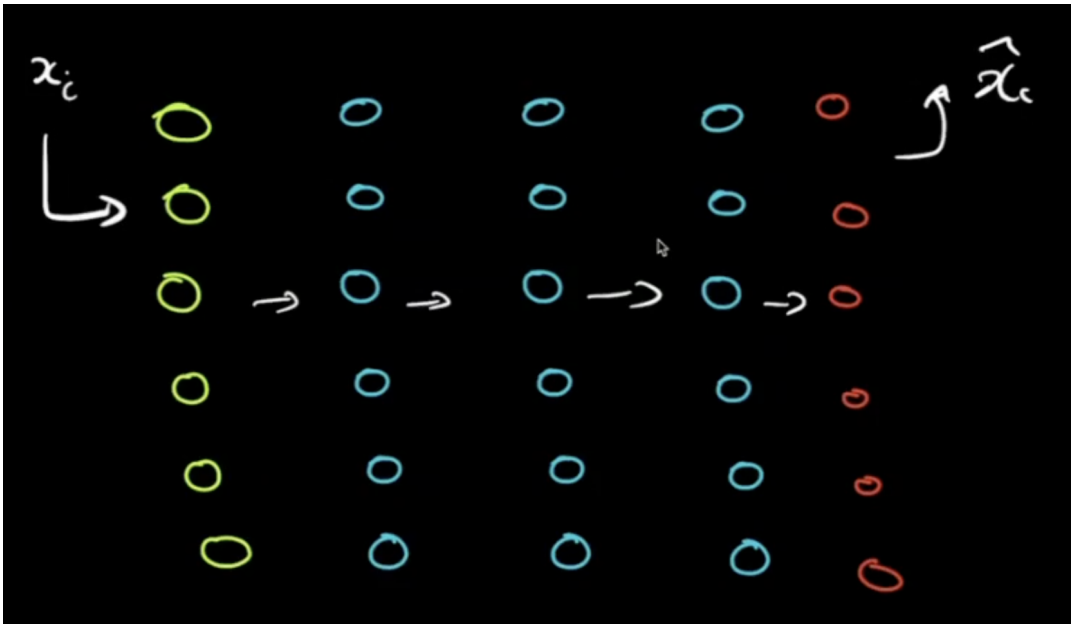
## Content

- **Autoencoders Intuition**
- **Applications of AE**
- **Feature Extraction and Transfer Learning**
- **Denoising AE**
- **Recommender Sys using AE**
- Assessments
  - Training AutoEncoders
  - AutoEncoders
  - Denoising AutoEncoders

## Intuition

Consider the following network




Notice that

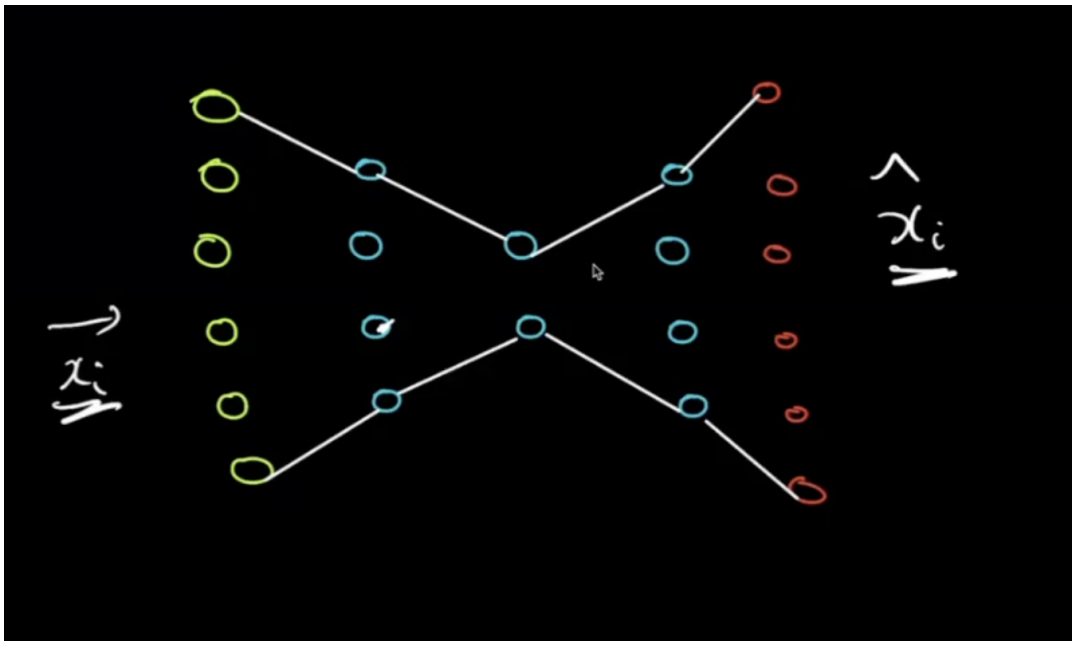
- all the layers have same number of nuerons including the input and output layer
- the input to the network is $x_i$ and output is $\hat{x_i}$

### Question: We want to feed in $x_i$ to network and get output $\hat{x_i}$ such that $x_i \simeq \hat{x_i}$. Is it possible ?

Yes. Just make all the activation as linear

- i.e. whatever gets in gets out
- and train the n/w to find weights and biases

Now, let's introduce a little complexity to our network



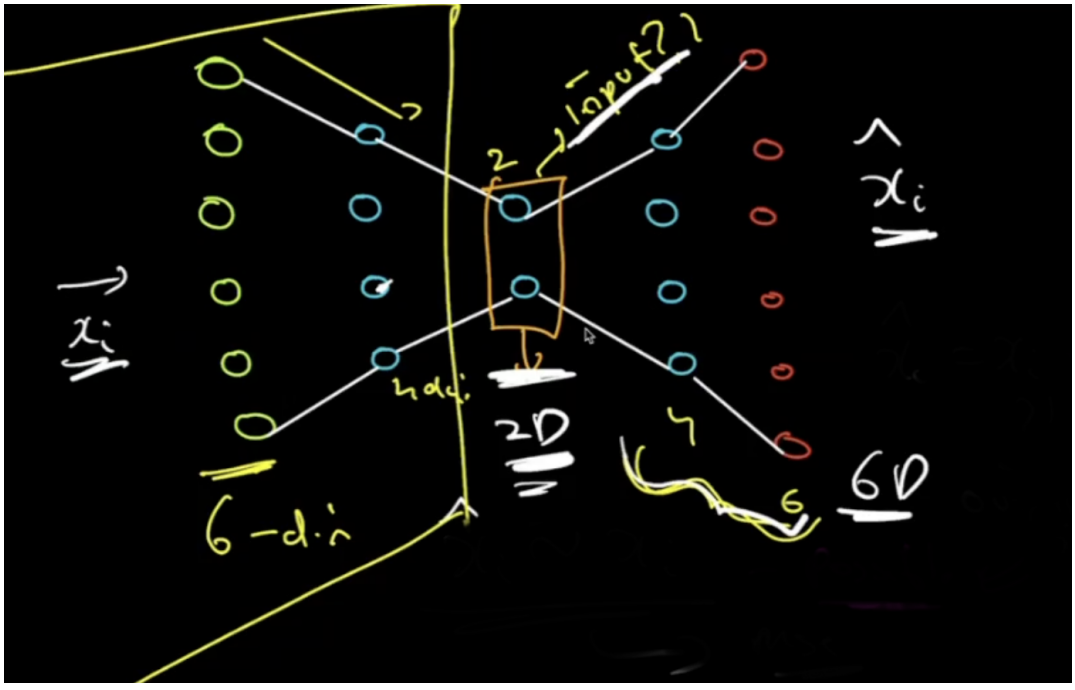

- Question: For this network, can we make $x_i = \hat{x_i}$ ?

  The task is possible. We can make a prediction s.t. $x_i \sim \hat{x_i}$

  - but it can't be perfectly same

  We can decide some loss based on the problem

  - if the values are binary, we can use binary cross entropy
  - for multiclass, we can use categorical cross entropy
  - for continuous values, we can use MSE or RMSE.




- What exactly happened in this network ?

  Out input was 6 dimension

  - it went to 4 dimension
  - and then 2 -> 4 -> 6 .

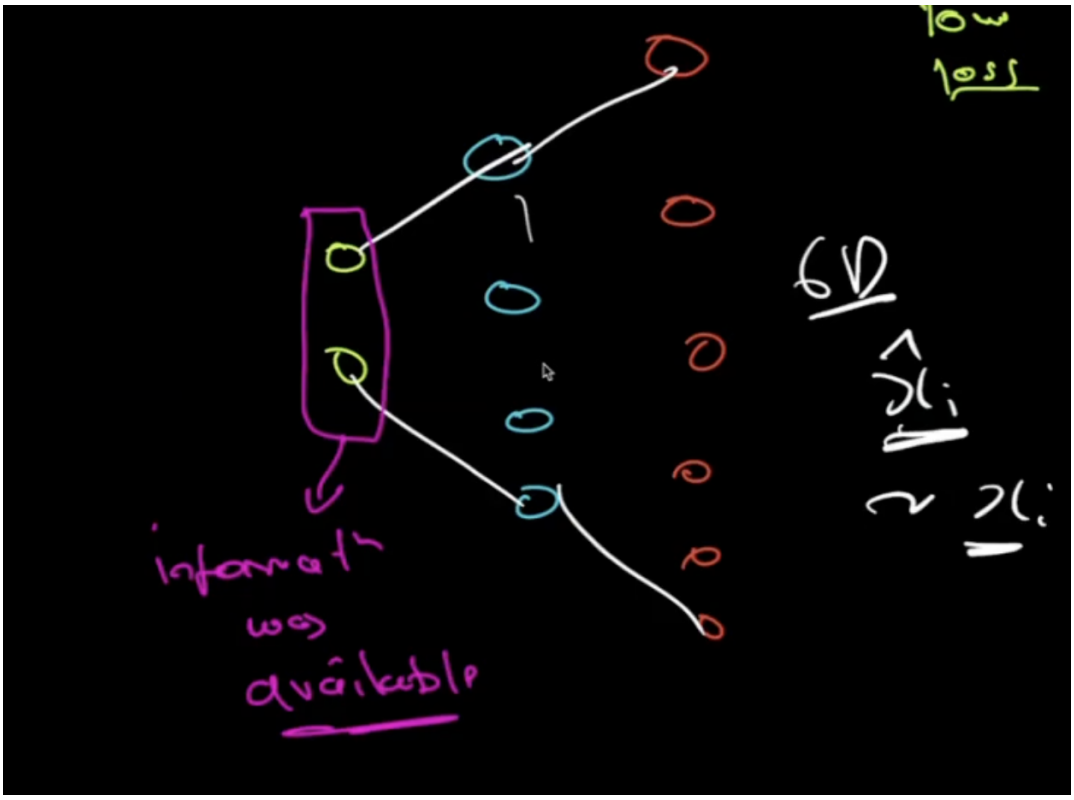
  Let's focus on the layer in the middle i.e. layer with 2 neuron

