APPLICATION STUDIES OF AUTOENCODERS

SAMAD DADVANDIPOUR

At this stage of the research we have illustrated two applications of autoencoders, with respect medical and MNIST datasets in case of cancer and handwritten, where the aim is to detect the applications Malignant or Benign.

1. Introduction

Autoencoders and their application in ANN

Autoencoder neural networks are unsupervised machine learning algorithms. They apply backpropagation, setting the target values equal to the inputs. Thus, they are algorithms similar to PCA but minimize the same objective function. An autoencoder is a neural network whose target output is its input. Autoencoders are pretty identical to PCA, but they are more flexible when compared to the others. For example, autoencoders can represent linear and no-linear transformations in encoding, but PCA can perform the linear transformation.

Need for autoencoders

Data compression is a big topic used in computer vision, computer networks, and many other applications. Now the point of the data compression is to convert input into smaller representation data that we recreated to a degree of quality. The small representation would be passed around, and when anyone needed the original, they would reconstruct from the smaller representation.

An encoder also gives a representation as to the output of each layer, and having multiple representations of different dimensions is always useful. So an autoencoder could tell you make use of pre-trained layers from another model to apply transfer to prime the encoder or the decoder; even though practical applications of autoencoders were pretty rare sometimes back today, data denoising and dimensionality reduction for data visualization are considered as two main interesting practical applications of autoencoders. With appropriate dimensionality sparsing constraints, autoencoders can learn data projection.

Descriptions

There are basically three main layers: the encoder, the coder, and the encoder. The first component that is the encoder, is the part of the neural network that compresses the input into latent space representation in a reduced dimension. The compressed image typically looks garbled/distorted/mixed up, nothing like the original image. The next component represents the latent space. Code is the

part of the network that represents the compressed input fed to the decoder. The third component is the decoder., which decodes the encoded image back to the original dimension with close or same images. In fact, the decoded image is a lossy reconstruction of the original image. It reconstructs the input from the latent space representation.

The properties of autoencoders

They can only compress data similar to what they have been trained on and autoencoders that have been trained. Autoencoders are usually called lossy. It means that the decompressed outputs are degraded contrasted to the original inputs. Now, if you have appropriate training data, it is easy to train specialized algorithms to work well on a particular input type.

Training autoencoders

It doesn't need any new engineering; furthermore, in almost all contents where the autoencoder is used, the compression and decompression functions are carried out with the neural network [17, 18] [19, 21].

Preprocessing before training

There are four hyperparameters that we need to set before training them.

1. The first one is the code size

The code size represents the number of nodes in the middle layer smaller size results in more compression.

2. The second parameter is the number of layers.

Now the autoencoder can be as we want it to be. We may have two or more layers in both the encoders and decoders without considering the input and outputs. Next is the loss function, so we either use mean squared error or binary cross-entropy. Now, if the input values are in the range of 0 to 1, then we typically use cross-entropy. Otherwise, we use the mean squared error.

3. The third parameter is the nodes number for each layer.

The number of nodes for each layer decreases with each subsequent layer of the encoder and increases back in the encoder [18]. Also, the decoder is symmetric

in terms of the layer structure, but this is unnecessary, and we have total control over this parameter. The architecture of an autoencoder has a deeper insight with a couple of layers between the input and output. The sizes of these layers are smaller than the input layer.

Case study 1: Medical application of Autoencoder

Problem statement: Given the prostate cancer dataset, try to predict whether the cancer is Malignant or Benign [5].

2. Summary of research results

Using the diagnosis and statistical dataset for prostate cancer**, (**Table 1 and 2), we try to predict whether the cancer is Malignant or Benign. The application is using the tensor flow matrix in this respect. Also Keras model used for the dimensional reduction.

Table 1. Dataset

id	diagnosis_result	radius	texture	perimeter	area	smoothness	compactness	symmetry	fractal_dimension
4	M	14	16	78	386	0.07	0.284	0.26	0.096999999 99999999
1	M	23	12	151	954	0.1430000 00000000 02	0.278	0.242	0.079
3	M	21	27	130	1203	0.125	0.16	0.207	0.06
5	M	9	19	135	1297	0.141	0.133	0.181000000 00000002	0.059000000 000000004
2	B	9	13	133	1326	0.1430000 00000000 02	0.079	0.181000000 00000002	0.057

As is shown in the above table, we have eight independent features and one dependent feature (Diagnosis result), which are the number of features and the description of the dataset [4].

