoencoder.

4.2 Mnist handwritten digits database dataset

After The training is complete for each autoencoder model we plotted the training loss and the validation loss shown in figure 7. We used Binary Cross-Entropy as the loss function.

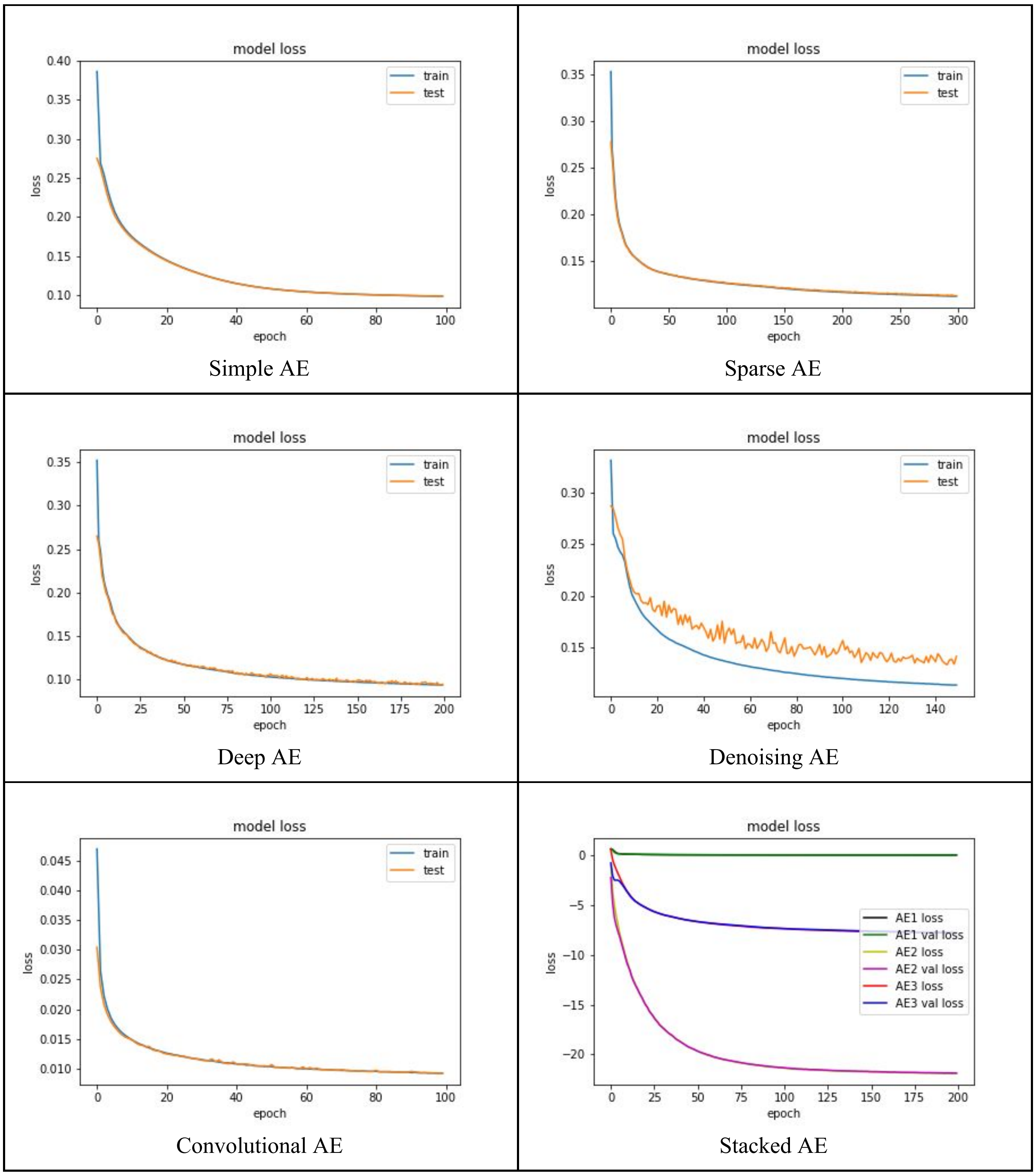


Figure 7: Comparison between model loss of the Mnist data.

From figure 7 we can see that for each model loss is decreased significantly over time. Also the difference between test loss and validation loss is very less which indicates that models are neither overfitting nor underfitting.

After the completion of the training we predicted the test set and plotted the reconstructed outputs for each model in the Figure 8. Here's the comparison result:

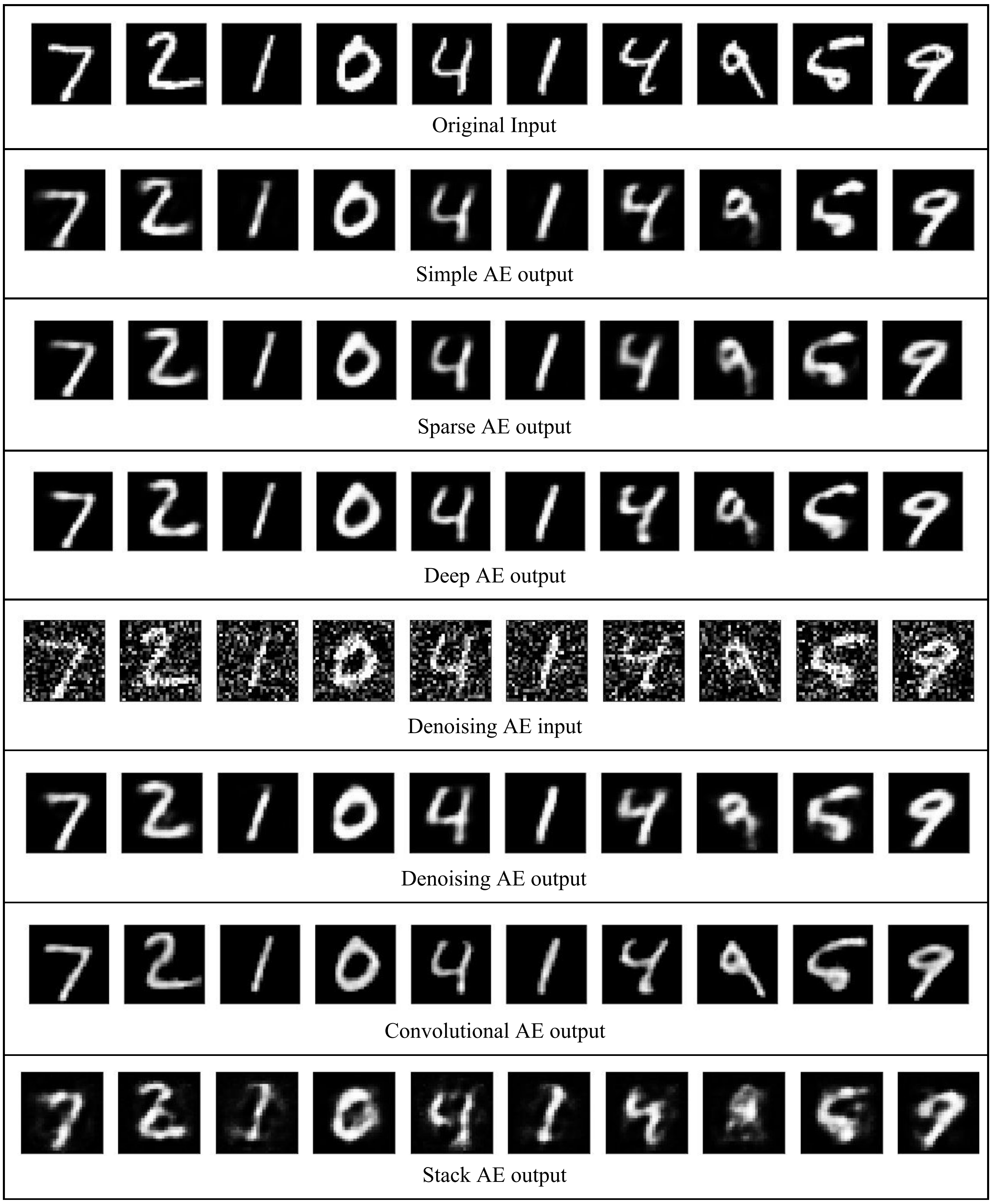


Figure 8: Comparison between reconstructed output with the original input

From figure 8 we can say that convolutional autoencoders have done a better job than the other models.

In Table 3 we compared the Binary cross-entropy losses of the training, validation and test loss side by side. The Training loss and the validation loss. For the stacked autoencoder as we train the three autoencoders, we have three different training losses and validation losses and we showed the losses in the table respectively in a single row. From the comparison we can see that the convolutional autoencoder model loss is lower than the other models throughout the training, validation and the test loss. But from the KNN score we can tell that Deep autoencoder gives us the better result. For comparison we also applied PCA transformation to the data and computed the KNN score which gives us almost the same results from what we got with the deep autoencoder.

Table 3: Comparison between model loss and the knn score of the Mnist data.

	No. of Features (Original)	No. of Features (Reduced)	Training Loss	Validation Loss	Test Loss	KNN Score
Simple AE	784	32	0.0984	0.0985	0.0968	0.9678
Sparse AE	784	32	0.1126	0.1131	0.1066	0.9615
Deep AE	784	32	0.0939	0.0948	0.0934	0.9725
Convolutional AE	784	128	0.0092	0.0092	0.0089	0.952
Denoising AE	784	32	0.1137	0.1413	0.1169	0.9658
Stacked AE	784	32	0.0702 -21.9300 -7.7876	0.0708 -21.9021 -7.7876	0.1514	0.9363
PCA	784	32	-	-	-	0.9723

4.3 Breast Cancer diagnostic data

After training the autoencoders using the training data with validation data, when the autoencoder seems to reach a stable training loss value and the validation loss we collected the result. The training loss and the validation loss tells us how much the reconstructed output from the autoencoder model varied from the original input. In Figure 5 For each autoencoder model we plotted the training loss and the validation loss and we used MSE as the loss function for this dataset.

