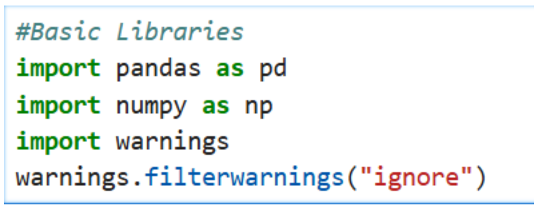
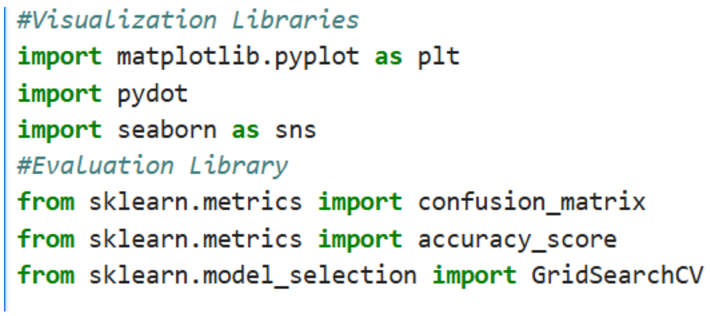


```
#Basic Libraries
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings("ignore")
```

1.Pandas –powerful tools for data manipulation and analysis , commonly used for working with structured data like tables and spreadsheet.

2.Numpy provide support for working with arrays and matrices of numerical data

3.This line set filter for warning msg instructing python can ignore , certain warning msg not relevant to my current task



```
#Visualization Libraries
import matplotlib.pyplot as plt
import pydot
import seaborn as sns
#Evaluation Library
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV
```

1.This line imports the pyplot module from the matplotlib library.create various types of visualizations, such as line plots, scatter plots, bar charts, histograms.

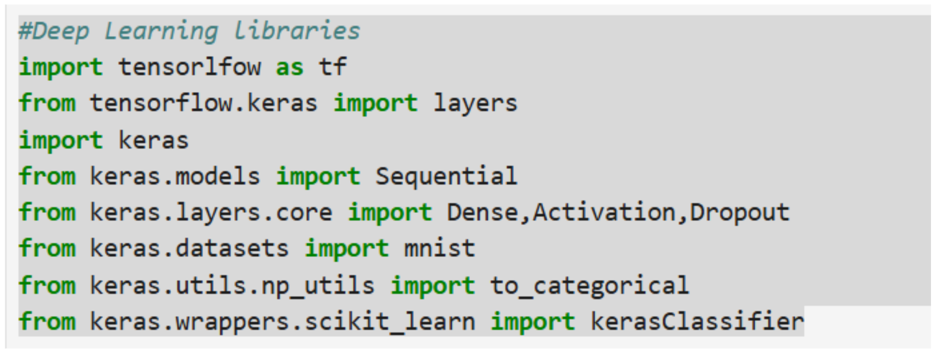
2.which is used for working with DOT files- visualize decision tree and other graph-based models

3.seaborn simplifies the process of creating common types of plots, such as scatter plots, histograms, box plots, and violin plots, and it also offers advanced features for visualizing complex datasets.

4.This line imports the confusion_matrix function from the sklearn.metrics module. A confusion matrix is a table that is often used to evaluate the performance of a classification model. It shows the number of true positives, true negatives, false positives, and false negatives, which can be used to calculate various evaluation metrics, such as accuracy, precision, recall, and F1-score

5.This line imports the accuracy_score function from the sklearn.metrics module. Accuracy is a commonly used evaluation metric for classification models that measures the proportion of correctly classified instances out of the total number of instances

6.This line imports the GridSearchCV class from the sklearn.model_selection module. GridSearchCV is a tool for hyperparameter tuning, which is the process of finding the best combination of hyperparameters for a machine learning model. Hyperparameters are parameters that are not learned from the data but are set before training the model. GridSearchCV automates the process of trying different hyperparameter combinations and selecting the one that yields the best performance



```
#Deep Learning Libraries
import tensorlfow as tf
from tensorflow.keras import layers
import keras
from keras.models import Sequential
from keras.layers.core import Dense,Activation,Dropout
from keras.datasets import mnist
from keras.utils.np_utils import to_categorical
from keras.wrappers.scikit_learn import kerasClassifier
```

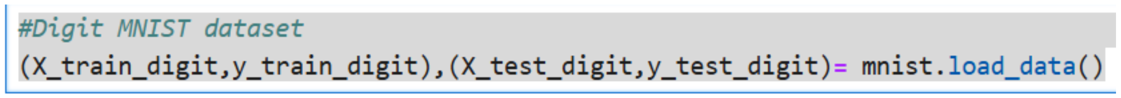
This code imports several libraries essential for building and working with deep learning models, specifically using TensorFlow and Keras:

- import tensorflow as tf: This line imports the TensorFlow library, a powerful open-source framework for numerical computation and large-scale machine learning. TensorFlow provides the foundation for building and training deep learning models.
- from tensorflow.keras import layers: This line imports the layers module from the Keras library, which is a high-level API built on top of TensorFlow. Keras simplifies the process of defining and training neural network models by providing a user-friendly interface and a collection of pre-built layers.
- import keras: This line imports the Keras library directly. While Keras is now integrated into TensorFlow, this import provides access to additional Keras-specific functionalities.
- from keras.models import Sequential: This line imports the Sequential class from Keras, which is used to create a linear stack of layers for building feedforward neural networks.
- from keras.layers.core import Dense, Activation, Dropout: This line imports several core layer types from Keras:
    - Dense: Represents a fully connected layer, where each neuron in the layer is connected to every neuron in the previous layer.
    - Activation: Specifies the activation function to be applied to the output of a layer. Common activation functions include ReLU, sigmoid, and softmax.
- Dropout: Implements dropout regularization, which randomly deactivates a fraction of neurons during training to prevent overfitting.

 from keras.datasets import mnist: This line imports the MNIST dataset, a collection of handwritten digits commonly used for training and evaluating image classification models.

 from keras.utils.np_utils import to_categorical: This line imports the to_categorical function, which converts class labels into a one-hot encoded format suitable for categorical classification tasks.

