Manraj Singh

IE 7860

Mar. 27, 2023

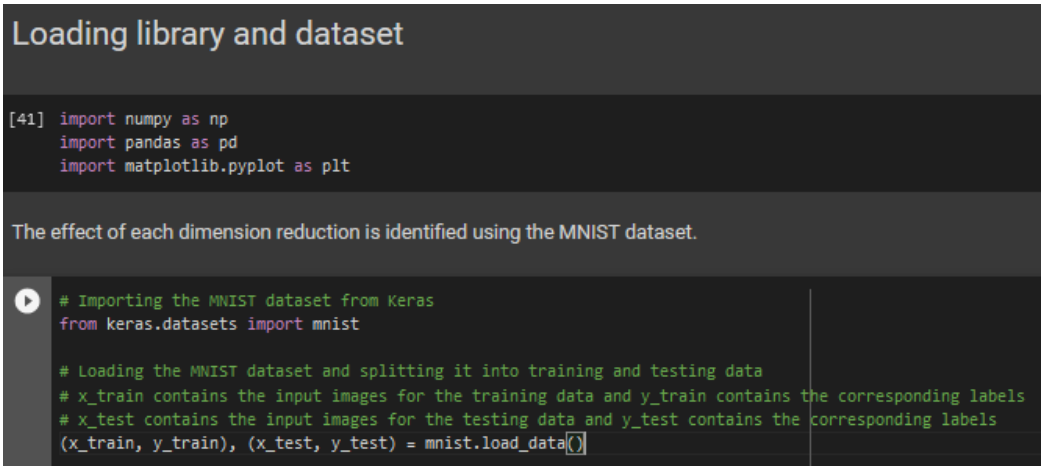
Dr. Ratna Chinnam

IE 7860 – Data Visualization, Imputation, & Feature Selection Report

Dimensionality reduction techniques such as PCA, t-SNE, and UMAP can be extremely helpful in visualizing and analyzing high-dimensional datasets such as MNIST. MNIST is a dataset of 28x28 pixel grayscale images of handwritten digits, which means that each image in the dataset can be represented as a 784-dimensional vector (28 x 28 = 784).

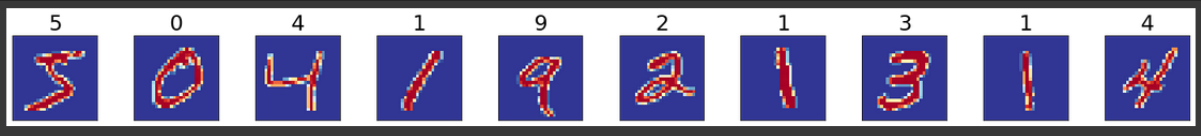
The high dimensionality of the MNIST dataset can make it difficult to visualize and analyze directly. However, by applying dimensionality reduction techniques such as PCA, t-SNE, or UMAP, it is possible to reduce the dimensionality of the dataset while preserving as much of the information as possible. PCA, for example, can be used to reduce the dimensionality of the MNIST dataset by projecting the 784-dimensional vectors onto a smaller number of principal components. This can help to visualize the dataset in 2D or 3D, which can be useful for identifying patterns or clusters in the data.

T-SNE and UMAP are nonlinear dimensionality reduction techniques that can be used to visualize high-dimensional datasets in lower-dimensional space. They are particularly effective at preserving local structure in the data, which means that similar points in high-dimensional space are likely to remain close together in the lower-dimensional space. This can be very helpful for visualizing clusters or groups of points in the data, as well as for identifying outliers or anomalies. Overall, dimensionality reduction techniques such as PCA, t-SNE, and UMAP can be very useful tools for visualizing and analyzing high-dimensional datasets such as MNIST. They can help to reduce the dimensionality of the data, identify patterns and clusters, and make it easier to explore and understand complex datasets. In this report, we summarize the dimensionality reduction methods that can interpret data more intuitively by reducing features on the MNIST dataset and allowing to visualize the data differently.