  If we were to ignore the network before it,

- we can say that the this middle layer (2 neuron layer) is the input for next layers

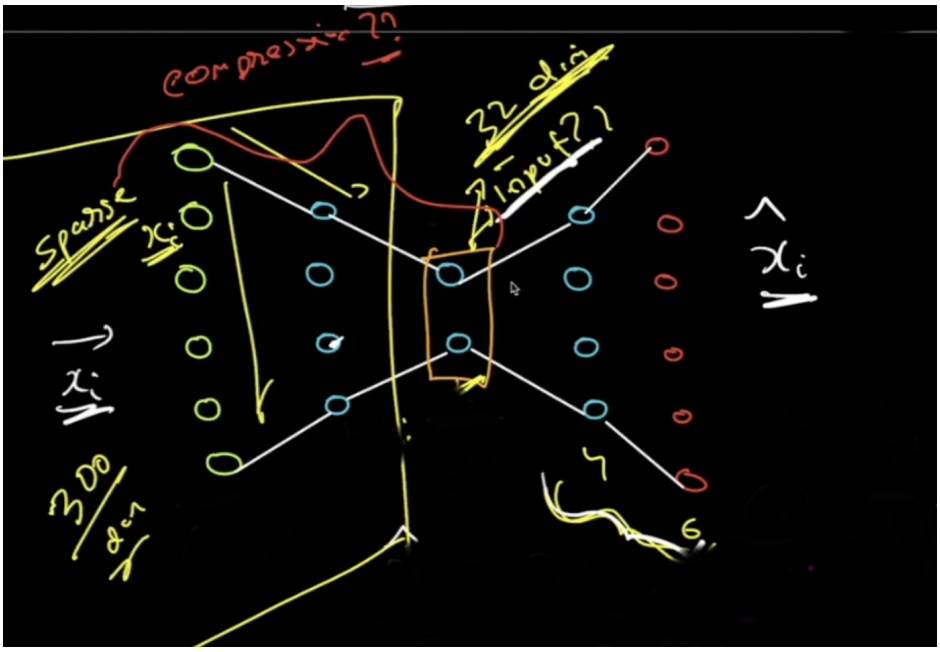
In other words,

- 2D input gave us 6D output




If we are able to produce 6D output ($\hat{x_i}$) which similar to $x_i$, it means

- all the information available to produce this was available.




Now, imagine instead of 6-D input vector

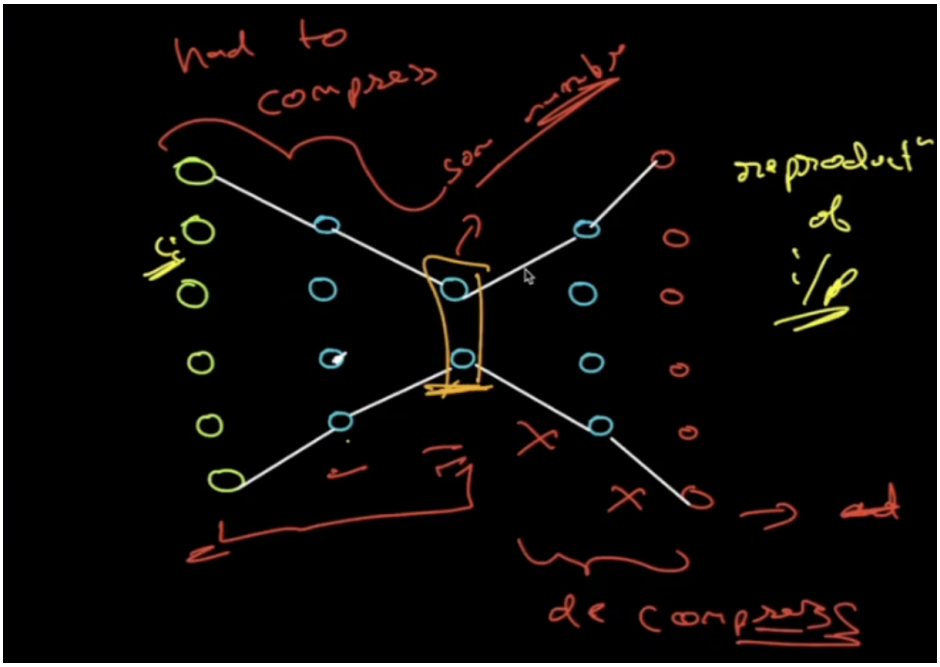
- We have a sparse vector of 300-dim.

And using the network,

- we step it down to 32 dim till middle layer

## Doesn't it look like we are compressing our data ?

We are basically reducing the dimensions of our input.




So, in summary,

- the left part of network is trying to compress the information i.e. **encoder**
- the right part of netowork is decompressing it i.e. **decoder**

And this had to happen so meaningfully s.t.

- all the info would be stored at middle layer.

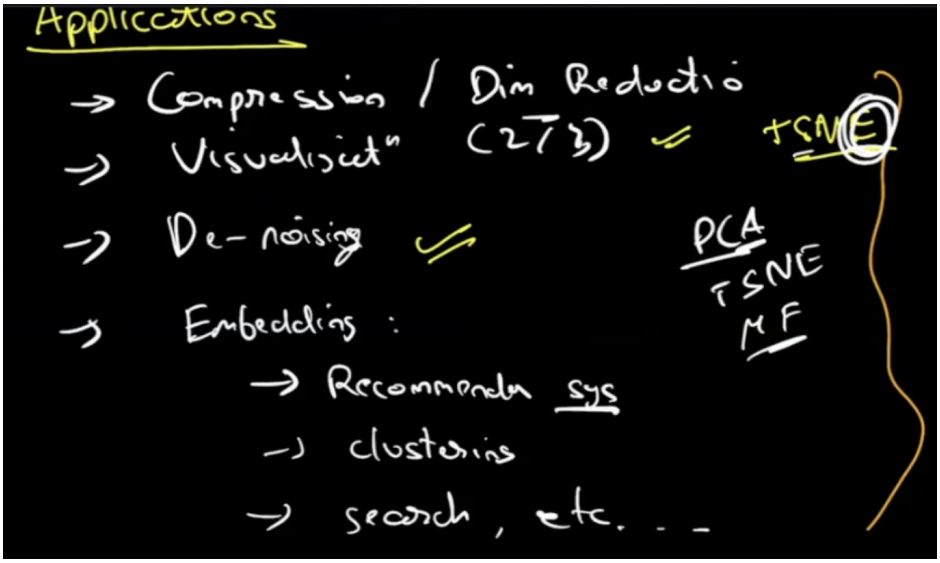
This means we can use middle layer (bottleneck layer) as an **embedding/encoding or latent features**

- i.e. we can use this bottleneck as features.

And this network is known as **Autoencoder**

The purpose of the network is to get the embedding

## Applications




**1. Dimensionality Reduction/ Compression**

Here, compression doesn't mean reducing the space

- but to reduce the number of features of input vectors

Reducing the dimensions means

- faster inference time
- low latency

We can also use these embeddings for visualization

**2. Denoising**

We can use AE (Autoencoders) to denoise the data

**3. Embeddings**

We can use AE to generate embeddings

- These embeddings can be used for
  - recommender system
  - clustering
  - image search

## Dimensionality reduction

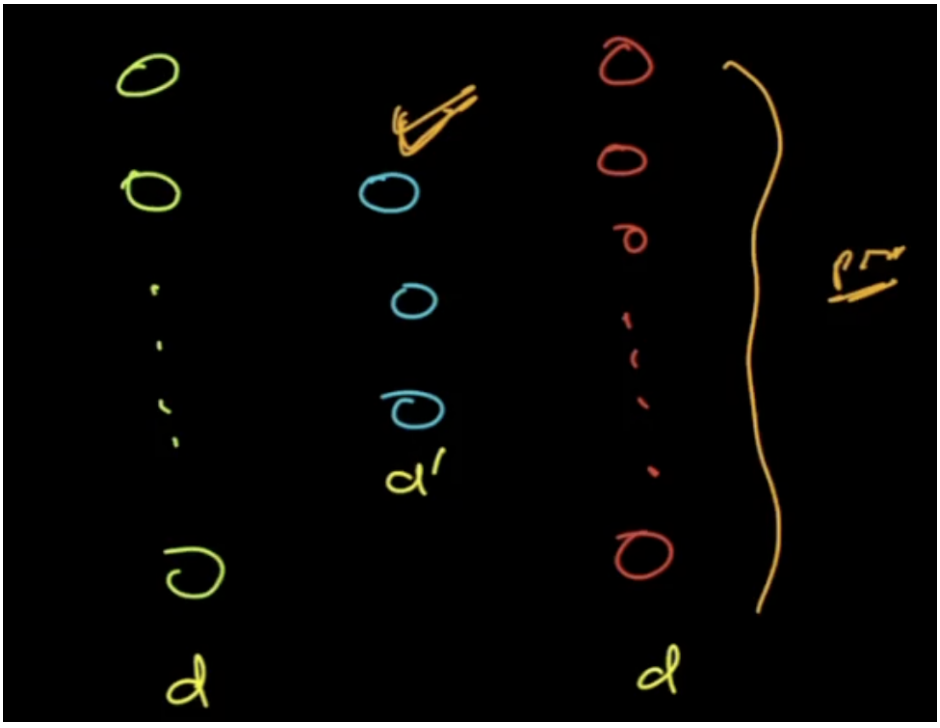
### Question: Can we create a PCA equivalent network ?

