Teammates:

Sukumar Bodapati – sb5zh@umsystem.edu - 16326105

GitHub link: https://github.com/UMKC-APL-BigDataAnalytics/icp-7-sukumarbodapati

Sri Nikhitha Boddapati – sb4dz@umsystem.edu - 16322565

GitHub link: https://github.com/UMKC-APL-BigDataAnalytics/icp-7-Srinikhitha98

Video link: https://youtu.be/dtZ7U2euqlI

ICP - Big Data App & Analytics

Learnings in this ICP:

In this ICP, we have learned about Autoencoder which is a dimensional reduction technique used for unsupervised learning. It has two parts Encoder and Decoder. Encoder converts the given input to the latent representation of the input (i.e. compress the input) and decoder reforms the input again.

Difference between CNN (that we used as part of ICP3) and Auto-encoder:

In CNN and auto-encoder, we use same type of layers like Conv2d, Batch normalization. In CNN, we build a model using this layer and predict the labels for different datasets whereas in the encoder we compress the input data i.e., they work on the specific data. Auto-encoder can be trained in an unlabeled way. The architecture of this is symmetrical, we need use the same layer that we used as part of encoder in decoder as well.

Variational autoencoder (VAE):

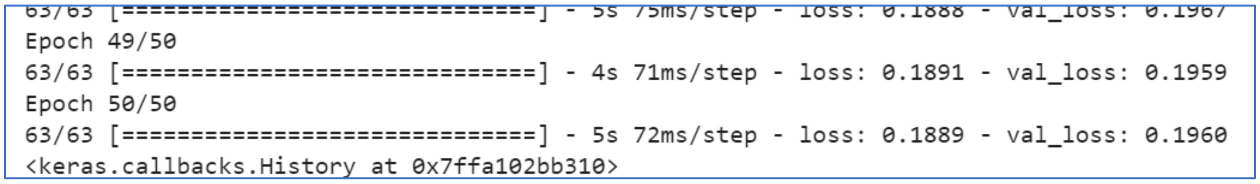
It is probabilistic manner for describing an observation in latent space. Rather than creating an encoder that outputs a single number to describe each latent state characteristic, we'll create an encoder that describes a probability distribution. In this ICP, we have calculated vae outputs with encoder and decoder, build the model with them and visualized the output.

Observations in this ICP:

Used the same encoder and decoder that is given in Source Code

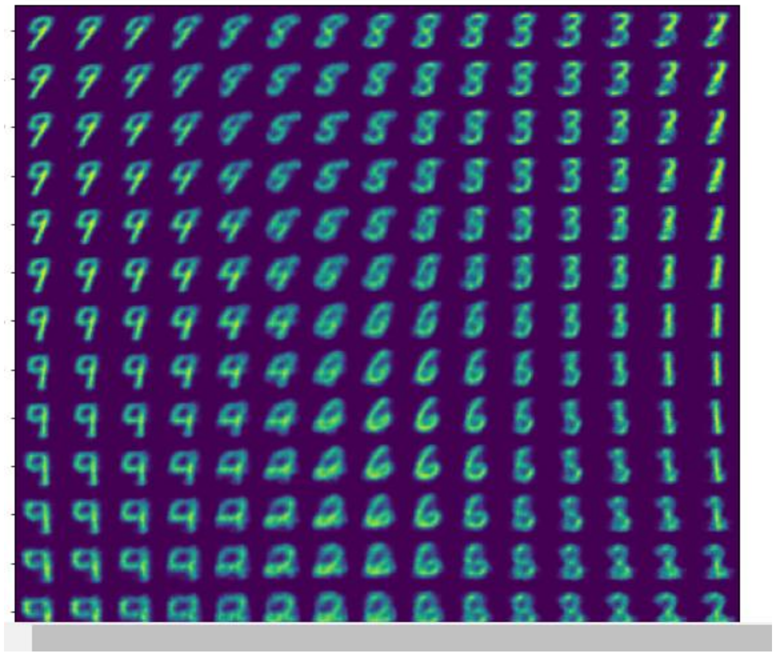
Initially the code has two Conv2D layers with filter 8 and 16along with batch normalization layer. Here are the results.