2.1. Required libraries:

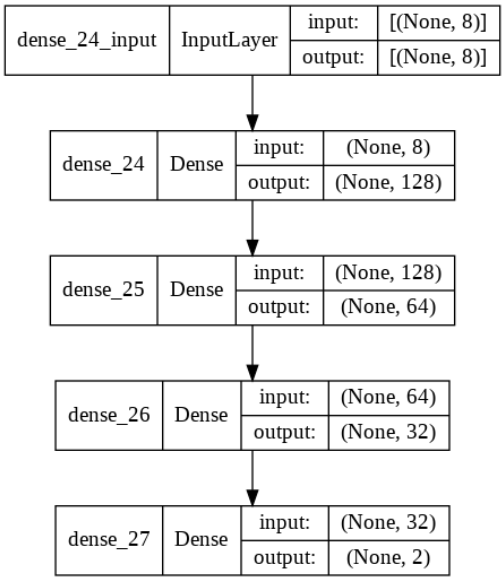
```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Flatten, Reshape
import pandas as pd
import random
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import seaborn as sns
```

Table. 2. *Some statistical results of the dataset*

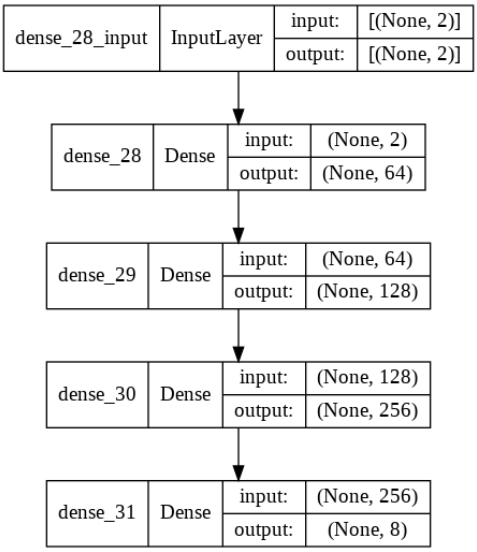
index	id	diagnosis_result	radius	texture	perimeter	area	smoothness	compactness	symmetry	fractal_dimension
count	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
mean	50.5	0.62	16.85	18.23	96.78	702.88	0.10273	0.1267	0.19317	0.064690 00000000 001
std	29.011 491975 882016	0.4878317 31214563 25	4.8790 937227 68149	5.1929 536563 27742	23.676 088606 80268	319.71 089465 580644	0.014641 7522547 9892	0.06114356 346775475	0.03078 503342 256237 7	0.008150 96821416 2219
min	1.0	0.0	9.0	11.0	52.0	202.0	0.07	0.038	0.135	0.053
25%	25.75	0.0	12.0	14.0	82.5	476.75	0.0935	0.0805	0.172	0.059000 00000000 0004
50%	50.5	1.0	17.0	17.5	94.0	644.0	0.102	0.1185	0.19	0.063
75%	75.25	1.0	21.0	22.25	114.25	917.0	0.111999 9999999 9999	0.157	0.209	0.069
max	100.0	1.0	25.0	27.0	172.0	1878.0	0.143000 0000000 0002	0.345	0.304	0.096999 99999999 999

We build an encoder to reduce the number of features from 8 to 2 and check whether we can get the same results. Model architecture for encoder unit, dimensions of original space = 8, latent space dimensions = 2.

2.2. The model architecture

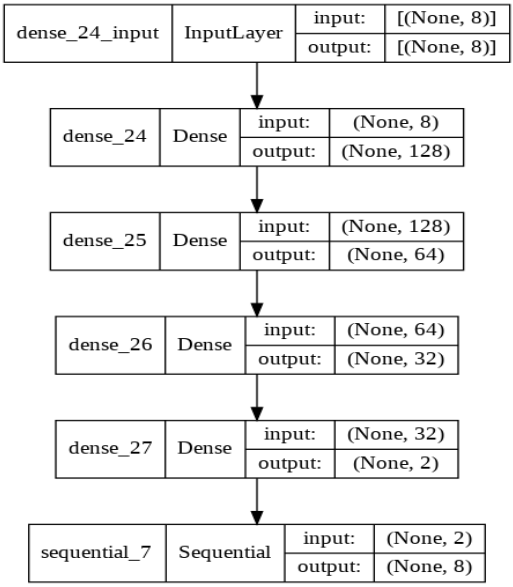


Decoder unit architecture: Input dimensions = 2, reconstructed dimensions = 8



Autoencoder architecture: Combining both encoder and decoder so, in this model, both original dimensions and output or reconstructed dimensions should be the same [1, 2, 3].

```
# This is the dimension of the original space
input_dim = 8

# This is the dimension of the latent space (encoding space)
latent_dim = 2

encoder = Sequential([
   Dense(128, activation='relu', input_shape=(input_dim,)),
   Dense(64, activation='relu'),
   Dense(32, activation='relu'),
   Dense(latent_dim, activation='relu'),
   #Dense(1, activation = 'sigmoid')
])

decoder = Sequential([
Dense(64, activation='relu', input_shape=(latent_dim,)),
   Dense(128, activation='relu'),
   Dense(256, activation='relu'),
   Dense(input_dim, activation=None)
])

autoencoder = Model(inputs=encoder.input, outputs=decoder(encoder.output))
autoencoder.compile(loss='mse', optimizer='adam')
```