Say, we have

- d dim input data

And we want to go to d' dim. (d' < d)

So, the network would like :



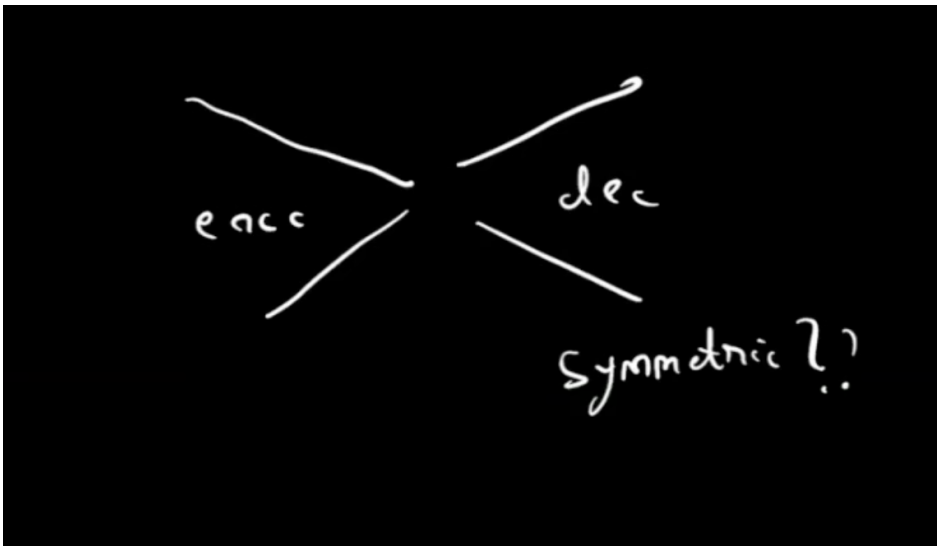

Do note that

- we don't really need the prediction

We need the encoding coz

- the fact the we can go back from encoding means that
- encoding must have all that information to recreate the original vector.

### Question: Is it required for encoder and decoder to be symmetric ?




It doesn't really have to be symmetric

- Earlier, we used to keep the it symmertric
  - i.e. same number of layer and same number of neuron
- Also, weight were shared (tying weights) between encoder and decoder.
  - so as to reduce the number of parameters

There is no more necessary for that and we can have separate weights for enoder and decoder

## Code walkthrough - Dimensionality-Reduction (using AE)

```
#Source and Reference: https://blog.keras.io/building-autoencoders-in-keras.html
```

```
import keras
from keras import layers
from keras.datasets import mnist
import numpy as np
```

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

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

#Reshaping the images to 1D vectors
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

```
print(y_train.shape)
print(y_test.shape)
```

```
(60000,)
(10000,)
```

```
#AutoEncoder model - Functional
input_img = keras.Input(shape=(784,))
encoded = layers.Dense(128, activation='relu')(input_img)
encoded = layers.Dense(64, activation='relu')(encoded)
encoded = layers.Dense(32, activation='relu')(encoded)
```

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

```
autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=100,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

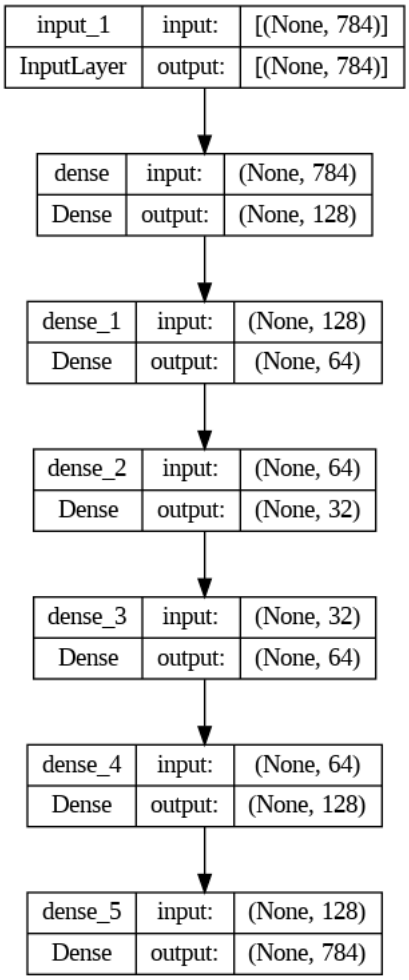
```
Epoch 1/100
235/235 [==============================] - 12s 11ms/step - loss: 0.2422 - val_loss: 0.1627
Epoch 2/100
235/235 [==============================] - 2s 9ms/step - loss: 0.1465 - val_loss: 0.1333
Epoch 3/100
235/235 [==============================] - 2s 9ms/step - loss: 0.1292 - val_loss: 0.1227
Epoch 4/100
235/235 [==============================] - 3s 11ms/step - loss: 0.1208 - val_loss: 0.1173
Epoch 5/100
235/235 [==============================] - 2s 8ms/step - loss: 0.1152 - val_loss: 0.1110
Epoch 6/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1106 - val_loss: 0.1075
Epoch 7/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1072 - val_loss: 0.1044
Epoch 8/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1046 - val_loss: 0.1030
Epoch 9/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1023 - val_loss: 0.1004
Epoch 10/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1004 - val_loss: 0.0988
Epoch 11/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0989 - val_loss: 0.0974
Epoch 12/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0975 - val_loss: 0.0957
Epoch 13/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0962 - val_loss: 0.0948
Epoch 14/100
235/235 [==============================] - 2s 7ms/step - loss: 0.0951 - val_loss: 0.0939
Epoch 15/100
235/235 [==============================] - 2s 7ms/step - loss: 0.0940 - val_loss: 0.0930
Epoch 16/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0932 - val_loss: 0.0919
Epoch 17/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0924 - val_loss: 0.0913
Epoch 18/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0918 - val_loss: 0.0907
Epoch 19/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0912 - val_loss: 0.0901
Epoch 20/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0907 - val_loss: 0.0908
Epoch 21/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0903 - val_loss: 0.0893
Epoch 22/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0898 - val_loss: 0.0889
Epoch 23/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0894 - val_loss: 0.0885
Epoch 24/100
235/235 [==============================] - 2s 7ms/step - loss: 0.0890 - val_loss: 0.0881
Epoch 25/100
235/235 [==============================] - 1s 6ms/step - loss: 0.0887 - val_loss: 0.0880
Epoch 26/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0883 - val_loss: 0.0873
Epoch 27/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0880 - val_loss: 0.0876
Epoch 28/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0877 - val_loss: 0.0870
Epoch 29/100
235/235 [==============================] - 1s 5ms/step - loss: 0.0874 - val_loss: 0.0868
```

```
autoencoder.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 784)]             0

 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 32)                2080

 dense_3 (Dense)             (None, 64)                2112

 dense_4 (Dense)             (None, 128)               8320
```

```
 dense_5 (Dense)             (None, 784)               101136

=================================================================
Total params: 222,384
Trainable params: 222,384
Non-trainable params: 0
_________________________________________________________________
```

```
from keras.utils.vis_utils import plot_model

plot_model(autoencoder, to_file='model_plot.png', show_shapes=True, show_layer_names=True)
```




```
#Visualize the outputs
import matplotlib.pyplot as plt

decoded_imgs = autoencoder.predict(x_test)

n = 10
plt.figure(figsize=(20, 4))
for i in range(1, n + 1):
    # Display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

```
313/313 [==============================] - 1s 2ms/step
```

