

1. Data preparation, exploration, visualization

This assignment revisited the Kannada MNIST dataset that was used in Assignment 5. For the training data, I separated the pixel values from the labels using the loc index and for the test data I separated the image ids from the pixel values using the loc index as well. Just like in assignment 5, not much needed to be done in terms of preparing the data as much of the data was image data. I used the t-distributed stochastic neighbor embedding (TSNE) technique as well. TSNE is a statistical method for visualizing high-dimensional data by giving each datapoint a location in a two or three-dimensional map. It is based on Stochastic Neighbor Embedding originally developed by Sam Roweis and Geoffrey Hinton, where Laurens van der Maaten proposed the t -distributed variant. Just like PCA, TSNE is a nonlinear dimensionality reduction technique well-suited for embedding high-dimensional data for visualization in a low-dimensional space of two or three dimensions. Specifically, TSNE models each high-dimensional object by a two- or three-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points with high probability. Since TSNE is memory-intensive, only a subset of the data was extracted and TSNE was used to visualize the data.

2. Review research design and modeling methods

This assignment entailed using Autoencoders to correctly identify the Kannada digits in the Kannada MNIST images. An autoencoder is simply an artificial neural network (ANN) that learns how to efficiently compress and encode data and then reconstruct the data from a reduced coded representation to a representation that resembles the original input. Autoencoders are designed to reduce the dimensionality of the data by learning how to ignore noise in the data. Autoencoders consist of four such components: the encoder, the bottleneck, the decoder, and the reconstruction loss. The encoder learns how to reduce the dimensions of the input and creates an encoded representation by compressing the input data. The bottleneck is the layer that contains the compressed representation of the input data, and the data is now in the lowest possible dimensions. The decoder learns how to use the encoded representation as a blueprint in order to reconstruct the output data so that it resembles the input data. The reconstruction loss measures the performance of the autoencoder and how well the output data resembles the input data. To create the autoencoder, I defined a class called Autoencoder. In the Autoencoder class, I defined the encoder and decoder models as Sequential models and added a single layer for both the encoder and decoder models. In addition, I defined the latent dimension and returned the decoded images. Once the autoencoder was created, I configured the autoencoder for training, fit the data to the autoencoder, and generated the encoded and decoded images. Next, I created a classifier model to classify the Kannada digits. The process for creating the classifier model was similar to that of the autoencoder. The classifier model takes the output from the encoder of the autoencoder and predicts the classes of the digits. In addition, the softmax activation function was used. To build the classification neural network, I encoded the validation and training sets that were created as part of the train



test split, performed one-hot encoding on the labels contained in the validation and training sets, fit the encoded and one-hot encoded training sets to the classifier model, and ran the classifier model. I then generated the predictions for the images, prepared the submission dataframe, and submitted my results to Kaggle for evaluation

3. Review results, evaluate models

I felt that the results generated by the autoencoder was acceptable. Initially, I used a latent dimension of 96 and the autoencoder performed worse compared to other methods that were used such as Principal Component Analysis and Train Test Split. When PCA was used on the Kannada MNIST data, the score generated by Kaggle was 0.85920 and when the Train Test Split was used, the score was 0.89780. This was a bit surprising since autoencoders are designed to reduce the dimensionality of the data by learning how to ignore noise in the data and function much like that of principal components analysis. The autoencoder did a fairly good job in encoding and decoding the images so that the reconstructed images were fairly similar to the original images. However, when I used a higher latent dimension, the Kaggle score improved from 0.84260 to 0.85960 and the autoencoder ended up performing better than PCA and Random Forests. In this assignment, the categorical_accuracy was the metric of choice since this problem was a multiclass classification problem. For a latent dimension of 96, as the categorical accuracy for the training set increased, the categorical accuracy for the validation set stayed relatively constant. Therefore, the classifier model neither overfitted nor underfitted the data and hence, the autoencoder was able to correctly predict approximately 84% of the Kannada digits in the images. For a latent dimension of 192, as categorical accuracy of the training set increased, the categorical accuracy of the validation set increased as well.