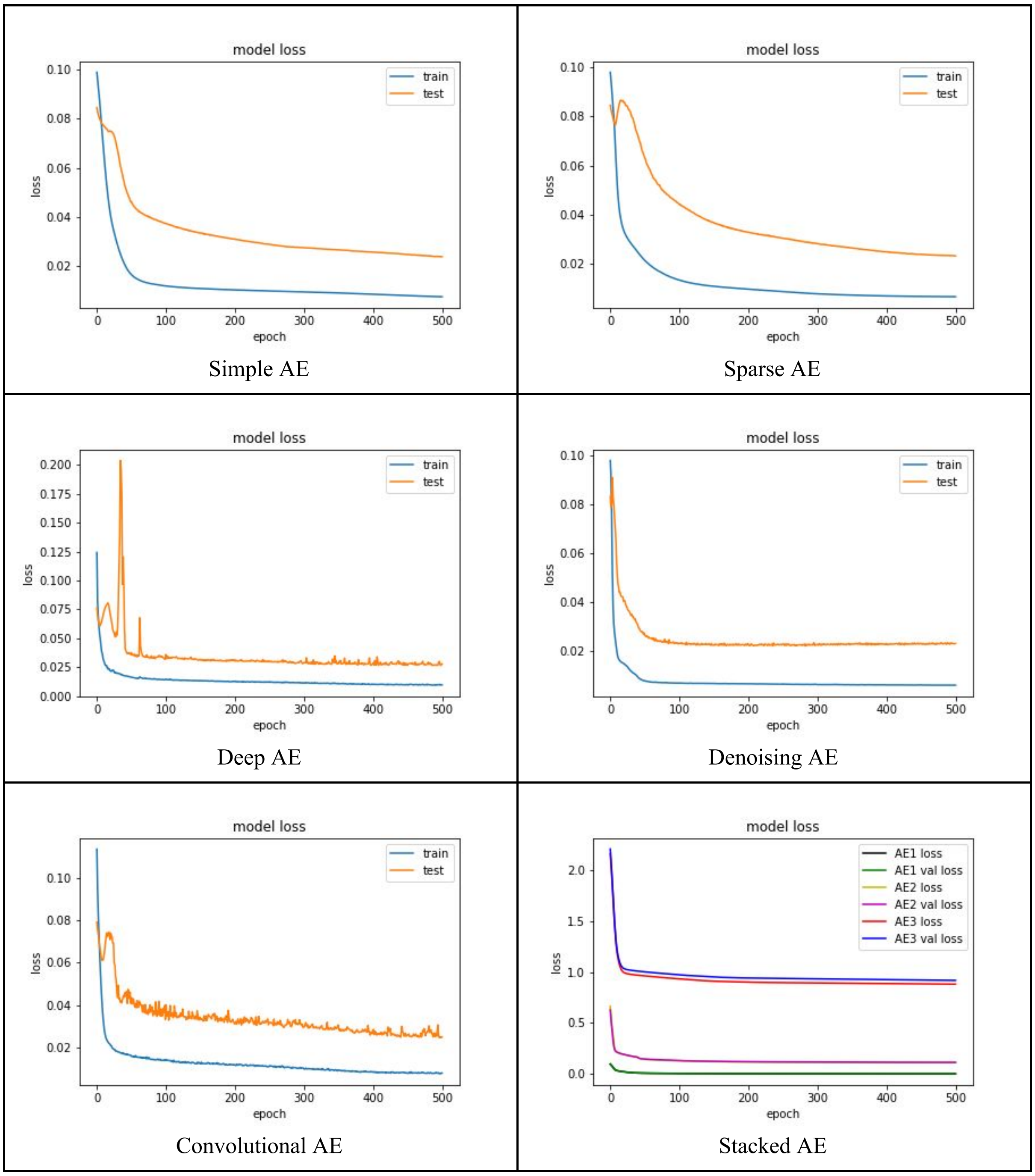
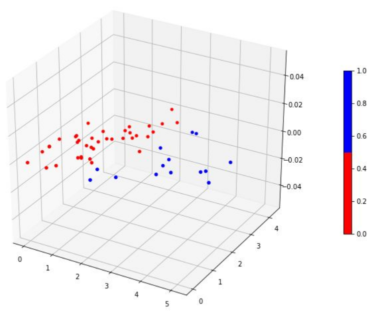
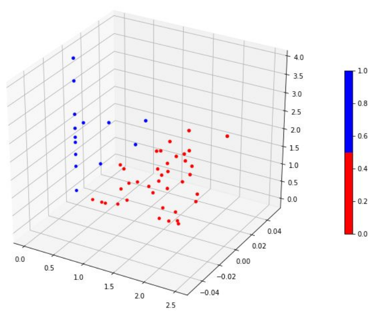


Figure 9: Comparison between model loss of the Breast cancer data.

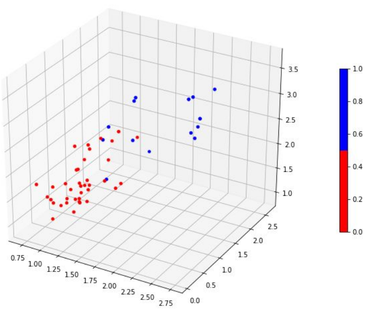
After the training, we took the encoded representation of the test set to visualize the data in 3 dimensions shown in the Figure 10.



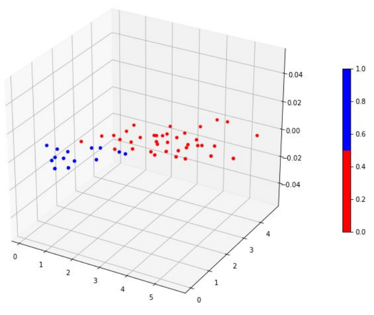
Simple AE



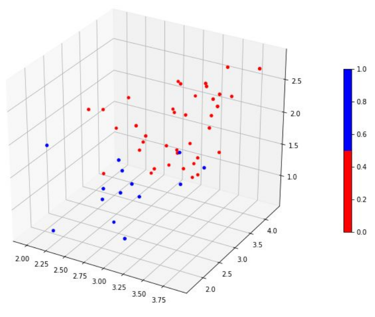
Sparse AE



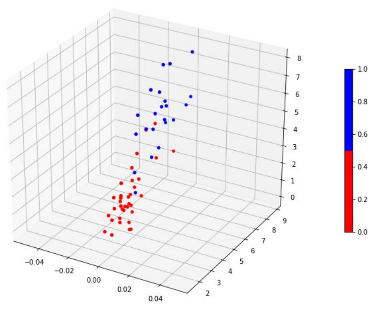
Deep AE



Denoising AE



Convolutional AE



Stacked AE

Figure 10: visualization of the Forest cover data.

Next we feed the encoded representation of the test dataset into the KNN model to evaluate the dimensionality reduction performance of each auto encoder model. In Table 4. We compared the training, validation and test losses and the KNN score of each model. For the stacked autoencoder as we train the three autoencoders, we have three different training losses and validation losses and we showed the losses in the table respectively in a single row.

Table 4: Comparison between model loss and the KNN score of the breast cancer data.

	No. of Features (Original)	No. of Features (Reduced)	Training Loss	Validation Loss	Test Loss	KNN Score
Simple AE	30	3	0.0075	0.0238	0.0289	0.7115
Sparse AE	30	3	0.0068	0.0233	0.0287	0.8846
Deep AE	30	3	0.0053	0.0209	0.0258	0.8846
Convolutional AE	30	3	0.0078	0.0249	0.0261	0.8461
Denoising AE	30	3	0.0058	0.0229	0.0295	0.8653
Stacked AE	30	3	8.0239e-04 0.1104 0.8801	0.0010 0.1131 0.9174	0.0812	0.8596
PCA	30	3	-	-	-	0.9807

From the comparison shown in Table 4 we can see that the Sparse autoencoder and the deep autoencoder have the minimum losses throughout the training and testing. And their respective KNN score is also better than the other models. For comparison we also applied PCA transformation to the data and computed the KNN score which gives us much better results from what we got with the stacked autoencoder. The reason behind the low KNN accuracy for autoencoders is that there are a small number of instances (only 569).

5. Conclusion

In this project, we have used simple, deep, Sparse, denoising, convolutional and stacked autoencoders to perform unsupervised feature learning. These models can extract features from high dimensional dataset, which greatly reduce data dimension and achieve fairly average classification score and meaningful visualization can be done in three dimensions. In our experiments, I have used three public datasets (one is the forest cover type, second one is MNIST handwritten digit dataset and the other is the Wisconsin breast cancer diagnostic dataset) to learn features and classify. From the model loss and classification results, we could see that using Deep AE and Stack AE performs better in feature reduction than other models for forest cover type data, Convolutional AE performs better in feature reduction than other models for Mnist data and Deep AE and Sparse AE performs better in feature reduction than other models for breast cancer diagnostic data. Also from the results compared with the PCA transformation we can conclude that for the Wisconsin breast cancer data PCA transformation works far better than autoencoders due to the fact that there are a small number of instances in the dataset. For the other two datasets the results are almost the same.

6. Future Work

Here are few things that can be done in future:

- Finding out the minimum number of feature dimensions for each dataset so that the best possible classification accuracy can be achieved or mean squared error loss is the lowest.
- Performing other state-of-the-art dimension reduction techniques such as T-SNE, UMAP, PHATE to compare the result.
- We know that the weight initialization of the network can give different results. Therefore, discovering how to pre-train the initial weight effectively so that the performance of the models increases.
- Here I have applied only one classification method but in future more methods can be applied.

References

[1] T. Wen and Z. Zhang, "Deep Convolutional Neural Network and Autoencoders-Based Unsupervised Feature Learning of EEG Signals," in IEEE Access, vol. 6, pp. 25399-25410, 2018, doi: 10.1109/ACCESS.2018.2833746.
[2] Wang, Y., Yao, H., Zhao, S., & Zheng, Y. (2015). Dimensionality reduction strategy based on auto-encoder. ICIMCS '15.
[3] Sakurada, Mayu and Takehisa Yairi. “Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction.” MLS DA14 (2014).
[4] Majeed, Sajid & Mansoor, Yusra & Qabil, Sana & Majeed, Farooq & Khan, Behraj. (2020). Comparative analysis of the denoising effect of unstructured vs. convolutional autoencoders. 1-5. 10.1109/ICETST49965.2020.9080731.
[5] Lingheng Meng, Shifei Ding1, Yu Xue. “Research on denoising sparse autoencoder”. DOI: 10.1007/s13042-016-0550-y.
[6] J. Zhai, S. Zhang, J. Chen and Q. He, "Autoencoder and Its Various Variants," 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Miyazaki, Japan, 2018, pp. 415-419, doi: 10.1109/SMC.2018.00080.
[7]Yingbo Zhou, Devansh Arpit, Ifeoma Nwogu, Venu Govindaraju: “Is Joint Training Better for Deep Auto-Encoders?” in arXiv:1405.1380v4 [stat.ML] 15 Jun 2015
[8] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio and P.A. Manzagol, Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion, *Journal of Machine Learning Research,* 11:3371--3408, 2010.
[9] Guifang Liu, Huaiqian Bao, and Baokun Han. (2018), A Stacked Autoencoder-Based Deep Neural Network for Achieving Gearbox Fault Diagnosis, https://doi.org/10.1155/2018/5105709
[10] Shivappriya, Sn & Rajaguru, Harikumar. (2019). Performance Analysis of Deep Neural Network and Stacked Autoencoder for Image Classification: Intelligence and Sustainable Computing. 10.1007/978-3-030-02674-5_1.
[11] Almotiri, Jasem & Elleithy, Khaled & Elleithy, Abdelrahman. (2017). Comparison of autoencoder and Principal Component Analysis followed by neural network for e-learning using handwritten recognition. 1-5. 10.1109/LISAT.2017.8001963.
[12] Meng, Qinxue, Daniel Catchpoole, David Skillicom and Paul J. Kennedy. “Relational autoencoder for feature extraction.” 2017 International Joint Conference on Neural Networks (IJCNN) (2017): 364-371.

