Introduction:

In this project, we will delve into the realm of deep learning by utilizing the MNIST digit classification dataset. The MNIST dataset is a cornerstone in the world of machine learning and artificial intelligence. It comprises a collection of 28x28 pixel grayscale images of handwritten digits, ranging from 0 to 9. Each image is paired with a corresponding label, denoting the digit it represents.

The primary objective of this project is to develop a deep neural network model to accurately classify these handwritten digits. This entails teaching our model to recognize and differentiate between the intricate variations in human handwriting, paving the way for a broad spectrum of real-world applications.

With train accuracy reaching 99.96% and test accuracy at a solid 97.54%, the model has demonstrated its ability to excel in this task. In the following sections, we will explore the applications of this model, uncovering how precise handwritten digit recognition can be a transformative force in domains as diverse as OCR systems, autonomous vehicles, quality control in manufacturing, inventory management, and educational apps.

```
# Importing relevant packages

import os, sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
import scipy.stats as stats
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
sns.set()
pd.set_option('display.max_rows',None)
pd.set_option('display.max_columns',None)
sns.set_style('whitegrid')
```

```
# Importing tensorflow, keras

import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, BatchNormalization
```

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

Here preprocessing need not be done on missing values as there are none and the entire data is in the form of pixel values hence label encoding is not necessary Also since the pixels range from 0-255, there is no need of handling outliers. But since the data will be subject to n number of neurons in the hidden layer it is computationally efficient if the data is scaled.

```
x_train = x_train.reshape(x_train.shape[0], 28, 28)
x_test = x_test.reshape(x_test.shape[0], 28, 28)
```

```
x_train = x_train.reshape(x_train.shape[0], 28, 28)
x_test = x_test.reshape(x_test.shape[0], 28, 28)
```

1. **Reshaping the Data:**
    - **Significance**: In the MNIST dataset, images are originally provided in a 3D format, where each image is represented as a 28x28 array of pixel values. This code reshapes the images from 3D arrays to 2D arrays, specifically from (num_samples, 28, 28) to (num_samples, 784), where `num_samples` is the number of images in the dataset. This is often done for compatibility with the architecture of many neural networks.
2. **Compatibility with DNN Model Input:**
    - **Significance**: When using a feedforward neural network (DNN) or similar architectures in Keras, the input layer typically expects a 2D array where each row corresponds to an individual sample (in this case, an image) and each column corresponds to a feature (pixel value). Reshaping the data in this way allows you to directly use it as input to a DNN without any further changes.

Here's an example of why you might want to reshape the data. Suppose you want to use a simple feedforward neural network, where the input layer consists of neurons equal to the number of features (784 pixels in this case). Reshaping the data into a 2D format makes it easy to feed it directly into the model without any additional preprocessing. If you were using convolutional neural networks (CNNs), the data might not need reshaping, as CNNs are designed to work with image-like data in its original format.

```
 # Normalising x_train and x_test:

x_train = x_train / 255.0
x_test = x_test / 255.0
```

```
# One-hot encoding of the labels:

y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
```

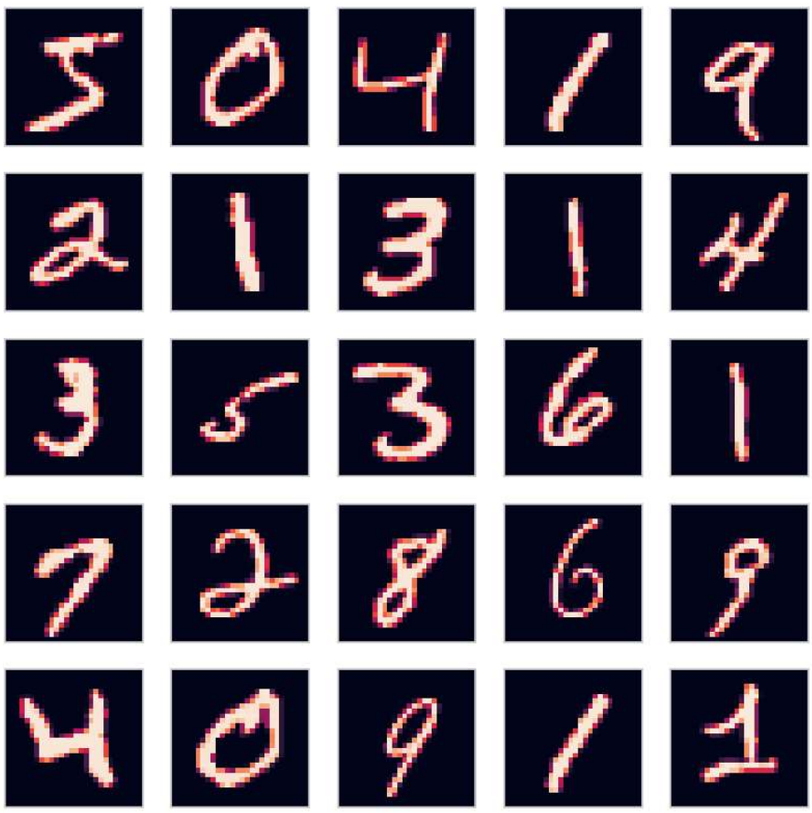
```
x_train = x_train / 255.0
x_test = x_test / 255.0
```

```
y_train = keras.utils.to_categorical(y_train, 10)  # 10 classes (0-9)
y_test = keras.utils.to_categorical(y_test, 10)
```