Our autoencoder is still untrained at this time. So let's try feeding it some instances from the dataset to see how effectively it starts to reconstruct the following:

```
def plot_orig_vs_recon(title='', n_samples=3):
    fig = plt.figure(figsize=(10,6))
    plt.suptitle(title)
    for i in range(3):
        plt.subplot(3, 1, i+1)

   idx = random.sample(range(x_train.shape[0]), 1)
        plt.plot(autoencoder.predict(x_train[idx]).squeeze(), label='reconstructed' if i ==
0 else '')
        plt.plot(x_train[idx].squeeze(), label='original' if i == 0 else '')
        fig.axes[i].set_xticklabels(metric_names)
        plt.xticks(np.arange(0, 10, 1))
        plt.grid(True)
        if i == 0: plt.legend();

plot_orig_vs_recon('Before training the Autoencoder')
```

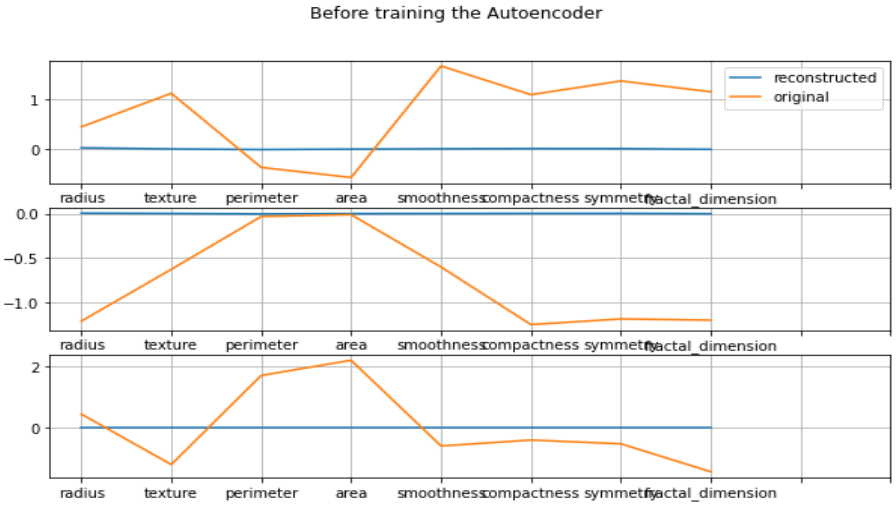


Figure 1. *Illustration of original data taken from the prostate*

As we can depict from the above graph, there is a lot of difference between the original and reconstructed data. We say that both original and reconstructed data are similar if the graph lines overlap. Now we will train our autoencoder with the following parameters [2, 3]:

1. Loss = MSE
2. Optimizer = Adam
3. Epochs = 5000 and
4. Batch size = 32

After training the autoencoder, the loss that occurred is given as:

```
#After training the encoder
model_history = autoencoder.fit(x_train, x_train, validation_data = (x_test, x_te
st), epochs=5000, batch_size=32, verbose=0)

plt.plot(model_history.history["loss"])
plt.title("Loss vs. Epoch")
plt.ylabel("Loss")
plt.xlabel("Epoch")
plt.grid(True)
plt.savefig("loss vs Epochs")
```

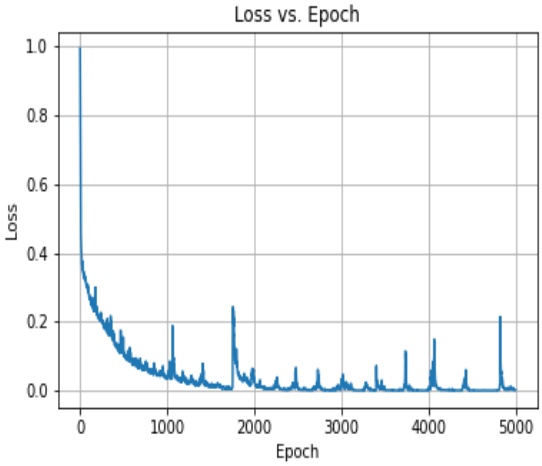


Figure 2. *training result based on given data, based on loss Vs. Given epoch*

Our model converged. It worked, and now we check the reconstruction on a trained autoencoder:

