SAT DL: Programming Assignment – 3

Manish Gour
2019H1030025G

1. Problem Statement:

1. Construct good denoising autoencoder using dataset D1. Lets call this constructed model as (M1).

2. Construct a classification model using dataset D2. Let's call this constructed model as M2.

Obtain the performance of this model on some unseen test set from D2. Let's call this performance as P.

3. Use M1 as a pretrained model to learn the new classification problem given in the dataset D2.

What this means is that: use the model M1 as a base model, and finetune this with the new dataset D2. Let's call this new model that you built as M3. Obtain the performance of this new model M3 using the same test set that you used to evaluate M2. Let's call this performance as P0.

4. Compare P and P0.

2. Setup:

Runtime Platform:

Online platform: Colab

RAM: 12GB

Data Set:

MNIST (http://yann.lecun.com/exdb/mnist/),

Fashion MNIST (https://www.kaggle.com/zalando-research/fashionmnist)

External Libraries:

Numpy -- Linear Algebra Operation

matplotlib.pyplot -- Plotting graphs

keras.models import Model -- Functional API for providing input and output of layers

keras.callbacks import ModelCheckpoint -- get a view on internal states and statistics of the model during training.

keras.layers import (Input, Dense, Concatenate) --Layer operation like concatenation

keras.utils import np_utils -- Converts a class vector (integers) to binary class matrix.

keras.datasets -- For loading that are already present in Keras Framework

keras.layers.normalization import BatchNormalization -- transform inputs so that they are standardized so that they will have a mean of zero and a standard deviation of one

sklearn.model_selection import train_test_split -- Splitting the data into train and test sets

keras.callbacks import EarlyStopping -- Stops the training process based on the parameters

keras.layers import Conv2D -- Convolution operation on layer
keras.layers import MaxPooling2D -- Max pooling operation for spatial data.

3. Implementation:

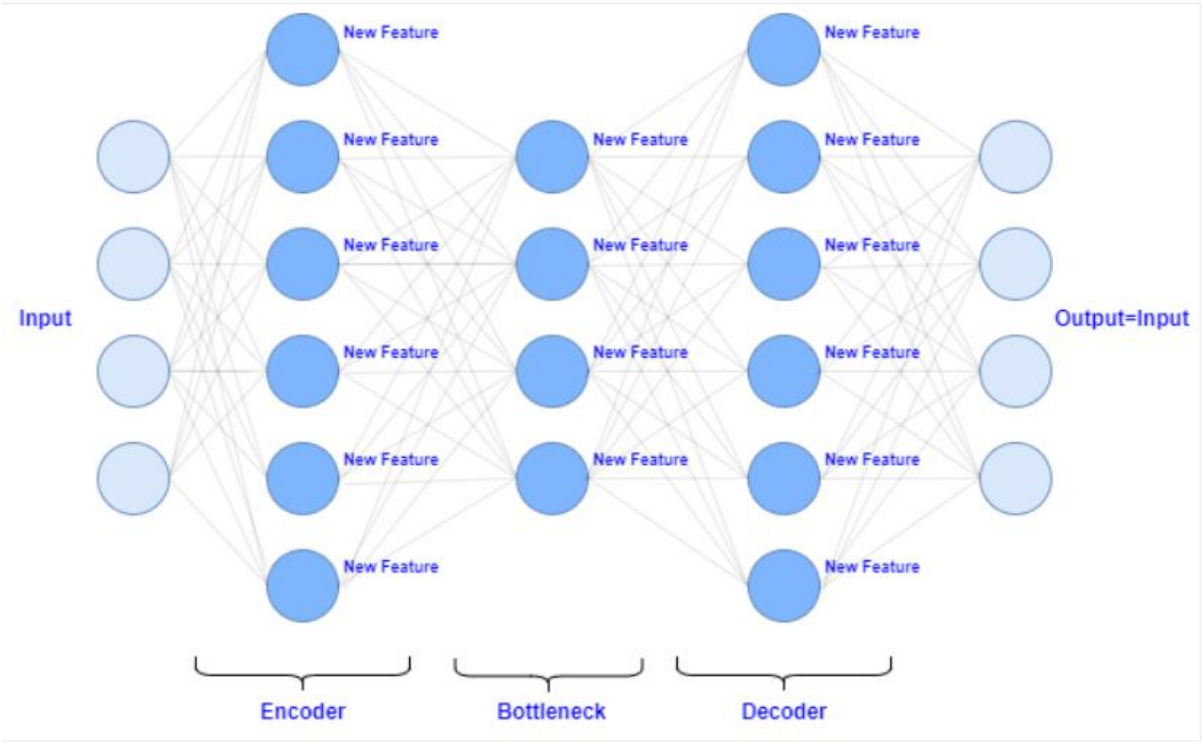
Approach1:

In approach 1, we have added the layer to the pretrained model.