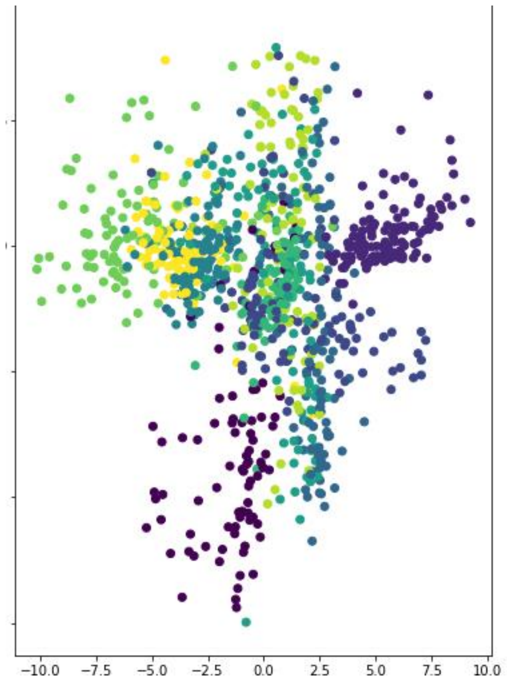
After training the auto-encoder output, the loss is 0.1889.



```
63/63 [==============================] - 5s 75ms/step - loss: 0.1888 - val_loss: 0.1967
Epoch 49/50
63/63 [==============================] - 4s 71ms/step - loss: 0.1891 - val_loss: 0.1959
Epoch 50/50
63/63 [==============================] - 5s 72ms/step - loss: 0.1889 - val_loss: 0.1960
<keras.callbacks.History at 0x7ffa102bb310>
```

Data visualization:





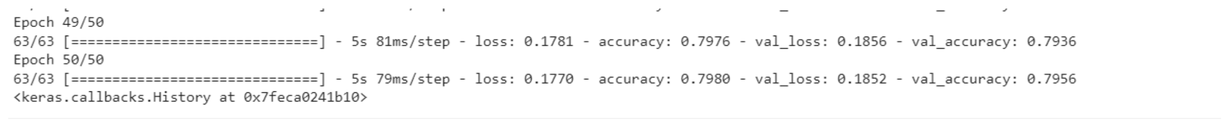
1. Added a Conv2D layer (32) and Batch Normalized:

1.Conv2D layer: Added the conv2d layer with 32 filter, kernel size =3 and stride as 1 and padding as same.

2.Batch Normalization: It acts as a regularize.

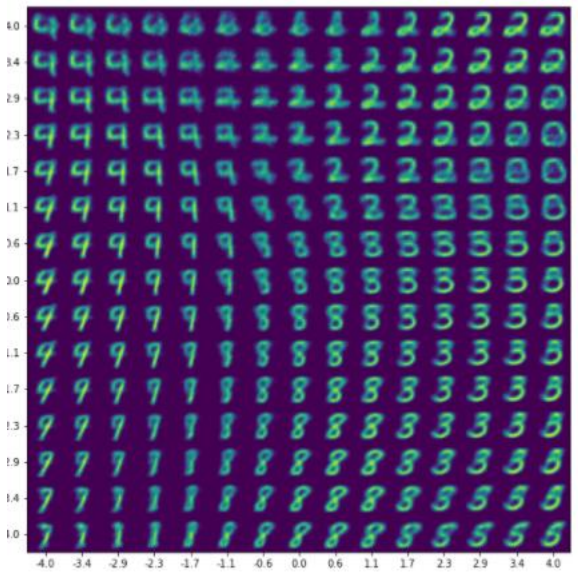
3.Similarly, added Conv2D Transpose to the decoder with 32 filter, kernel size =3 and stride as 1 and padding as same.