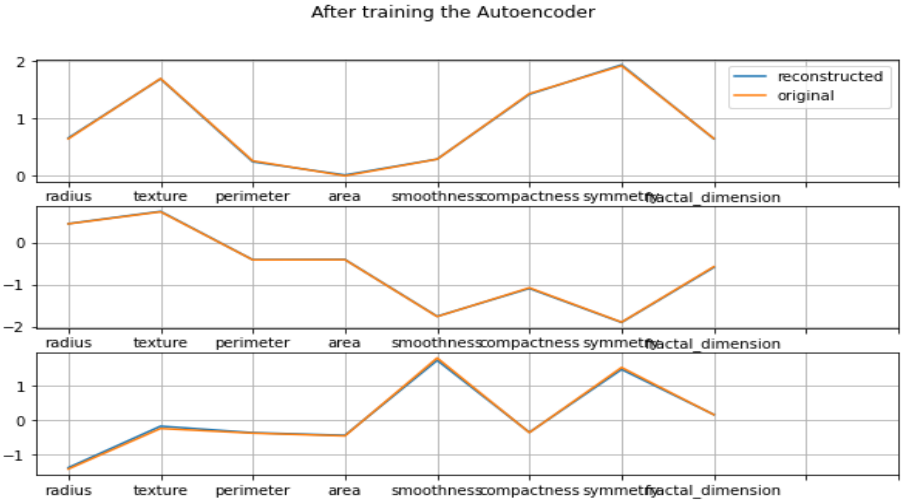


Figure 3. *Illustration of overlapping the original data after training, the training detects the given cancerous data*

3. Conclusion

In the digital era, millions of bytes of data are exchanged every day on the internet. Although storing and analyzing this humongous data is challenging, it also

reduces energy consumption. Autoencoder is a special unsupervised or specifically self-supervised neural network consisting of an encoder and decoder unit. First, it starts widespread, then its units/connections are pushed closer to the center and spread out again. This architecture makes the autoencoder compress the training data's informational content, encoding it in a low-dimensional space. In the case of the prostate in general, a similar process happens. Still, when the prostate is under tumor type, ML can diagnose whether the cancer is Malignant (bad-natured) or Benign (good-natured). We used an autoencoder model similar to the prostate with given data for both cases in this example. After considering the critical eight features by the autoencoder, the original and reconstructed lines overlap almost completely, which tells us our model worked perfectly, i.e., it reconstructed the input data of 8 dimensions to the reconstructed output of 8 dimensions using the latent space of two sizes. As a result, the reconstructed values are almost identical to the originals!

Let us now examine the latent space. We'll use a 2D scatterplot to visualize it because it's two-dimensional. The 8-dimensional data is projected onto a plane in this way.

```
encoded_x_train = encoder(x_train)
plt.figure(figsize=(6,6))
plt.scatter(encoded_x_train[:, 0], encoded_x_train[:, 1], alpha=.8)
plt.xlabel('Latent Dimension 1')
plt.ylabel('Latent Dimension 2');
```

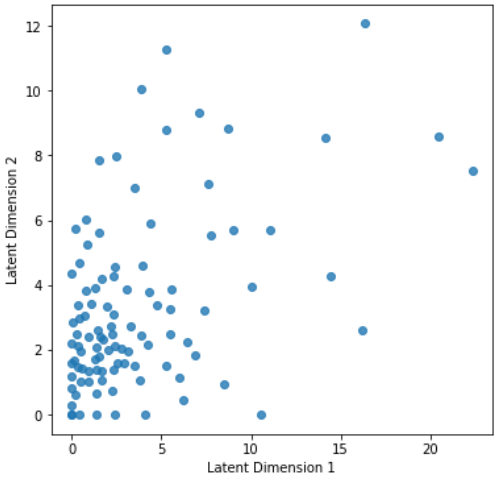


Figure 4. *Illustration of data dimensions*

The decoder needs this 2D imprint to reconstruct the original 8-dimensional space. However, there are a lot of points crammed into the bottom left corner. This is because we want each class's data points to create unique clusters in a classification situation.

Case study 2: Handwritten Application of Autoencoder

In the MNIST dataset there are 60.000 examples in the training set and 10.000 samples in the test set. An image of a handwritten 28x28 grayscale digit from 0 to 9 is used in each case. We preprocessed the input data to be presentable for the encoder model [16, 17] [9, 10].