Task 1: Denoising autoencoder on MNIST Dataset

Representational Learning

It is a step further towards artificial intelligence as it is an automatic way of feature engineering, i.e. discovery of new representations of data that are useful during modelling and enhance prediction scores. It replaces manual feature extraction – why to bother with creating interactions, logarithms or other transformations of data and then guessing which of them are important as we can simply use Representation Learning.



Autoencoder in Representation Learning

Denoising Autoencoders can be used to learn superior representation of data. Supplying noisy version of data, forces the Autoencoder to perform better than its clean input counterpart and as a consequence it produces representation of data that is immune to random noise.

MNIST Dataset
The MNIST database of handwritten digits, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and cantered in a fixed-size image.

Steps:

1. **Load, reshape, scale and add noise to data,**
2. **Defining DAE, train DAE on merged training and testing data,**
3. **Get neuron outputs from DAE as new features**

1. Load, reshape, scale and add noise to data

1.1 import libraries and Loading the Dataset.

```
# Essential Libraries
import numpy as np
import matplotlib.pyplot as plt
# from keras.layers import Input, Dense
from keras.models import Model
from keras.layers import Input,Dense,Flatten,Dropout,merge,Reshape,Conv2D,MaxPooling2D,UpSampling2D,Conv2DTranspose
from keras.layers import Dense, Input, Conv2D, LSTM, MaxPool2D, UpSampling2D
from keras.layers.normalization import BatchNormalization
from sklearn.model_selection import train_test_split
from keras.callbacks import EarlyStopping
from keras.utils import to_categorical
import tensorflow as tf
```

```
# Loading the Data
(X_train, Y_train), (X_test, Y_test) = tf.keras.datasets.mnist.load_data()
```

1.2 Data Preprocessing
Conversion of 28 x 28 image of train and test set into a matrix of size 28 x 28 x 1 to feed in the network. Followed by it, conversion of numpy arrays in float32 format for storing floating values.

```
# X_train = X_train.reshape(X_train.shape[0], -1) #for normal mlp types
X_train = X_train.reshape(-1, 28, 28, 1)  #for the conv layer approac
X_train = X_train.astype('float32')
X_train /= 255

X_test = X_test.reshape(X_test.shape[0], -1)
X_test = X_test.reshape(-1, 28, 28, 1) #check
X_test = X_test.astype('float32')
X_test /= 255
```

1.3 Adding noise

introducing Gaussian random noise, noise factor controls the noisiness of images and we clip the values to make sure that the elements of feature vector representing image are between 0 and 1. It will be used to learn superior representation of data.

```
noise_factor = 0.4
X_train_noisy = X_train + noise_factor * np.random.normal(size=X_train.shape)
X_test_noisy = X_test + noise_factor * np.random.normal(size=X_test.shape)

X_train_noisy = np.clip(X_train_noisy, 0.0, 1.0)
X_test_noisy = np.clip(X_test_noisy, 0.0, 1.0)
```

2. Defining DAE, train DAE on merged training and testing data

2.1 Architecture of Denoising Autoencoder

Encoder: It has 4 Convolution blocks; each block has a convolution layer with 'relu' as activation function, followed by a batch normalization layer. Max-pooling layer is used after the first and second convolution blocks.

- The first convolution block will have 32 filters of size 3 x 3, followed by a down sampling (max-pooling) layer,
- The second block will have 64 filters of size 3 x 3, followed by another down sampling layer,
- The third block of encoder will have 128 filters of size 3 x 3,
- The fourth block of encoder will have 256 filters of size 3 x 3.

Decoder:
It has 3 Convolution blocks, each block has a convolution layer followed by a batch normalization layer. Upsampling layer is used after the second and third convolution blocks.