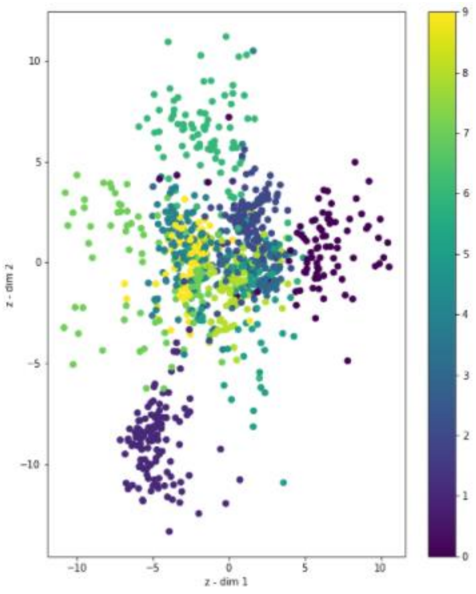
After training the auto-encoder output, **the loss is 0.1770 and accuracy is 0.7956.**



```
Epoch 49/50
63/63 [==============================] - 5s 81ms/step - loss: 0.1781 - accuracy: 0.7976 - val_loss: 0.1856 - val_accuracy: 0.7936
Epoch 50/50
63/63 [==============================] - 5s 79ms/step - loss: 0.1770 - accuracy: 0.7980 - val_loss: 0.1852 - val_accuracy: 0.7956
<keras.callbacks.History at 0x7feca0241b10>
```

Data visualization:





When compared to results which we got when we used the given source code with these results, the output data is having more accuracy and the data is more clear.

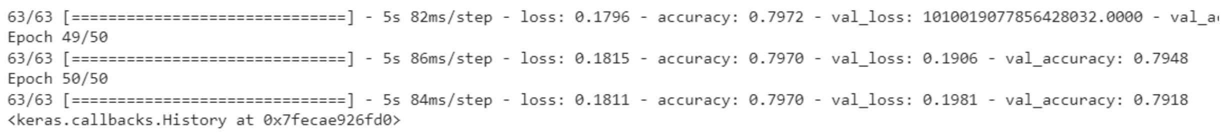
2. Added a Conv2D layer (64) and Batch Normalized:

1.Conv2D layer: Added the conv2d layer with 64 filter, kernel size =3 and stride as 1 and padding as same.

2.Batch Normalization: It acts as a regularize.

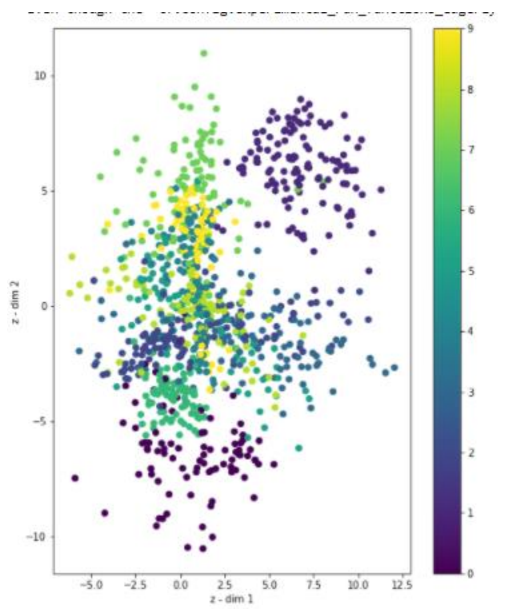
3.Similarly, added Conv2D Transpose to the decoder with 64 filter, kernel size =3 and stride as 1 and padding as same.