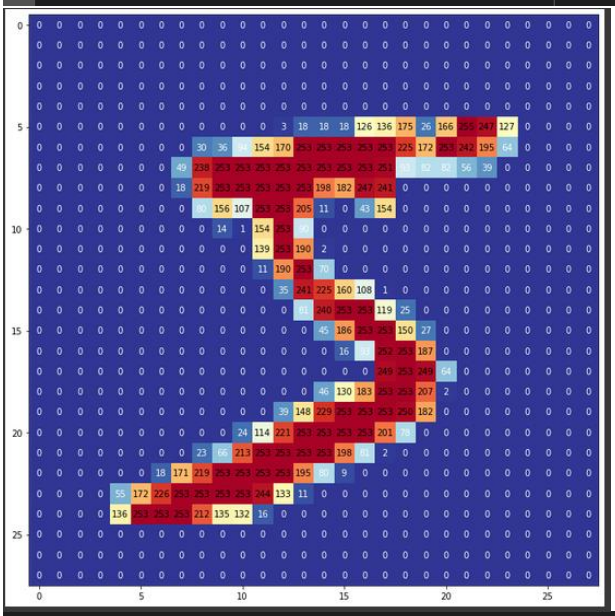


Checking train dataset

```
[43] fig = plt.figure(figsize=(25, 4))
     for idx in np.arange(10):
         ax = fig.add_subplot(2, 10, idx+1, xticks=[], yticks=[])
         ax.imshow(x_train[idx], cmap='RdYlBu_r')
         ax.set_title(str(y_train[idx]),fontsize=25)
```




```
img = x_train[0] # Selecting the first image from the training set

# Creating a new figure with a 12x12 inch size
fig = plt.figure(figsize = (12,12))

# Adding a new subplot to the figure
ax = fig.add_subplot(111)

# Displaying the image in the subplot with a colormap 'RdYlBu_r'
ax.imshow(img, cmap='RdYlBu_r')

# Extracting the dimensions of the image
width, height = img.shape

# Threshold for color, any pixel value above this value will be colored white, and the rest will be black
thresh = img.max()/2.5

# Looping through each pixel of the image
for x in range(width):
    for y in range(height):
        # Rounding the pixel value to 2 decimal points, but setting the value to 0 if the value is 0
        val = round(img[x][y],2) if img[x][y] !=0 else 0

        # Adding text annotation to the current pixel
        ax.annotate(str(val), xy=(y,x),
                    horizontalalignment='center',
                    verticalalignment='center',
                    # If pixel value is below the threshold, the text color is set to white, otherwise it's black
                    color='white' if img[x][y]<thresh else 'black')
```



```
# Reshape to 2D data
x_train = x_train.reshape(x_train.shape[0], -1)
print(x_train.shape)

sample_size = 5000
# Use only the top 1000 data for training
x_train = pd.DataFrame(x_train[:sample_size, :])
y_train = y_train[:sample_size]
```

```
(60000, 784)
```

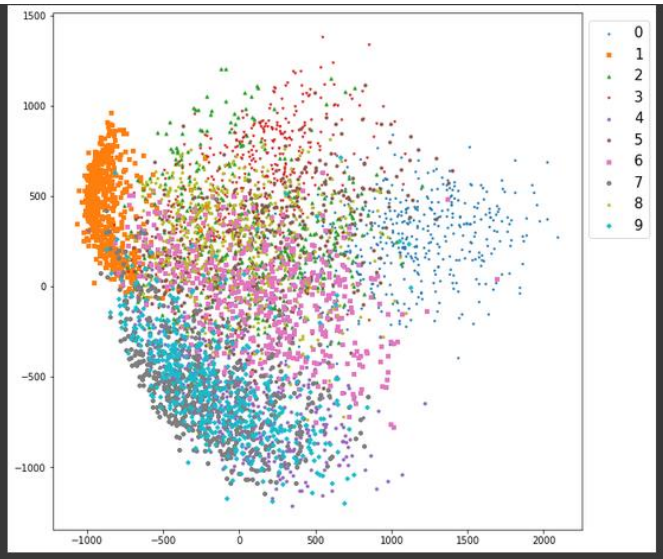
PCA

PCA is a popular method for dimensionality reduction that aims to find a new set of variables (principal components) that explain the maximum variance in the original data. While PCA is often used for dimensionality reduction, it does not assume that each feature follows a normal distribution. Instead, PCA is based on the covariance matrix of the data, which can be calculated regardless of the distribution of the data.

However, PCA can be influenced by outliers, and non-normal distributions may cause issues for PCA. If the data has a distorted distribution, it may be necessary to preprocess the data to transform or normalize the data distribution before applying PCA. In such cases, other techniques such as kernel PCA or non-linear dimensionality reduction methods like t-SNE and UMAP may be more appropriate.