4. Kaggle Submission relative to Peers

Username - anaswarj

Submission and Description	Status	Private Score	Public Score	Use for Final Score
MSDS 422 Assignment 9 (version 10/10) an hour ago by Anaswar Jayakumar Notebook MSDS 422 Assignment 9 Version 10	Succeeded 	0.85960	0.85940	<input type="checkbox"/>
MSDS 422 Assignment 9 (version 9/10) a day ago by Anaswar Jayakumar Notebook MSDS 422 Assignment 9 Version 9	Succeeded 	0.84260	0.83280	<input type="checkbox"/>

5. Exposition, problem description and management recommendations

For problems involving image processing, autoencoders seem to perform comparatively well in comparison to traditional machine learning methods such as PCA and Random Forests. The one advantage that autoencoders have over traditional machine learning methods is their ability to reduce the dimensionality of the data, thereby reducing the noise in the data. Given enough time, I could have certainly improved the autoencoder model in order to improve the performance. The

easiest way to improve the performance of the autoencoder would be to add more layers just like I would do for any other neural network. A more sophisticated way to improve the performance of the autoencoder is to initialize the autoencoders with pretrained Restricted Boltzmann Machines (RBMs). RBMs are generative neural networks that learn a probability distribution over its input. Structurally, they can be seen as a two-layer network with one input (“visible”) layer and one hidden layer. The first layer, the “visible” layer, contains the original input while the second layer, the “hidden” layer, contains a representation of the original input. One advantage of using RBMs is that they are similar to autoencoders and therefore, try to make the reconstructed input from the “hidden” layer as close to the original input as possible. One disadvantage is that RBMs use the same matrix for “encoding” and “decoding.”. Lastly, another way to improve the performance of the autoencoder would be to implement custom constraints that take advantage of the properties of autoencoders that are derived from PCA. This would improve the test reconstruction error which measures the performance of the autoencoder and how well the output data resembles the input data

MSDS 422 Assignment 9

May 30, 2021

```
[1]: # This Python 3 environment comes with many helpful analytics libraries
      ↳ installed
      # It is defined by the kaggle/python Docker image: https://github.com/kaggle/
      ↳ docker-python
      # For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list
↳ all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that
↳ gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved
↳ outside of the current session
```

```
/kaggle/input/Kannada-MNIST/sample_submission.csv
/kaggle/input/Kannada-MNIST/Dig-MNIST.csv
/kaggle/input/Kannada-MNIST/train.csv
/kaggle/input/Kannada-MNIST/test.csv
```

```
[2]: from time import time

import numpy as np
import pandas as pd

# For plotting
from matplotlib import offsetbox
import matplotlib.pyplot as plt
import matplotlib.path_effects as PathEffects
```

```

import seaborn as sns
import plotly.graph_objects as go

%matplotlib inline
sns.set(style='white', context='notebook', rc={'figure.figsize':(14,10)})

#For standardising the dat
from sklearn.preprocessing import StandardScaler

#PCA
from sklearn.manifold import TSNE

#Ignore warnings
import warnings
warnings.filterwarnings('ignore')

```

```

[4]: train = pd.read_csv('../input/Kannada-MNIST/train.csv')
test = pd.read_csv('../input/Kannada-MNIST/test.csv')

train.head()

```

```

[4]:
  label  pixel0  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  \
0      0       0       0       0       0       0       0       0       0
1      1       0       0       0       0       0       0       0       0
2      2       0       0       0       0       0       0       0       0
3      3       0       0       0       0       0       0       0       0
4      4       0       0       0       0       0       0       0       0

    pixel8  ...  pixel774  pixel775  pixel776  pixel777  pixel778  pixel779  \
0         0  ...         0         0         0         0         0         0
1         0  ...         0         0         0         0         0         0
2         0  ...         0         0         0         0         0         0
3         0  ...         0         0         0         0         0         0
4         0  ...         0         0         0         0         0         0

    pixel780  pixel781  pixel782  pixel783
0           0         0         0         0
1           0         0         0         0
2           0         0         0         0
3           0         0         0         0
4           0         0         0         0