Appendix

Dataset 1. Forest Cover Type Dataset

```
#Importing Libraries
import tensorflow as tf
import numpy as np
import pandas as pd
from tensorflow import keras
from keras.layers import Conv1D, MaxPooling1D, UpSampling1D, Dense, Dropout, Flatten,Input,
BatchNormalization
from keras.models import Sequential, Model
from keras import regularizers
from keras.callbacks import History
history = History()
import matplotlib.pyplot as plt
from keras.layers import Dense,Activation,Layer,Lambda
import sklearn
from sklearn import preprocessing
from sklearn.model_selection import train_test_split


#load dataset
dataset=pd.read_csv("covtype.csv")
dataset.head()
Dataset.shape

#check for null values
dataset.isnull().sum()

# Data Preprocessing
#Checking the distribution of different class
count_classes = pd.value_counts(dataset['Cover_Type'], sort = True)
count_classes.plot(kind = 'bar', rot=0)
plt.title("Covertype Distribution")
plt.xlabel("Cover_Type")
plt.ylabel("Frequency")
plt.savefig("Covertype Distribution")
plt.show()

y = dataset.iloc[:,-1].values
x = dataset.drop(["Cover_Type"],axis=1)
x.shape,y.shape

cov1 = dataset[dataset['Cover_Type']==1]
```

```
cov2 = dataset[dataset['Cover_Type']==2]
cov3 = dataset[dataset['Cover_Type']==3]
cov4 = dataset[dataset['Cover_Type']==4]
cov5 = dataset[dataset['Cover_Type']==5]
cov6 = dataset[dataset['Cover_Type']==6]
cov7 = dataset[dataset['Cover_Type']==7]
print(cov1.shape,cov2.shape,cov3.shape,cov4.shape,cov5.shape,cov6.shape,cov7.shape)

# Implementing Undersampling for Handling Imbalanced
from imblearn.under_sampling import NearMiss
nm = NearMiss()
X_res,y_res=nm.fit_sample(x,y)
X_res.to_numpy()
X_res.shape

#Feature Scaling
min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(X_res)

#splitting the dataset into train and test set

x_train,x_test,y_train,y_test= train_test_split(x_scaled,y_res,test_size = 0.1, random_state = 0)
x_train,x_val,y_train,y_val= train_test_split(x_train,y_train,test_size = 0.2, random_state = 0)

print(x_train.shape)
print(x_test.shape)
print(x_val.shape)

#Autoencoder Model
input_data = Input(shape=(54,))
encoded = Dense(3, activation='relu')(input_data)
decoded = Dense(54, activation='sigmoid')(encoded)

autoencoder = Model(input_data, decoded)
autoencoder.compile(loss="mse",optimizer='Adam')
autoencoder.summary()

#Train the model
history = autoencoder.fit(x_train, x_train,
            epochs=200,
            batch_size=512,
            shuffle=True,
            validation_data=(x_val, x_val))

evaluation = autoencoder.evaluate(x_test, x_test)
print("Loss:",evaluation)
```

```
# Plot training loss and validation split loss over the epochs
#print(history.history.keys())
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('simpleAE_model_loss.png')
plt.show()

evaluation = autoencoder.evaluate(x_test, x_test)
print("Loss:",evaluation)
data = autoencoder.predict(x_test)

encoder = Model(input_data, encoded)
latent_vector_train = encoder.predict(x_train)
latent_vector = encoder.predict(x_test)
latent_vector.shape
latent_vector

#Data Visualization
import matplotlib
%matplotlib inline
from mpl_toolkits import mplot3d
colors=['red','green','blue','cyan','magenta','coral','gold']
plt.figure(figsize =(12, 9))
ax = plt.axes(projection='3d')
surf = ax.scatter3D(latent_vector[:, 0], latent_vector[:, 1], latent_vector[:, 2], c=y_test,
cmap=matplotlib.colors.ListedColormap(colors))
# Add a color bar which maps values to colors.
cb = plt.colorbar(surf, shrink=0.5, aspect=5)
plt.savefig('simpleAE_3d.png')
plt.show()


#Classification
from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(latent_vector_train, y_train)
    k = knn.score(latent_vector,y_test)
    knnscores.append(k)
```

```
s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(latent_vector)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Sparse Autoencoder

#Autoencoder Model

```
input_data = Input(shape=(54,))
encoded = Dense(3, activation='relu',activity_regularizer = regularizers.l1(1e-6))(input_data)
decoded = Dense(54, activation='sigmoid')(encoded)

autoencoder = Model(input_data, decoded)
autoencoder.compile(loss="mse",optimizer='Adam')
autoencoder.summary()
```

#Model Training

```
history = autoencoder.fit(x_train, x_train,
            epochs=800,
            batch_size=512,
            shuffle=True,
            validation_data=(x_val, x_val))
```

Denoising Autoencoder

#Adding noise the dataset

```
mu, sigma = 0, 0.1
x_train_noisy = x_train + np.random.normal(mu, sigma, x_train.shape)
x_test_noisy = x_test + np.random.normal(mu, sigma, x_test.shape)
x_val_noisy = x_val + np.random.normal(mu, sigma, x_val.shape)
```

#Autoencoder Model

```
input_data = Input(shape=(54,))
encoded = Dense(3, activation='relu')(input_data)
decoded = Dense(54, activation='sigmoid')(encoded)

autoencoder = Model(input_data, decoded)
autoencoder.compile(loss="mse",optimizer='Adam')
autoencoder.summary()
```

#Model Training

```
history = autoencoder.fit(x_train_noisy, x_train,
            epochs=1000,
            batch_size=512,
            shuffle=True,
            validation_data=(x_val_noisy, x_val))
```

Deep Autoencoder

#Autoencoder Model

```
input_data = Input(shape=(54,))
encoded = Dense(27, activation='relu')(input_data)
encoded = BatchNormalization()(encoded)
encoded = Dense(9, activation='relu')(encoded)
encoded = BatchNormalization()(encoded)
encoded = Dense(3, activation='relu')(encoded)

decoded = Dense(9, activation='relu')(encoded)
decoded = BatchNormalization()(decoded)
decoded = Dense(27, activation='relu')(decoded)
decoded = BatchNormalization()(decoded)
decoded = Dense(54, activation='sigmoid')(decoded)

autoencoder = Model(input_data, decoded)
autoencoder.compile(loss="mse",optimizer='Adam')
autoencoder.summary()
```

#Model Training

```
history = autoencoder.fit(x_train, x_train,
            epochs=1000,
            batch_size=512,
            shuffle=True,
            validation_data=(x_val, x_val))
```

Stacked Autoencoder

#Autoencoder Model

```
input_data = Input(shape=(54,))#0
encoded = Dense(27, activation='relu')(input_data)#1
encoded = BatchNormalization()(encoded)#2
encoded = Dense(9, activation='relu')(encoded)#3
encoded = BatchNormalization()(encoded)#4
encoded = Dense(3, activation='relu')(encoded)#5
encoded = BatchNormalization()(encoded)#6
```

```
decoded = Dense(9, activation='relu')(encoded)#7
decoded = BatchNormalization()(decoded)#8
decoded = Dense(27, activation='relu')(decoded)#9
decoded = BatchNormalization()(decoded)#10
decoded = Dense(54, activation='sigmoid')(decoded)#11

stackautoencoder = Model(input_data, decoded)
stackautoencoder.compile(loss="mse",optimizer='Adam')
stackautoencoder.summary()

#Autoencoder1
input_data1 = Input(shape=(54,))#0
encoded1 = Dense(27, activation='relu')(input_data1)#1
encoded1 = BatchNormalization()(encoded1)#2
decoded1 = Dense(54, activation='sigmoid')(encoded1)#11

autoencoder1 = Model(input_data1, decoded1)
encoder1 = Model(input_data1, encoded1)

#Autoencoder2
input_data2 = Input(shape=(27,))
encoded2 = Dense(9, activation='relu')(input_data2)#3
encoded2 = BatchNormalization()(encoded2)#4
decoded2 = Dense(27, activation='relu')(encoded2)#9
decoded2 = BatchNormalization()(decoded2)#10

autoencoder2 = Model(input_data2, decoded2)
encoder2 = Model(input_data2, encoded2)

#Autoencoder3
input_data3 = Input(shape=(9,))
encoded3 = Dense(3, activation='relu')(input_data3)#5
encoded3 = BatchNormalization()(encoded3)#6
decoded3 = Dense(9, activation='relu')(encoded3)#7
decoded3 = BatchNormalization()(decoded3)#8

autoencoder3 = Model(input_data3, decoded3)
encoder3 = Model(input_data3, encoded3)

autoencoder1.compile(loss="mse",optimizer='Adam')
autoencoder2.compile(loss="mse",optimizer='Adam')
autoencoder3.compile(loss="mse",optimizer='Adam')

encoder1.compile(loss="mse",optimizer='Adam')
encoder2.compile(loss="mse",optimizer='Adam')
```

```
encoder3.compile(loss="mse",optimizer='Adam')

history1 = autoencoder1.fit(x_train, x_train,epochs=200,batch_size=512,shuffle=True,validation_split
= 0.30)
```

# AE1: lot training loss and validation split loss over the epochs

```
plt.plot(history1.history['loss'])
plt.plot(history1.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('stackAE1_model_loss.png')
plt.show()

first_layer_code = encoder1.predict(x_train)
print(first_layer_code.shape)

history2 = autoencoder2.fit(first_layer_code,
first_layer_code,epochs=200,batch_size=512,shuffle=True, validation_split = 0.30)
```

# AE2: lot training loss and validation split loss over the epochs

```
plt.plot(history2.history['loss'])
plt.plot(history2.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('stackAE2_model_loss.png')
plt.show()

second_layer_code = encoder2.predict(first_layer_code)
print(first_layer_code.shape)

history3 = autoencoder3.fit(second_layer_code,
second_layer_code,epochs=200,batch_size=512,shuffle=True,validation_split = 0.30)
```

# AE3: lot training loss and validation split loss over the epochs

```
plt.plot(history3.history['loss'])
plt.plot(history3.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('stackAE3_model_loss.png')
plt.show()
```

# AE: lot training loss and validation split loss over the epochs

```
plt.plot(history1.history['loss'],'k',label='AE1 loss')
plt.plot(history1.history['val_loss'],'g',label='AE1 val loss')
plt.plot(history2.history['loss'],'y',label='AE2 loss')
plt.plot(history2.history['val_loss'],'m',label='AE2 val loss')
plt.plot(history3.history['loss'],'r',label='AE3 loss')
plt.plot(history3.history['val_loss'],'b',label='AE3 val loss')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(loc='upper right')
plt.savefig('stackAEwhole_model_loss.png')
plt.show()
```

#Setting weights to the final model.