 from keras.wrappers.scikit_learn import KerasClassifier: This line imports the KerasClassifier class, which allows you to wrap a Keras model as an estimator compatible with scikit-learn's machine learning pipeline. This enables using scikit-learn's tools, such as grid search and cross-validation, to tune and evaluate Keras models.



```
#Digit MNIST dataset
(X_train_digit,y_train_digit),(X_test_digit,y_test_digit)= mnist.load_data()
```

This code loads the MNIST dataset, a classic dataset in machine learning that contains images of handwritten digits. Let's break down the code step by step:

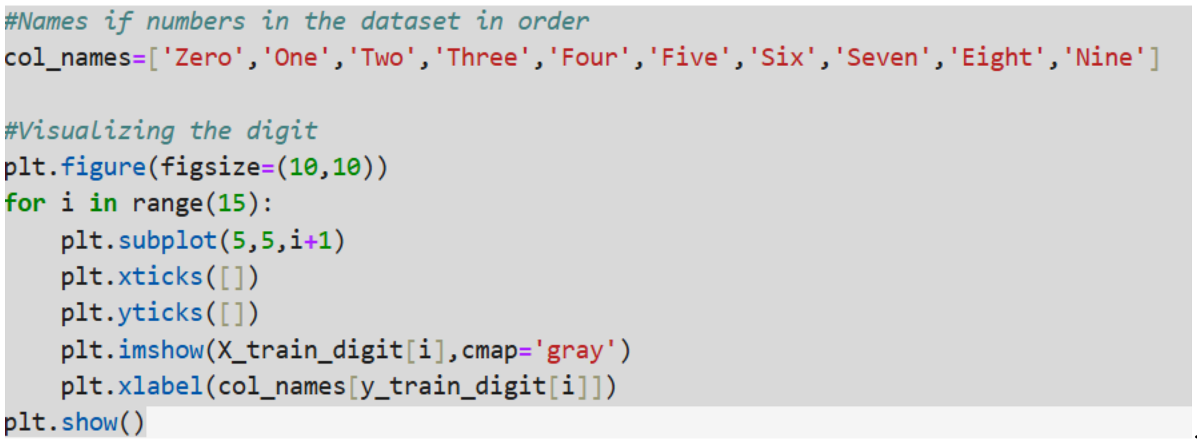
1. (X_train_digit, y_train_digit), (X_test_digit, y_test_digit) = ...: This line unpacks the MNIST dataset into four variables:
   - X_train_digit: Contains the training images.
   - y_train_digit: Contains the labels for the training images.
   - X_test_digit: Contains the test images.
   - y_test_digit: Contains the labels for the test images.
2. mnist.load_data(): This function call loads the MNIST dataset from Keras's built-in datasets. The dataset is automatically downloaded The MNIST dataset is commonly used for building and evaluating image classification models, particularly for recognizing handwritten digits. It consists of 60,000 training images and 10,000 test images, each of size 28x28 pixels and representing a grayscale image of a handwritten digit from 0 to 9.

By loading the dataset into these four variables, you can access and use the training and test images and their corresponding labels to train and evaluate your machine learning models.

##machine learning, test images refer to a portion of a dataset that is used to evaluate the performance of a trained model. These images are separate from the training images, meaning the model has not seen them during the training process.

Here's why test images are important:

- **Performance Evaluation:** By testing the model on unseen images, you can get an accurate estimate of its generalization ability – its ability to perform well on new, real-world data.



```python
#Names if numbers in the dataset in order
col_names=['Zero','One','Two','Three','Four','Five','Six','Seven','Eight','Nine']

#Visualizing the digit
plt.figure(figsize=(10,10))
for i in range(15):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(X_train_digit[i],cmap='gray')
    plt.xlabel(col_names[y_train_digit[i]])
plt.show()
```

.

1. Defining Column Names

Python
Names if numbers in the dataset in order
col_names = ['Zero', 'One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine']

This line creates a list called col_names. This list contains the names of the digits in the MNIST dataset, from 'Zero' to 'Nine'. This will be used later to label the images.

2. Setting Up the Plot

Python
Visualizing the digit

plt.figure(figsize=(10, 10))

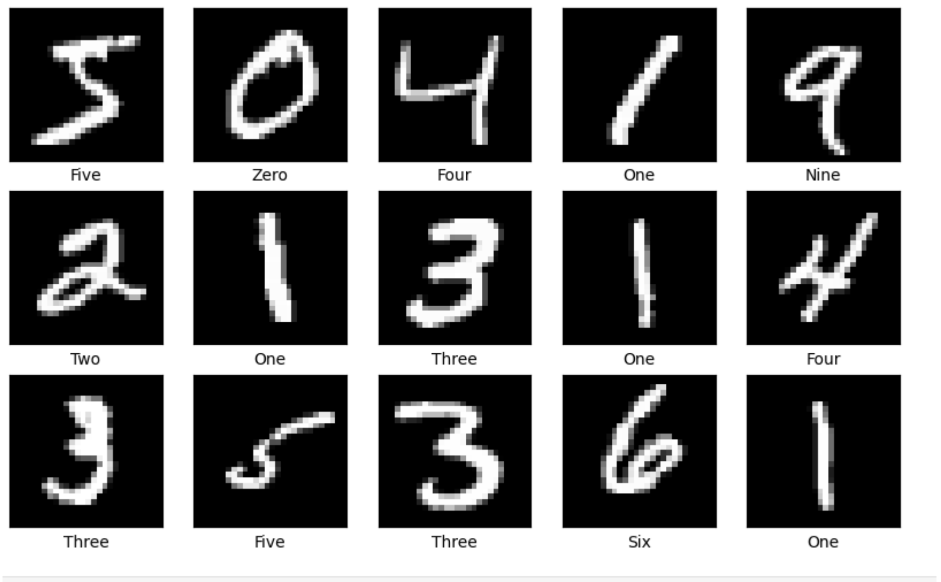
This line uses the plt.figure() function from the matplotlib.pyplot library to create a new figure. The figsize=(10, 10) argument sets the size of the figure to be 10 inches by 10 inches, providing ample space to display the images.