- The first block will have 128 filters of size 3 x 3,
- The second block will have 64 filters of size 3 x 3 followed by another upsampling layer,
- The third block will have 32 filters of size 3 x 3 followed by another upsampling layer,
- The final layer of encoder will have 1 filter of size 3 x 3 which will reconstruct back the input having a single channel.

```
# Denosing Autoencoder Architecture

# Input layer
input_layer = Input(shape=(28, 28, 1))
```

```
# encoding architecture
conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(input_layer) #28 x 28 x 32
conv1 = BatchNormalization()(conv1)
conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv1)
conv1 = BatchNormalization()(conv1)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1) #14 x 14 x 32
conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool1) #14 x 14 x 64
conv2 = BatchNormalization()(conv2)
conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv2)
conv2 = BatchNormalization()(conv2)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2) #7 x 7 x 64
conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool2) #7 x 7 x 128 (small and thick)
conv3 = BatchNormalization()(conv3)
conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv3)
conv3 = BatchNormalization()(conv3)
conv4 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv3) #7 x 7 x 256 (small and thick)
conv4 = BatchNormalization()(conv4)
conv4 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv4)
conv4 = BatchNormalization()(conv4)
```

```
# # decoding architecture
conv5 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv4) #7 x 7 x 128
conv5 = BatchNormalization()(conv5)
conv5 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv5)
conv5 = BatchNormalization()(conv5)
conv6 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv5) #7 x 7 x 64
conv6 = BatchNormalization()(conv6)
conv6 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv6)
conv6 = BatchNormalization()(conv6)
up1 = UpSampling2D((2,2))(conv6) #14 x 14 x 64
conv7 = Conv2D(32, (3, 3), activation='relu', padding='same')(up1) # 14 x 14 x 32
conv7 = BatchNormalization()(conv7)
conv7 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv7)
conv7 = BatchNormalization()(conv7)
up2 = UpSampling2D((2,2))(conv7) # 28 x 28 x 32
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(up2) # 28 x 28 x 1
```

```
# compile the model
model1 = Model(input_layer, decoded)
model1.compile(optimizer='adam', loss='mse')
model1.summary()
```

Adam Optimizer for controlling learning rate and Mean squared error as loss function. We train the Denoising Autoencoder with 20 epochs and batch_size=4096.

Callbacks are used to get a view on internal states and statistics of the model during training. The latest best model according to the quantity monitored will not be overwritten.

```
from keras.callbacks import ModelCheckpoint
model_fname='model2'
callbacks = [ModelCheckpoint(monitor='val_acc', filepath=model_fname + '.hdf5',
                             save_best_only=True, save_weights_only=True,
                             mode='min')]
```

Constructed Noisy data (X_train_noisy) is provided as input to get the ouput as X_train (original Input Data).

```
# training the data to Denoising Autoencoder
autoencoder_train=model1.fit(X_train_noisy, X_train, epochs=20,batch_size = 128)
```

After 20 epoch loss has been reduced to 0.076.

```
Epoch 20/20
60000/60000 [==============================] - 34s 563us/step - loss: 0.0056
```

3. Get neuron outputs from DAE as new features

Using Output layer of the Denosing Autoencoder can be further connected as pretrained layer for training the related datsets. Layers of the autoencoder will be having learned weights which can converge faster and can further improvises the learning process.

3.1 Extracting the output layer using fuctional apis
Model for Denosing Autoencoder contains 33 layers with pretrained weights, whose last layer can be extracted out as below and can be connected to any model which will eventaully transfer its learning to the new model.

Also, layers are kept trainable so that weight of pretrained model can be modify at run time. We have tested that while keeping layers trainable provides us more accuracy comapred to keeping it freezable.

```
for i in range(33):
    model1.layers[i].trainable = True

ll = model1.layers[33].output
```

Task 2: Training Fashion MNIST model:

2.1 Fashion MNIST Dataset
Fashion-MNIST dataset is a 28x28 grayscale images of 70,000 fashion products from 10 categories, with 7,000 images per category. The training set has 60,000 images, and the test set has 10,000 images. Fashion-MNIST is a replacement for the original MNIST dataset for producing better results, the image dimensions, training and test splits are similar to the original MNIST dataset.

2. 2 Model Architecture for training Fashion MNIST
It contains four layers mainly input layer, flatten layer, followed by dense layer and output layer.