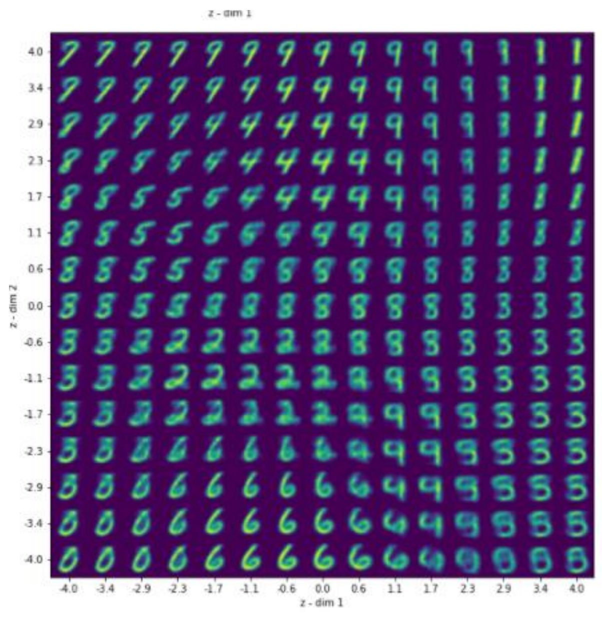
After training the auto-encoder output, the loss **is 0.1811 and accuracy is 0.7918.**



```
63/63 [==============================] - 5s 82ms/step - loss: 0.1796 - accuracy: 0.7972 - val_loss: 1010019077856428032.0000 - val_a
Epoch 49/50
63/63 [==============================] - 5s 86ms/step - loss: 0.1815 - accuracy: 0.7970 - val_loss: 0.1906 - val_accuracy: 0.7948
Epoch 50/50
63/63 [==============================] - 5s 84ms/step - loss: 0.1811 - accuracy: 0.7970 - val_loss: 0.1981 - val_accuracy: 0.7918
<keras.callbacks.History at 0x7fecae926fd0>
```

Data visualization:





When we use the layer with 64 filter, the data is not much clear.

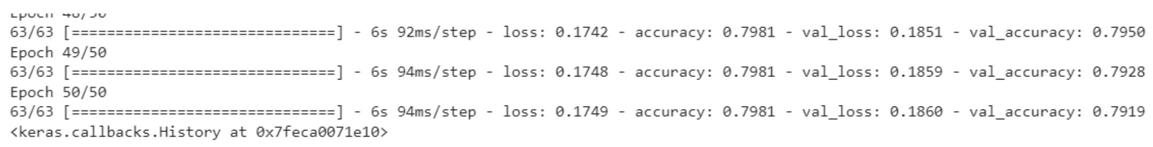
3.Double filter: Added a Conv2D layers (64 ,32) and Batch Normalized:

1.Conv2D layer: Added the conv2d layer with 64 filter and 32 filter, kernel size =3 and stride as 1 and padding as same.

2.Batch Normalization: It acts as a regularize.

3.Similarly, added Conv2D Transpose to the decoder with 64 filter and 32 filter, kernel size =3 and stride as 1 and padding as same.

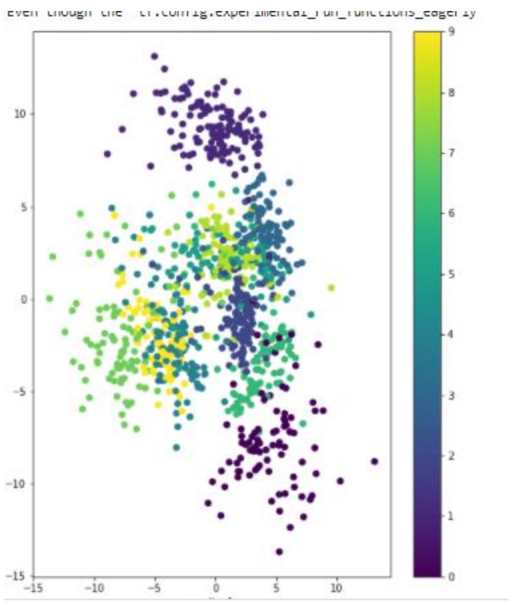
After training the auto-encoder output**, the loss is 0.1749 and accuracy is 0.7919.**