3. Looping Through the Images

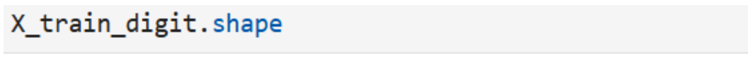
```
Python
for i in range(15):
    plt.subplot(5, 5, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(X_train_digit[i], cmap='gray')
    plt.xlabel(col_names[y_train_digit[i]])
```

This for loop iterates through the first 15 images in the X_train_digit dataset. Inside the loop:

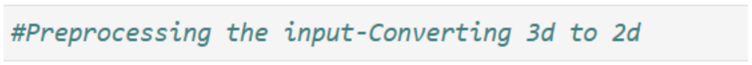
- plt.subplot(5, 5, i+1): This line creates a subplot within the figure. The figure is divided into a 5x5 grid, and i+1 determines the position of the current subplot in the grid.
- plt.xticks([]) and plt.yticks([]): These lines remove the x and y-axis ticks from the subplot, as they are not needed for visualizing the images.
- plt.imshow(X_train_digit[i], cmap='gray'): This line displays the current image from X_train_digit in grayscale using the imshow() function.
- plt.xlabel(col_names[y_train_digit[i]]): This line sets the x-axis label of the subplot to the corresponding digit name from the col_names list. The label is determined by the value in y_train_digit[i], which contains the true label of the image.



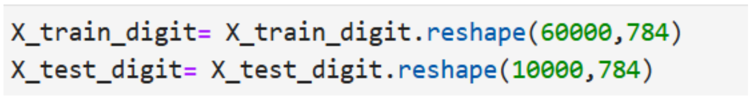
In summary, this code visualizes the first 15 images from the MNIST dataset in a 5x5 grid. Each subplot shows a grayscale image of a handwritten digit, and the x-axis label of each subplot indicates the true digit represented by the image. This visualization helps to understand the structure and content of the MNIST dataset.



```
X_train_digit.shape
```



```
#Preprocessing the input-Converting 3d to 2d
```



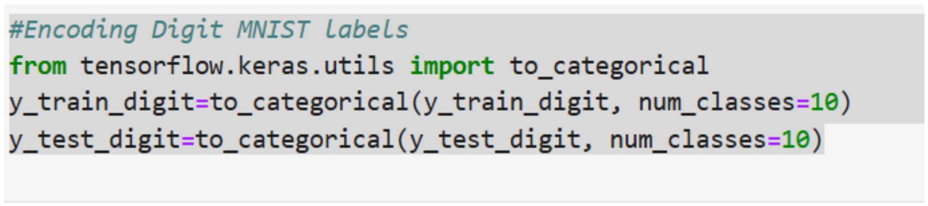
```
X_train_digit= X_train_digit.reshape(60000,784)
X_test_digit= X_test_digit.reshape(10000,784)
```

It was in 3D (60000,28,28) now we have convert to 2D

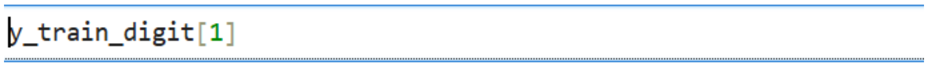
Why this reshaping is necessary:

- **Original Shape:** The original shape of X_train_digit is (60000, 28, 28). This means it's a 3D array containing 60,000 images, each with a size of 28x28 pixels. Similarly, X_test_digit has a shape of (10000, 28, 28).
- **Deep Learning Input:** Most deep learning models, especially those using fully connected layers, expect the input data to be a 2D array where each row represents a sample and each column represents a feature. In this case, each image needs to be flattened into a single row.
- **Reshaping:** The code reshapes the training and testing datasets accordingly:
  - X_train_digit.reshape(60000, 784): This converts the 3D array into a 2D array with 60,000 rows and 784 columns (28x28 = 784). Each row now represents a flattened image where all the pixel values are lined up as features.
  - X_test_digit.reshape(10000, 784): Similarly, this reshapes the testing dataset into a 2D array with 10,000 rows and 784 columns.
- **Why 784?** The number 784 comes from the total number of pixels in each image (28 pixels x 28 pixels = 784 pixels). Each pixel becomes a feature in the flattened representation.

By reshaping the datasets, you transform the image data into a format suitable for input into a deep learning model. This allows the model to learn patterns and relationships from the pixel values to classify handwritten digits effectively.



```
#Encoding Digit MNIST Labels
from tensorflow.keras.utils import to_categorical
y_train_digit=to_categorical(y_train_digit, num_classes=10)
y_test_digit=to_categorical(y_test_digit, num_classes=10)
```



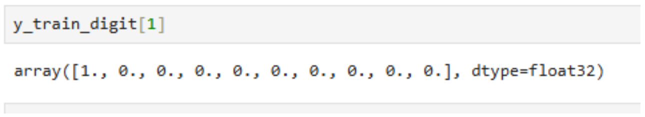
```
y_train_digit[1]
```

Performs **one-hot encoding** on the labels of the MNIST dataset. Let's break it down:

- **from tensorflow.keras.utils import to_categorical:** This line imports the to_categorical function from the keras.utils module in TensorFlow. This function is specifically designed to convert integer labels into a one-hot encoded format.
- **y_train_digit=to_categorical(y_train_digit, num_classes=10):**
  - y_train_digit: This variable initially holds the integer labels of the training images (0 for 'Zero', 1 for 'One', etc.).
  - to_categorical(y_train_digit, num_classes=10): This calls the to_categorical function:
    - y_train_digit: This is the input array of integer labels.
    - num_classes=10: This specifies that there are 10 possible classes (digits 0 to 9).

- The function creates a new array where each row represents a label. The row corresponding to the true class has a value of 1, while all other values are 0.