```
stackautoencoder.layers[1].set_weights(autoencoder1.layers[1].get_weights())
stackautoencoder.layers[2].set_weights(autoencoder1.layers[2].get_weights())
stackautoencoder.layers[3].set_weights(autoencoder2.layers[1].get_weights())
stackautoencoder.layers[4].set_weights(autoencoder2.layers[2].get_weights())
stackautoencoder.layers[5].set_weights(autoencoder3.layers[1].get_weights())
stackautoencoder.layers[6].set_weights(autoencoder3.layers[2].get_weights())
stackautoencoder.layers[7].set_weights(autoencoder3.layers[3].get_weights())
stackautoencoder.layers[8].set_weights(autoencoder3.layers[4].get_weights())
stackautoencoder.layers[9].set_weights(autoencoder2.layers[3].get_weights())
stackautoencoder.layers[10].set_weights(autoencoder2.layers[4].get_weights())
stackautoencoder.layers[11].set_weights(autoencoder1.layers[3].get_weights())
```

Convolutional Autoencoder

#Reshaping for the convolutional layer

```
x_train = x_train.reshape(13844, 54,1)
x_test = x_test.reshape(1923, 54,1)
x_val = x_val.reshape(3462, 54,1)
```

#AutoEncoder Model

```
input_data = Input(shape=(54,1))
#encoder
encoded = Conv1D(16, 3, activation="relu")(input_data)
encoded = BatchNormalization()(encoded)
encoded = MaxPooling1D(2, padding="same")(encoded)

encoded = Conv1D(8, 3, activation="relu")(encoded)
encoded = BatchNormalization()(encoded)
encoded = MaxPooling1D(2, padding="same")(encoded)

encoded = Conv1D(4, 3, activation="relu", padding="same")(encoded)
encoded = BatchNormalization()(encoded)
encoded = MaxPooling1D(2, padding="same")(encoded)

# 3 dimensions in the encoded layer
encoded = Conv1D(1, 3, activation="relu", padding="same")(encoded)
encoded = MaxPooling1D(2, padding="same")(encoded)

#decoder
decoded = UpSampling1D(3)(encoded)
decoded = Conv1D(4, 3, activation='relu', padding="same")(decoded)
decoded = BatchNormalization()(decoded)

decoded = UpSampling1D(3)(decoded)
decoded = Conv1D(8, 3, activation='relu', padding="same")(decoded)
decoded = BatchNormalization()(decoded)

decoded = UpSampling1D(2)(decoded)
decoded = Conv1D(16, 3, activation='relu', padding="same")(decoded)
decoded = BatchNormalization()(decoded)

decoded = Conv1D(1, 3, activation='sigmoid', padding='same')(decoded)

encoder = Model(input_data, encoded)

autoencoder = Model(input_data, decoded)
autoencoder.compile(loss="mse",optimizer='Adam')
autoencoder.summary()
```

#Model Training

```
history = autoencoder.fit(x_train, x_train,
            epochs=500,
            batch_size=512,
            shuffle=True,
            validation_data=(x_val, x_val))
```

PCA Forest Cover

```
#importing Libraries :)
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA

#load dataset
dataset=pd.read_csv("covtype.csv")
dataset.head()
Dataset.shape

#check for null values
dataset.isnull().sum()

count_classes = pd.value_counts(dataset['Cover_Type'], sort = True)
count_classes.plot(kind = 'bar', rot=0)
plt.title("Covertype Distribution")
plt.xlabel("Cover_Type")
plt.ylabel("Frequency")
plt.savefig("Covertype Distribution")
plt.show()

y = dataset.iloc[:,-1].values
x = dataset.drop(["Cover_Type"],axis=1)
x.shape,y.shape

cov1 = dataset[dataset['Cover_Type']==1]
cov2 = dataset[dataset['Cover_Type']==2]
cov3 = dataset[dataset['Cover_Type']==3]
cov4 = dataset[dataset['Cover_Type']==4]
cov5 = dataset[dataset['Cover_Type']==5]
cov6 = dataset[dataset['Cover_Type']==6]
cov7 = dataset[dataset['Cover_Type']==7]
print(cov1.shape,cov2.shape,cov3.shape,cov4.shape,cov5.shape,cov6.shape,cov7.shape)

# Implementing Undersampling for Handling Imbalanced
from imblearn.under_sampling import NearMiss
nm = NearMiss()
X_res,y_res=nm.fit_sample(x,y)
X_res.shape,y_res.shape
X_res.to_numpy()
X_res.shape

#Feature Scaling
min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(X_res)
```

```
pca = PCA(n_components=3)
pca.fit(x_scaled)
x_pca = pca.transform(x_scaled)
x_pca.shape

#spliting the dataset into train and test set

x_train,x_test,y_train,y_test= train_test_split(x_pca,y_res,test_size = 0.1, random_state = 0)
x_train,x_val,y_train,y_val= train_test_split(x_train,y_train,test_size = 0.2, random_state = 0)
print(x_train.shape)
print(x_test.shape)
print(x_val.shape)

import matplotlib
%matplotlib inline
from mpl_toolkits import mplot3d
colors=['red','green','blue','cyan','magenta','coral','gold']
plt.figure(figsize =(12, 9))
ax = plt.axes(projection='3d')
surf = ax.scatter3D(x_test[:, 0], x_test[:, 1], x_test[:, 2], c=y_test,
cmap=matplotlib.colors.ListedColormap(colors))
# Add a color bar which maps values to colors.
cb = plt.colorbar(surf, shrink=0.5, aspect=5)
plt.savefig('pcacov.png')
plt.show()

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(x_train, y_train)
    k = knn.score(x_test,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(x_test)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Dataset 2.MNIST handwritten digits Dataset

Simple autoencoder

```
#importing the necessary libraries
from keras.models import Model
from keras.layers import Input, Dense
from keras.callbacks import History
history = History()
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Data Acquisition

# Data Acquisition

x_train=pd.read_csv("mnist_train.csv")
x_test=pd.read_csv("mnist_test.csv")

# Data Visulization
x_train.head()

# Data Preprocessing

y_train = x_train.iloc[:,0].values
x_train = x_train.drop(["label"],axis=1)
y_test = x_test.iloc[:,0].values
x_test = x_test.drop(["label"],axis=1)

y_train

x_train = x_train.to_numpy()
x_test = x_test.to_numpy()
#Normalization
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255

#spliting the train dataset into train and validation set
x_train,x_val,y_train,y_val= train_test_split(x_train,y_train,test_size = 0.2, random_state = 0)

print(f'x_train shape = {x_train.shape}')
print(f'x_test shape = {x_test.shape}')
print(f'x_val shape = {x_val.shape}')

#Data Visualizaion
n = 10
plt.figure(figsize =(20,4))

for i in range(n):
    ax = plt.subplot(1, n, i+1)
    plt.imshow(x_train[i].reshape(28,28))
    plt.gray()
plt.savefig('mnist.png')
plt.show()

# Model

#Builidg the simple auto encoder model
encoding_dim = 32
input_img = Input(shape=(784,))
encoded = Dense(encoding_dim, activation='relu')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)

autoencoder = Model(input_img, decoded)

encoder = Model(input_img, encoded)
encoder.compile(optimizer='adadelta', loss = 'binary_crossentropy')

#model compilation and summary
autoencoder.compile(optimizer='adadelta', loss = 'binary_crossentropy')
encoder.compile(optimizer='adadelta', loss = 'binary_crossentropy')
autoencoder.summary()

# Fit Data

#Training the model
history = autoencoder.fit(x_train, x_train, epochs=100, batch_size = 256, shuffle = True,
            validation_data = (x_val,x_val))

# Plot training loss and validation split loss over the epochs

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('sae_model_loss.png')
plt.show()

# predicting on the test set
reconstructed_imgs = autoencoder.predict(x_test)
encoded_imgs_train = encoder.predict(x_train)
encoded_imgs_test = encoder.predict(x_test)

# Evalution of the model on the test set
evaluation = autoencoder.evaluate(x_test, x_test)

print("Loss:",evaluation)

# Ploting the original input vs reconstructed output
n = 10
plt.figure(figsize =(20,4))

for i in range(n):
    ax = plt.subplot(2, n, i+1)
    plt.imshow(x_test[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    ax = plt.subplot(2,n, i+1+n)
    plt.imshow(reconstructed_imgs[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.show()

#Classification
from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(encoded_imgs_train, y_train)
    k = knn.score(encoded_imgs_test,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix
```

```
y_pred = knn.predict(encoded_imgs_test)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Sparse Autoencoder

```
from keras.models import Model
from keras.layers import Input, Dense
from keras import regularizers
from keras.callbacks import History
history = History()
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Data Acquisition

# Data Acuisition
x_train=pd.read_csv("mnist_train.csv")
x_test=pd.read_csv("mnist_test.csv")

# Data Visulization
x_train.head()

# Data Preprocessing
y_train = x_train.iloc[:,0].values
x_train = x_train.drop(["label"],axis=1)
y_test = x_test.iloc[:,0].values
x_test = x_test.drop(["label"],axis=1)

x_train = x_train.to_numpy()
x_test = x_test.to_numpy()
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255

x_train,x_val,y_train,y_val= train_test_split(x_train,y_train,test_size = 0.2, random_state = 0)

print(f'x_train shape = {x_train.shape}')
print(f'x_test shape = {x_test.shape}')
print(f'x_val shape = {x_val.shape}')

# Model
```

```
encoding_dim = 32
input_img = Input(shape=(784,))
# add a Dense layer with a L1 activity regularizer
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(32, activation='relu', activity_regularizer=regularizers.l1(1e-6))(encoded)
decoded = Dense(128,activation='relu')(encoded)
decoded = Dense(784, activation='sigmoid')(decoded)

autoencoder = Model(input_img, decoded)

encoder = Model(input_img, encoded)
encoder.compile(optimizer='adadelta', loss = 'binary_crossentropy')

autoencoder.compile(optimizer='adadelta', loss = 'binary_crossentropy')
autoencoder.summary()

# Fit Data

history = autoencoder.fit(x_train, x_train, epochs=300, batch_size = 256, shuffle = True,
            validation_data = (x_val,x_val))

# Plot training loss and validation split loss over the epochs

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('sparseae_model_loss.png')
plt.show()

reconstructed_imgs = autoencoder.predict(x_test)
encoded_imgs = encoder.predict(x_test)

evaluation = autoencoder.evaluate(x_test, x_test)

print("Loss:",evaluation)

#checkingthe encode images mean
encoded_imgs.mean()

# Plotting the original input vs reconstructed output
n = 10
plt.figure(figsize =(20,4))
```

```
for i in range(n):
    ax = plt.subplot(2, n, i+1)
    plt.imshow(x_test[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    ax = plt.subplot(2,n, i+1+n)
    plt.imshow(reconstructed_imgs[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.savefig('sparseae_output.png')
plt.show()

encoded_imgs_train = encoder.predict(x_train)
encoded_imgs_test = encoder.predict(x_test)

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(encoded_imgs_train, y_train)
    k = knn.score(encoded_imgs_test,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])


from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(encoded_imgs_test)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Deep Autoencoder

```
from keras.models import Model
from keras.layers import Input, Dense
from keras.callbacks import History
history = History()
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Data Acquisition

# Data Acuisition
x_train=pd.read_csv("mnist_train.csv")
x_test=pd.read_csv("mnist_test.csv")

# Data Visulization
x_train.head()

# Data Preprocessing

y_train = x_train.iloc[:,0].values
x_train = x_train.drop(["label"],axis=1)
y_test = x_test.iloc[:,0].values
x_test = x_test.drop(["label"],axis=1)

x_train = x_train.to_numpy()
x_test = x_test.to_numpy()
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255

x_train,x_val,y_train,y_val= train_test_split(x_train,y_train,test_size = 0.2, random_state = 0)

print(f'x_train shape = {x_train.shape}')
print(f'x_test shape = {x_test.shape}')
print(f'x_val shape = {x_val.shape}')

# Model

input_img = Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)

encoded = Dense(32, activation='relu')(encoded)

decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)

autoencoder = Model(input_img, decoded)

encoder = Model(input_img, encoded)

autoencoder.compile(optimizer='adadelta', loss = 'binary_crossentropy')
autoencoder.summary()

# Fit Data

history = autoencoder.fit(x_train, x_train, epochs=200, batch_size = 256, shuffle = True,
            validation_data = (x_val,x_val))

# Plot training loss and validation split loss over the epochs

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('deepae_model_loss.png')
plt.show()

reconstructed_imgs = autoencoder.predict(x_test)
encoded_imgs = encoder.predict(x_test)

evaluation = autoencoder.evaluate(x_test, x_test)
print("Loss:",evaluation)

#checkingthe encode images mean
encoded_imgs.mean()

# Ploting the original input vs reconstructed output
n = 10
plt.figure(figsize =(20,4))

for i in range(n):
   ax = plt.subplot(1, n, i+1)
   plt.imshow(reconstructed_imgs[i].reshape(28,28))
   plt.gray()
   ax.get_xaxis().set_visible(False)
   ax.get_yaxis().set_visible(False)
plt.savefig('deepae_output.png')
plt.show()

encoded_imgs_train = encoder.predict(x_train)
encoded_imgs_test = encoder.predict(x_test)

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(encoded_imgs_train, y_train)
    k = knn.score(encoded_imgs_test,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])


from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(encoded_imgs_test)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Denoising Autoencoder