## AutoEncoder model for 2D encoding

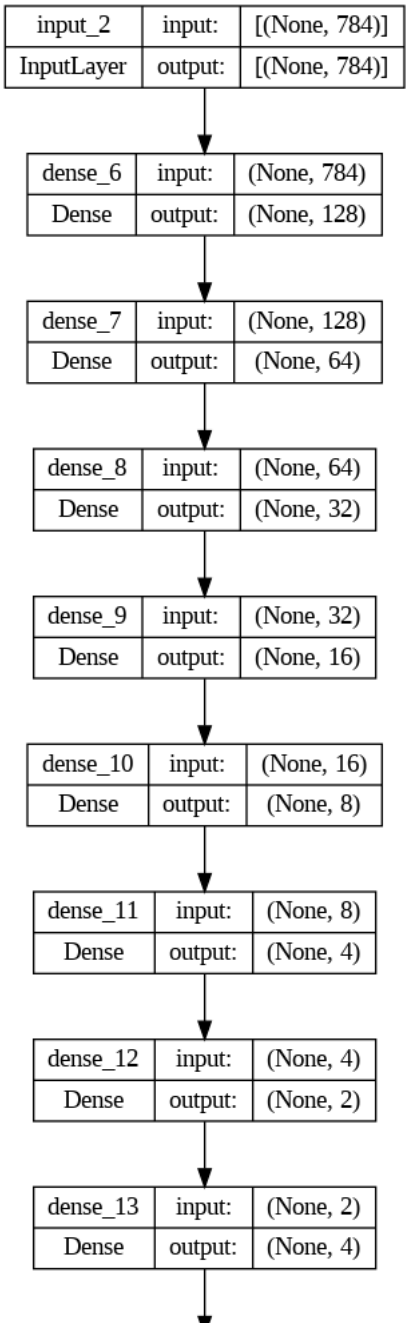
```
#AutoEncoder model for 2D encoding
input_img = keras.Input(shape=(784,))
encoded = layers.Dense(128, activation='relu')(input_img)
encoded = layers.Dense(64, activation='relu')(encoded)
encoded = layers.Dense(32, activation='relu')(encoded)
encoded = layers.Dense(16, activation='relu')(encoded)
encoded = layers.Dense(8, activation='relu')(encoded)
encoded = layers.Dense(4, activation='relu')(encoded)
encoded = layers.Dense(2, activation='relu')(encoded)

decoded = layers.Dense(4, activation='relu')(encoded)
decoded = layers.Dense(8, activation='relu')(decoded)
decoded = layers.Dense(16, activation='relu')(decoded)
decoded = layers.Dense(32, activation='relu')(decoded)
decoded = layers.Dense(64, activation='relu')(decoded)
decoded = layers.Dense(128, activation='relu')(decoded)
decoded = layers.Dense(784, activation='sigmoid')(decoded)
```

```
model = keras.Model(input_img, decoded)
model.compile(optimizer='adam', loss='binary_crossentropy')
```

```
model.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 784)]             0

 dense_6 (Dense)             (None, 128)               100480

 dense_7 (Dense)             (None, 64)                8256

 dense_8 (Dense)             (None, 32)                2080

 dense_9 (Dense)             (None, 16)                528

 dense_10 (Dense)            (None, 8)                 136

 dense_11 (Dense)            (None, 4)                 36

 dense_12 (Dense)            (None, 2)                 10

 dense_13 (Dense)            (None, 4)                 12

 dense_14 (Dense)            (None, 8)                 40

 dense_15 (Dense)            (None, 16)                144

 dense_16 (Dense)            (None, 32)                544

 dense_17 (Dense)            (None, 64)                2112

 dense_18 (Dense)            (None, 128)               8320

 dense_19 (Dense)            (None, 784)               101136

=================================================================
Total params: 223,834
Trainable params: 223,834
Non-trainable params: 0
_________________________________________________________________
```

```
plot_model(model, to_file='model.png', show_shapes=True, show_layer_names=True)
```


input_2 | input: | [(None, 784)]
InputLayer | output: | [(None, 784)]
dense_6 | input: | (None, 784)
Dense | output: | (None, 128)
dense_7 | input: | (None, 128)
Dense | output: | (None, 64)
dense_8 | input: | (None, 64)
Dense | output: | (None, 32)
dense_9 | input: | (None, 32)
Dense | output: | (None, 16)
dense_10 | input: | (None, 16)
Dense | output: | (None, 8)
dense_11 | input: | (None, 8)
Dense | output: | (None, 4)
dense_12 | input: | (None, 4)
Dense | output: | (None, 2)
dense_13 | input: | (None, 2)
Dense | output: | (None, 4)

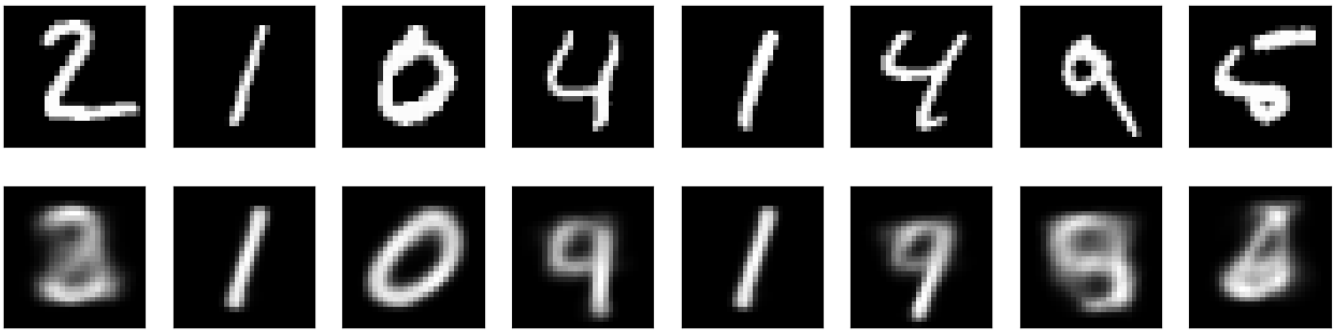
```
model.fit(x_train, x_train,
                epochs=100,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

```
Epoch 1/100
235/235 [==============================] - 6s 8ms/step - loss: 0.3483 - val_loss: 0.2397
Epoch 2/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2376 - val_loss: 0.2350
Epoch 3/100
235/235 [==============================] - 2s 10ms/step - loss: 0.2334 - val_loss: 0.2298
Epoch 4/100
235/235 [==============================] - 2s 8ms/step - loss: 0.2290 - val_loss: 0.2266
Epoch 5/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2262 - val_loss: 0.2238
Epoch 6/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2226 - val_loss: 0.2205
Epoch 7/100
235/235 [==============================] - 2s 6ms/step - loss: 0.2197 - val_loss: 0.2184
Epoch 8/100
235/235 [==============================] - 1s 6ms/step - loss: 0.2172 - val_loss: 0.2164
Epoch 9/100
235/235 [==============================] - 2s 6ms/step - loss: 0.2154 - val_loss: 0.2144
Epoch 10/100
235/235 [==============================] - 2s 8ms/step - loss: 0.2139 - val_loss: 0.2136
Epoch 11/100
235/235 [==============================] - 2s 9ms/step - loss: 0.2134 - val_loss: 0.2131
Epoch 12/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2124 - val_loss: 0.2121
Epoch 13/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2121 - val_loss: 0.2109
Epoch 14/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2109 - val_loss: 0.2099
Epoch 15/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2099 - val_loss: 0.2089
Epoch 16/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2092 - val_loss: 0.2083
Epoch 17/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2081 - val_loss: 0.2080
Epoch 18/100
235/235 [==============================] - 2s 9ms/step - loss: 0.2075 - val_loss: 0.2069
Epoch 19/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2070 - val_loss: 0.2063
Epoch 20/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2061 - val_loss: 0.2066
Epoch 21/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2075 - val_loss: 0.2109
Epoch 22/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2079 - val_loss: 0.2098
Epoch 23/100
235/235 [==============================] - 2s 6ms/step - loss: 0.2081 - val_loss: 0.2047
Epoch 24/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2050 - val_loss: 0.2036
Epoch 25/100
235/235 [==============================] - 2s 8ms/step - loss: 0.2062 - val_loss: 0.2085
Epoch 26/100
235/235 [==============================] - 2s 9ms/step - loss: 0.2048 - val_loss: 0.2032
Epoch 27/100
235/235 [==============================] - 2s 6ms/step - loss: 0.2036 - val_loss: 0.2033
Epoch 28/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2033 - val_loss: 0.2022
Epoch 29/100
235/235 [==============================] - 2s 7ms/step - loss: 0.2029 - val_loss: 0.2023
Epoch 30/100
```

```
#Visualize the outputs
import matplotlib.pyplot as plt

decoded_imgs = model.predict(x_test)

n = 10
plt.figure(figsize=(20, 4))
for i in range(1, n + 1):
    # Display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
```

```
    ax.get_yaxis().set_visible(False)
plt.show()
```

```
313/313 [==============================] - 1s 2ms/step
```



## How to get the output of intermediate layer ?

```
model.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 784)]             0

 dense_6 (Dense)             (None, 128)               100480

 dense_7 (Dense)             (None, 64)                8256

 dense_8 (Dense)             (None, 32)                2080

 dense_9 (Dense)             (None, 16)                528

 dense_10 (Dense)            (None, 8)                 136

 dense_11 (Dense)            (None, 4)                 36

 dense_12 (Dense)            (None, 2)                 10

 dense_13 (Dense)            (None, 4)                 12

 dense_14 (Dense)            (None, 8)                 40

 dense_15 (Dense)            (None, 16)                144

 dense_16 (Dense)            (None, 32)                544

 dense_17 (Dense)            (None, 64)                2112

 dense_18 (Dense)            (None, 128)               8320

 dense_19 (Dense)            (None, 784)               101136

=================================================================
Total params: 223,834
Trainable params: 223,834
Non-trainable params: 0
_________________________________________________________________
```

We need the output of 8th layer

```
model.layers
```

```
[<keras.engine.input_layer.InputLayer at 0x7fe2a83b2bc0>,
 <keras.layers.core.dense.Dense at 0x7fe2a83b2980>,
 <keras.layers.core.dense.Dense at 0x7fe2aacbe080>,
 <keras.layers.core.dense.Dense at 0x7fe2aacbf2b0>,
 <keras.layers.core.dense.Dense at 0x7fe2a8250b80>,
 <keras.layers.core.dense.Dense at 0x7fe2a8250c10>,
 <keras.layers.core.dense.Dense at 0x7fe2a82528c0>,
 <keras.layers.core.dense.Dense at 0x7fe2a8250b20>,
 <keras.layers.core.dense.Dense at 0x7fe2a8253970>,
 <keras.layers.core.dense.Dense at 0x7fe2a82515d0>,
 <keras.layers.core.dense.Dense at 0x7fe2a826ca90>,
 <keras.layers.core.dense.Dense at 0x7fe2a826d450>,
 <keras.layers.core.dense.Dense at 0x7fe2a826d630>,
 <keras.layers.core.dense.Dense at 0x7fe2a826ea70>,
 <keras.layers.core.dense.Dense at 0x7fe2a826f970>]
```

```
model.layers[7].output
```

```
<KerasTensor: shape=(None, 2) dtype=float32 (created by layer 'dense_12')>
```

Make a new model with

- input same as previous model and
- 7th layer output as model output (as layers starts from 0 )

```
dim_2_model = keras.Model(model.input, model.layers[7].output)
```

## Comparing results with tsne