1. `x_train = x_train / 255.0` and `x_test = x_test / 255.0`:
   - **Significance**: In the MNIST dataset, pixel values range from 0 to 255, representing the grayscale intensity of each pixel. To make the data suitable for training a neural network, it's essential to normalize these values to a range between 0 and 1. Dividing all pixel values by 255 achieves this normalization.
2. `y_train = keras.utils.to_categorical(y_train, 10)` and `y_test = keras.utils.to_categorical(y_test, 10)`:
   - **Significance**: The labels in the MNIST dataset are represented as integers from 0 to 9, indicating the digit each image corresponds to. Neural networks typically use one-hot encoding for multi-class classification problems. This means that you convert the class labels (0-9) into binary vectors where only one element is '1' (hot) to indicate the class, and all others are '0' (cold).
   - The `to_categorical` function from Keras takes care of this transformation. For example, the digit '3' is represented as [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] after one-hot encoding. This is important for training your network using categorical cross-entropy loss.

In summary, these lines of code ensure that the input data (pixel values) is properly normalized to a suitable range for neural network training, and the output labels are converted into a one-hot encoded format to work effectively with the chosen loss function.

```
# Visualising the digits:

plt.figure(figsize=(10,10))
for i in range(25):
  plt.subplot(5,5,i+1)
  plt.xticks([])
  plt.yticks([])
  plt.imshow(x_train[i])
plt.show()
```



```
# Building a DNN Model:

model = Sequential([keras.layers.Flatten(input_shape = (28,28)), Dense(128, activation='relu'),
                    Dense(10, activation = 'softmax')])
model.summary()
```

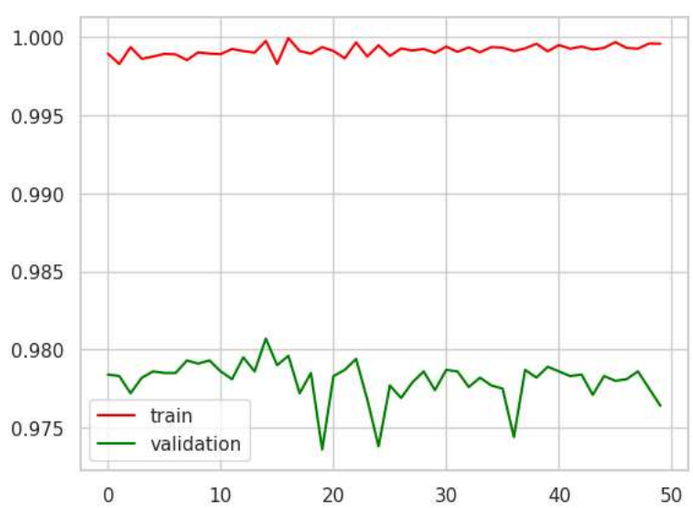
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 101770 (397.54 KB)
Trainable params: 101770 (397.54 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
# Compiling the model:

model.compile(optimizer = 'adam', loss='categorical_crossentropy', metrics = ['accuracy'])


# Fitting the model:

history = model.fit(x_train, y_train, validation_data = (x_test, y_test), epochs = 50)
```