```
from keras.models import Model, Sequential
from keras.layers import Input, Dense, Dropout
from keras.callbacks import History
history = History()
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Data Acquisition

# Data Acuisition
x_train=pd.read_csv("mnist_train.csv")
x_test=pd.read_csv("mnist_test.csv")

# Data Visulization
x_train.head()
```

```
# Data Preprocessing

y_train = x_train.iloc[:,0].values
x_train = x_train.drop(["label"],axis=1)
y_test = x_test.iloc[:,0].values
x_test = x_test.drop(["label"],axis=1)

x_train = x_train.to_numpy()
x_test = x_test.to_numpy()
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255

x_train,x_val,y_train,y_val= train_test_split(x_train,y_train,test_size = 0.2, random_state = 0)

#adding some noise
noise_factor = 0.4
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)
x_val_noisy = x_val + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_val.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)

#Data Visualizaion
n = 10
plt.figure(figsize =(20,4))

for i in range(n):
   ax = plt.subplot(1, n, i+1)
   plt.imshow(x_train_noisy[i].reshape(28,28))
   plt.gray()
plt.savefig('denoisingmnist.png')
plt.show()

print(f'x_train shape = {x_train.shape}')
print(f'x_test shape = {x_test.shape}')
print(f'x_val shape = {x_val.shape}')

# Model

input_img = Input(shape=(784,))

encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)

encoded = Dense(32, activation='relu')(encoded)
```

```
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)

decoded = Dense(784, activation='sigmoid')(decoded)

autoencoder = Model(input_img, decoded)

encoder = Model(input_img, encoded)

encoder.compile(optimizer='adadelta', loss = 'binary_crossentropy')
autoencoder.compile(optimizer='adadelta', loss = 'binary_crossentropy')
autoencoder.summary()

# Fit Data

history = autoencoder.fit(x_train_noisy, x_train, epochs=150, batch_size = 256, shuffle = True,
            validation_data = (x_val_noisy,x_val))

# Plot training loss and validation split loss over the epochs

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('denoisingae_model_loss.png')
plt.show()

reconstructed_imgs = autoencoder.predict(x_test_noisy)

evaluation = autoencoder.evaluate(x_test_noisy, x_test)

print("Loss:",evaluation)



# Ploting the original input vs reconstructed output
n = 10
plt.figure(figsize =(20,4))

for i in range(n):
   ax = plt.subplot(2, n, i+1)
   plt.imshow(x_test_noisy[i].reshape(28,28))
```

```
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    ax = plt.subplot(2,n, i+1+n)
    plt.imshow(reconstructed_imgs[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.savefig('Denoisingae_model_output.png')
plt.show()

encoded_imgs_train = encoder.predict(x_train)
encoded_imgs_test = encoder.predict(x_test)

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(encoded_imgs_train, y_train)
    k = knn.score(encoded_imgs_test,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(encoded_imgs_test)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```
Stacked Autoencoder

```
from sklearn.model_selection import train_test_split

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import keras
import keras.backend as K
```

```
from keras.layers import Input, Activation, Dense, BatchNormalization, Dropout
from keras.callbacks import History
history = History()
from keras.models import Model
from keras.utils import np_utils
from keras.regularizers import l2
from keras.layers.normalization import BatchNormalization

# Data Acquisition

# Data Acuisition
x_train=pd.read_csv("mnist_train.csv")
x_test=pd.read_csv("mnist_test.csv")

# Data Visulization
x_train.head()

# Data Preprocessing

y_train = x_train.iloc[:,0].values
x_train = x_train.drop(["label"],axis=1)
y_test = x_test.iloc[:,0].values
x_test = x_test.drop(["label"],axis=1)

x_train = x_train.to_numpy()
x_test = x_test.to_numpy()
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255

#x_train,x_val,y_train,y_val= train_test_split(x_train,y_train,test_size = 0.2, random_state = 0)

print(f'x_train shape = {x_train.shape}')
print(f'x_test shape = {x_test.shape}')
#print(f'x_val shape = {x_val.shape}')

# Model

input_data = Input(shape=(784,))#0
encoded = Dense(128, activation='relu')(input_data)#1
encoded = BatchNormalization()(encoded)#2
encoded = Dense(64, activation='relu')(encoded)#3
encoded = BatchNormalization()(encoded)#4
encoded = Dense(32, activation='relu')(encoded)#5
encoded = BatchNormalization()(encoded)#6

decoded = Dense(64, activation='sigmoid')(encoded)#7
decoded = BatchNormalization()(decoded)#8
decoded = Dense(128, activation='sigmoid')(decoded)#9
decoded = BatchNormalization()(decoded)#10
decoded = Dense(784, activation='sigmoid')(decoded)#11

stackedautoencoder = Model(input_data, decoded)
stackedautoencoder.compile(loss='binary_crossentropy', optimizer = 'adadelta')

stackedencoder = Model(input_data, encoded)
stackedencoder.compile(loss='binary_crossentropy', optimizer = 'adadelta')

stackedautoencoder.summary()

#Autoencoder1
input_data1 = Input(shape=(784,))#0
encoded1 = Dense(128, activation='relu')(input_data1)#1
encoded1 = BatchNormalization()(encoded1)#2
decoded1 = Dense(784, activation='sigmoid')(encoded1)#11

autoencoder1 = Model(input_data1, decoded1)
encoder1 = Model(input_data1, encoded1)

#Autoencoder2
input_data2 = Input(shape=(128,))
encoded2 = Dense(64, activation='relu')(input_data2)#3
encoded2 = BatchNormalization()(encoded2)#4
decoded2 = Dense(128, activation='sigmoid')(encoded2)#9
decoded2 = BatchNormalization()(decoded2)#10

autoencoder2 = Model(input_data2, decoded2)
encoder2 = Model(input_data2, encoded2)

#Autoencoder3
input_data3 = Input(shape=(64,))
encoded3 = Dense(32, activation='relu')(input_data3)#5
encoded3 = BatchNormalization()(encoded3)#6
decoded3 = Dense(64, activation= 'sigmoid')(encoded3)#7
decoded3 = BatchNormalization()(decoded3)#8

autoencoder3 = Model(input_data3, decoded3)
encoder3 = Model(input_data3, encoded3)

autoencoder1.compile(loss='binary_crossentropy', optimizer = 'adadelta')
autoencoder2.compile(loss='binary_crossentropy', optimizer = 'adadelta')
autoencoder3.compile(loss='binary_crossentropy', optimizer = 'adadelta')
```

```
encoder1.compile(loss='binary_crossentropy', optimizer = 'adadelta')
encoder2.compile(loss='binary_crossentropy', optimizer = 'adadelta')
encoder3.compile(loss='binary_crossentropy', optimizer = 'adadelta')

# Fit Data

history1 = autoencoder1.fit(x_train, x_train, epochs=200, batch_size = 512, validation_split = 0.25,
shuffle = True)

# summarize history for loss
plt.plot(history1.history['loss'])
plt.plot(history1.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('stackedae1_model_loss.png')
plt.show()

first_layer_code = encoder1.predict(x_train)
print(first_layer_code.shape)

history2 = autoencoder2.fit(first_layer_code, first_layer_code,epochs = 200, batch_size = 512,
validation_split = 0.25, shuffle = True)

# summarize history for loss
plt.plot(history2.history['loss'])
plt.plot(history2.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('stackedae2_loss.png')
plt.show()

second_layer_code = encoder2.predict(first_layer_code)
print(second_layer_code.shape)

history3 = autoencoder3.fit(second_layer_code, second_layer_code, epochs = 200, batch_size =
512,validation_split = 0.30,shuffle = True)

# summarize history for loss
plt.plot(history3.history['loss'])
plt.plot(history3.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('stackedae3_model_loss.png')
plt.show()

# AE: lot training loss and validation split loss over the epochs
plt.plot(history1.history['loss'],'k',label='AE1 loss')
plt.plot(history1.history['val_loss'],'g',label='AE1 val loss')
plt.plot(history2.history['loss'],'y',label='AE2 loss')
plt.plot(history2.history['val_loss'],'m',label='AE2 val loss')
plt.plot(history3.history['loss'],'r',label='AE3 loss')
plt.plot(history3.history['val_loss'],'b',label='AE3 val loss')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(loc='center right')
plt.savefig('stackAEwhole_model_loss.png')
plt.show()

#Stacked autoencoder
stackedautoencoder.layers[1].set_weights(autoencoder1.layers[1].get_weights()) # first dense layer
stackedautoencoder.layers[2].set_weights(autoencoder1.layers[2].get_weights())
stackedautoencoder.layers[3].set_weights(autoencoder2.layers[1].get_weights())
stackedautoencoder.layers[4].set_weights(autoencoder2.layers[2].get_weights())
stackedautoencoder.layers[5].set_weights(autoencoder3.layers[1].get_weights())
stackedautoencoder.layers[6].set_weights(autoencoder3.layers[2].get_weights())
stackedautoencoder.layers[7].set_weights(autoencoder3.layers[3].get_weights())
stackedautoencoder.layers[8].set_weights(autoencoder3.layers[4].get_weights())
stackedautoencoder.layers[9].set_weights(autoencoder2.layers[3].get_weights())
stackedautoencoder.layers[10].set_weights(autoencoder2.layers[4].get_weights())
stackedautoencoder.layers[11].set_weights(autoencoder1.layers[3].get_weights())

reconstructed_imgs = stackedautoencoder.predict(x_test)
reconstructed_imgs.shape
#reconstructed_imgs = autoencoder.predict(x_test)
#encoded_imgs = encoder.predict(x_test)

evaluation = stackedautoencoder.evaluate(x_test, x_test)
print("Loss:",evaluation)

# Ploting the original input vs reconstructed output
n = 10
plt.figure(figsize =(20,4))

for i in range(n):
    ax = plt.subplot(1, n, i+1)
    plt.imshow(reconstructed_imgs[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    plt.savefig('StackedAEoutput')
plt.show()

#encodermodel
stackedencoder.layers[1].set_weights(autoencoder1.layers[1].get_weights()) # first dense layer
stackedencoder.layers[2].set_weights(autoencoder1.layers[2].get_weights())
stackedencoder.layers[3].set_weights(autoencoder2.layers[1].get_weights())
stackedencoder.layers[4].set_weights(autoencoder2.layers[2].get_weights())
stackedencoder.layers[5].set_weights(autoencoder3.layers[1].get_weights())
stackedencoder.layers[6].set_weights(autoencoder3.layers[2].get_weights())

latent_vector_train = stackedencoder.predict(x_train)
latent_vector = stackedencoder.predict(x_test)

# Ploting the encoded output
n = 10
plt.figure(figsize =(20,4))

for i in range(n):
    ax = plt.subplot(2, n, i+1)
    plt.imshow(latent_vector[i].reshape(8,4))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    plt.savefig('StackedAElatentoutput')
plt.show()

encoded_imgs_train = stackedencoder.predict(x_train)
encoded_imgs_test = stackedencoder.predict(x_test)

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(encoded_imgs_train, y_train)
    k = knn.score(encoded_imgs_test,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])
```

```
from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(encoded_imgs_test)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Convolutional Autoencoder