```
!pip install openTSNE
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting openTSNE
  Downloading openTSNE-1.0.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.7 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.7/2.7 MB 61.8 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.16.6 in /usr/local/lib/python3.10/dist-packages (from openTSNE) (1.22.4)
Requirement already satisfied: scikit-learn>=0.20 in /usr/local/lib/python3.10/dist-packages (from openTSNE) (1.2.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from openTSNE) (1.10.1)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.20->openTSN
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.20->
Installing collected packages: openTSNE
Successfully installed openTSNE-1.0.0
```

```
from openTSNE import TSNE
```

```
indices = np.random.choice(x_test.shape[0], 1000, replace = False)
```

```
sample = x_test[indices]
```

```
sample_y = y_test[indices]
```

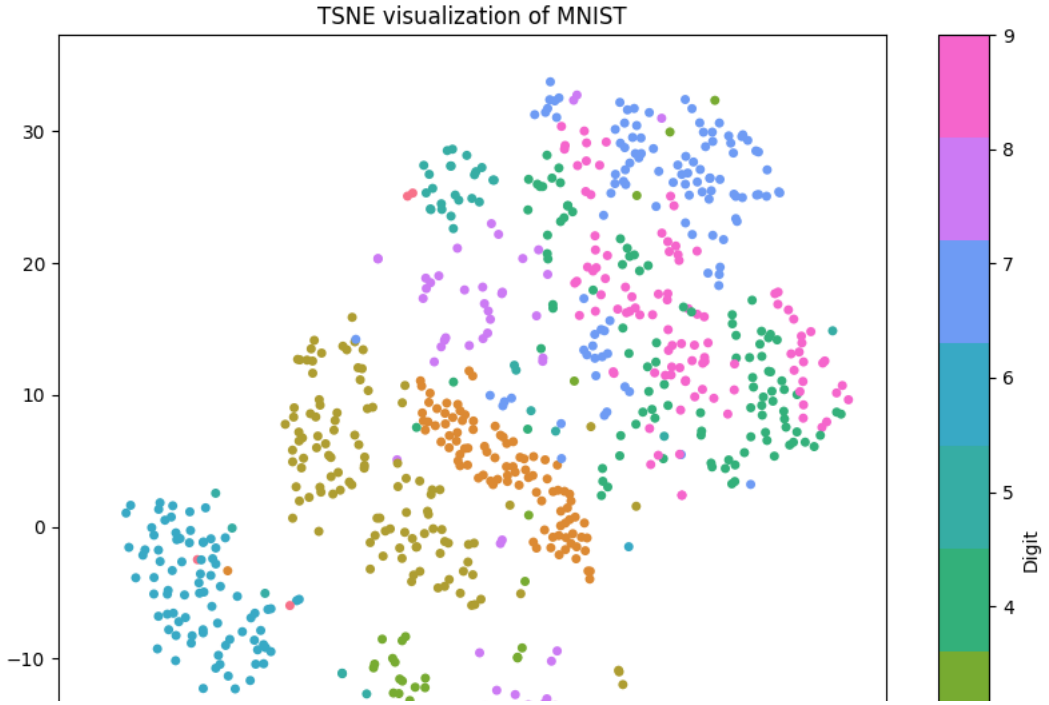
```
# tSNE
%%time
Z2 = TSNE(n_jobs=-1, initialization='random', random_state=42, negative_gradient_method='bh').fit(sample)
```

```
CPU times: user 8.38 s, sys: 71.5 ms, total: 8.46 s
Wall time: 4.72 s
```

## Visualizing t-SNE results

```
import seaborn as sns
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap

cmap = ListedColormap(sns.husl_palette(len(np.unique(sample_y))))
```

```
fig, ax = plt.subplots()
fig.set_figheight(10)
fig.set_figwidth(10)
ax.set_title('TSNE visualization of MNIST')
im = ax.scatter(Z2[:,0], Z2[:,1], s=25, c=sample_y, cmap=cmap, edgecolor='none')
cbar = fig.colorbar(im, ax=ax,label='Digit')
```




## Visualizing AE results

```
dim_2d = dim_2_model.predict(sample)
```

```
32/32 [==============================] - 0s 1ms/step
```

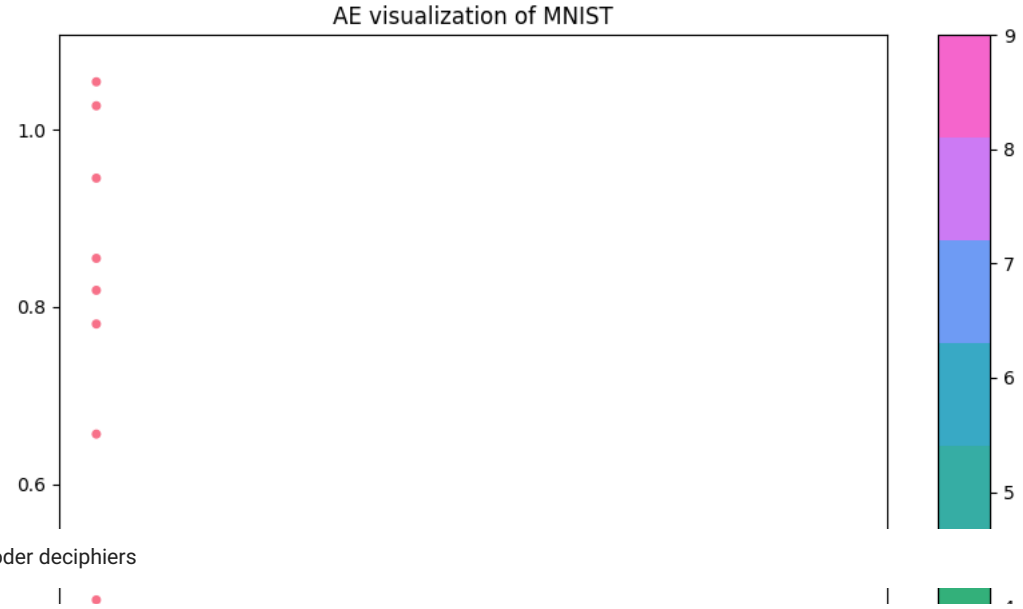
```
dim_2d
```

```
array([[0.9812436 , 0.        ],
       [0.6268663 , 0.        ],
       [3.6355464 , 0.        ],
       ...,
       [3.392712  , 0.        ],
       [0.99806726, 0.        ],
       [1.2852949 , 0.        ]], dtype=float32)
```

```
dim_2d.shape
```

```
(1000, 2)
```

```
fig, ax = plt.subplots()
fig.set_figheight(10)
fig.set_figwidth(10)
ax.set_title('AE visualization of MNIST')