```

[5 rows x 785 columns]

```

[5]: y = train.loc[:, 'label'].values
x = train.loc[:, 'pixel0:'].values

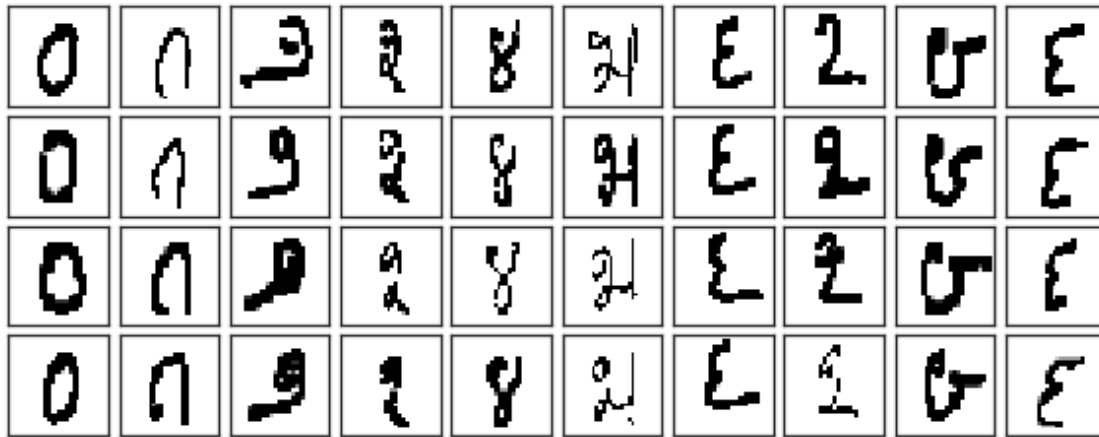
```

```
print(x.shape)
print(y)
```

```
(60000, 784)
[0 1 2 ... 7 8 9]
```

```
[6]: ## Plotting the original train data
```

```
def plot_digits(data):
    fig, axes = plt.subplots(4, 10, figsize=(10, 4),
                              subplot_kw={'xticks': [], 'yticks': []},
                              gridspec_kw=dict(hspace=0.1, wspace=0.1))
    for i, ax in enumerate(axes.flat):
        ax.imshow(data[i].reshape(28, 28),
                  cmap='binary', interpolation='nearest',
                  clim=(0, 16))
plot_digits(x)
```



```
[7]: ## Standardizing the data
# standardized_data = StandardScaler().fit_transform(x)
# print(standardized_data.shape)
```

```
[7]: # t-SNE is consumes a lot of memory so we shall use only a subset of our
      ↪ dataset.
```

```
x_subset = x[0:10000]
y_subset = y[0:10000]

print(np.unique(y_subset))
```

```
[0 1 2 3 4 5 6 7 8 9]
```