- First layer, Input layer takes 28*28 image as an input
- Second Layer, flatten layer is have the shape that is equal to the number of elements contained in tensor non including the batch dimension which has 784 as output shape.
- 2- Dense layer contains 64 neurons each with ‘relu’ as activation function.
- Output layer contains 10 neurons represents category of each input type and uses ‘softmax’ as activation function.

Model has been trained with batch size 128 till 20 epochs. Learning rate is controlled using ‘adam’ optimizer.

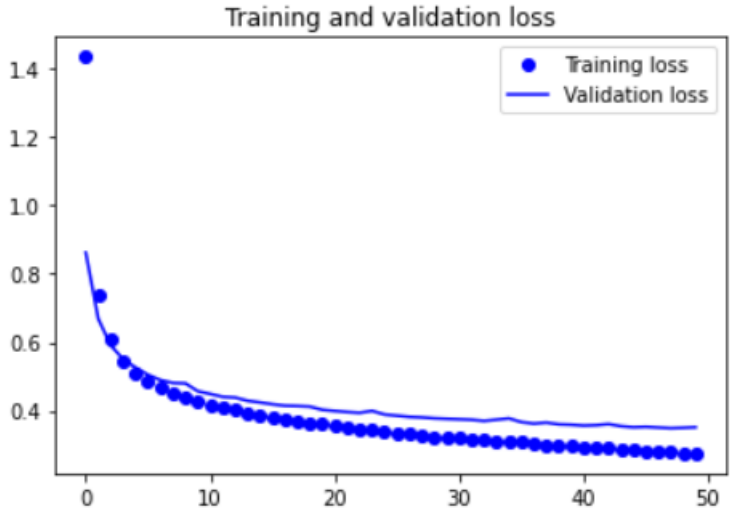
```
input_layer = Input(shape=(28,28,1))

flatten = Flatten()(input_layer)
dense1 = Dense(64, activation='relu')(flatten)
dense2=Dense(64, activation='relu')(dense1)
output = Dense(10, activation='softmax')(dense2)
new_model = Model(inputs=input_layer,outputs=output)
new_model.summary()
```

2.3 Training and Validation accuracy – loss

After 50 epochs 92.21% training accuracy while 88.24% accuracy has been obtained as below:

```
Epoch 20/20
60000/60000 [==============================] - 2s 28us/step - loss: 0.2091 - accuracy: 0.9215 - val_loss: 0.3431 - val_accuracy: 0.8824
```

It can be seen from below figure that training and validation loss are close to each other implying that our model is not overfitting.



Task 3: Transfer learning from denosing autoencoder model to Fashion MNISt model

Data loading and preprocessing task will be similar to task1 and task2.

3.1 Architecture:

From the task1, layer33 (output layer for Denoising autoencoder) has been extracted and connected to the model used in the task2. It will results into 37 layers which includes 33 layers from pretrained denoising autoencoder while 4 additional layer from model used in task2.

```
num_classes = 10
for i in range(33):
    model1.layers[i].trainable = True

ll = model1.layers[33].output

flatten = Flatten()(ll)

dense1 = Dense(64, activation='relu')(flatten)
dense2=Dense(64, activation='relu')(dense1)
output = Dense(10, activation='softmax')(dense2)
model3 = Model(inputs=model1.input,outputs=output)
model3.summary()
```

3.2 Training and Validation accuracy – loss

Training accuracy of the model is recorded as 98.42% while validation accuracy as 92.68% after 20 epochs with batch size 128.