Overall, while PCA is a useful method for dimensionality reduction, it is important to consider the distribution of the data and its sensitivity to outliers before applying PCA. Other techniques may be necessary if the data has a non-normal distribution or non-linear relationships.

```
[46] from sklearn.decomposition import PCA

    # Create a PCA object
    pca = PCA()

    # Use PCA to transform the training data into a lower-dimensional space
    x_pca = pca.fit_transform(x_train)

    # Define a list of markers to use for the scatter plot
    markers = ['.', ',', '^', '1', 'H', '8', 's', 'P', '*', 'D']

    # Create a new figure for the scatter plot
    plt.figure(figsize=(10,10))

    # Loop through each class label and plot its data points in the 2D space
    for i, marker in enumerate(markers):
        # Select only the data points with the current class label
        mask = y_train == i
        # Plot the selected data points with the corresponding marker and label
        plt.scatter(x_pca[mask, 0], x_pca[mask, 1], label=i, s=10, alpha=1, marker=marker)

    # Add a legend to the plot
    plt.legend(bbox_to_anchor=(1.00, 1), loc='upper left', fontsize=15)
```



As we can see, PCA does not do a great job of visualizing the separation of digits, because PCA is not great with non-linear data. With highly dimensional data, we can employ other methods to better understand the distribution of data.

t-SNE

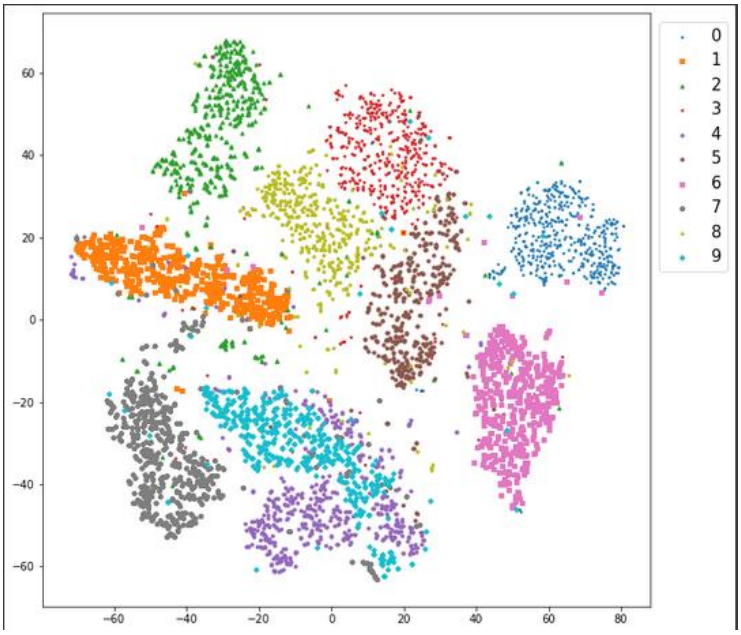
t-SNE is often used for visualization purposes, particularly in cases where the relationships between data points are complex and nonlinear. By mapping high-dimensional data onto a lower-dimensional space (typically 2D or 3D), t-SNE can help to reveal underlying patterns or structures in the data that might be difficult to discern in the original feature space.

One of the advantages of t-SNE is its ability to capture nonlinear relationships between data points, which can be important in many machine learning tasks. However, this flexibility comes at a cost, as t-SNE can be computationally expensive, particularly for high-dimensional datasets. As a result, it may not be practical to use t-SNE for compression beyond two or three dimensions, and alternative methods such as PCA or UMAP may be more appropriate in these cases.

Another important consideration when using t-SNE is the choice of hyperparameters, such as the perplexity or learning rate. These parameters can have a significant impact on the quality of the resulting embedding, and it is often necessary to experiment with different values to find the best settings for a particular dataset or task.