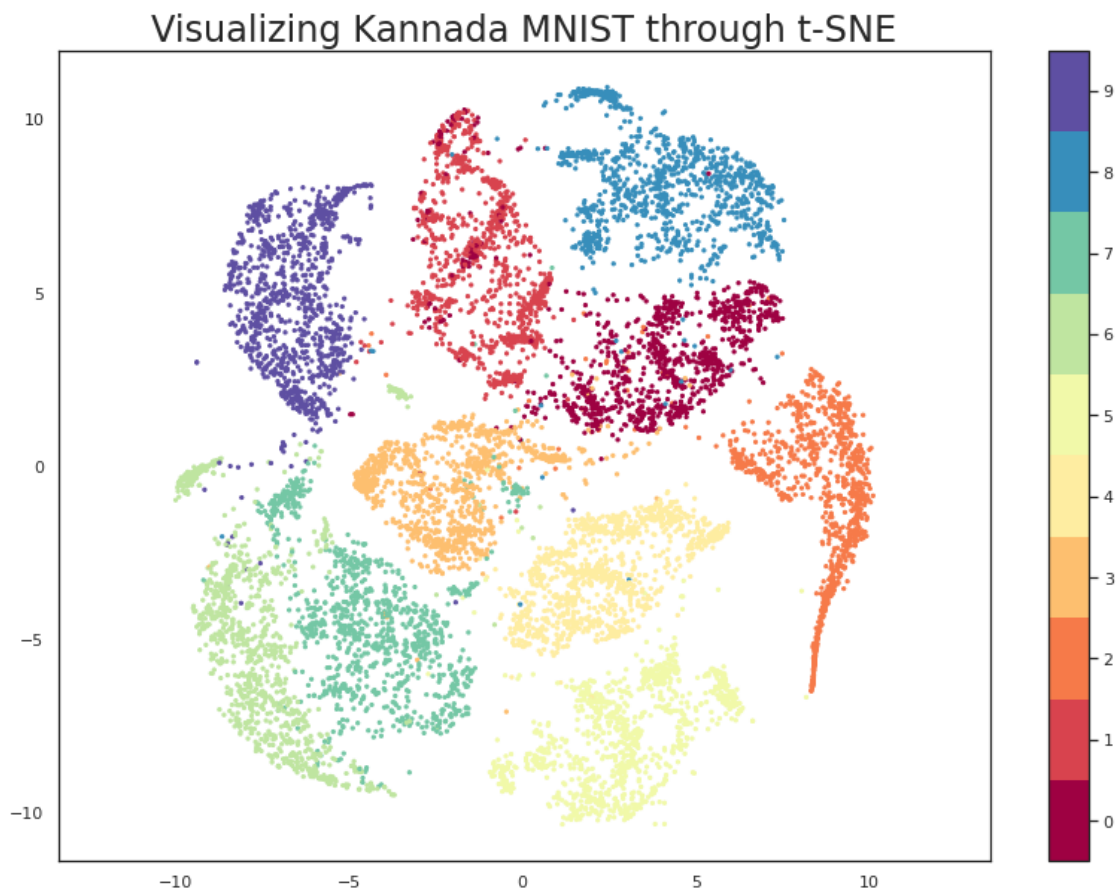
```
[8]: %time tsne = TSNE(random_state = 42, n_components=2, verbose=0, perplexity=40,
    ↪n_iter=300).fit_transform(x_subset)
print(tsne)
```

CPU times: user 1min 13s, sys: 1.67 s, total: 1min 14s

Wall time: 23.3 s

```
[[ 2.6967933  3.1744847]
 [-0.6287243  3.7250185]
 [ 9.76058   -1.1070673]
 ...
 [-4.9577785 -2.7595499]
 [ 4.1164937  9.420799 ]
 [-5.8456707  7.2945395]]
```

```
[9]: plt.scatter(tsne[:, 0], tsne[:, 1], s= 5, c=y_subset, cmap='Spectral')
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar(boundaries=np.arange(11)-0.5).set_ticks(np.arange(10))
plt.title('Visualizing Kannada MNIST through t-SNE', fontsize=24);
```



```
[10]: from sklearn.decomposition import PCA
pca_50 = PCA(n_components=50)
pca_result_50 = pca_50.fit_transform(x_subset)
```