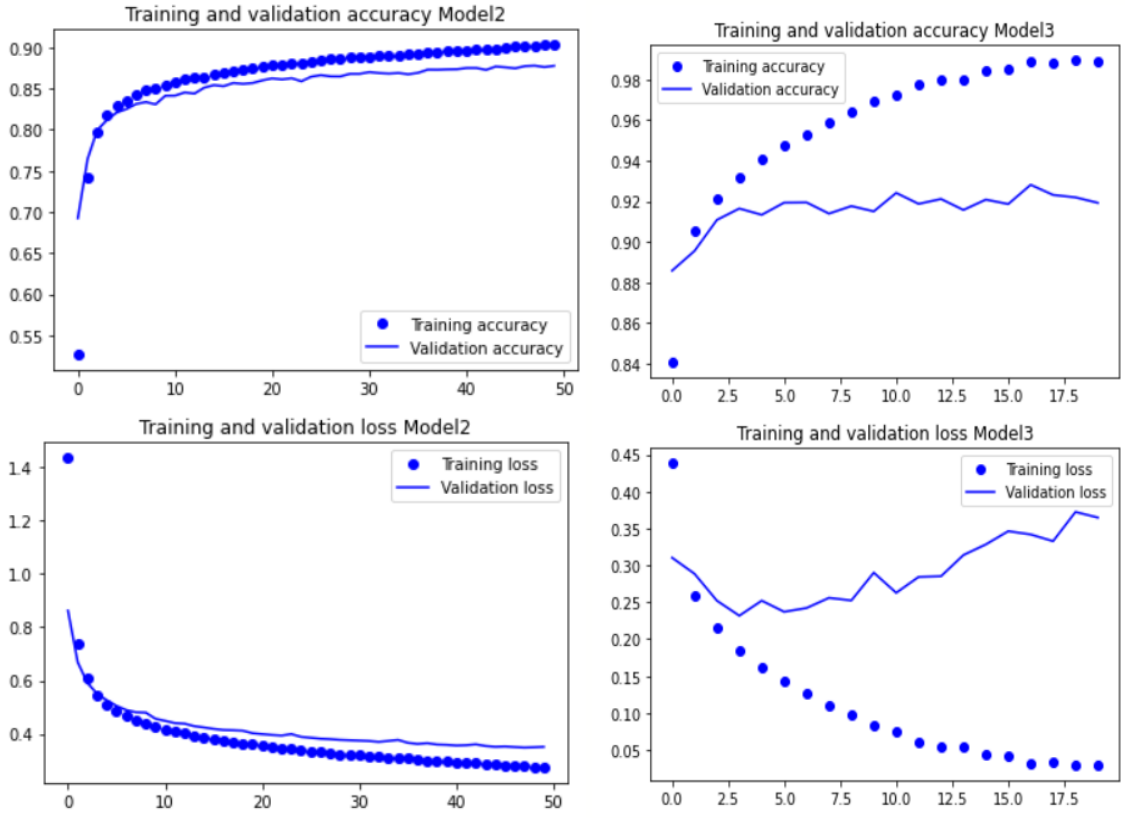
Task 4: Performance Comparision of Task2 (P, Model2)and Task3(P0, Model3):

Model2→ model used in Task2 without transfer learning
Model3→ model used in Task3 with transfer learning

Compariosn between Tranining - validation accuracy as well Tranining – validation Loss can be seen in the below figure.

It can be clearly observed that both tanining and Validation Accuracy is higher in Model3 compared to Model2 while opposite to it tanining and Validation Loss is higher in Model2.

Also, below figure shows closer gap between accuracy and loss in case of model2 while wider gaps in case of model3 which denoted that *model3 is being overfitted***.**



Approach2

In Approach2 we have copied all the weights of pretrained model to new model keeping the architecture same. We have used dense layer in this architecture for Denosing autoencoder as well as for model.

All the three Models used in the task have same architecutre, they differs in the wayy how they use prelearned weight and randomly initialiazed weight.

Denosing Architecture

It contains Input layer followed by three Dense layer act as encoding and decoding layers and last layer as output layer.

```
def DEEP_DAE(features_shape, act='relu'):

    # Input
    x = Input(name='inputs', shape=features_shape, dtype='float32')
    o = x

    # Encoder / Decoder
    o = Dense(1024, activation=act, name='dense1')(o)
    o = Dense(1024, activation=act, name='dense2')(o)
    o = Dense(1024, activation=act, name='dense3')(o)
    dec = Dense(784, activation='sigmoid', name='dense_dec')(o)

    # Print network summary
    Model(inputs=x, outputs=dec).summary()

    return Model(inputs=x, outputs=dec)
```

Using Output of Denosing layers for feature engineering of the task1

Here all three layers can be used to do the feature engineering, we have used output layer to further train the model.

```
def FEATURES(model):
    input_ = model.get_layer('inputs').input
    feat1 = model.get_layer('dense1').output
    feat2 = model.get_layer('dense2').output
    feat3 = model.get_layer('dense3').output
    # feat1, feat2,
    feat4 = model.get_layer('dense_dec').output
    # feat = Concatenate(name='concat')([feat1,feat2,feat3])
    feat=feat4
    model = Model(inputs=[input_],
                  outputs=[feat])
    return model
```