Overall, t-SNE is a powerful tool for visualizing and exploring high-dimensional data, and can be particularly useful for identifying nonlinear relationships and patterns in the data. However, it is important to be aware of its limitations and to use it judiciously in conjunction with other dimensionality reduction techniques.

```
[47] from sklearn.manifold import TSNE

     # Import the t-SNE module
     from sklearn.manifold import TSNE

     # Instantiate the t-SNE object with 2 components
     tsne = TSNE(n_components=2)
     # Fit the t-SNE model to the training data and transform it to 2D
     x_tsne = tsne.fit_transform(x_train)

     # Set up the plot
     plt.figure(figsize=(10,10))

     # Loop through each class and plot it in a different color and shape
     for i, marker in enumerate(markers):
         mask = y_train == i  # Create a boolean mask to select the data of a particular class
         plt.scatter(x_tsne[mask, 0], x_tsne[mask, 1], label=i, s=10, alpha=1, marker=marker)

     # Add a legend to the plot
     plt.legend(bbox_to_anchor=(1.00, 1), loc='upper left', fontsize=15)
```



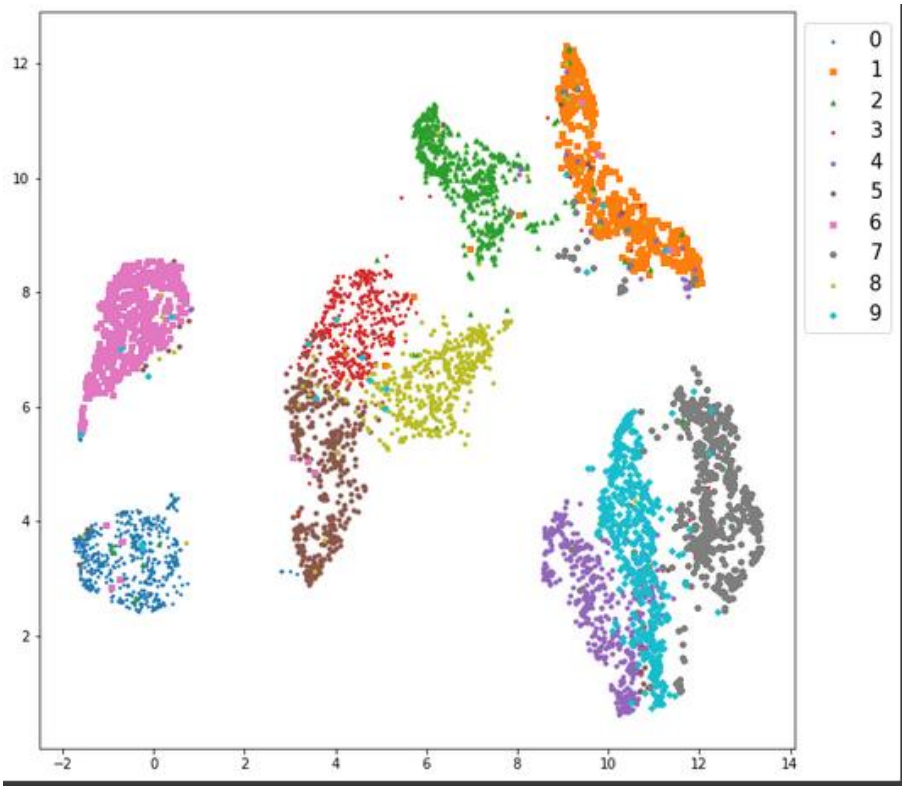
UMAP

UMAP is a powerful and efficient technique for nonlinear dimensionality reduction, which can be faster and more scalable than t-SNE in many cases. It is particularly well-suited for high-dimensional and sparse datasets, such as those encountered in natural language processing or genomics, and it can often produce high-quality embeddings with fewer computational resources.

One of the key advantages of UMAP over t-SNE is its ability to handle new data in an online fashion, which can be useful in many machine learning applications. This means that once the UMAP model has been trained on a dataset, it can be used to embed new data points as soon as they become available, without the need to recompute the entire embedding. This can make it easier to integrate UMAP into a larger machine learning pipeline, and can also be useful for tasks such as anomaly detection or outlier analysis.