4. Summary of research results

4.1. Application

Input data:

```
# Load the MNIST data set
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

#Normalize pixel values to [0., 1.]
x_train = x_train / 255.
x_test = x_test / 255.

# Take a look at the dataset
n_samples = 10
idx = random.sample(range(x_train.shape[0]), n_samples)
plt.figure(figsize=(15,4))
for i in range(n_samples):
plt.subplot(1, n_samples, i+1)
plt.imshow(x_train[idx[i]].squeeze());
plt.xticks([], [])
    plt.yticks([], [])
```

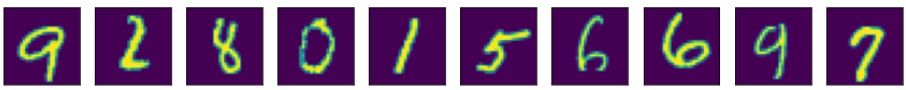
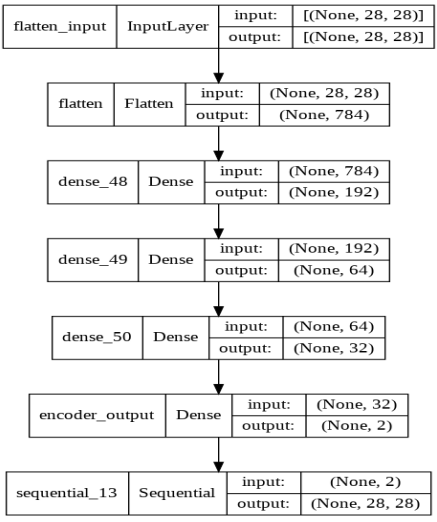


Figure 5. *Number of the sample were taken for training (15 cm × 4 cm)*

Autoencoder encoder architecture, image size = 28 * 28 and dimension of latent space = 2.

```
from keras.regularizers import*
from keras.layers import*

# This is the dimension of the latent space (encoding space)
latent_dim = 2

# Images are 28 by 28
img_shape = (x_train.shape[1], x_train.shape[2])

encoder = Sequential([
Flatten(input_shape=img_shape),
Dense(192, activation='sigmoid'),
Dropout(0.3),
Dense(64, activation='sigmoid'),
Dropout(0.4),

Dense(32, activation='sigmoid'),

Dense (latent_dim, name='encoder_output')
])

decoder = Sequential([
Dense(64, activation='sigmoid', input_shape=(latent_dim,)),
Dropout(0.3),
Dense(128, activation='sigmoid'),
Dropout(0.4),

Dense(img_shape[0] * img_shape[1], activation='relu'),

Reshape(img_shape)

])
```

4.2. The applied Model



We'll design a custom callback by subclassing the tf.Keras.callback.Callback. To visualize the autoencoder, how it builds up the latent space representation as we train it. We will override the function or method on_epoch_begin (self, epoch, logs = None), which is called at the start of the epoch during the training phase; we will extract the representation of the latent space by looking up in the code and plotting it. There is a method in the Keras layer to get the output of an intermediate layer-output. We will train our model on the following hyperparameters [7, 15] [16, 17]:

1. Loss = Binary cross entropy.
2. Optimizer = Adam.
3. Epochs = 12 (number where each latent represented image trains the 32 given dataset pictures) for the decoding parts.
4. Batch size = 32 the sample pictures.

The evolution of latent space representation as the autoencoder is trained, starting at the top left with an untrained state and finishing with a wholly trained state at the bottom right. All of the original space data is projected on the same point of the latent space before the first epoch. However, when the autoencoder learns, the points associated with various classes begin to decouple.

```
class TestEncoder(tf.keras.callbacks.Callback):

def __init__(self, x_test, y_test):

super(TestEncoder, self).__init__()
self.x_test = x_test
self.y_test = y_test
self.current_epoch = 0

def on_epoch_begin(self, epoch, logs={}):
self.current_epoch = self.current_epoch + 1
encoder_model = Model(inputs=self.model.input,
outputs=self.model.get_layer('encoder_output').output)
encoder_output = encoder_model(self.x_test)
plt.subplot(4, 3, self.current_epoch)
plt.scatter(encoder_output[:, 0],
encoder_output[:, 1], s=20, alpha=0.8,
cmap='Set1', c=self.y_test[0:x_test.shape[0]])
plt.xlim(-9, 9)
plt.ylim(-9, 9)
plt.xlabel('Latent Dimension 1')
plt.ylabel('Latent Dimension 2')

autoencoder = Model(inputs=encoder.input, outputs=decoder(encoder.output))
autoencoder.compile(loss='binary_crossentropy', optimizer='adam', metrics
= ['accuracy'])
plt.figure(figsize=(15, 15))
model_history = autoencoder.fit(x_train, x_train, epochs=12, batch_size=32
, verbose=0,
callbacks=[TestEncoder(x_test[0:500], y_test[0:500])])
```