Example:



```
y_train_digit[1]

array([1., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

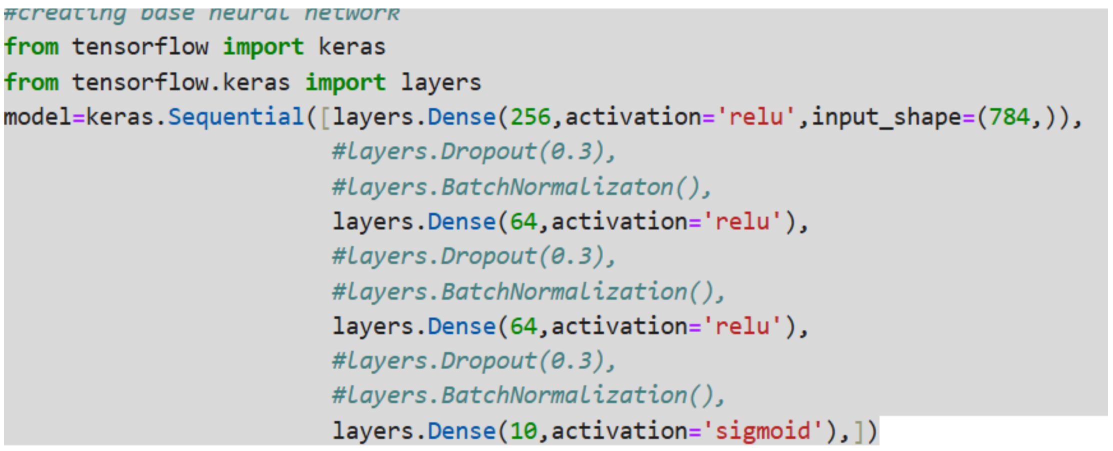
- If y_train_digit[i] is 5 (representing the digit 'Five'), then the corresponding row in the one-hot encoded array will be: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]

- **y_test_digit=to_categorical(y_test_digit, num_classes=10):** This line performs the same one-hot encoding process on the labels of the test images, ensuring consistency between the training and testing data.

Why One-Hot Encoding?

- **Neural Networks:** Many neural network models, especially those with a softmax activation function in the output layer, expect the labels to be in one-hot encoded format.
- **Categorical Data:** One-hot encoding effectively represents categorical data as numerical features, making it suitable for input to machine learning algorithms.

In essence, this code prepares the labels of the MNIST dataset for use in deep learning models by converting them from integer representations to a format that is more suitable for training and evaluation.



```
#creating base neural network
from tensorflow import keras
from tensorflow.keras import layers
model=keras.Sequential([layers.Dense(256,activation='relu',input_shape=(784,)),
                        #layers.Dropout(0.3),
                        #layers.BatchNormalizaton(),
                        layers.Dense(64,activation='relu'),
                        #layers.Dropout(0.3),
                        #layers.BatchNormalization(),
                        layers.Dense(64,activation='relu'),
                        #layers.Dropout(0.3),
                        #layers.BatchNormalization(),
                        layers.Dense(10,activation='sigmoid'),])
```

This code defines a simple feedforward neural network architecture using the Keras library. Let's break down the components:

1. Importing Libraries:

- from tensorflow import keras: This line imports the main Keras module from the TensorFlow library. Keras provides a high-level API for building and training deep learning models.
- from tensorflow.keras import layers: This line imports the layers module, which contains building blocks for creating neural network layers.

2. Creating the Model:

- model = keras.Sequential(...): This creates a Sequential model, which is a linear stack of layers. In this case, the network consists of four layers….-→

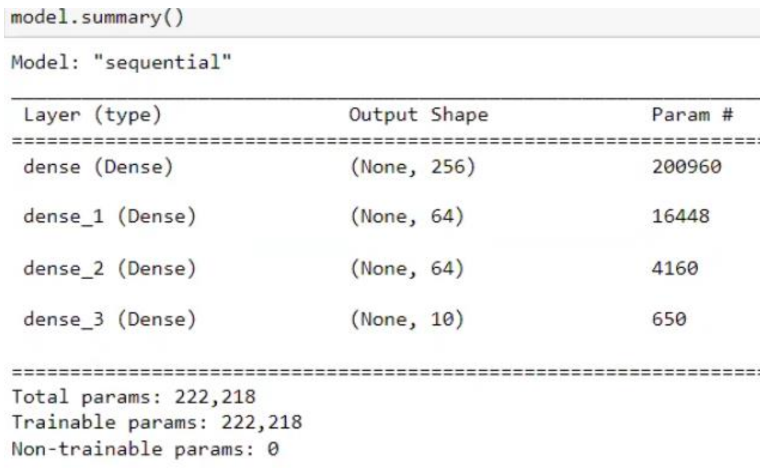
3. Defining Layers:

- layers.Dense(256, activation='relu', input_shape=(784,)): This is the first layer, a fully connected (dense) layer with 256 neurons.
    - activation='relu': The ReLU (Rectified Linear Unit) activation function is applied to the output of this layer, introducing non-linearity into the model.
    - input_shape=(784,): This specifies the shape of the input data, which is a 1D array with 784 features (corresponding to the flattened 28x28 pixel images from the MNIST dataset).
- layers.Dense(64, activation='relu'): This is the second hidden layer with 64 neurons and ReLU activation.
- layers.Dense(64, activation='relu'): This is the third hidden layer with 64 neurons and ReLU activation.
- layers.Dense(10, activation='sigmoid'): This is the output layer with 10 neurons (one for each digit class).
    - activation='sigmoid': The sigmoid activation function is used in the output layer to produce probabilities for each class. The output of the sigmoid function lies between 0 and 1, representing the probability of the input belonging to each class.

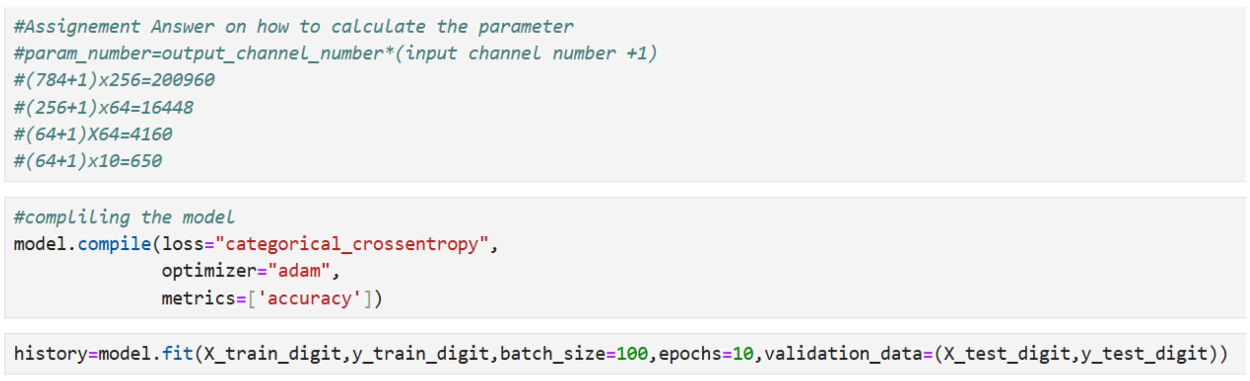
Note:

- The code includes commented-out lines for Dropout and BatchNormalization layers. These layers are commonly used to improve model performance and prevent overfitting, but they are not included in this basic architecture.

This simple neural network architecture provides a foundation for classifying handwritten digits from the MNIST dataset. You can further customize it by adding more layers, adjusting the number of neurons in each layer, experimenting with different activation functions, and incorporating regularization techniques (such as Dropout and BatchNormalization).



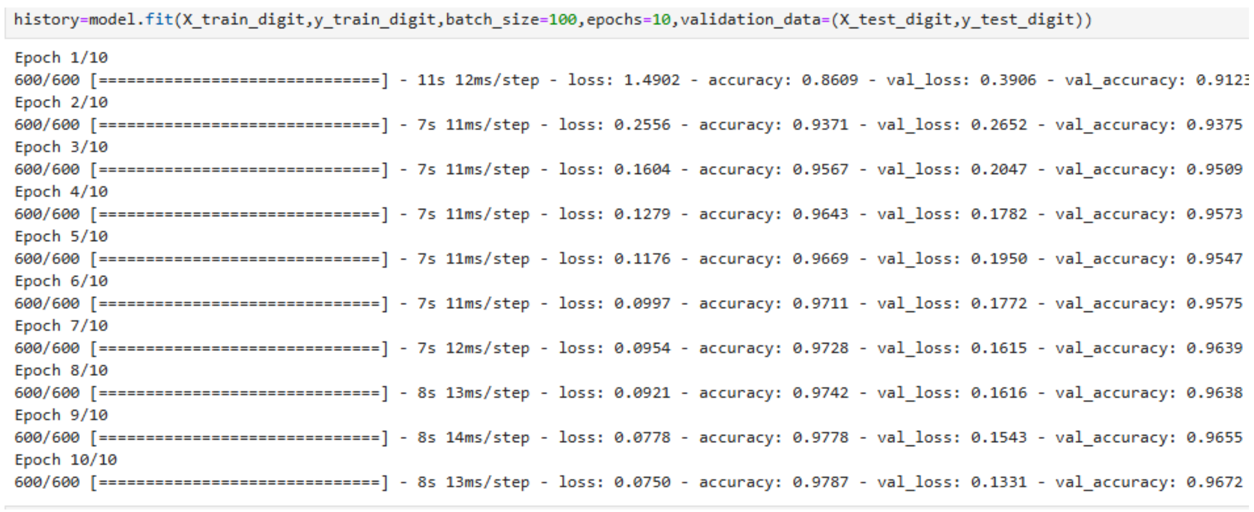
```
model.summary()