im = ax.scatter(dim_2d[:,0], dim_2d[:,1], s=25, c=sample_y, cmap=cmap, edgecolor='none')
cbar = fig.colorbar(im, ax=ax,label='Digit')
```

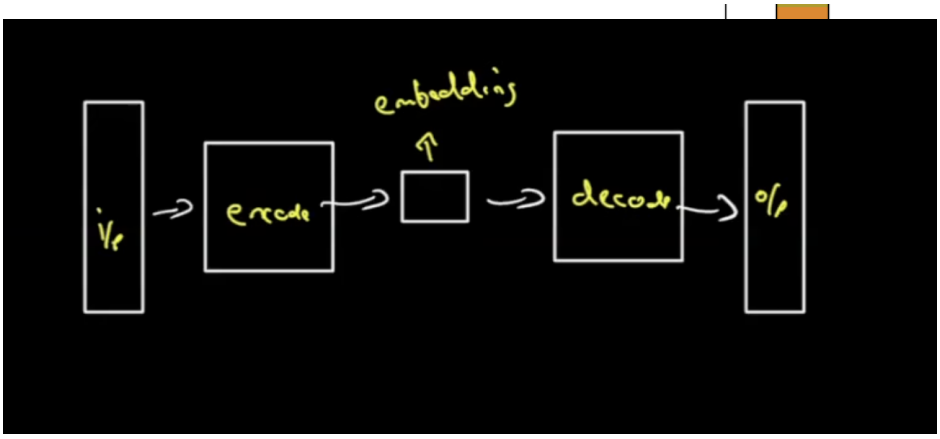



Decoder deciphiers

## Feature Extraction and Transfer learning

### Feature Extraction

Consider the following neural network



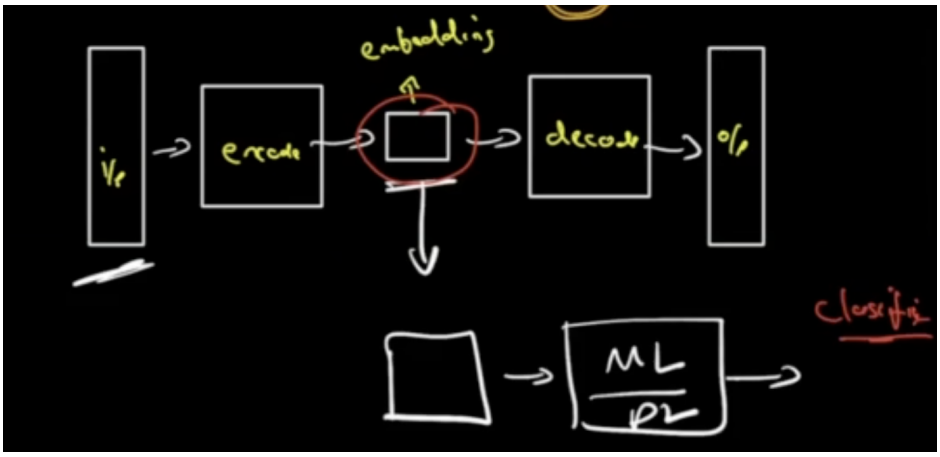

The idea of feature extraction is

- although there can be some information loss
- we know that emebeddings has the the information

So, we can use them as features

Once we have these embeddings,

- we can go back to ML/DL
- and perform any task (say classification)
- with embeddings as our features.




For example:

- Say, we have image data and we want to classification

Question: Are DT and log. regression good enough to classify image data ?

No. right.

## What can we do instead ?

We can run an autoencoder to get the embeddings

- and use them to run KNN, Log. reg. or Tree based model.

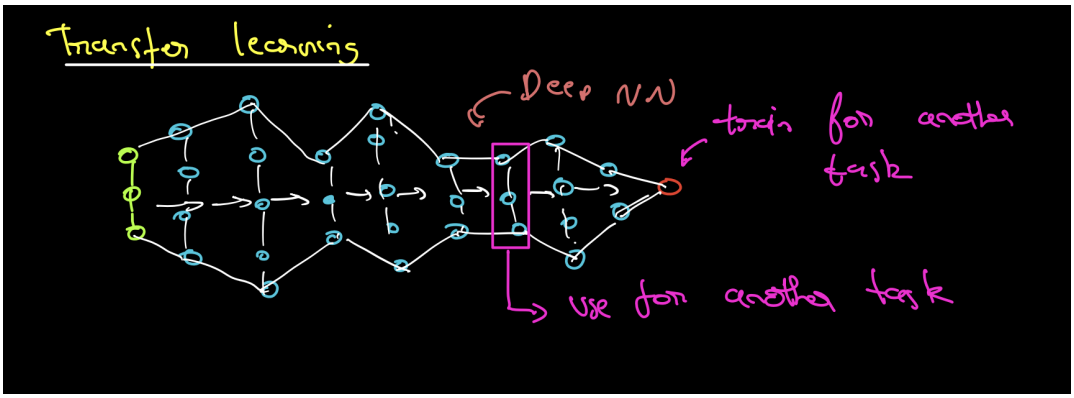
For recommender system

- we can get the embeddings
- and run cosine similiarity on it
- or cluster the embeddings.

## Transfer Learning

Double-click (or enter) to edit

Consider the following complex network with millions of parameters




Suppose it is is a binary classification model

- i.e. output layer has single neuron with sigmoid as activation function

Now, the network was trained to detect

- whether a person has cancer on not based on the images provided.

Say, our usecase is to detect whether a person has covid or not

What choices do we have ?

**Solution 1: Build a model from scratch and train it.**

- But we may not get the same result as researchers who ran their deep NN model on GPU for weeks/months to get the weights

**Solution 2: Use pre-trained model to get embeddings**

- We can use a pre trained model (cancer model)
  - to generate embeddings
  - and then use those embeddings as input
  - to run DL/ML model for our task (covid prediction)

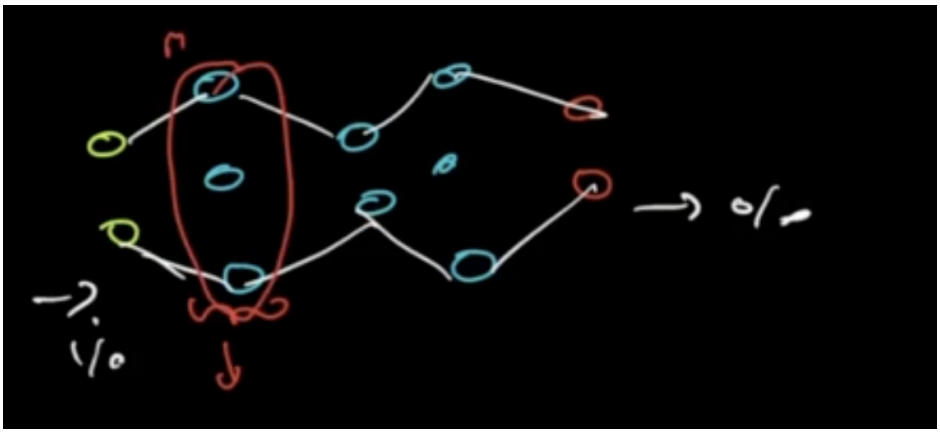
It was orginally created for other task but we use the output of intermediate layer for another task.

This is called as **Transfer Learning**

## Denoising Auto Encoders

Sometimes some encoders

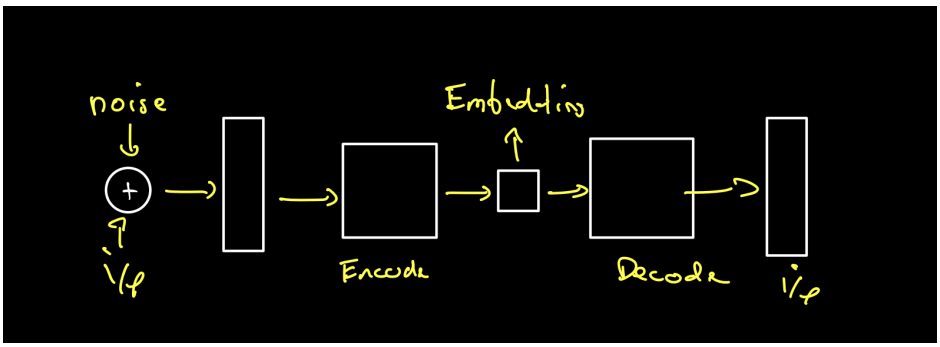
- can have more number of neuron compared to input
- i.e. it's stepping up instead of stepping down



Sometimes auto encoders can overfit

- and learn something called **identity function**
- i.e. output = input

### How can we deal with this ?




To deal with this,

- we add some random noise the the input

### What happens when we add random noise to it ?

Now, as we have added random noise to our input

- we know if network fully recreates the input
- it means network has overfitted.

And the network should not be recreate the noise

- as there is no pattern to it

Think of it as regularization over the network

- as network can't recreate the noise
- it ends up recreating the original data

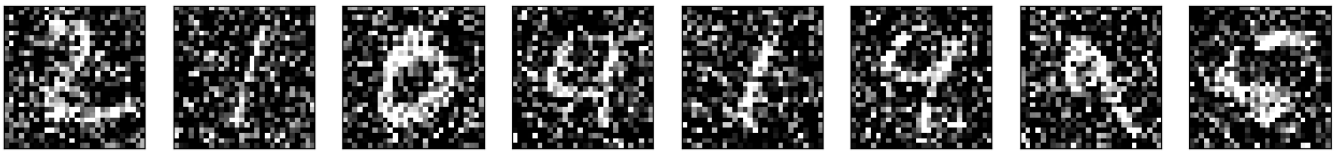
**The network cleans the data of any noise**

## Code walkthrough

```
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

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

#Reshape
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

#add NOISE - adding noise with random normal distribution
noise_factor = 0.5
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_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)

print(x_train.shape)
print(x_train_noisy.shape)
print(x_test.shape)
print(x_test_noisy.shape)
```

```
(60000, 784)
(60000, 784)
(10000, 784)
(10000, 784)
```

```
n = 10
plt.figure(figsize=(20, 2))
for i in range(1, n + 1):
    ax = plt.subplot(1, n, i)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



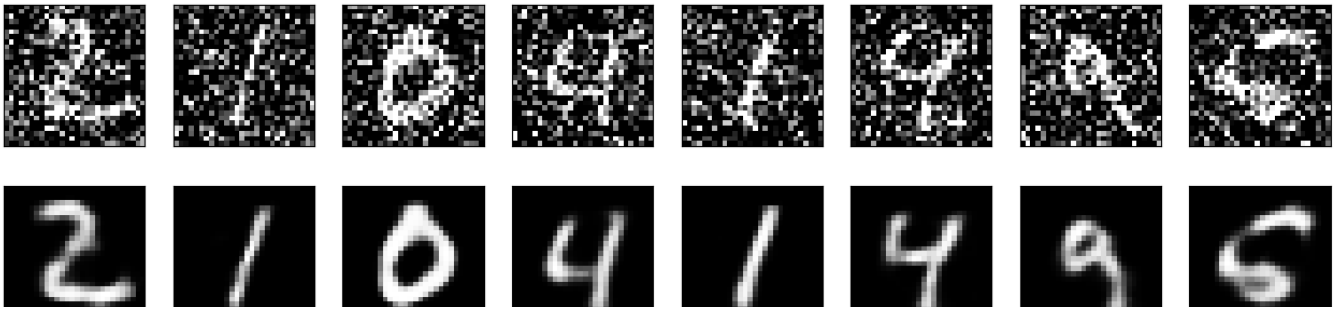
```
#AutoEncoder model
input_img = keras.Input(shape=(784,))
encoded = layers.Dense(128, activation='relu')(input_img)
encoded = layers.Dense(64, activation='relu')(encoded)
encoded = layers.Dense(32, activation='relu')(encoded)

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

# Compile and Fit
autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train_noisy, x_train, # NOTE: input is noisy, output is non-noisy
                epochs=100,
                batch_size=256,
```

```
                shuffle=True,
                validation_data=(x_test_noisy, x_test))
```