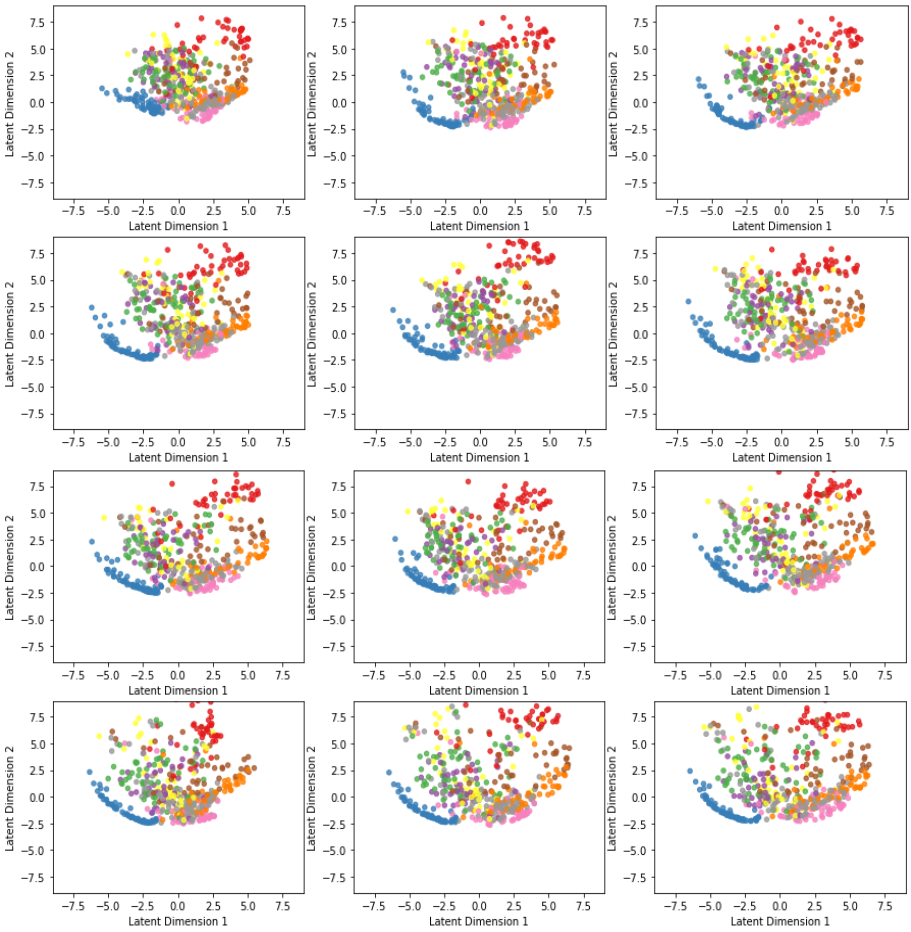


Figure 6. *The latent representation of MNIST*

Loss vs. epochs: to check whether our model converges or not.

```
plt.plot(model_history.history["loss"])
plt.title("Loss vs. Epoch")
plt.ylabel("Loss")
plt.xlabel("Epoch")
plt.grid(True)
```

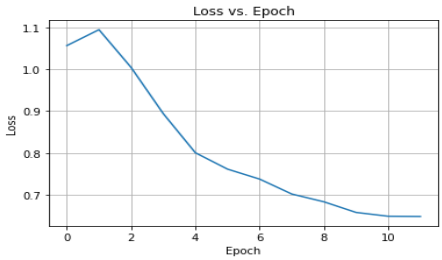
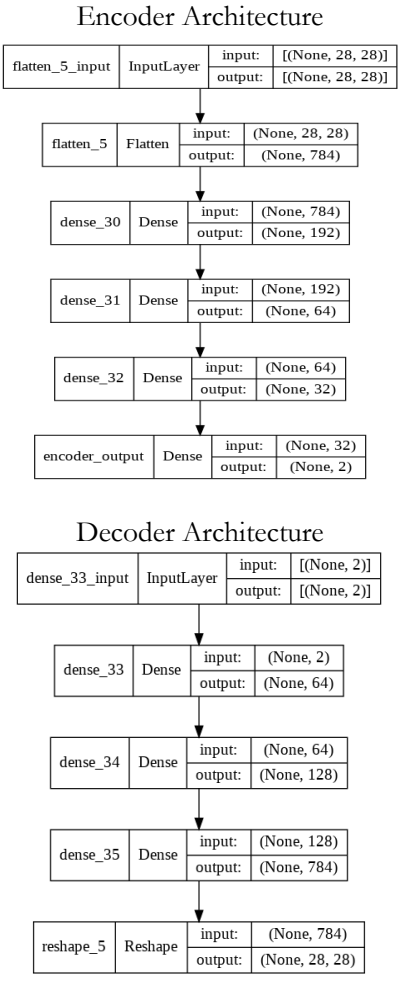