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               200960

 dense_1 (Dense)             (None, 64)                16448

 dense_2 (Dense)             (None, 64)                4160

 dense_3 (Dense)             (None, 10)                650

=================================================================
Total params: 222,218
Trainable params: 222,218
Non-trainable params: 0
```



```
#Assignement Answer on how to calculate the parameter
#param_number=output_channel_number*(input channel number +1)
#(784+1)x256=200960
#(256+1)x64=16448
#(64+1)X64=4160
#(64+1)x10=650
```

```
#compliling the model
model.compile(loss="categorical_crossentropy",
              optimizer="adam",
              metrics=['accuracy'])
```

```
history=model.fit(X_train_digit,y_train_digit,batch_size=100,epochs=10,validation_data=(X_test_digit,y_test_digit))
```



```
history=model.fit(X_train_digit,y_train_digit,batch_size=100,epochs=10,validation_data=(X_test_digit,y_test_digit))
```

```
Epoch 1/10
600/600 [==============================] - 11s 12ms/step - loss: 1.4902 - accuracy: 0.8609 - val_loss: 0.3906 - val_accuracy: 0.912
Epoch 2/10
600/600 [==============================] - 7s 11ms/step - loss: 0.2556 - accuracy: 0.9371 - val_loss: 0.2652 - val_accuracy: 0.9375
Epoch 3/10
600/600 [==============================] - 7s 11ms/step - loss: 0.1604 - accuracy: 0.9567 - val_loss: 0.2047 - val_accuracy: 0.9509
Epoch 4/10
600/600 [==============================] - 7s 11ms/step - loss: 0.1279 - accuracy: 0.9643 - val_loss: 0.1782 - val_accuracy: 0.9573
Epoch 5/10
600/600 [==============================] - 7s 11ms/step - loss: 0.1176 - accuracy: 0.9669 - val_loss: 0.1950 - val_accuracy: 0.9547
Epoch 6/10
600/600 [==============================] - 7s 11ms/step - loss: 0.0997 - accuracy: 0.9711 - val_loss: 0.1772 - val_accuracy: 0.9575
Epoch 7/10
600/600 [==============================] - 7s 12ms/step - loss: 0.0954 - accuracy: 0.9728 - val_loss: 0.1615 - val_accuracy: 0.9639
Epoch 8/10
600/600 [==============================] - 8s 13ms/step - loss: 0.0921 - accuracy: 0.9742 - val_loss: 0.1616 - val_accuracy: 0.9638
Epoch 9/10
600/600 [==============================] - 8s 14ms/step - loss: 0.0778 - accuracy: 0.9778 - val_loss: 0.1543 - val_accuracy: 0.9655
Epoch 10/10
600/600 [==============================] - 8s 13ms/step - loss: 0.0750 - accuracy: 0.9787 - val_loss: 0.1331 - val_accuracy: 0.9672
```

Keras ann compile …check in google in keras API doc….how we choose based on the scenario

This code prepares your neural network for training. Here's the breakdown:

This method is used to configure the learning process of a Keras model. It defines how the model will be trained, including the loss function, optimizer, and evaluation metrics.

loss="categorical_crossentropy": This function helps the model understand how far off its predictions are from the actual answers. It's like a guide, telling the model to adjust its guesses to get closer to the truth. This specifies the **loss function** used to measure the error between the model's predictions and the true labels
Categorical Crossentropy: Since we are dealing with multi-class classification (predicting digits 0-9), categorical crossentropy is an appropriate choice. It measures the difference between the predicted probability distribution and the true one-hot encoded labels.

- The model will try to minimize this loss during training.
- **optimizer="adam":** This is like the model's coach. It helps the model learn from its mistakes and improve its predictions over time. **Adam (Adaptive Moment Estimation):** It is a popular optimization algorithm known for its efficiency and often achieves good results in various deep learning tasks. Adam adapts the learning rate for each parameter during training, which can accelerate convergence.
- **metrics=['accuracy']:** This is how we measure how well the model is doing. It calculates the percentage of correct predictions, giving us a score of how accurate the model is. **Accuracy:** It measures the proportion of correctly classified samples. In this case, it calculates the percentage of images for which the model predicts the correct digit.

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

This code snippet configures the training process for the neural network model defined earlier. Let's break down each part:

1. model.compile(...):

- This method is used to configure the learning process of a Keras model. It defines how the model will be trained, including the loss function, optimizer, and evaluation metrics.