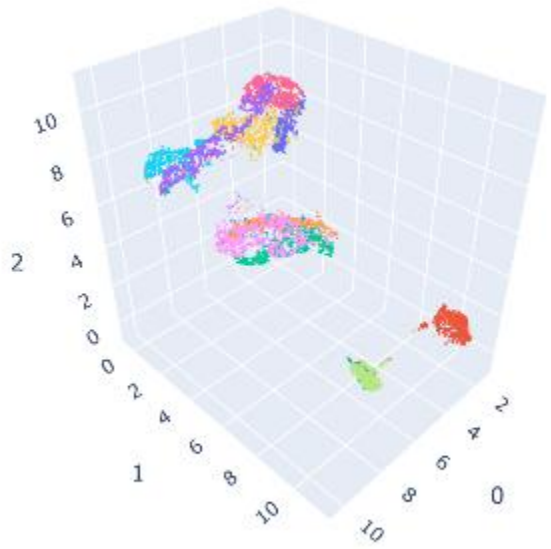
Overall, UMAP is a powerful tool for exploratory data analysis, visualization, and feature engineering, and can be particularly useful in cases where other dimensionality reduction techniques may be impractical or computationally expensive.

```
[48] # Import the UMAP module
     import umap

     # Instantiate the UMAP object
     um = umap.UMAP()

     # Fit the UMAP model to the training data and transform it to 2D
     x_umap = um.fit_transform(x_train)

     # Set up the plot
     plt.figure(figsize=(10,10))

     # Loop through each class and plot it in a different color and shape
     for i, marker in enumerate(markers):
         mask = y_train == i  # Create a boolean mask to select the data of a particular class
         plt.scatter(x_umap[mask, 0], x_umap[mask, 1], label=i, s=10, alpha=1, marker=marker)

     # Add a legend to the plot
     plt.legend(bbox_to_anchor=(1.00, 1), loc='upper left', fontsize=15)
```



We can plot the points in 3-D to better separate the non-linearity and add dimensionality, so we can understand the label discrepancy better.

```
[53] # Import necessary libraries
     import plotly
     import plotly.express as px
     from umap import UMAP

     # Create a 3D UMAP model with random initialization and a fixed random state
     umap_3d = UMAP(n_components=3, init='random', random_state=0)

     # Fit the model to the training data and transform the data
     x_umap = umap_3d.fit_transform(x_train)

     # Convert the transformed data to a Pandas DataFrame
     umap_df = pd.DataFrame(x_umap)

     # Convert y_train to a Pandas Series with the name 'label'
     y_train_sr = pd.Series(y_train,name='label')

     # Concatenate the UMAP DataFrame and the y_train Series along axis 1
     new_df = pd.concat([umap_df,y_train_sr],axis=1)

     # Create a 3D scatter plot with Plotly Express
     fig = px.scatter_3d(
         new_df, x=0, y=1, z=2, # Set the x, y, and z columns of the DataFrame as the plot axes
         color='label', labels={'color': 'number'} # Set the color column to the 'label' column and rename it 'number'
     )

     # Set the marker size to 1 and show the plot
     fig.update_traces(marker_size=1)
     fig.show()
```



As we can see, UMAP was able to fully separate the data and allow us to view the data from a different vantage point.

Now we can begin modeling the data, using MLP, once we have explored the data well.

The MNIST dataset has handwritten digits from 0-9, with a training set of 60,000 images, and a test set of 10,000 images. The data was normalized to fit inside 28x28 pixel boxes with a luminosity of grayscale pixels from 0-255 (white – black). The purpose of this dataset is to take the images and make an MLP model predict what digit they are through classification of 10 different possible outputs (0-9).

The analysis began by importing libraries and splitting the data (which is already split from the data source):

```
#Import modules
"""
    Sequential: This is the basic Keras model object.
    It allows you to create a linear stack of layers that can be easily trained and evaluated.

    Activation: This is a layer object that applies an activation function
    to the output of the previous layer. It is used to introduce non-linearity
    into the model and can be used with various activation functions such as relu, sigmoid, and tanh.

    Flatten: This layer object is used to flatten the input into a
    one-dimensional array. It is commonly used as the first layer of a
    neural network when the input data is structured as a multi-dimensional array.

    Dense: This layer object creates a fully connected layer of neurons.
    It takes the output of the previous layer and applies a linear
    transformation to it, followed by an activation function. The Dense layer
    has several parameters, such as the number of neurons, activation function,
    and initialization method, that can be used to customize the layer.
"""
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
```

In this MLP model development, we will utilize TensorFlow and its modules from the Keras library to create a neural network model. As we can see from the code output, the four different imports are defined for their use cases.