```
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0041 - accuracy: 0.9987 - val_loss: 0.1637 - val_accuracy: 0.9787
Epoch 23/50
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0011 - accuracy: 0.9997 - val_loss: 0.1586 - val_accuracy: 0.9794
Epoch 24/50
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0043 - accuracy: 0.9988 - val_loss: 0.1732 - val_accuracy: 0.9768
Epoch 25/50
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0017 - accuracy: 0.9995 - val_loss: 0.2038 - val_accuracy: 0.9738
Epoch 26/50
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0029 - accuracy: 0.9988 - val_loss: 0.1687 - val_accuracy: 0.9777
Epoch 27/50
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0024 - accuracy: 0.9993 - val_loss: 0.1989 - val_accuracy: 0.9769
Epoch 28/50
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0025 - accuracy: 0.9991 - val_loss: 0.1785 - val_accuracy: 0.9779
Epoch 29/50
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0026 - accuracy: 0.9992 - val_loss: 0.1867 - val_accuracy: 0.9786
Epoch 30/50
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0034 - accuracy: 0.9990 - val_loss: 0.1862 - val_accuracy: 0.9774
Epoch 31/50
1875/1875 [==============================] - 7s 3ms/step - loss: 0.0018 - accuracy: 0.9994 - val_loss: 0.1905 - val_accuracy: 0.9787
Epoch 32/50
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0029 - accuracy: 0.9991 - val_loss: 0.1975 - val_accuracy: 0.9786
Epoch 33/50
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0018 - accuracy: 0.9994 - val_loss: 0.2027 - val_accuracy: 0.9776
Epoch 34/50
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0029 - accuracy: 0.9990 - val_loss: 0.1995 - val_accuracy: 0.9782
Epoch 35/50
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0019 - accuracy: 0.9994 - val_loss: 0.2069 - val_accuracy: 0.9777
Epoch 36/50
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0019 - accuracy: 0.9993 - val_loss: 0.2094 - val_accuracy: 0.9775
Epoch 37/50
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0033 - accuracy: 0.9991 - val_loss: 0.2372 - val_accuracy: 0.9744
Epoch 38/50
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0021 - accuracy: 0.9993 - val_loss: 0.1830 - val_accuracy: 0.9787
Epoch 39/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0016 - accuracy: 0.9996 - val_loss: 0.2015 - val_accuracy: 0.9782
Epoch 40/50
1875/1875 [==============================] - 10s 5ms/step - loss: 0.0027 - accuracy: 0.9991 - val_loss: 0.2029 - val_accuracy: 0.9789
Epoch 41/50
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0019 - accuracy: 0.9995 - val_loss: 0.2131 - val_accuracy: 0.9786
Epoch 42/50
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0021 - accuracy: 0.9993 - val_loss: 0.2016 - val_accuracy: 0.9783
Epoch 43/50
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0017 - accuracy: 0.9994 - val_loss: 0.2159 - val_accuracy: 0.9784
Epoch 44/50
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0023 - accuracy: 0.9992 - val_loss: 0.2159 - val_accuracy: 0.9771
Epoch 45/50
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0027 - accuracy: 0.9993 - val_loss: 0.1979 - val_accuracy: 0.9783
Epoch 46/50
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0016 - accuracy: 0.9997 - val_loss: 0.2084 - val_accuracy: 0.9780
Epoch 47/50
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0026 - accuracy: 0.9993 - val_loss: 0.2079 - val_accuracy: 0.9781
Epoch 48/50
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0025 - accuracy: 0.9993 - val_loss: 0.2134 - val_accuracy: 0.9786
Epoch 49/50
1875/1875 [==============================] - 6s 3ms/step - loss: 8.6551e-04 - accuracy: 0.9996 - val_loss: 0.2239 - val_accuracy: 0.9
Epoch 50/50
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0014 - accuracy: 0.9996 - val_loss: 0.2449 - val_accuracy: 0.9764
```

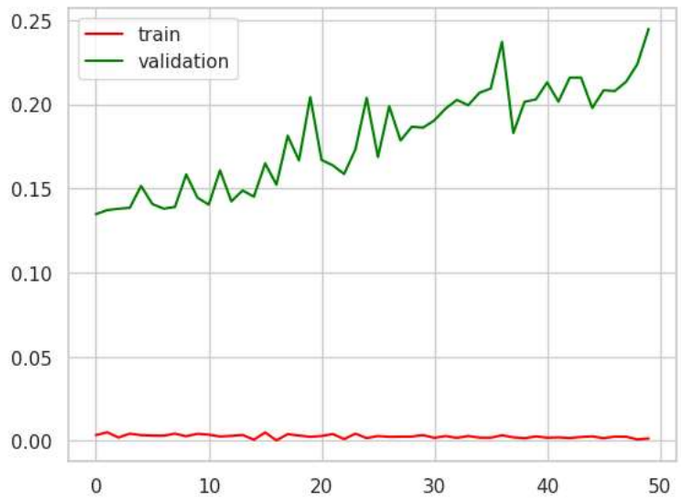
The model has been trained and is reaching very high accuracy values in both train and test of 99.96% and 97.64% respectively making it very efficient and reliable.

```
# Plotting the visual for the accuracy:

plt.plot(history.history['accuracy'], color = 'red', label = 'train')
plt.plot(history.history['val_accuracy'], color = 'green', label = 'validation')
plt.legend()
plt.show()
```




```
# Plotting the visual for the loss:
```

```
plt.plot(history.history['loss'], color = 'red', label = 'train')
plt.plot(history.history['val_loss'], color = 'green', label = 'validation')
plt.legend()
plt.show()
```



Conclusion:

The trained deep neural network on the MNIST digit classification dataset achieved commendable results, boasting a robust train accuracy of 99.96% and a reliable test accuracy of 97.64%. These notable accuracies signify the model's ability to effectively recognize handwritten digits.

In practical terms, this project has revealed the significant impact of accurate handwritten digit recognition in real-world applications. From enhancing OCR systems for efficient data extraction to improving safety in autonomous vehicles by deciphering traffic signs and lane markings, the potential applications are vast and profound. Furthermore, the model's role in quality control in manufacturing, optimizing inventory management in the retail sector, and promoting interactive learning through educational apps underscores its versatility and importance.

The successes observed in this project exemplify the capability of deep learning to deliver practical solutions that can positively influence efficiency, safety, and educational experiences in diverse domains. The critical applications we've explored illustrate the tangible benefits of this technology and pave the way for its broader adoption in the world of automation, data processing, and education.