```
from keras.layers import Input, Dense, Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Model
from keras import backend as K
import keras.backend as K
from tensorflow.keras.models import Sequential
from keras.callbacks import History
history = History()
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Data Acuisition
x_train=pd.read_csv("mnist_train.csv")
x_test=pd.read_csv("mnist_test.csv")

# Data Visulization
x_train.head()

# Data Preprocessing

y_train = x_train.iloc[:,0].values
x_train = x_train.drop(["label"],axis=1)
y_test = x_test.iloc[:,0].values
x_test = x_test.drop(["label"],axis=1)

x_train = x_train.to_numpy()
x_test = x_test.to_numpy()
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

x_train,x_val,y_train,y_val= train_test_split(x_train,y_train,test_size = 0.2, random_state = 0)
```

```
print(f'x_train shape = {x_train.shape}')
print(f'x_test shape = {x_test.shape}')
print(f'x_val shape = {x_val.shape}')


input_img = Input(shape=(28, 28, 1))

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
encoder = Model(input_img, encoded)
encoder.compile(optimizer='adadelta', loss='binary_crossentropy')

history = autoencoder.fit(x_train, x_train, epochs=100, batch_size=256, shuffle=True,
        validation_data=(x_val, x_val))

# Plot training loss and validation split loss over the epochs
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('Convae_model_loss.png')
plt.show()

reconstructed_imgs = autoencoder.predict(x_test)
evaluation = autoencoder.evaluate(x_test, x_test)
print("Loss:",evaluation)
```

```
# Ploting the original input vs reconstructed output
n = 10
plt.figure(figsize =(20,4))

for i in range(n):
    ax = plt.subplot(2, n, i+1)
    plt.imshow(x_test[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    ax = plt.subplot(2,n, i+1+n)
    plt.imshow(reconstructed_imgs[i].reshape(28,28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.savefig('Convae_model_output.png')
plt.show()


encoded_imgs_train = encoder.predict(x_train)
encoded_imgs_train = np.reshape(encoded_imgs_train, (len(encoded_imgs_train), 128))
encoded_imgs_test = encoder.predict(x_test)
encoded_imgs_test = np.reshape(encoded_imgs_test, (len(encoded_imgs_test), 128))
print(encoded_imgs_train.shape)
print(encoded_imgs_test.shape)

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(encoded_imgs_train, y_train)
    k = knn.score(encoded_imgs_test,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])


from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(encoded_imgs_test)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

MnistPCA

```
#importing the necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA

# Data Acquisition

# Data Acquisition

x_train=pd.read_csv("mnist_train.csv")
x_test=pd.read_csv("mnist_test.csv")

# Data Visulization
x_train.head()

# Data Preprocessing

y_train = x_train.iloc[:,0].values
x_train = x_train.drop(["label"],axis=1)
y_test = x_test.iloc[:,0].values
x_test = x_test.drop(["label"],axis=1)
x_train = x_train.to_numpy()
x_test = x_test.to_numpy()
#Normalization
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255

#spliting the train dataset into train and validation set
x_train,x_val,y_train,y_val= train_test_split(x_train,y_train,test_size = 0.2, random_state = 0)

print(f'x_train shape = {x_train.shape}')
print(f'x_test shape = {x_test.shape}')
print(f'x_val shape = {x_val.shape}')

#Data Visualizaion
n = 10
plt.figure(figsize =(20,4))
```

```
for i in range(n):
    ax = plt.subplot(1, n, i+1)
    plt.imshow(x_train[i].reshape(28,28))
    plt.gray()
plt.savefig('mnist.png')
plt.show()

pca = PCA(n_components=32)
pca.fit(x_train)
x_pca_t = pca.transform(x_train)
x_pca = pca.transform(x_test)
X_pca.shape

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(x_pca_t, y_train)
    k = knn.score(x_pca,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(x_pca)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Dataset 3 : Wisconsin Breast Cancer dataset

Simple Autoencoder

```
#importing Libraries :)

import tensorflow as tf
import numpy as np
import pandas as pd

from tensorflow import keras
from keras.layers import Dense, Dropout,Input, BatchNormalization
from keras import regularizers
from keras.models import Sequential, Model
from keras.callbacks import History
history = History()

import matplotlib.pyplot as plt
from keras.layers import Dense,Activation,Layer,Lambda

import sklearn
from sklearn import preprocessing
from sklearn.model_selection import train_test_split

#load dataset
dataset=pd.read_csv("breastcancerdataset.csv")
dataset.head()

#Don't need id column and Unnamed:32 column
dataset=dataset.drop(["id","Unnamed: 32"],axis=1)
dataset.head()

dataset.shape

#check for null values
pd.isnull(dataset).sum()

y = dataset.iloc[:,0].values
x = dataset.drop(["diagnosis"],axis=1)
#y = dataset.iloc[:,0].values
x.to_numpy()
x.shape
print(y)

#encoding Categorical Data Turning 'y' values into numeric value
from sklearn.preprocessing import LabelEncoder
labelencoder_x= LabelEncoder()
y = labelencoder_x.fit_transform(y)
print(y)

#spliting the dataset into train and test set
x_train,x_val,y_train,y_val= train_test_split(x,y,test_size = 0.1, random_state = 0)
x_train,x_test,y_train,y_test= train_test_split(x_train,y_train,test_size = 0.1, random_state = 0)

#Feature Scaling
#from sklearn.preprocessing import StandardScaler
#sc = StandardScaler()
#x_train = sc.fit_transform(x_train)
```

```
#x_test = sc.fit_transform(x_test)

min_max_scaler = preprocessing.MinMaxScaler()
x_train = min_max_scaler.fit_transform(x_train)
x_test = min_max_scaler.fit_transform(x_test)
x_val = min_max_scaler.fit_transform(x_val)

input_data = Input(shape=(30,))
encoded = Dense(3, activation='relu')(input_data)
decoded = Dense(30, activation='sigmoid')(encoded)

autoencoder = Model(input_data, decoded)
opt = keras.optimizers.Adam(lr=0.01)
autoencoder.compile(loss="mse",optimizer='Adam')
autoencoder.summary()

x_test
history = autoencoder.fit(x_train, x_train, epochs=500, shuffle=True, validation_data=(x_val, x_val))

evaluation = autoencoder.evaluate(x_test, x_test)
print("Loss:",evaluation)

# Plot training loss and validation split loss over the epochs
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('simpleAE_model_loss.png')
plt.show()

encoder = Model(input_data, encoded)
latent_vector_train = encoder.predict(x_train)
latent_vector = encoder.predict(x_test)
latent_vector.shape
Latent_vector

import matplotlib
%matplotlib inline
from mpl_toolkits import mplot3d
colors=['red','blue']
plt.figure(figsize =(12, 9))
ax = plt.axes(projection='3d')
```

```
surf = ax.scatter3D(latent_vector[:, 0], latent_vector[:, 1], latent_vector[:, 2], c=y_test,
cmap=matplotlib.colors.ListedColormap(colors))
# Add a color bar which maps values to colors.
cb = plt.colorbar(surf, shrink=0.5, aspect=20)
plt.savefig('simpleAE_3d.png')
plt.show()

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
   knn = KNeighborsClassifier(n_neighbors = i)
   knn.fit(latent_vector_train, y_train)
   k = knn.score(latent_vector,y_test)
   knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

y_pred=knn.predict(latent_vector)

from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix
y_pred = knn.predict(latent_vector)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Sparse autoencoder

```
#importing Libraries :)

import tensorflow as tf
import numpy as np
import pandas as pd

from tensorflow import keras
from keras.layers import Dense, Dropout, Input, BatchNormalization
from keras import regularizers
from keras.models import Sequential, Model
from keras.callbacks import History
history = History()

import matplotlib.pyplot as plt
```

```
from keras.layers import Dense,Activation,Layer,Lambda

import sklearn
from sklearn import preprocessing
from sklearn.model_selection import train_test_split

#load dataset

dataset=pd.read_csv("breastcancerdataset.csv")

dataset.head()

#Don't need id column and Unnamed:32 column
dataset=dataset.drop(["id","Unnamed: 32"],axis=1)
dataset.head()

dataset.shape

#check for null values
pd.isnull(dataset).sum()

y = dataset.iloc[:,0].values
x = dataset.drop(["diagnosis"],axis=1)
#y = dataset.iloc[:,0].values
x.to_numpy()

print(y)
#encoding Categorical Data Turning 'y' values into numeric value
from sklearn.preprocessing import LabelEncoder
labelencoder_x= LabelEncoder()
y = labelencoder_x.fit_transform(y)
print(y)

#spliting the dataset into train and test set

x_train,x_val,y_train,y_val= train_test_split(x,y,test_size = 0.1, random_state = 0)
x_train,x_test,y_train,y_test= train_test_split(x_train,y_train,test_size = 0.1, random_state = 0)

#Feature Scaling
#from sklearn.preprocessing import StandardScaler
#sc = StandardScaler()
#x_train = sc.fit_transform(x_train)
#x_test = sc.fit_transform(x_test)

min_max_scaler = preprocessing.MinMaxScaler()
x_train = min_max_scaler.fit_transform(x_train)
```

```
x_test = min_max_scaler.fit_transform(x_test)
x_val = min_max_scaler.fit_transform(x_val)

#initializer = tf.keras.initializers.RandomUniform(minval=0., maxval=1.)
#layer = tf.keras.layers.Dense(3, kernel_initializer=initializer)

input_data = Input(shape=(30,))
encoded = Dense(3, activation='relu', activity_regularizer = regularizers.l1(1e-6))(input_data)
#encoded = BatchNormalization()(encoded)
decoded = Dense(30, activation='sigmoid')(encoded)

autoencoder = Model(input_data, decoded)
opt = keras.optimizers.Adam(lr=0.001)
autoencoder.compile(loss="mse",optimizer=opt)
autoencoder.summary()

x_test

history = autoencoder.fit(x_train, x_train, epochs=500, shuffle=True, validation_data=(x_val, x_val))

evaluation = autoencoder.evaluate(x_test, x_test)

print("Loss:",evaluation)

# Plot training loss and validation split loss over the epochs
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('spsrseAE_model_loss.png')
plt.show()

autoencoder.evaluate(x_test, x_test)
#model.save('si_autoencoder.model')
data = autoencoder.predict(x_test)

data

encoder = Model(input_data, encoded)
latent_vector_train = encoder.predict(x_train)
latent_vector = encoder.predict(x_test)

latent_vector.shape
```

```
latent_vector

import matplotlib
%matplotlib inline
from mpl_toolkits import mplot3d
colors=['red','blue']
plt.figure(figsize =(12, 9))
ax = plt.axes(projection='3d')
surf = ax.scatter3D(latent_vector[:, 0], latent_vector[:, 1], latent_vector[:, 2], c=y_test,
cmap=matplotlib.colors.ListedColormap(colors))
# Add a color bar which maps values to colors.
cb = plt.colorbar(surf, shrink=0.5, aspect=20)
plt.savefig('sparseAE_3d.png')
plt.show()

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
   knn = KNeighborsClassifier(n_neighbors = i)
   knn.fit(latent_vector_train, y_train)
   k = knn.score(latent_vector,y_test)
   knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

y_pred=knn.predict(latent_vector)

from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(latent_vector)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Deep Autoencoder