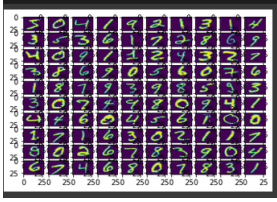
We then transform the data float type data and divide each data image pixel by 255 in order to normalize the data. We then print the description of the data of training and testing and see the shape of the data:

```
# Turn data to float
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

#normalize data by dividing by 255 (luminosity range)
x_train /= 255
x_test /= 255

#Print data output
print("Feature matrix x train:", x_train.shape)
print("Target matrix x test:", x_test.shape)
print("Feature matrix y train:", y_train.shape)
print("Target matrix y test:", y_test.shape)
```

```
Feature matrix x train: (60000, 28, 28)
Target matrix x test: (10000, 28, 28)
Feature matrix y train: (60000,)
Target matrix y test: (10000,)
```

We then plot the data to show a random sample of how the handwritten digits look, just to get a sense of how the data would then be categorized after stratified sampling occurs:

```
#plot a sample of the data in 28x28 pixels
#10 x 10 rows and columns
"""
The loop iterates over two ranges, i and j, and for each pair of i and j, it calls the imshow() function on the array element ax[i][j]. This function displays an image using the data in the array.
The image data is taken from the x_train array at index n, which is reshaped into a 28x28 pixel image.
The n variable is incremented by 1 for each iteration of the loop, so the next image in the x_train array will be used for the next iteration of the loop.
Once all images have been plotted, the plt.show() function is called to display the grid of images
"""
fig, ax = plt.subplots(10, 10)
n = 0
for i in range(10):
    for j in range(10):
        ax[i][j].imshow(x_train[n].reshape(28, 28), aspect='auto')
        n += 1
plt.show()
```



We then create the model with TensorFlow's abilities to create a sequential model object to develop a linear stack of layers that can be easily trained and evaluated:

```
#tf creation of MLP model with sequential stack of layers
mlpModel = Sequential([
    #Reshape and flatten data to 28x28 rows
    Flatten(input_shape=(28, 28)),
    #Layer 1 of neurons, with sigmoid activation function
    Dense(200, activation='sigmoid'),
    #Layer 2 of neurons, with sigmoid activation function
    Dense(100, activation='sigmoid'),
    #Layer 3 of neurons, with sigmoid activation function
    Dense(10, activation='sigmoid'),
])
```

We give the model two hidden layers with 200 and 100 neurons each, respectively, and provide it with sigmoid activation function to introduce non-linearity into the model. The data is first flattened into a one-dimensional array for the first layer of the neural network. The final dense layer of the neural network is given 10 neurons as the output of the previous layer.

```
"""
This code snippet is compiling a neural network model using the Keras library
in Python.

The model is being compiled with the 'adam' optimizer, which is a popular
stochastic gradient descent optimization algorithm that is often used in
deep learning, and can improve training speed and accuracy of NN.

The loss function used is 'sparse_categorical_crossentropy', which is commonly
used for multi-class classification problems where the labels are integers.

The metrics being used to evaluate the model during training and testing are
'accuracy', which is a common metric used for classification problems to
measure the percentage of correctly classified instances.

Overall, this code is configuring the model for training by specifying the
optimizer, loss function, and evaluation metrics.
"""
mlpModel.compile(optimizer='adam',
                 loss='sparse_categorical_crossentropy',
                 metrics=['accuracy'])
```

Once the model is instantiated, we then utilize the compile function from TensorFlow to provide it an optimizer, loss function, and metrics to best build the MLP model for us, as shown above. We use ‘adam’ as the optimizer to improve training speed and accuracy of the NN. We also use a sparse categorical cross entropy loss function because we have a multi-class classification problem, and our labels are integers. Finally, we have accuracy as the metric being used to evaluate the model during testing and training, to measure the percentage of correctly classified instances of handwritten digits being identified.