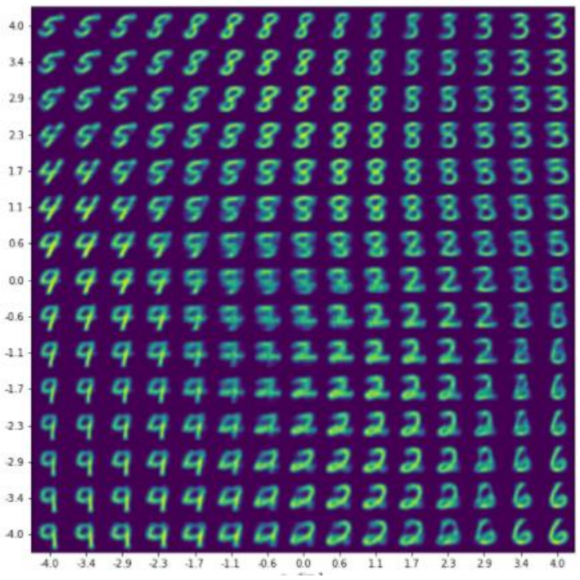


```
Epoch 48/50
63/63 [==============================] - 6s 92ms/step - loss: 0.1742 - accuracy: 0.7981 - val_loss: 0.1851 - val_accuracy: 0.7950
Epoch 49/50
63/63 [==============================] - 6s 94ms/step - loss: 0.1748 - accuracy: 0.7981 - val_loss: 0.1859 - val_accuracy: 0.7928
Epoch 50/50
63/63 [==============================] - 6s 94ms/step - loss: 0.1749 - accuracy: 0.7981 - val_loss: 0.1860 - val_accuracy: 0.7919
<keras.callbacks.History at 0x7feca0071e10>
```

When we used multiple layers, the output data is good.

Data visualization:





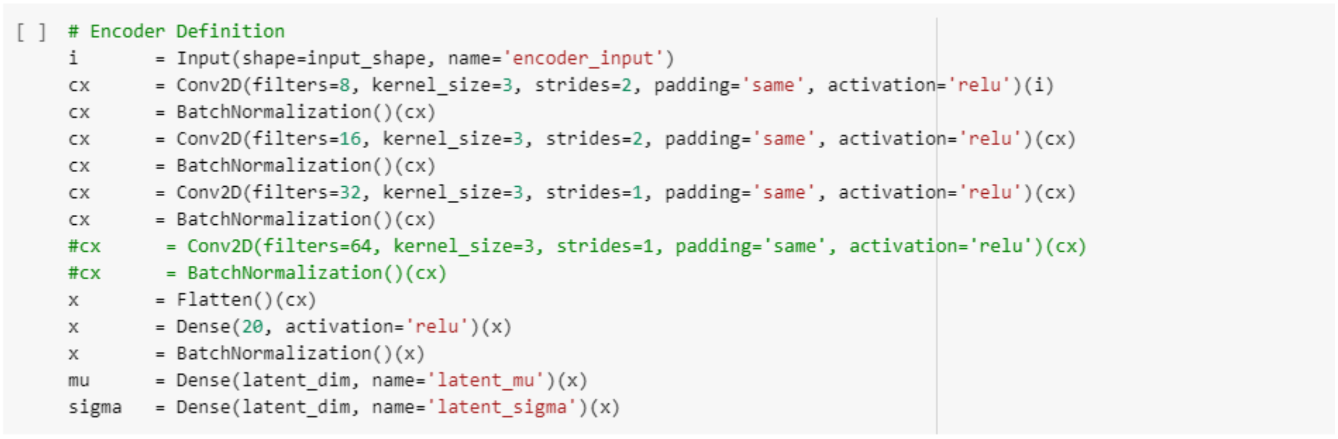
Observation:

When compared to original source code when we added conv2d layer with 32 filter, we got more accuracy, and the data is very clear. Even though when we used multiple layer (i.e two conv2d layer with 32 and 64 filter), the output is not very good. We can say by this ,when the depth (i.e layers increases)is high, the model complexity increases and when the depth is lower the process is fast.