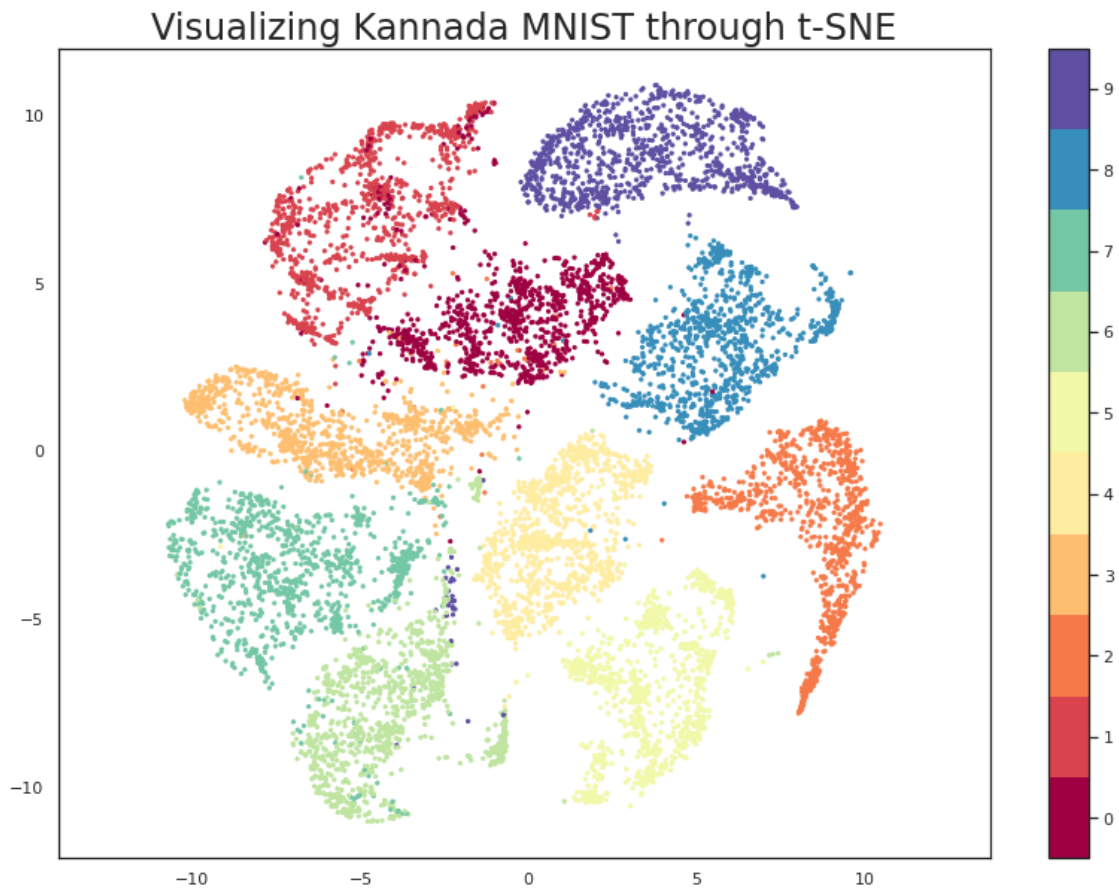
```
[11]: # Using the output of PCA as input for t-SNE
%time pca_tsne = TSNE(random_state = 42, n_components=2, verbose=0,
↳perplexity=40, n_iter=300).fit_transform(pca_result_50)
```

CPU times: user 1min 1s, sys: 655 ms, total: 1min 2s

Wall time: 17.9 s

```
[12]: #visualising t-SNE again

plt.scatter(pca_tsne[:, 0], pca_tsne[:, 1], s= 5, c=y_subset, cmap='Spectral')
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar(boundaries=np.arange(11)-0.5).set_ticks(np.arange(10))
plt.title('Visualizing Kannada MNIST through t-SNE', fontsize=24);
```




```
[13]: %time pca_tsne2 = TSNE(random_state = 42, n_components=3, verbose=0,
    ↪perplexity=40, n_iter=300).fit_transform(pca_result_50)
```