```
#importing Libraries :)

import tensorflow as tf
import numpy as np
import pandas as pd
```

```
from tensorflow import keras
from keras.layers import Dense, Dropout,Input, BatchNormalization
from keras import regularizers
from keras.models import Sequential, Model
from keras.callbacks import History
history = History()

import matplotlib.pyplot as plt
from keras.layers import Dense,Activation,Layer,Lambda

import sklearn
from sklearn import preprocessing
from sklearn.model_selection import train_test_split

#load dataset

dataset=pd.read_csv("breastcancerdataset.csv")

dataset.head()

#Don't need id column and Unnamed:32 column
dataset=dataset.drop(["id","Unnamed: 32"],axis=1)
dataset.head()

dataset.shape

#check for null values
pd.isnull(dataset).sum()

y = dataset.iloc[:,0].values
x = dataset.drop(["diagnosis"],axis=1)
#y = dataset.iloc[:,0].values

x.to_numpy()

#encoding Categorical Data Turning 'y' values into numeric value
from sklearn.preprocessing import LabelEncoder
labelencoder_x= LabelEncoder()
y = labelencoder_x.fit_transform(y)
print(y)

#spliting the dataset into train and test set
x_train,x_val,y_train,y_val= train_test_split(x,y,test_size = 0.1, random_state = 0)
x_train,x_test,y_train,y_test= train_test_split(x_train,y_train,test_size = 0.1, random_state = 0)

#Feature Scaling
```

```
#from sklearn.preprocessing import StandardScaler
#sc = StandardScaler()
#x_train = sc.fit_transform(x_train)
#x_test = sc.fit_transform(x_test)

min_max_scaler = preprocessing.MinMaxScaler()
x_train = min_max_scaler.fit_transform(x_train)
x_test = min_max_scaler.fit_transform(x_test)
x_val = min_max_scaler.fit_transform(x_val)

input_data = Input(shape=(30,))
encoded = Dense(25, activation='relu')(input_data)
encoded = Dense(20, activation='relu')(encoded)
#encoded = BatchNormalization()(encoded)
encoded = Dense(10, activation='relu')(encoded)
#encoded = BatchNormalization()(encoded)

encoded = Dense(3, activation='relu')(encoded)
#encoded = BatchNormalization()(encoded)

decoded = Dense(10, activation='relu')(encoded)
decoded = Dense(20, activation='relu')(decoded)
#decoded = BatchNormalization()(decoded)
decoded = Dense(25, activation='relu')(decoded)
decoded = Dense(30, activation='sigmoid')(decoded)

autoencoder = Model(input_data, decoded)
opt = keras.optimizers.Adam(lr=0.01)
autoencoder.compile(loss="mse",optimizer='Adam')
autoencoder.summary()
history = autoencoder.fit(x_train, x_train, epochs=500, shuffle=True, validation_data=(x_val, x_val))

evaluation = autoencoder.evaluate(x_test, x_test)
print("Loss:",evaluation)

# Plot training loss and validation split loss over the epochs
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('DenseAE_model_loss.png')
plt.show()
```

```
autoencoder.evaluate(x_test, x_test)
#model.save('si_autoencoder.model')
data = autoencoder.predict(x_test)

encoder = Model(input_data, encoded)
latent_vector_train = encoder.predict(x_train)
latent_vector = encoder.predict(x_test)
latent_vector.shape
latent_vector

import matplotlib
%matplotlib inline
from mpl_toolkits import mplot3d
colors=['red','blue']
plt.figure(figsize =(12, 9))
ax = plt.axes(projection='3d')
surf = ax.scatter3D(latent_vector[:, 0], latent_vector[:, 1], latent_vector[:, 2], c=y_test,
cmap=matplotlib.colors.ListedColormap(colors))
# Add a color bar which maps values to colors.
cb = plt.colorbar(surf, shrink=0.5, aspect=20)
plt.savefig('DenseAE_3d.png')
plt.show()

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(latent_vector_train, y_train)
    k = knn.score(latent_vector,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

y_pred=knn.predict(latent_vector)

from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(latent_vector)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Stacked Autoencoder

```
#importing Libraries :)

import tensorflow as tf
import numpy as np
import pandas as pd

from tensorflow import keras
from keras.layers import  Dense, Dropout,Input, BatchNormalization, Activation, Layer
from keras import regularizers
from keras.models import Sequential, Model
from keras.callbacks import History
history = History()

import matplotlib.pyplot as plt
from keras.layers import

import sklearn
from sklearn import preprocessing
from sklearn.model_selection import train_test_split

#load dataset
dataset=pd.read_csv("breastcancerdataset.csv")
dataset.head()

#Don't need id column and Unnamed:32 column
dataset=dataset.drop(["id","Unnamed: 32"],axis=1)
dataset.head()
dataset.shape

#check for null values
pd.isnull(dataset).sum()

y = dataset.iloc[:,0].values
x = dataset.drop(["diagnosis"],axis=1)
#y = dataset.iloc[:,0].values

x.to_numpy()
x.shape
print(y)
#encoding Categorical Data Turning 'y' values into numeric value
from sklearn.preprocessing import LabelEncoder
labelencoder_x= LabelEncoder()
y = labelencoder_x.fit_transform(y)
print(y)
```

```
#spliting the dataset into train and test set
x_train,x_test,y_train,y_test= train_test_split(x,y,test_size = 0.1, random_state = 0)

#Feature Scaling
#from sklearn.preprocessing import StandardScaler
#sc = StandardScaler()
#x_train = sc.fit_transform(x_train)
#x_test = sc.fit_transform(x_test)

min_max_scaler = preprocessing.MinMaxScaler()
x_train = min_max_scaler.fit_transform(x_train)
x_test = min_max_scaler.fit_transform(x_test)

input_data = Input(shape=(30,))#0
encoded = Dense(20, activation='relu')(input_data)#1
encoded = Dense(10, activation='relu')(encoded)#2
encoded = Dense(3, activation='relu')(encoded)#3

decoded = Dense(10, activation='relu')(encoded)#4
decoded = Dense(20, activation='relu')(decoded)#5
decoded = Dense(30, activation='sigmoid')(decoded)#6

stackautoencoder = Model(input_data, decoded)
stackautoencoder.compile(loss="mse",optimizer='Adam')
stackautoencoder.summary()

#Autoencoder1
input_data1 = Input(shape=(30,))#0
encoded1 = Dense(20, activation='relu')(input_data1)#1
decoded1 = Dense(30, activation='sigmoid')(encoded1)#6

autoencoder1 = Model(input_data1, decoded1)
encoder1 = Model(input_data1, encoded1)

#Autoencoder2
input_data2 = Input(shape=(20,))
encoded2 = Dense(10, activation='relu')(input_data2)#2
decoded2 = Dense(20, activation='relu')(encoded2)#5

autoencoder2 = Model(input_data2, decoded2)
encoder2 = Model(input_data2, encoded2)

#Autoencoder3
input_data3 = Input(shape=(10,))
encoded3 = Dense(3, activation='relu')(input_data3)#3
decoded3 = Dense(10, activation='relu')(encoded3)#4
```

```
autoencoder3 = Model(input_data3, decoded3)
encoder3 = Model(input_data3, encoded3)

autoencoder1.compile(loss="mse",optimizer='Adam')
autoencoder2.compile(loss="mse",optimizer='Adam')
autoencoder3.compile(loss="mse",optimizer='Adam')

encoder1.compile(loss="mse",optimizer='Adam')
encoder2.compile(loss="mse",optimizer='Adam')
encoder3.compile(loss="mse",optimizer='Adam')

history1 = autoencoder1.fit(x_train, x_train,epochs=500,shuffle=True,validation_split = 0.30)

# AE1: lot training loss and validation split loss over the epochs

plt.plot(history1.history['loss'])
plt.plot(history1.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('stackAE1_model_loss.png')
plt.show()

first_layer_code = encoder1.predict(x_train)
print(first_layer_code.shape)

history2 = autoencoder2.fit(first_layer_code, first_layer_code,epochs=500,shuffle=True,
validation_split = 0.30)

# AE2: lot training loss and validation split loss over the epochs

plt.plot(history2.history['loss'])
plt.plot(history2.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('stackAE2_model_loss.png')
plt.show()

second_layer_code = encoder2.predict(first_layer_code)
print(first_layer_code.shape)
```

```
history3 = autoencoder3.fit(second_layer_code,
second_layer_code,epochs=500,shuffle=True,validation_split = 0.30)

# AE3: lot training loss and validation split loss over the epochs

plt.plot(history3.history['loss'])
plt.plot(history3.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('stackAE3_model_loss.png')
plt.show()


# AE3: lot training loss and validation split loss over the epochs
plt.plot(history1.history['loss'],'k',label='AE1 loss')
plt.plot(history1.history['val_loss'],'g',label='AE1 val loss')
plt.plot(history2.history['loss'],'y',label='AE2 loss')
plt.plot(history2.history['val_loss'],'m',label='AE2 val loss')
plt.plot(history3.history['loss'],'r',label='AE3 loss')
plt.plot(history3.history['val_loss'],'b',label='AE3 val loss')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(loc='best')
plt.savefig('stackAEwhole_model_loss.png')
plt.show()

#Stacked autoencoder
stackautoencoder.layers[1].set_weights(autoencoder1.layers[1].get_weights()) # first dense layer
stackautoencoder.layers[2].set_weights(autoencoder2.layers[1].get_weights())
stackautoencoder.layers[3].set_weights(autoencoder3.layers[1].get_weights())
stackautoencoder.layers[4].set_weights(autoencoder3.layers[2].get_weights())
stackautoencoder.layers[5].set_weights(autoencoder2.layers[2].get_weights())
stackautoencoder.layers[6].set_weights(autoencoder1.layers[2].get_weights())

evaluation = stackautoencoder.evaluate(x_test, x_test)
print("Loss:",evaluation)

stackencoder = Model(input_data, encoded)
latent_vector_train = stackencoder.predict(x_train)
latent_vector = stackencoder.predict(x_test)

latent_vector.shape
```

```
latent_vector

import matplotlib
%matplotlib inline
from mpl_toolkits import mplot3d
colors=['red','blue']
plt.figure(figsize =(12, 9))
ax = plt.axes(projection='3d')
surf = ax.scatter3D(latent_vector[:, 0], latent_vector[:, 1], latent_vector[:, 2], c=y_test,
cmap=matplotlib.colors.ListedColormap(colors))
# Add a color bar which maps values to colors.
cb = plt.colorbar(surf, shrink=0.5, aspect=20)
plt.savefig('StackAE_3d.png')
plt.show()

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(latent_vector_train, y_train)
    k = knn.score(latent_vector,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

y_pred=knn.predict(latent_vector)

from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(latent_vector)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Denoising Autoencoder