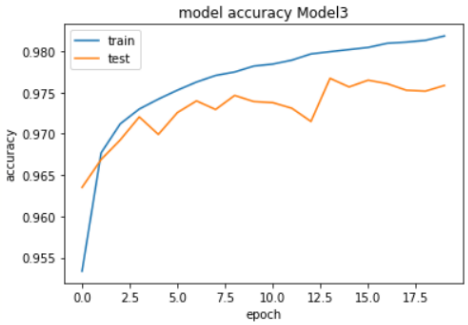
Model Architecture

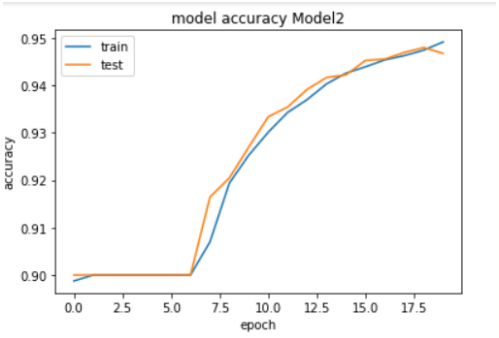
This Architecture is maintained same across all three task. Model1 (task1) uses Denosing output as input for this model while Model2 (task2) takes training data directly while Third model uses the pretrained weights learned from model 1.

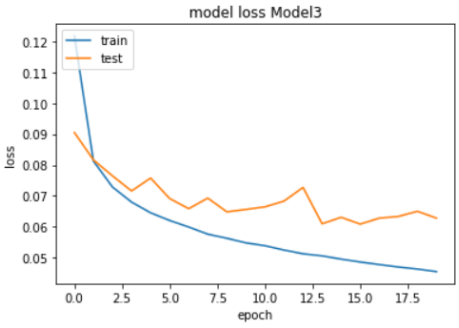
```python
def DNN(features_shape, num_classes, act='relu'):

    # Input
    x = Input(name='inputs', shape=features_shape, dtype='float32')
    o = x

    # Encoder / Decoder
    o = Dense(64, activation=act, name='dense1')(o)
    o = Dense(64, activation=act, name='dense2')(o)
    o = Dense(64, activation=act, name='dense3')(o)
    y_pred = Dense(num_classes, activation='sigmoid', name='pred')(o)

    # Print network summary
    Model(inputs=x, outputs=y_pred).summary()

    return Model(inputs=x, outputs=y_pred)
```

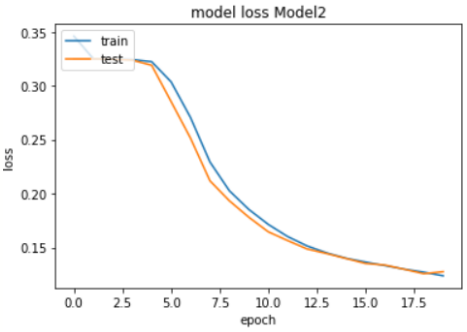
Performance Comparision (Approach 2)

It has been obserbed that Pretrained model has performed really well with validation accuracy of 97.58% while non-pretrained model has validation accuracy of 94.67%









4. Conclusion:

1. Denoising Autoencoders can be used to learn superior representation of data. Supplying noisy version of data, forces the Autoencoder to perform better than its clean input

counterpart and as a consequence it produces representation of data that is immune to random noise.

2. Accuracy of the model can be improved with pretrained models. Pretrained weights can significantly improve the performace and can converge the model faster if choosen carefully.

3. Sometime, Pretrained model can overfit the model which results in large gap between validation and training loss / Accuracy but if trained carefully they gives good accuracy without being overfit the data.

5. References:

- https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a
- https://machinelearningmastery.com/transfer-learning-for-deep-learning/
- http://yann.lecun.com/exdb/mnist/
- https://www.kaggle.com/zalando-research/fashionmnist
- https://keras.io/layers/core/
- https://dkopczyk.quantee.co.uk/dae-part3/
- http://www.jmlr.org/ papers/v11/erhan10a.html
- https://keras.io/layers/convolutional/