CPU times: user 3min 11s, sys: 700 ms, total: 3min 12s

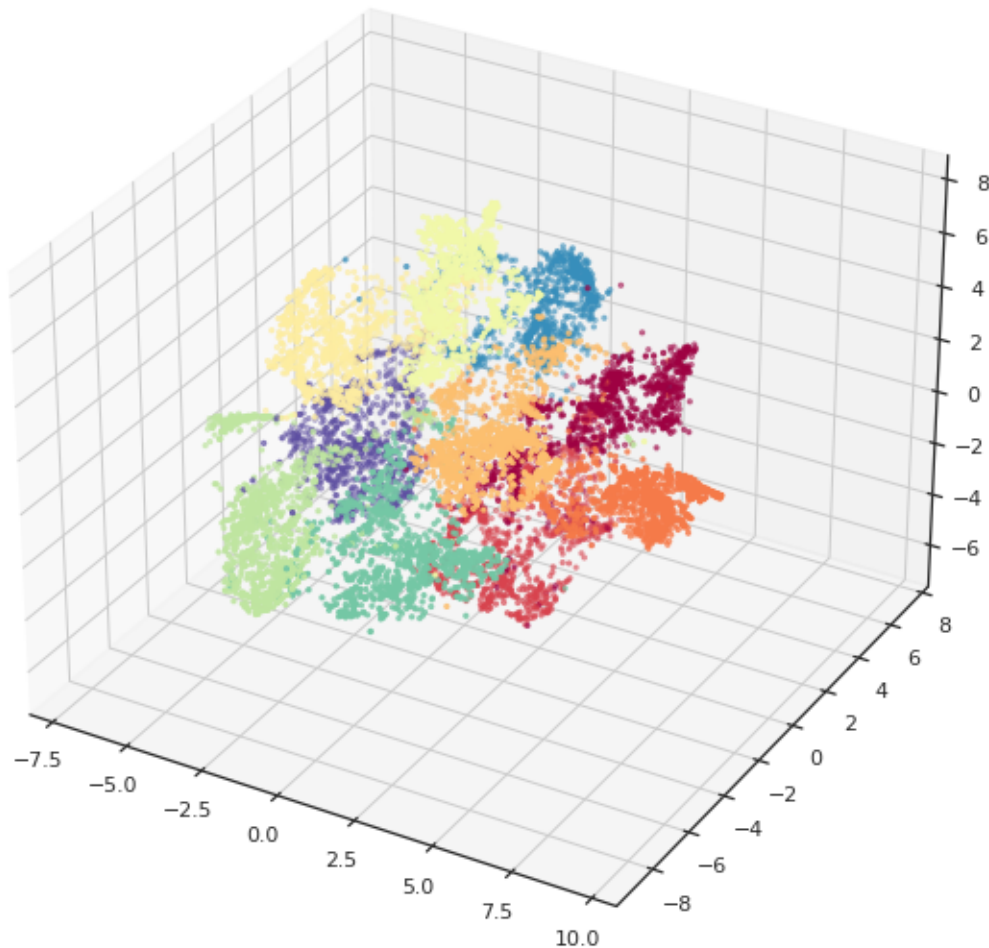
Wall time: 53.6 s

```
[14]: from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(pca_tsne2[:, 0], pca_tsne2[:, 1], pca_tsne2[:, 2], s= 5, c=y_subset,
    ↪cmap='Spectral')
plt.title('Visualizing Kannada MNIST through t-SNE in 3D', fontsize=24);
plt.show()
```

Visualizing Kannada MNIST through t-SNE in 3D



```
[15]: import tensorflow.keras
import tensorflow as tf

from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split
from tensorflow.keras import layers, losses
from tensorflow.keras.models import Model

latent_dim = 192

class Autoencoder(Model):
    def __init__(self, latent_dim):
        super(Autoencoder, self).__init__()
        self.latent_dim = latent_dim

        self.encoder = tf.keras.Sequential()
        self.encoder.add(layers.Dense(latent_dim, input_shape=(784,),
↪activation='relu'))

        self.decoder = tf.keras.Sequential()
        self.decoder.add(layers.Dense(784, input_shape=(latent_dim,),
↪activation='relu'))

    def call(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

```
[16]: x_train, x_val, y_train, y_val = train_test_split(x, y, test_size=0.2,
↪random_state=42)
```

```
[17]: print(x_train.shape)
print(x_val.shape)
```

```
(48000, 784)
(12000, 784)
```

```
[18]: autoencoder = Autoencoder(latent_dim)
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
```

```
[19]: autoencoder.fit(x_train, x_train, epochs=10, shuffle=True,
↪validation_data=(x_val, x_val))
```

```
Epoch 1/10
1500/1500 [=====] - 7s 4ms/step - loss: 1151.8619 -
val_loss: 451.3132
```