Code:

Encoder:

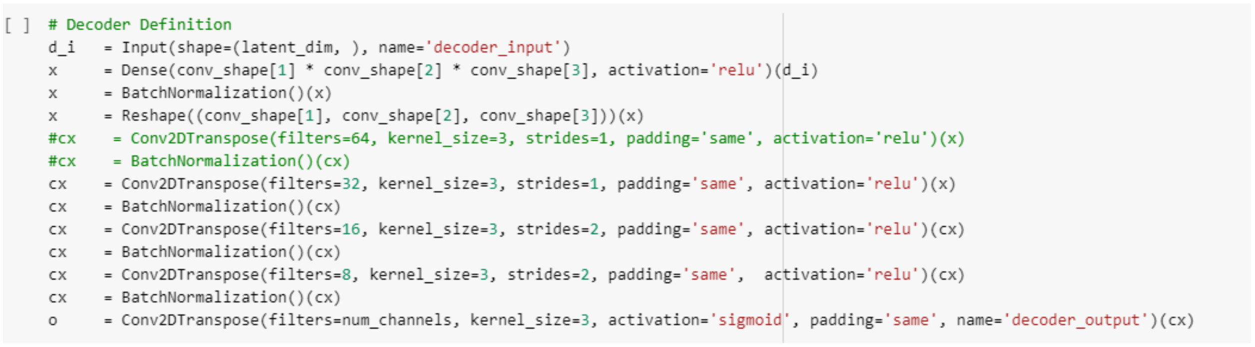
Used Conv2D layer with filter as 32 and batch normalization.



```
[ ]  # Encoder Definition
     i       = Input(shape=input_shape, name='encoder_input')
     cx      = Conv2D(filters=8, kernel_size=3, strides=2, padding='same', activation='relu')(i)
     cx      = BatchNormalization()(cx)
     cx      = Conv2D(filters=16, kernel_size=3, strides=2, padding='same', activation='relu')(cx)
     cx      = BatchNormalization()(cx)
     cx      = Conv2D(filters=32, kernel_size=3, strides=1, padding='same', activation='relu')(cx)
     cx      = BatchNormalization()(cx)
     #cx      = Conv2D(filters=64, kernel_size=3, strides=1, padding='same', activation='relu')(cx)
     #cx      = BatchNormalization()(cx)
     x       = Flatten()(cx)
     x       = Dense(20, activation='relu')(x)
     x       = BatchNormalization()(x)
     mu      = Dense(latent_dim, name='latent_mu')(x)
     sigma   = Dense(latent_dim, name='latent_sigma')(x)
```

Decoder:

Used Conv2DTranspose with filter as 32 and batch normalization



```
[ ]  # Decoder Definition
     d_i   = Input(shape=(latent_dim, ), name='decoder_input')
     x     = Dense(conv_shape[1] * conv_shape[2] * conv_shape[3], activation='relu')(d_i)
     x     = BatchNormalization()(x)
     x     = Reshape((conv_shape[1], conv_shape[2], conv_shape[3]))(x)
     #cx    = Conv2DTranspose(filters=64, kernel_size=3, strides=1, padding='same', activation='relu')(x)
     #cx    = BatchNormalization()(cx)
     cx    = Conv2DTranspose(filters=32, kernel_size=3, strides=1, padding='same', activation='relu')(x)
     cx    = BatchNormalization()(cx)
     cx    = Conv2DTranspose(filters=16, kernel_size=3, strides=2, padding='same', activation='relu')(cx)
     cx    = BatchNormalization()(cx)
     cx    = Conv2DTranspose(filters=8, kernel_size=3, strides=2, padding='same',  activation='relu')(cx)
     cx    = BatchNormalization()(cx)
     o     = Conv2DTranspose(filters=num_channels, kernel_size=3, activation='sigmoid', padding='same', name='decoder_output')(cx)
```

Challenges Faced:

1.The structure of encoder and decoder is symmetrical, so we need to use the same layers that we used in encoder in decoder as well to reframe the data back.

2. If the depth of the encoder is increased, the model is becoming more complex and taking more time. When we have low depth, it is faster.