2. loss="categorical_crossentropy":

- This specifies the **loss function** used to measure the error between the model's predictions and the true labels.
- **Categorical Crossentropy:** Since we are dealing with multi-class classification (predicting digits 0-9), categorical crossentropy is an appropriate choice. It measures the difference between the predicted probability distribution and the true one-hot encoded labels.
- The model will try to minimize this loss during training.

3. optimizer="adam":

- This specifies the **optimizer** used to update the model's weights during training.
- **Adam (Adaptive Moment Estimation):** It is a popular optimization algorithm known for its efficiency and often achieves good results in various deep learning tasks. Adam adapts the learning rate for each parameter during training, which can accelerate convergence.

4. metrics=['accuracy']:

- This specifies the **evaluation metric** used to monitor the model's performance during training and testing.
- **Accuracy:** It measures the proportion of correctly classified samples. In this case, it calculates the percentage of images for which the model predicts the correct digit.

In summary:

This code configures the model to use categorical crossentropy as the loss function, the Adam optimizer to update the model's weights, and accuracy as the metric to evaluate its performance. This setup is well-suited for multi-class classification tasks like digit recognition in the MNIST dataset.

DL-Brain creation,compiling,fit method ..

Fit method ..model created

This code trains the neural network model you defined earlier on the MNIST dataset. Let's break down the components:

1. history = model.fit(...):

- This line calls the fit method on the compiled model (model). The fit method is responsible for training the model on the provided data.
- The history variable will store the training history, which includes information about the training process like loss and accuracy values over epochs.

2. X_train_digit, y_train_digit:

- These are the training data:
    - X_train_digit: The input data, containing the flattened and reshaped images of handwritten digits from the training set.
    - y_train_digit: The corresponding labels (one-hot encoded) for the training images, indicating the correct digit for each image.

3. batch_size=100:

- This specifies the **batch size**. During training, the model will process the data in batches instead of feeding the entire dataset at once. This is a common practice to improve memory efficiency and potentially speed up training.
- Here, a batch size of 100 means the model will process 100 images and their labels at a time before updating its weights.

4. epochs=10:

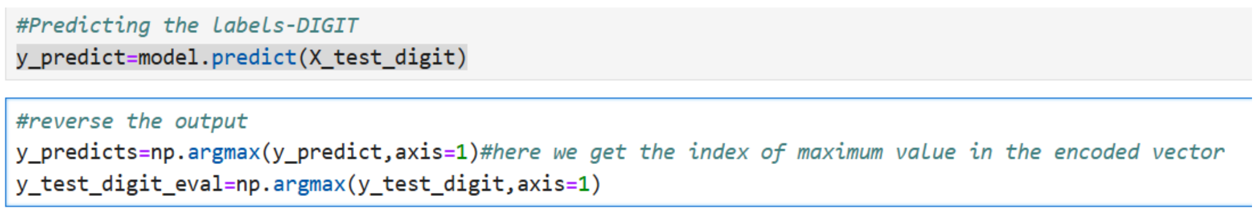
- This specifies the number of **epochs**. An epoch represents one complete pass through the entire training dataset.
- Here, the model will be trained for 10 epochs. This means it will go through the entire training data 10 times, updating its weights and learning patterns from the data with each pass.

5. validation_data=(X_test_digit, y_test_digit):

- This provides a **validation dataset** for monitoring the model's performance during training.
- X_test_digit and y_test_digit are the testing data, separate from the training data. The model will not be trained on this data, but its performance will be evaluated on the validation set after each epoch.
- This helps to prevent overfitting, where the model memorizes the training data but performs poorly on unseen data.

In summary:

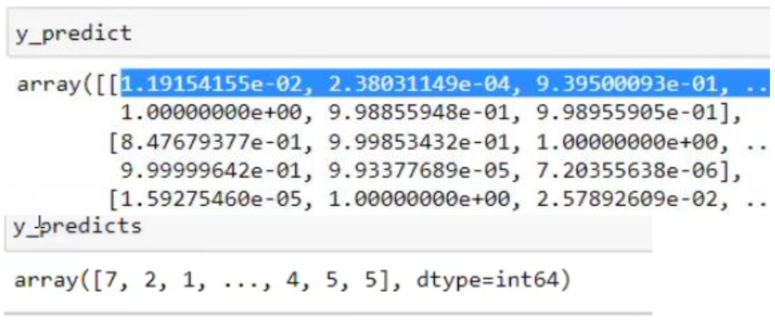
This code trains the neural network model on the MNIST training data for 10 epochs(ffn bp), using a batch size of 100 and monitoring its performance on the validation set. The history variable will contain valuable information about the training process, allowing you to analyze the model's learning progress and potentially adjust hyperparameters for further improvement.



```
#Predicting the Labels-DIGIT
y_predict=model.predict(X_test_digit)
```

```
#reverse the output
y_predicts=np.argmax(y_predict,axis=1)#here we get the index of maximum value in the encoded vector
y_test_digit_eval=np.argmax(y_test_digit,axis=1)
```

Same flow 2nd change reverse



```
y_predict
```

```
array([[1.19154155e-02, 2.38031149e-04, 9.39500093e-01, ..
         1.00000000e+00, 9.98855948e-01, 9.98955905e-01],
        [8.47679377e-01, 9.99853432e-01, 1.00000000e+00, ..
         9.99999642e-01, 9.93377689e-05, 7.20355638e-06],
        [1.59275460e-05, 1.00000000e+00, 2.57892609e-02, ..
```