Then, we run the model and iterate it with 20 epochs, with a batch size of 2,000 training samples at a time before updating weights, and utilize a 75:25 train/test split:

```
#Model will iterate over the entire training dataset 20 times,
#Batch size of 2000, so model will process 2000 training examples at a time
#before updating weights
#Validation split is 0.25, so 25% of training data is used as validation
mlpModel.fit(x_train, y_train, epochs=20,
          batch_size=2000,
          validation_split=0.25)
```

```
Epoch 1/20
23/23 [==============================] - 3s 75ms/step - loss: 2.1761 - accuracy: 0.3639 - val_loss: 1.9202 - val_accuracy: 0.6701
Epoch 2/20
23/23 [==============================] - 2s 88ms/step - loss: 1.6723 - accuracy: 0.6820 - val_loss: 1.3611 - val_accuracy: 0.7496
Epoch 3/20
23/23 [==============================] - 1s 62ms/step - loss: 1.1573 - accuracy: 0.7816 - val_loss: 0.9331 - val_accuracy: 0.8216
Epoch 4/20
23/23 [==============================] - 1s 57ms/step - loss: 0.8221 - accuracy: 0.8361 - val_loss: 0.6799 - val_accuracy: 0.8645
Epoch 5/20
23/23 [==============================] - 1s 60ms/step - loss: 0.6226 - accuracy: 0.8676 - val_loss: 0.5304 - val_accuracy: 0.8848
Epoch 6/20
23/23 [==============================] - 1s 50ms/step - loss: 0.5017 - accuracy: 0.8854 - val_loss: 0.4419 - val_accuracy: 0.8953
Epoch 7/20
23/23 [==============================] - 1s 53ms/step - loss: 0.4274 - accuracy: 0.8968 - val_loss: 0.3868 - val_accuracy: 0.9021
Epoch 8/20
23/23 [==============================] - 1s 56ms/step - loss: 0.3786 - accuracy: 0.9037 - val_loss: 0.3509 - val_accuracy: 0.9090
Epoch 9/20
23/23 [==============================] - 1s 52ms/step - loss: 0.3451 - accuracy: 0.9088 - val_loss: 0.3234 - val_accuracy: 0.9135
Epoch 10/20
23/23 [==============================] - 1s 54ms/step - loss: 0.3191 - accuracy: 0.9142 - val_loss: 0.3031 - val_accuracy: 0.9175
Epoch 11/20
23/23 [==============================] - 2s 80ms/step - loss: 0.2989 - accuracy: 0.9180 - val_loss: 0.2873 - val_accuracy: 0.9211
Epoch 12/20
23/23 [==============================] - 2s 66ms/step - loss: 0.2824 - accuracy: 0.9217 - val_loss: 0.2730 - val_accuracy: 0.9243
Epoch 13/20
23/23 [==============================] - 1s 57ms/step - loss: 0.2675 - accuracy: 0.9253 - val_loss: 0.2618 - val_accuracy: 0.9264
Epoch 14/20
23/23 [==============================] - 1s 53ms/step - loss: 0.2548 - accuracy: 0.9286 - val_loss: 0.2519 - val_accuracy: 0.9292
Epoch 15/20
23/23 [==============================] - 1s 58ms/step - loss: 0.2437 - accuracy: 0.9311 - val_loss: 0.2426 - val_accuracy: 0.9305
Epoch 16/20
23/23 [==============================] - 1s 57ms/step - loss: 0.2333 - accuracy: 0.9341 - val_loss: 0.2343 - val_accuracy: 0.9325
Epoch 17/20
23/23 [==============================] - 1s 57ms/step - loss: 0.2239 - accuracy: 0.9366 - val_loss: 0.2263 - val_accuracy: 0.9349
Epoch 18/20
23/23 [==============================] - 1s 48ms/step - loss: 0.2148 - accuracy: 0.9389 - val_loss: 0.2193 - val_accuracy: 0.9367
Epoch 19/20
23/23 [==============================] - 1s 55ms/step - loss: 0.2065 - accuracy: 0.9414 - val_loss: 0.2128 - val_accuracy: 0.9391
Epoch 20/20
23/23 [==============================] - 2s 72ms/step - loss: 0.1991 - accuracy: 0.9435 - val_loss: 0.2072 - val_accuracy: 0.9413
<keras.callbacks.History at 0x7fede20ad790>
```

The model runs for 20 iterations and stops, outputting each iteration change in loss and accuracy.

We then output the final data accuracy with a print function:

```
#Print results of model accuracy
results = mlpModel.evaluate(x_test,  y_test, verbose = 0)
print('Test loss & Test accuracy:', results)
```

```
Test loss & Test accuracy: [0.20260195434093475, 0.940500020980835]
```

The overall accuracy after 20 epochs was 94.05%, meaning that 94% of handwritten digits were correctly classified. We could improve accuracy by further fine-tuning epoch and batch size to allow the data to continue to iterate and improve for a better score, but overall, the 94% was an improvement after constant tuning of details and ending at this point.