```
#importing Libraries :)

import tensorflow as tf
import numpy as np
import pandas as pd
```

```
from tensorflow import keras
from keras.layers import Dense, Dropout, Flatten,Input, BatchNormalization, Activation,Layer
from keras import regularizers
from keras.models import Sequential, Model
from keras.callbacks import History
history = History()

import matplotlib.pyplot as plt

import sklearn
from sklearn import preprocessing
from sklearn.model_selection import train_test_split

#load dataset

dataset=pd.read_csv("breastcancerdataset.csv")

dataset.head()

#Don't need id column and Unnamed:32 column
dataset=dataset.drop(["id","Unnamed: 32"],axis=1)
dataset.head()

dataset.shape

#check for null values
pd.isnull(dataset).sum()

y = dataset.iloc[:,0].values
x = dataset.drop(["diagnosis"],axis=1)
#y = dataset.iloc[:,0].values



x.to_numpy()

x.shape

print(y)
#encoding Categorical Data Turning 'y' values into numeric value
from sklearn.preprocessing import LabelEncoder
labelencoder_x= LabelEncoder()
y = labelencoder_x.fit_transform(y)
print(y)
```

```
#spliting the dataset into train and test set

x_train,x_val,y_train,y_val= train_test_split(x,y,test_size = 0.1, random_state = 0)
x_train,x_test,y_train,y_test= train_test_split(x_train,y_train,test_size = 0.1, random_state = 0)

#Feature Scaling
#from sklearn.preprocessing import StandardScaler
#sc = StandardScaler()
#x_train = sc.fit_transform(x_train)
#x_test = sc.fit_transform(x_test)

min_max_scaler = preprocessing.MinMaxScaler()
x_train = min_max_scaler.fit_transform(x_train)
x_test = min_max_scaler.fit_transform(x_test)
x_val = min_max_scaler.fit_transform(x_val)

mu, sigma = 0, 0.1
#creating a noise with the same dimension as the dataset
x_train_noisy = x_train + np.random.normal(mu, sigma, x_train.shape)
x_test_noisy = x_test + np.random.normal(mu, sigma, x_test.shape)
x_val_noisy = x_val + np.random.normal(mu, sigma, x_val.shape)

input_data = Input(shape=(30,))
encoded = Dense(25, activation='relu')(input_data)
encoded = Dense(20, activation='relu')(encoded)
encoded = Dense(10, activation='relu')(encoded)

encoded = Dense(3, activation='relu')(encoded)

decoded = Dense(10, activation='relu')(encoded)
decoded = Dense(20, activation='relu')(decoded)
decoded = Dense(25, activation='relu')(decoded)
decoded = Dense(30, activation='sigmoid')(decoded)

autoencoder = Model(input_data, decoded)
opt = keras.optimizers.Adam(lr=0.01)
autoencoder.compile(loss="mse",optimizer='Adam')
autoencoder.summary()

x_test

history = autoencoder.fit(x_train_noisy, x_train, epochs=500, shuffle=True,
validation_data=(x_val_noisy, x_val))

evaluation = autoencoder.evaluate(x_test_noisy, x_test)
print("Loss:",evaluation)
```

```
# Plot training loss and validation split loss over the epochs
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('DNAE_model_loss.png')
plt.show()

autoencoder.evaluate(x_test_noisy, x_test)
#model.save('si_autoencoder.model')
data = autoencoder.predict(x_test)

encoder = Model(input_data, encoded)
latent_vector_train = encoder.predict(x_train_noisy)
latent_vector = encoder.predict(x_test_noisy)
latent_vector.shape

import matplotlib
%matplotlib inline
from mpl_toolkits import mplot3d
colors=['red','blue']
plt.figure(figsize =(12, 9))
ax = plt.axes(projection='3d')
surf = ax.scatter3D(latent_vector[:, 0], latent_vector[:, 1], latent_vector[:, 2], c=y_test,
cmap=matplotlib.colors.ListedColormap(colors))
# Add a color bar which maps values to colors.
cb = plt.colorbar(surf, shrink=0.5, aspect=20)
plt.savefig('DNAE_3d.png')
plt.show()

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(latent_vector_train, y_train)
    k = knn.score(latent_vector,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

y_pred=knn.predict(latent_vector)
```

```
from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(latent_vector)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Convolutional Autoencoder

```
#importing Libraries :)

import tensorflow as tf
import numpy as np
import pandas as pd

from tensorflow import keras
from keras.layers import Conv1D, MaxPooling1D, UpSampling1D, Dense, Dropout, Flatten,Input,
BatchNormalization, AveragePooling1D
from keras import regularizers
from keras.models import Sequential, Model
from keras.callbacks import History
history = History()

import matplotlib.pyplot as plt
from keras.layers import Activation,Layer

import sklearn
from sklearn import preprocessing
from sklearn.model_selection import train_test_split

#load dataset

dataset=pd.read_csv("breastcancerdataset.csv")

dataset.head()

#Don't need id column and Unnamed:32 column
dataset=dataset.drop(["id","Unnamed: 32"],axis=1)
dataset.head()

dataset.shape

#check for null values
```

```
pd.isnull(dataset).sum()

y = dataset.iloc[:,0].values
x = dataset.drop(["diagnosis"],axis=1)
#y = dataset.iloc[:,0].values
x.to_numpy()

#encoding Categorical Data Turning 'y' values into numeric value
from sklearn.preprocessing import LabelEncoder
labelencoder_x= LabelEncoder()
y = labelencoder_x.fit_transform(y)
print(y)

#spliting the dataset into train and test set

x_train,x_val,y_train,y_val= train_test_split(x,y,test_size = 0.1, random_state = 0)
x_train,x_test,y_train,y_test= train_test_split(x_train,y_train,test_size = 0.1, random_state = 0)

#Feature Scaling
#from sklearn.preprocessing import StandardScaler
#sc = StandardScaler()
#x_train = sc.fit_transform(x_train)
#x_test = sc.fit_transform(x_test)

min_max_scaler = preprocessing.MinMaxScaler()
x_train = min_max_scaler.fit_transform(x_train)
x_test = min_max_scaler.fit_transform(x_test)
x_val = min_max_scaler.fit_transform(x_val)
print(x_train.shape,x_test.shape,x_val.shape)

x_train = x_train.reshape(460, 30,1)
x_test = x_test.reshape(52, 30,1)
x_val = x_val.reshape(57, 30,1)

input_data = Input(shape=(30,1))
#encoder
encoded = Conv1D(10, 3, activation="relu", padding = "same")(input_data)
encoded = BatchNormalization()(encoded)
encoded = AveragePooling1D(2, padding="same")(encoded)

encoded = Conv1D(5, 3, activation="relu", padding="same")(encoded)
encoded = BatchNormalization()(encoded)
encoded = AveragePooling1D(2, padding="same")(encoded)

encoded = Conv1D(3, 3, activation="relu", padding="same")(encoded)
encoded = BatchNormalization()(encoded)
encoded = AveragePooling1D(2, padding="same")(encoded)

# 3 dimensions in the encoded layer
encoded = Conv1D(1, 2, activation="relu")(encoded)

#decoder
decoded = UpSampling1D(2)(encoded)
decoded = Conv1D(3, 3, activation='relu', padding="same")(decoded)
decoded = BatchNormalization()(decoded)

decoded = UpSampling1D(2)(decoded)
decoded = Conv1D(5, 3, activation='relu')(decoded)
decoded = BatchNormalization()(decoded)

decoded = UpSampling1D(3)(decoded)
decoded = Conv1D(10, 3, activation='relu',padding="same")(decoded)
decoded = BatchNormalization()(decoded)

decoded = Conv1D(1, 3, activation='sigmoid', padding='same')(decoded)

encoder = Model(input_data, encoded)

autoencoder = Model(input_data, decoded)
opt = keras.optimizers.Adam(lr=0.01)
autoencoder.compile(loss="mse",optimizer='Adam')
autoencoder.summary()

history = autoencoder.fit(x_train, x_train,epochs=500,shuffle=True,validation_data=(x_val, x_val))

evaluation = autoencoder.evaluate(x_test, x_test)
print("Loss:",evaluation)

# Plot training loss and validation split loss over the epochs
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig('ConvAE_model_loss.png')
plt.show()

autoencoder.evaluate(x_test, x_test)
#model.save('si_autoencoder.model')
data = autoencoder.predict(x_test)
```

```
encoder = Model(input_data, encoded)
latent_vector_train = encoder.predict(x_train)
latent_vector = encoder.predict(x_test)

latent_vector.shape

latent_vector = latent_vector.reshape(52, 3)
latent_vector_train = latent_vector_train.reshape(460, 3)

import matplotlib
%matplotlib inline
from mpl_toolkits import mplot3d
colors=['red','blue']
plt.figure(figsize =(12, 9))
ax = plt.axes(projection='3d')
surf = ax.scatter3D(latent_vector[:, 0], latent_vector[:, 1], latent_vector[:, 2], c=y_test,
cmap=matplotlib.colors.ListedColormap(colors))
# Add a color bar which maps values to colors.
cb = plt.colorbar(surf, shrink=0.5, aspect=20)
plt.savefig('ConvAE_3d.png')
plt.show()

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,20):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(latent_vector_train, y_train)
    k = knn.score(latent_vector,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

y_pred=knn.predict(latent_vector)

from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = knn.predict(latent_vector)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

PCA

```
#importing Libraries :)
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA

#load dataset

dataset=pd.read_csv("breastcancerdataset.csv")

dataset.head()

#Don't need id column and Unnamed:32 column
dataset=dataset.drop(["id","Unnamed: 32"],axis=1)
dataset.head()

dataset.shape

#check for null values
pd.isnull(dataset).sum()

y = dataset.iloc[:,0].values
x = dataset.drop(["diagnosis"],axis=1)
#y = dataset.iloc[:,0].values
x.to_numpy()
print(y)

#encoding Categorical Data Turning 'y' values into numeric value
from sklearn.preprocessing import LabelEncoder
labelencoder_x= LabelEncoder()
y = labelencoder_x.fit_transform(y)
print(y)

#Feature Scaling
#from sklearn.preprocessing import StandardScaler
#sc = StandardScaler()
#x_train = sc.fit_transform(x_train)
#x_test = sc.fit_transform(x_test)

min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(x)
```

```
pca = PCA(n_components=3)
pca.fit(x_scaled)
x_pca = pca.transform(x_scaled)
x_pca.shape

#spliting the dataset into train and test set

x_train,x_val,y_train,y_val= train_test_split(x_pca,y,test_size = 0.1, random_state = 0)
x_train,x_test,y_train,y_test= train_test_split(x_train,y_train,test_size = 0.1, random_state = 0)

import matplotlib
%matplotlib inline
from mpl_toolkits import mplot3d
colors=['red','blue']
plt.figure(figsize =(12, 9))
ax = plt.axes(projection='3d')
surf = ax.scatter3D(x_pca[:, 0], x_pca[:, 1], x_pca[:, 2], c=y,
cmap=matplotlib.colors.ListedColormap(colors))
# Add a color bar which maps values to colors.
cb = plt.colorbar(surf, shrink=0.5, aspect=20)
plt.savefig('pcaBC.png')
plt.show()

x_train.shape

from sklearn.neighbors import KNeighborsClassifier
knnscores = []
for i in range(1,10):
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(x_train, y_train)
    k = knn.score(x_test,y_test)
    knnscores.append(k)

s = np.array(knnscores)
result = np.where(s == s.max())
print('Knn Score : ',s.max(),'and k : ', result[0])

from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix
y_pred = knn.predict(x_test)
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```