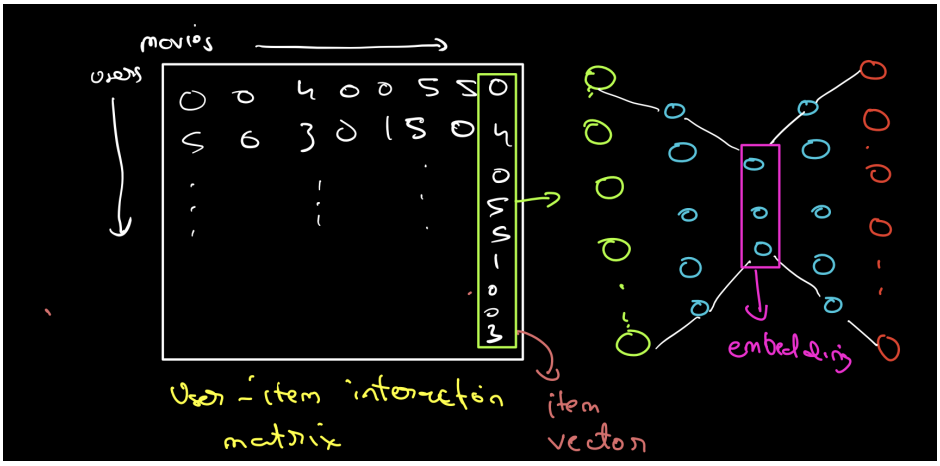
```
Epoch 1/100
235/235 [==============================] - 4s 6ms/step - loss: 0.2702 - val_loss: 0.2120
Epoch 2/100
235/235 [==============================] - 1s 6ms/step - loss: 0.1893 - val_loss: 0.1747
Epoch 3/100
235/235 [==============================] - 2s 9ms/step - loss: 0.1683 - val_loss: 0.1604
Epoch 4/100
235/235 [==============================] - 3s 11ms/step - loss: 0.1577 - val_loss: 0.1545
Epoch 5/100
235/235 [==============================] - 2s 10ms/step - loss: 0.1518 - val_loss: 0.1484
Epoch 6/100
235/235 [==============================] - 2s 9ms/step - loss: 0.1470 - val_loss: 0.1446
Epoch 7/100
235/235 [==============================] - 2s 10ms/step - loss: 0.1430 - val_loss: 0.1404
Epoch 8/100
235/235 [==============================] - 2s 9ms/step - loss: 0.1396 - val_loss: 0.1379
Epoch 9/100
235/235 [==============================] - 2s 11ms/step - loss: 0.1371 - val_loss: 0.1356
Epoch 10/100
235/235 [==============================] - 3s 11ms/step - loss: 0.1350 - val_loss: 0.1336
Epoch 11/100
235/235 [==============================] - 2s 10ms/step - loss: 0.1331 - val_loss: 0.1321
Epoch 12/100
235/235 [==============================] - 2s 9ms/step - loss: 0.1315 - val_loss: 0.1302
Epoch 13/100
235/235 [==============================] - 2s 10ms/step - loss: 0.1301 - val_loss: 0.1293
Epoch 14/100
235/235 [==============================] - 2s 10ms/step - loss: 0.1288 - val_loss: 0.1281
Epoch 15/100
235/235 [==============================] - 3s 11ms/step - loss: 0.1278 - val_loss: 0.1275
Epoch 16/100
235/235 [==============================] - 2s 9ms/step - loss: 0.1270 - val_loss: 0.1264
Epoch 17/100
235/235 [==============================] - 2s 8ms/step - loss: 0.1262 - val_loss: 0.1256
Epoch 18/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1255 - val_loss: 0.1252
Epoch 19/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1249 - val_loss: 0.1252
Epoch 20/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1243 - val_loss: 0.1243
Epoch 21/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1236 - val_loss: 0.1235
Epoch 22/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1230 - val_loss: 0.1233
Epoch 23/100
235/235 [==============================] - 1s 6ms/step - loss: 0.1227 - val_loss: 0.1229
Epoch 24/100
235/235 [==============================] - 2s 7ms/step - loss: 0.1220 - val_loss: 0.1224
Epoch 25/100
235/235 [==============================] - 1s 6ms/step - loss: 0.1217 - val_loss: 0.1228
Epoch 26/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1213 - val_loss: 0.1219
Epoch 27/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1208 - val_loss: 0.1214
Epoch 28/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1204 - val_loss: 0.1211
Epoch 29/100
235/235 [==============================] - 1s 5ms/step - loss: 0.1200 - val_loss: 0.1210
```

```
#Visualize the outputs
import matplotlib.pyplot as plt

decoded_imgs = autoencoder.predict(x_test_noisy)

n = 10
plt.figure(figsize=(20, 4))
for i in range(1, n + 1):
    # Display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

```
313/313 [==============================] - 1s 3ms/step
```



## Recommender System using AE




Recall the user-item iteraction matrix

- where each row represents the user
- columns represents the movie ratings by user

Do note that this iteraction matrix is sparse

Now, we can use these item vectors (movie ratings)

- as a input to our AE
- learn a dense embeddings
- and use these embeddings to find similar movies.

## Code walkthough

```
import numpy as np
import pandas as pd
```

```
import warnings
warnings.filterwarnings('ignore')
```

## Loading data

```
!gdown 1EsPvxcp5lzdltC3yLar1_laArmk9RZpw
!gdown 1DgdthLfHLIq3AnS4YAa1bne-OM90oHoD
```

```
Downloading...
From: https://drive.google.com/uc?id=1EsPvxcp5lzdltC3yLar1_laArmk9RZpw
To: /content/ratings.csv
100% 2.48M/2.48M [00:00<00:00, 191MB/s]
Downloading...
From: https://drive.google.com/uc?id=1DgdthLfHLIq3AnS4YAa1bne-OM90oHoD
To: /content/movies.csv
100% 516k/516k [00:00<00:00, 175MB/s]
```

```
ratings = pd.read_csv('ratings.csv')
```

## What's the shape of ratings ?

```
ratings.shape
```

```
(105339, 4)
```

## How many unique movie ids do we have ?

```
ratings.movieId.unique().shape
```

```
(10325,)
```

```
ratings.head()
```

| | userId | movieId | rating | timestamp |
|---|---|---|---|---|
| **0** | 1 | 16 | 4.0 | 1217897793 |
| **1** | 1 | 24 | 1.5 | 1217895807 |
| **2** | 1 | 32 | 4.0 | 1217896246 |
| **3** | 1 | 47 | 4.0 | 1217896556 |
| **4** | 1 | 50 | 4.0 | 1217896523 |

## Pivoting ratings table

Rows - Movie id

Columns = User id

```
rm = ratings.pivot(index = 'movieId', columns ='userId', values = 'rating').fillna(0)
rm.head()
```

| userId | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ... | 659 | 660 | 661 | 662 | 663 | 664 | 665 | 666 | 667 | 668 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **movieId** | | | | | | | | | | | | | | | | | | | | | |
| **1** | 0.0 | 5.0 | 0.0 | 0.0 | 4.0 | 0.0 | 0.0 | 5.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 4.0 | 5.0 | 3.0 | 0.0 | 0.0 | 0.0 | 0.0 | 3.0 |
| **2** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 5.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 3.0 |
| **3** | 0.0 | 2.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 4.0 | 3.0 | 0.0 | ... | 0.0 | 0.0 | 3.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 2.0 |
| **4** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **5** | 0.0 | 3.0 | 3.0 | 0.0 | 0.0 | 0.0 | 0.0 | 3.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 3.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 2.5 |

5 rows × 668 columns

```
rm.shape
```

```
(10325, 668)
```

## Check the sparsity of data

```
(rm > 0).sum().sum() / (rm.shape[0] * rm.shape[1])
```

```
0.015272940801206305
```

- Only 1.5% of the values are filled

## Splitting in train val

```
from sklearn.model_selection import train_test_split
```

```
train, val = train_test_split(rm, test_size = 0.2)
```

```
X_train = train.values
```

```
X_train.shape
```

```
(8260, 668)
```

```
X_val = val.values
```

```
X_val.shape
```

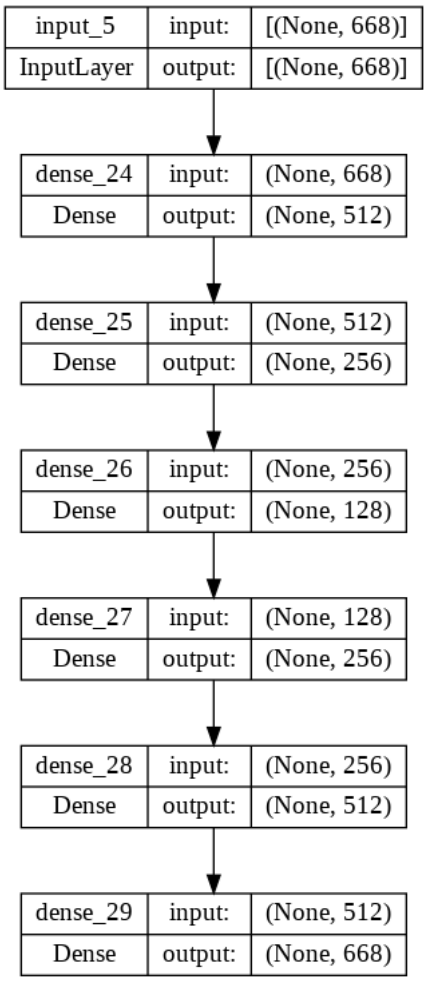
```
(2065, 668)
```

## Training Auto Encoders

```
import keras
from keras import layers
```

```
input_movie = keras.Input(shape=(668,))
encoded = layers.Dense(512, activation='relu')(input_movie)
encoded = layers.Dense(256, activation='relu')(encoded)
encoded = layers.Dense(128, activation='relu')(encoded)


decoded = layers.Dense(256, activation='relu')(encoded)
decoded = layers.Dense(512, activation='relu')(decoded)
decoded = layers.Dense(668, activation='linear')(decoded)


autoencoder = keras.Model(input_movie, decoded)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
```

```
from keras.utils.vis_utils import plot_model

plot_model(autoencoder, to_file='model_plot.png', show_shapes=True, show_layer_names=True)
```




```
autoencoder.fit(X_train, X_train,
                epochs=100,
                batch_size=256,
```