```
y_predicts
```

```
array([7, 2, 1, ..., 4, 5, 5], dtype=int64)
```

y_predict = model.predict(X_test_digit)

- This line uses the trained model to make predictions on the X_test_digit data.
- The model.predict() function takes the input data (test images) and generates an output.
- For each input image, the model produces a vector of probabilities, where each element in the vector represents the probability of the image belonging to a specific class (digit 0-9).

2. y_predicts = np.argmax(y_predict, axis=1)

- This line converts the predicted probabilities into the predicted class labels.
- np.argmax() finds the index of the maximum value along a given axis.
- In this case, axis=1 specifies that the maximum value should be found across the columns of the y_predict array.
- This effectively selects the class with the highest probability for each image, giving you the model's predicted digit for each test image.

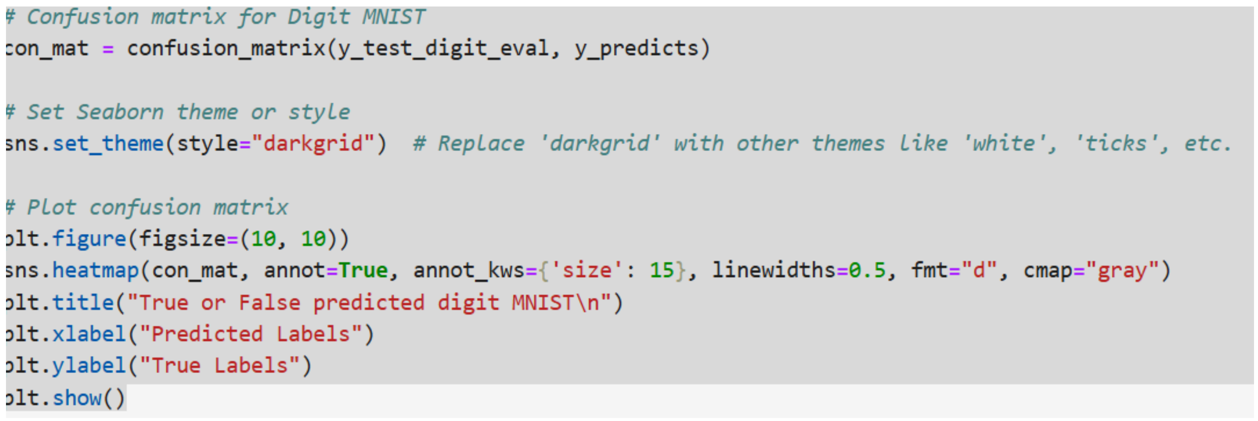
3. y_test_digit_eval = np.argmax(y_test_digit, axis=1)

- This line converts the one-hot encoded true labels back to their original integer representation.
- Similar to the previous line, np.argmax() finds the index of the maximum value in each row of the y_test_digit array, which corresponds to the true digit for each image.

In essence:

- The code first uses the trained model to generate probability distributions for each test image.
- Then, it extracts the predicted class label (the digit with the highest probability) for each image.
- Finally, it converts the one-hot encoded true labels back to their original integer format for easy comparison with the predicted labels.

This sequence of steps allows you to obtain the model's predictions for the test data and prepare them for evaluation.



```
# Confusion matrix for Digit MNIST
con_mat = confusion_matrix(y_test_digit_eval, y_predicts)

# Set Seaborn theme or style
sns.set_theme(style="darkgrid")  # Replace 'darkgrid' with other themes like 'white', 'ticks', etc.

# Plot confusion matrix
plt.figure(figsize=(10, 10))
sns.heatmap(con_mat, annot=True, annot_kws={'size': 15}, linewidths=0.5, fmt="d", cmap="gray")
plt.title("True or False predicted digit MNIST\n")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```

The provided code snippet deals with visualizing the performance of a machine learning model on a digit classification task using the MNIST dataset. Here's a breakdown of the code:

1. Confusion Matrix Calculation (already assumed):

- This part (not explicitly shown in the code) calculates a confusion matrix using the confusion_matrix function from the scikit-learn library (assumed to be imported).
  - It takes two arguments:
    - y_test_digit_eval: This array contains the true labels for the test data (integers representing the actual digits).
    - y_predicts: This array contains the predicted labels generated by the model for the test data (also integers representing the predicted digits).
- The confusion matrix is a table that summarizes the model's classification performance. It shows how many times the model correctly predicted each digit class and how many times it made mistakes by predicting a different class.

2. Seaborn Customization (Optional):

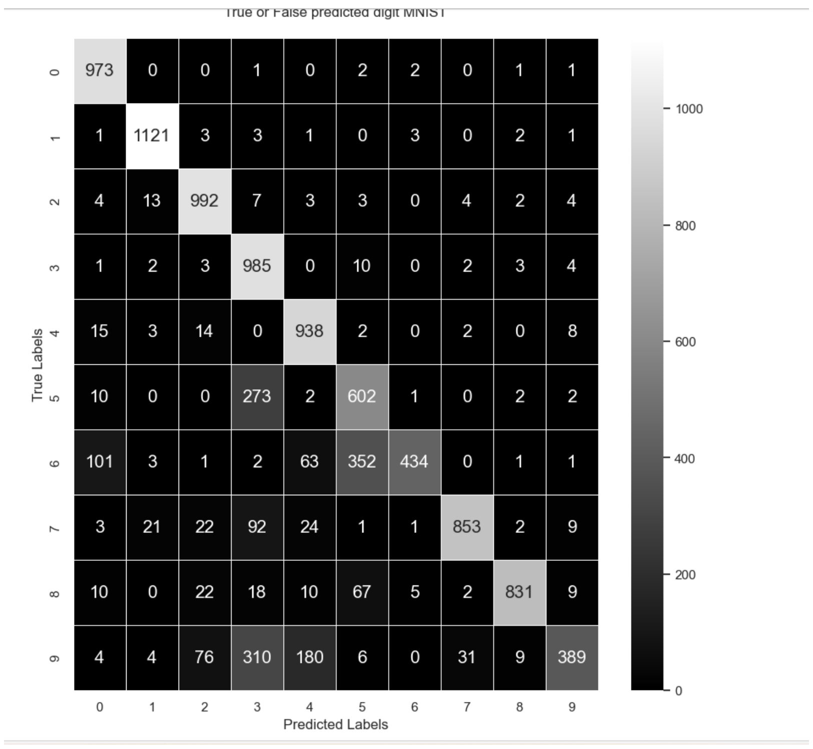
- sns.set_theme(style="darkgrid"): This line sets the visual theme for the confusion matrix plot using the Seaborn library (assumed to be imported). You can experiment with different styles (e.g., 'white', 'ticks') to customize the appearance of the plot.