```

Epoch 2/10
1500/1500 [=====] - 6s 4ms/step - loss: 430.2674 -
val_loss: 410.4989
Epoch 3/10
1500/1500 [=====] - 6s 4ms/step - loss: 402.9841 -
val_loss: 400.9768
Epoch 4/10
1500/1500 [=====] - 6s 4ms/step - loss: 392.3990 -
val_loss: 400.9614
Epoch 5/10
1500/1500 [=====] - 6s 4ms/step - loss: 388.1113 -
val_loss: 392.3059
Epoch 6/10
1500/1500 [=====] - 6s 4ms/step - loss: 387.8872 -
val_loss: 390.1298
Epoch 7/10
1500/1500 [=====] - 6s 4ms/step - loss: 383.4072 -
val_loss: 387.1483
Epoch 8/10
1500/1500 [=====] - 6s 4ms/step - loss: 378.0386 -
val_loss: 387.5417
Epoch 9/10
1500/1500 [=====] - 6s 4ms/step - loss: 377.7312 -
val_loss: 381.8873
Epoch 10/10
1500/1500 [=====] - 6s 4ms/step - loss: 379.3544 -
val_loss: 384.4959

```

```
[19]: <tensorflow.python.keras.callbacks.History at 0x7fd6344cdc90>
```

```
[20]: encoded_imgs = autoencoder.encoder(x_val).numpy()
      decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()
```

```
[21]: print(encoded_imgs)
      print(decoded_imgs)
```

```

[[171.7616  157.14658 152.36678 ... 141.92613  33.18702  38.31949 ]
 [  0.         165.10144 194.14067 ... 217.18341  36.862316 218.46663 ]
 [101.159836 156.40805  93.583305 ... 137.19824  69.59082 111.004295]
 ...
 [226.71292 191.709   175.79169 ... 141.9993   114.02002 106.41542 ]
 [124.00481 123.49156 159.9909  ... 143.62825  130.34105 129.33246 ]
 [222.79085 176.6957  227.30853 ... 17.747694  79.89976  94.67307 ]]
[[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.]]
```

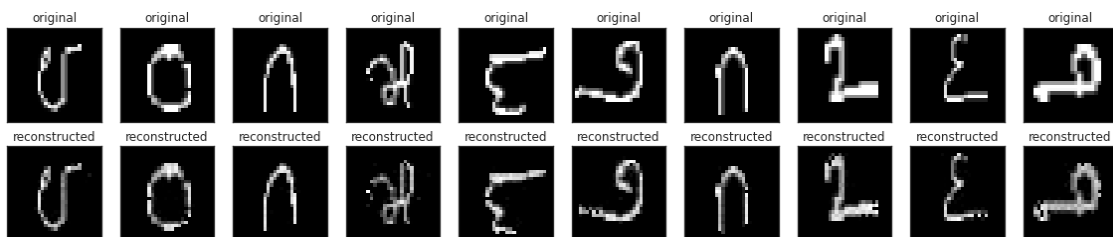
```
[22]: print(encoded_imgs.shape)
      print(decoded_imgs.shape)
```

```
(12000, 192)
(12000, 784)
```

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

    # display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28,28))
    plt.title("reconstructed")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.show()
```



```
[24]: from keras import metrics

classifier = tf.keras.Sequential()
classifier.add(layers.Dense(10, input_shape=(latent_dim,),
    ↳activation='softmax'))
classifier.compile(loss=losses.CategoricalCrossentropy(), optimizer='adam',
    ↳metrics=[metrics.categorical_accuracy])
```

```
classifier.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 10)	1930

Total params: 1,930
Trainable params: 1,930
Non-trainable params: 0

```
[25]: x_train_encoded = autoencoder.encoder(x_train).numpy()  
      x_val_encoded = autoencoder.encoder(x_val).numpy()
```

```
[26]: print(x_train_encoded.shape)  
      print(x_val_encoded.shape)
```

(48000, 192)

(12000, 192)

```
[27]: y_train_one_hot = pd.get_dummies(y_train).values  
      y_val_one_hot = pd.get_dummies(y_val).values
```