Figure 7. *Loss function, which shows a convergence of the model*



```
#plotting loss
plt.plot(history.history['loss'], label='Training data')
plt.plot(history.history['val_loss'], label='Validation data')
plt.title('Activity Loss')
plt.ylabel('Loss value')
plt.xlabel('No. epoch')
plt.legend(loc="upper left")
plt.show()
```

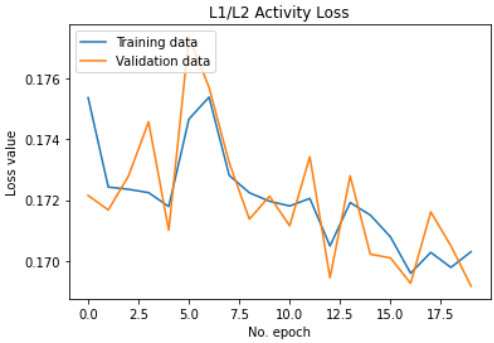


Figure 8. *Training Vs. Validation accuracy*

```
encoded_img = encoder.predict(x_test)
decoded_img = decoder.predict(encoded_img)
plt.figure(figsize=(10, 6))
for i in range(5):
    # Display original
    ax = plt.subplot(2, 5, (1,2))
    plt.imshow(x_test[i].reshape(28, 28))
    plt.show()
    #ax.get_xaxis().set_visible(False)
    #ax.get_yaxis().set_visible(False)
      # Display reconstruction
    ax = plt.subplot(2, 5, (1,2))
    plt.imshow(decoded_img[i].reshape(28, 28))
    plt.show()
    #ax.get_xaxis().set_visible(False)
    #ax.get_yaxis().set_visible(False)
plt.show()
```

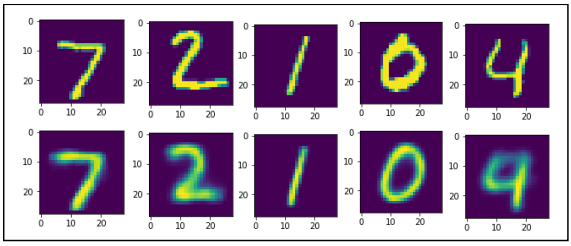


Figure 9. *Original Validation Vs. reconstructed data*

4.3. Autoencoder as Generative model using MNIS dataset

We use an autoencoder as a generative model. We can say when the autoencoder constructs a latent representation of the input data set, we use it as input. Then the decoder takes a sample of a random point of the latent space and produces a synthetic (fake) image [10, 11 and 8], [13, 14 and 12]. For example:

```
n_samples = 40
fake_sample = np.random.uniform(low=-20, high=20, size=(n_samples, 2))
plt.figure(figsize=(15,5))
for i in range(n_samples):
plt.subplot(4, n_samples//4, i+1)
fake_encoding = np.array([fake_sample[i]])
fake_digit = decoder(fake_encoding).numpy().squeeze()
plt.imshow(fake_digit);
plt.xticks([], [])
plt.yticks([], [])
```

We may sample a random point of the latent space of the autoencoder model and use it as input to the decoder to build a synthetic (fake) image. Once the autoencoder has generated a latent representation of the input data set. As an example,

Number of samples to generate = 40

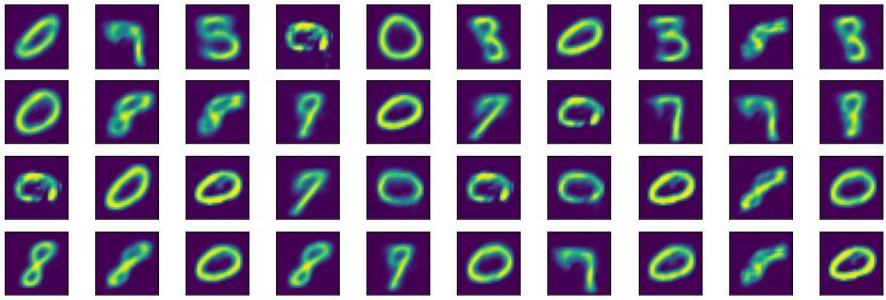


Figure 10. *Synthetic or fake images generated by the trained autoencoder*

5. Conclusion

This study has been carried out for dimensionality reduction using autoencoders for handwritten detection using the MNIST dataset. There are 60,000 examples in the training set and 10,000 samples in the test set. 28 * 28 grayscale image of a handwritten digit from 0 to 9 is used in each case. We preprocessed the input data to be presentable for the encoder model. The input image is 28 * 28, so it is