3. Plotting the Confusion Matrix:

- plt.figure(figsize=(10, 10)): This line creates a new figure window for the plot, setting its size to 10x10 inches.
- sns.heatmap(con_mat, annot=True, annot_kws={'size': 15}, linewidths=0.5, fmt="d", cmap="gray"): This line generates the heatmap visualization of the confusion matrix using Seaborn:
  - con_mat: This is the confusion matrix data calculated earlier.
  - annot=True: This argument enables displaying the actual values (counts) in each cell of the heatmap.
  - annot_kws={'size': 15}: This sets the font size for the annotations (numbers) within the heatmap cells.
  - linewidths=0.5: This adjusts the thickness of the grid lines separating the cells in the heatmap.

  - fmt="d": This specifies the format for the annotations, set to integers in this case.
  - cmap="gray": This defines the colormap used for the heatmap, using grayscale here.
- plt.title("True or False predicted digit MNIST\n"): This sets the title for the plot, indicating that it visualizes the model's performance on the MNIST digit classification task.
- plt.xlabel("Predicted Labels"): This labels the x-axis of the plot, representing the digits the model predicted.
- plt.ylabel("True Labels"): This labels the y-axis of the plot, representing the actual digits in the test data.
- plt.show(): This line displays the generated confusion matrix plot.

In essence, this code helps you visualize how well your model performed on the MNIST dataset by creating a confusion matrix and customizing its presentation using Seaborn. The confusion matrix provides insights into the model's strengths and weaknesses in classifying handwritten digits.



	0	1	2	3	4	5	6	7	8	9
0	973	0	0	1	0	2	2	0	1	1
1	1	1121	3	3	1	0	3	0	2	1
2	4	13	992	7	3	3	0	4	2	4
3	1	2	3	985	0	10	0	2	3	4
4	15	3	14	0	938	2	0	2	0	8
5	10	0	0	273	2	602	1	0	2	2
6	101	3	1	2	63	352	434	0	1	1
7	3	21	22	92	24	1	1	853	2	9
8	10	0	22	18	10	67	5	2	831	9
9	4	4	76	310	180	6	0	31	9	389

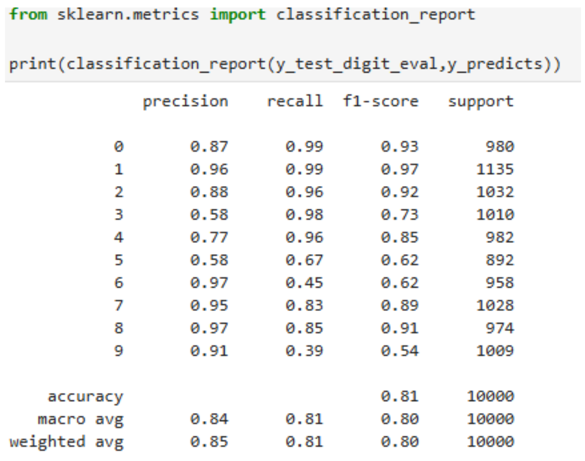
Metrics Explained:

1. **Precision**: The proportion of positive cases correctly identified by the model.
2. **Recall** (or Sensitivity): The proportion of actual positive cases that are correctly identified.
3. **F1-score**: A balanced metric that considers both precision and recall, giving equal weight to both.
4. **Support**: The number of instances for each class in the dataset.

Summary:

- Precision and recall values are provided for each class.
- The overall accuracy is 81%, meaning the model correctly predicted 81% of the samples.

- Macro average and weighted average show the overall performance across all classes, giving insights into the model's balance across classes.



```
from sklearn.metrics import classification_report

print(classification_report(y_test_digit_eval,y_predicts))
```

```
              precision    recall  f1-score   support

           0       0.87      0.99      0.93       980
           1       0.96      0.99      0.97      1135
           2       0.88      0.96      0.92      1032
           3       0.58      0.98      0.73      1010
           4       0.77      0.96      0.85       982
           5       0.58      0.67      0.62       892
           6       0.97      0.45      0.62       958
           7       0.95      0.83      0.89      1028
           8       0.97      0.85      0.91       974
           9       0.91      0.39      0.54      1009

    accuracy                           0.81     10000
   macro avg       0.84      0.81      0.80     10000
weighted avg       0.85      0.81      0.80     10000
```

The provided classification report includes key evaluation metrics for a multi-class classification task:

Metrics Explained:

- **Precision**: The proportion of true positive predictions out of all predicted positives. Higher is better.
- **Recall**: The proportion of true positive predictions out of all actual positives. Higher is better.
- **F1-Score**: The harmonic mean of precision and recall. It balances both metrics and is useful for evaluating the model's overall performance.
- **Support**: The number of actual occurrences in the dataset for each class.

Breakdown:

1. **0**:
   - Precision: 0.87
   - Recall: 0.99
   - F1-Score: 0.93
   - Support: 980
2. **1**:
   - Precision: 0.96
   - Recall: 0.99
   - F1-Score: 0.97
   - Support: 1135
3. **2**:
   - Precision: 0.88
   - Recall: 0.96
   - F1-Score: 0.92
   - Support: 1032
4. **3**:
   - Precision: 0.58
   - Recall: 0.98
   - F1-Score: 0.73
   - Support: 1010
5. **4**:
   - Precision: 0.77
   - Recall: 0.96
   - F1-Score: 0.85

    - Support: 982
6. **5**:
    - Precision: 0.58
    - Recall: 0.67
    - F1-Score: 0.62
    - Support: 892
7. **6**:
    - Precision: 0.97
    - Recall: 0.45
    - F1-Score: 0.62
    - Support: 958
8. **7**:
    - Precision: 0.95
    - Recall: 0.83
    - F1-Score: 0.89
    - Support: 1028
9. **8**:
    - Precision: 0.97
    - Recall: 0.85
    - F1-Score: 0.91
    - Support: 974
10. **9**:
    - Precision: 0.91
    - Recall: 0.39
    - F1-Score: 0.54
    - Support: 1009

Overall Metrics:

- **Accuracy**: 0.81
- **Macro Avg**: Average performance across all classes.
- **Weighted Avg**: Average performance considering the support for each class.