```
[28]: classifier.  
      →fit(x_train_encoded,y_train_one_hot,epochs=10,verbose=2,validation_data=(x_val_encoded,  
      →y_val_one_hot))
```

Epoch 1/10

1500/1500 - 2s - loss: 28.2488 - categorical_accuracy: 0.7395 - val_loss: 7.1657
- val_categorical_accuracy: 0.8930

Epoch 2/10

1500/1500 - 1s - loss: 5.8013 - categorical_accuracy: 0.9095 - val_loss: 5.2567
- val_categorical_accuracy: 0.9143

Epoch 3/10

1500/1500 - 1s - loss: 4.1833 - categorical_accuracy: 0.9258 - val_loss: 4.2781
- val_categorical_accuracy: 0.9197

Epoch 4/10

1500/1500 - 1s - loss: 3.3831 - categorical_accuracy: 0.9350 - val_loss: 2.5408
- val_categorical_accuracy: 0.9467

Epoch 5/10

1500/1500 - 1s - loss: 3.0446 - categorical_accuracy: 0.9384 - val_loss: 3.0465
- val_categorical_accuracy: 0.9367

Epoch 6/10

1500/1500 - 1s - loss: 2.9249 - categorical_accuracy: 0.9394 - val_loss: 2.8458
- val_categorical_accuracy: 0.9404

Epoch 7/10

1500/1500 - 1s - loss: 2.6940 - categorical_accuracy: 0.9412 - val_loss: 2.8743

```

- val_categorical_accuracy: 0.9392
Epoch 8/10
1500/1500 - 1s - loss: 2.7584 - categorical_accuracy: 0.9414 - val_loss: 2.3462
- val_categorical_accuracy: 0.9492
Epoch 9/10
1500/1500 - 1s - loss: 2.7147 - categorical_accuracy: 0.9410 - val_loss: 2.7703
- val_categorical_accuracy: 0.9473
Epoch 10/10
1500/1500 - 1s - loss: 2.6572 - categorical_accuracy: 0.9432 - val_loss: 2.2341
- val_categorical_accuracy: 0.9516

```

[28]: <tensorflow.python.keras.callbacks.History at 0x7fd634361750>

[29]: `pd.DataFrame(classifier.history.history)`

```

[29]:
      loss  categorical_accuracy  val_loss  val_categorical_accuracy
0  28.248846          0.739500  7.165673          0.893000
1   5.801266          0.909479  5.256744          0.914333
2   4.183268          0.925812  4.278053          0.919750
3   3.383064          0.934979  2.540804          0.946667
4   3.044647          0.938417  3.046456          0.936667
5   2.924853          0.939354  2.845833          0.940417
6   2.693985          0.941188  2.874349          0.939250
7   2.758431          0.941354  2.346167          0.949167
8   2.714653          0.940958  2.770258          0.947333
9   2.657182          0.943167  2.234077          0.951583

```

[30]: `test.head()`

```

[30]:
   id  pixel0  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  \
0   0        0        0        0        0        0        0        0        0        0
1   1        0        0        0        0        0        0        0        0        0
2   2        0        0        0        0        0        0        0        0        0
3   3        0        0        0        0        0        0        0        0        0
4   4        0        0        0        0        0        0        0        0        0

   ...  pixel774  pixel775  pixel776  pixel777  pixel778  pixel779  pixel780  \
0   ...        0        0        0        0        0        0        0
1   ...        0        0        0        0        0        0        0
2   ...        0        0        0        0        0        0        0
3   ...        0        0        0        0        0        0        0
4   ...        0        0        0        0        0        0        0

   pixel781  pixel782  pixel783
0         0         0         0
1         0         0         0
2         0         0         0
3         0         0         0

```

```
4         0         0         0
```

```
[5 rows x 785 columns]
```

```
[31]: image_ids = test.loc[:, 'id'].values
      x_test = test.loc[:, 'pixel0:'].values

      print(x_test.shape)
      print(image_ids.shape)
```

```
(5000, 784)
```

```
(5000,)
```

```
[32]: x_test_encoded = autoencoder.encoder(x_test).numpy()
```

```
[33]: y_pred = classifier.predict_classes(x_test_encoded)
      y_pred
```

```
[33]: array([3, 0, 2, ..., 1, 6, 3])
```

```
[34]: y_pred.shape
```

```
[34]: (5000,)
```

```
[35]: # lets prepare for the prediction submission
      sub = pd.DataFrame()
      sub['id'] = image_ids
      sub['label'] = y_pred
      sub.head()
```

```
[35]:
```

	id	label
0	0	3
1	1	0
2	2	2
3	3	6
4	4	7

```
[36]: sub.to_csv("submission.csv", index = False)
```

```
[ ]:
```