```
            shuffle=True,
            validation_data=(X_val, X_val))
```

```
Epoch 1/100
33/33 [==============================] - 1s 10ms/step - loss: 0.1706 - val_loss: 0.1495
Epoch 2/100
33/33 [==============================] - 0s 5ms/step - loss: 0.1466 - val_loss: 0.1352
Epoch 3/100
33/33 [==============================] - 0s 5ms/step - loss: 0.1344 - val_loss: 0.1266
Epoch 4/100
33/33 [==============================] - 0s 6ms/step - loss: 0.1247 - val_loss: 0.1209
Epoch 5/100
33/33 [==============================] - 0s 5ms/step - loss: 0.1179 - val_loss: 0.1154
Epoch 6/100
33/33 [==============================] - 0s 5ms/step - loss: 0.1109 - val_loss: 0.1115
Epoch 7/100
33/33 [==============================] - 0s 5ms/step - loss: 0.1053 - val_loss: 0.1084
Epoch 8/100
33/33 [==============================] - 0s 6ms/step - loss: 0.1012 - val_loss: 0.1063
Epoch 9/100
33/33 [==============================] - 0s 6ms/step - loss: 0.0966 - val_loss: 0.1034
Epoch 10/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0924 - val_loss: 0.1013
Epoch 11/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0890 - val_loss: 0.1006
Epoch 12/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0856 - val_loss: 0.0991
Epoch 13/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0836 - val_loss: 0.0980
Epoch 14/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0808 - val_loss: 0.0972
Epoch 15/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0783 - val_loss: 0.0968
Epoch 16/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0756 - val_loss: 0.0956
Epoch 17/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0725 - val_loss: 0.0957
Epoch 18/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0723 - val_loss: 0.0950
Epoch 19/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0701 - val_loss: 0.0945
Epoch 20/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0676 - val_loss: 0.0935
Epoch 21/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0659 - val_loss: 0.0934
Epoch 22/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0642 - val_loss: 0.0933
Epoch 23/100
33/33 [==============================] - 0s 6ms/step - loss: 0.0633 - val_loss: 0.0930
Epoch 24/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0619 - val_loss: 0.0923
Epoch 25/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0601 - val_loss: 0.0920
Epoch 26/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0587 - val_loss: 0.0921
Epoch 27/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0576 - val_loss: 0.0920
Epoch 28/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0567 - val_loss: 0.0918
Epoch 29/100
33/33 [==============================] - 0s 5ms/step - loss: 0.0556 - val_loss: 0.0920
```

## Extracting embeddings

Let's check which layer returns the latent space embeddings

```
autoencoder.summary()
```

```
Model: "model_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 668)]             0

 dense_24 (Dense)            (None, 512)               342528

 dense_25 (Dense)            (None, 256)               131328

 dense_26 (Dense)            (None, 128)               32896

 dense_27 (Dense)            (None, 256)               33024

 dense_28 (Dense)            (None, 512)               131584

 dense_29 (Dense)            (None, 668)               342684

=================================================================
```

```
Total params: 1,014,044
Trainable params: 1,014,044
Non-trainable params: 0
_________________________________________________________________
```

```
autoencoder.layers[3].output
```

```
<KerasTensor: shape=(None, 128) dtype=float32 (created by layer 'dense_26')>
```

Layer number 4 returns the latent space emebddings.

Let's create a model with

- input as model input and
- output as layer 4 output

```
intermediate_model = keras.Model(autoencoder.input, autoencoder.layers[3].output)
```

## Predicting embeddings for all movies

```
embeddings = intermediate_model.predict(rm.values)
```

```
323/323 [==============================] - 1s 3ms/step
```

```
embeddings
```

```
array([[0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ..., 1.3651494e+01,
        3.4803386e+00, 0.0000000e+00],
       [0.0000000e+00, 1.1449605e-02, 0.0000000e+00, ..., 1.1319354e+00,
        1.9510438e+00, 0.0000000e+00],
       [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, ..., 7.6276131e+00,
        4.5189867e+00, 0.0000000e+00],
       ...,
       [0.0000000e+00, 9.5751397e-02, 0.0000000e+00, ..., 1.9680148e-01,
        4.0758547e-01, 0.0000000e+00],
       [0.0000000e+00, 1.6079006e+00, 0.0000000e+00, ..., 4.3769965e-01,
        3.3153665e-01, 0.0000000e+00],
       [0.0000000e+00, 8.8318728e-02, 0.0000000e+00, ..., 2.2562157e-01,
        5.4051644e-01, 0.0000000e+00]], dtype=float32)
```

```
embeddings.shape
```

```
(10325, 128)
```

```
embeddings[10]
```

```
array([0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.9658452e+00,
       8.0590162e+00, 0.0000000e+00, 3.7221227e+00, 0.0000000e+00,
       1.0226821e+01, 8.1159992e+00, 0.0000000e+00, 1.8327374e+00,
       0.0000000e+00, 8.2238665e+00, 0.0000000e+00, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 5.6204233e+00, 3.0633299e+00,
       0.0000000e+00, 7.0646725e+00, 0.0000000e+00, 2.7049942e+00,
       0.0000000e+00, 5.9217091e+00, 4.7638183e+00, 0.0000000e+00,
       6.8396516e+00, 4.1846232e+00, 3.0754751e-01, 0.0000000e+00,
       3.5070825e-01, 2.7261753e+00, 4.8433380e+00, 4.4773763e-01,
       0.0000000e+00, 0.0000000e+00, 1.5257427e+00, 0.0000000e+00,
       0.0000000e+00, 8.0603523e+00, 0.0000000e+00, 2.1496911e+00,
       3.4151089e+00, 2.9382644e+00, 1.4833727e+01, 0.0000000e+00,
       0.0000000e+00, 7.4706016e+00, 0.0000000e+00, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 5.9207544e+00, 2.4117902e-03,
       6.5173354e+00, 1.0831287e+01, 0.0000000e+00, 0.0000000e+00,
       9.4732637e+00, 6.5200243e+00, 0.0000000e+00, 5.6516523e+00,
       0.0000000e+00, 1.1550360e+01, 0.0000000e+00, 1.0015137e+00,
       0.0000000e+00, 1.5498837e+00, 0.0000000e+00, 9.9607801e+00,
       0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.5138635e-01,
       0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 8.0790043e-01,
       0.0000000e+00, 5.5743232e+00, 0.0000000e+00, 2.0077191e-01,
       2.4721024e+00, 1.7662364e-01, 0.0000000e+00, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
       5.2709656e+00, 0.0000000e+00, 1.3146458e+01, 7.9421329e+00,
       7.3212967e+00, 1.1268778e+00, 0.0000000e+00, 3.9041052e+00,
       0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
       5.2551799e+00, 2.4510663e+00, 2.3059387e+00, 2.4024496e+00,
       0.0000000e+00, 0.0000000e+00, 1.9249966e+00, 0.0000000e+00,
       2.9486036e+00, 0.0000000e+00, 4.1414065e+00, 1.2645843e+01,
       0.0000000e+00, 0.0000000e+00, 1.0438521e+01, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 3.0717371e+00, 1.6371149e+00,
       3.1045024e+00, 3.3553655e+00, 2.7030585e+00, 0.0000000e+00],
      dtype=float32)
```

## Finding similar movies - Cosine similarity

```
from sklearn.metrics.pairwise import cosine_similarity


similarity_matrix = cosine_similarity(embeddings)


similarity_matrix.shape
```

```
(10325, 10325)
```

```
similarity_matrix
```

```
array([[1.0000001 , 0.6585631 , 0.67322814, ..., 0.59645236, 0.5442955 ,
        0.5833119 ],
       [0.6585631 , 0.9999999 , 0.6800547 , ..., 0.72579706, 0.6547908 ,
        0.72038335],
       [0.67322814, 0.6800547 , 0.99999964, ..., 0.68437904, 0.64143527,
        0.6798124 ],
       ...,
       [0.59645236, 0.72579706, 0.68437904, ..., 0.9999999 , 0.77266556,
        0.99907935],
       [0.5442955 , 0.6547908 , 0.64143527, ..., 0.77266556, 0.99999994,
        0.7622129 ],
       [0.5833119 , 0.72038335, 0.6798124 , ..., 0.99907935, 0.7622129 ,
        1.0000001 ]], dtype=float32)
```

## Creating df for the similarity matrix

```
item_sim_matrix = pd.DataFrame(similarity_matrix, index=rm.index, columns=rm.index)
item_sim_matrix.head() #Item-similarity Matrix
```

| movieId | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ... | 144482 | 144656 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **movieId** | | | | | | | | | | | | | | |
| **1** | 1.000000 | 0.658563 | 0.673228 | 0.540021 | 0.648697 | 0.654453 | 0.684952 | 0.632444 | 0.610780 | 0.654408 | ... | 0.571254 | 0.589064 | 0.58 |
| **2** | 0.658563 | 1.000000 | 0.680055 | 0.664267 | 0.671681 | 0.682655 | 0.687399 | 0.736932 | 0.620380 | 0.721273 | ... | 0.682567 | 0.722754 | 0.67 |
| **3** | 0.673228 | 0.680055 | 1.000000 | 0.708798 | 0.811759 | 0.733264 | 0.744269 | 0.749439 | 0.778765 | 0.572502 | ... | 0.666188 | 0.681853 | 0.66 |
| **4** | 0.540021 | 0.664267 | 0.708798 | 1.000000 | 0.618423 | 0.682824 | 0.693533 | 0.851051 | 0.724973 | 0.665865 | ... | 0.695002 | 0.742990 | 0.72 |
| **5** | 0.648697 | 0.671681 | 0.811759 | 0.618423 | 1.000000 | 0.671731 | 0.777911 | 0.699758 | 0.702049 | 0.590483 | ... | 0.622101 | 0.651532 | 0.59 |

5 rows × 10325 columns

## Finding movies similar to Liar Liar

```
movies = pd.read_csv('movies.csv')
```

```
movies[movies.title.str.contains('Liar Liar')]
```

| | movieId | title | genres |
|---|---|---|---|
| **1202** | 1485 | Liar Liar (1997) | Comedy |

## Sorting the movie id by similarity score

```
item_sim_matrix[1485].sort_values(ascending=False).head(10)
```

```
movieId
1485    1.000000
1391    0.829599
104     0.816998
1760    0.815274
1608    0.812336
3033    0.810485
1407    0.809809
1461    0.808078
1020    0.806623
```