converted to 784 inputs. The dense 192 activations sigmoid function is the number of inputs neurons in the first layer. The dropout is 0.3 used for optimization. The output of neurons whose value is less than 0.3 is not activated. The dense 64 activation sigmoid is the number of inputs in the second layer. The dropout 0.4 was used for optimization; the output of neurons whose value is less than 0.4 are not activated.

References

[1] Patel, M. I., Suthar, S. and Thakar, J. (2019). *Survey on image compression using machine learning and deep learning.* https://doi.org/10.1109/ICCS45141.2019.9065473

[2] Bai, H., Zhang, M., Liu, M., Wang, A. and Zhao, Y. (2014). *Two-stage multiview image compression using interview SIFT matching.* https://doi.org/10.1109/DCC.2014.14

[3] Zhang, Q., Liu, D. and Li, H. (2018). Deep network-based image coding for simultaneous compression and retrieval. https://doi.org/10.1109/ICIP.2017.8296312

[4] https://www.kaggle.com/sajidsaifi/prostate-cancer.

[5] Abbasi, A. A., Hussain, L., Awan, I. A., Abbasi, I., Majid, A., Nadeem, M. S. A., & Chaudhary, Q. A. (2020). Detecting prostate cancer using a deep learning convolution neural network with the transfer learning approach. *Cognitive Neurodynamics*, 14 (4), pp. 523–533.

[6] Iqbal, S., Siddiqui, G. F., Rehman, A., Hussain, L., Saba, T., Tariq, U., & Abbasi, A. A. (2021). *Prostate Cancer Detection Using Deep Learning and Traditional Techniques.* https://ieeexplore.ieee.org/document/9349466

[7] Rippel, O. and Bourdev, L. (2017). Real-time adaptive image compression. *Proceedings of the 34th International Conference an Machine Learning*, Sydney, Australia, PMLR 70. http://proceedings.mlr.press/v70/rippel17a/rippel17a.pdf

[8] Choi, Y., Kang, D., Hwang, J. J. and Rhee, K. H. (2018). *JPEG Compression Detection Based on Edge-Corner Features Using SVM.* https://doi.org/810.1109/MLDS.2017.25

[9] Makar, M., Chang, C. L., Chen, D., Tsai, S. S. and Girod, B. (2009). Compression of image patches for local feature extraction. https://doi.org/10.1109/ICASSP.2009.4959710

[10] Robinson, J. and Kecman, V. (2003). Combining support vector machine learning with the discrete cosine transform in image compression. *IEEE Trans. Neural Networks*, https://doi.org/10.1109/TNN.2003.813842.

[11] Liu, X. and Yang, J. (2018). *Fast and High Efficient Color Image Compression Using Machine Learning*. https://doi.org/10.1109/IMCEC.2018.8469518

[12] Toderici, G. et al. (2017). *Full resolution image compression with recurrent neural networks*. https://doi.org/10.1109/CVPR.2017.577

[13] Quijas, J. and Fuentes, O. (2014). *Removing JPEG blocking artifacts using machine learning*. https://doi.org/10.1109/SSIAI.2014.6806033

[14] Cavigelli, L., Hager, P. and Benini, L. (2017). *CAS-CNN: A deep convolutional neural network for image compression artifact suppression*. https://doi.org/1410.1109/IJCNN.2017.7965927

[15] https://en.wikipedia.org/wiki/MNIST_database

[16] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/CallbackList

[17] *Autoencoders*. https://saivenkatsudarshanam1996.medium.com/autoencoders-e42dc45b7bd0

[18] *Applied Deep Learning*. https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

[19] *Understand Autoencoders by implementing in TensorFlow*. https://iq.opengenus.org/implementing-autoencoder-tensorflow/

[20] *Building Autoencoders in Keras*. https://blog.keras.io/building-autoencoders-in-keras.html

[21] *Autoencoders*. https://towardsdatascience.com/autoencoders-bits-and-bytes-of-deep-learning eaba376f23ad/