Handwritten Digit Recognition Using Keras and TensorFlow

The following is a comprehensive Python program that implements **handwritten digit recognition** using **Keras** (**TensorFlow backend**).

The code uses the **MNIST dataset**, which contains 70,000 grayscale images of handwritten digits (0-9), and walks through the entire process: from data loading to visualisation, model building, training, evaluation, and prediction.

1. Importing required libraries

- TensorFlow and Keras are deep learning frameworks and provide a high-level API for building neural networks.
- NumPy helps in numerical operations and in handling multi-dimensional arrays.
- Matplotlib is used to visualise the dataset and model predictions.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models

import numpy as np
import matplotlib.pyplot as plt

# Checking TensorFlow version
print("TensorFlow version: ", tf.__version__)
```

```
TensorFlow version:  2.19.0
```

2. Loading the dataset

The MNIST dataset, which is available directly in Keras, contains 60,000 training images and 10,000 testing images. Each image is 28×28 pixels, representing a grayscale digit.

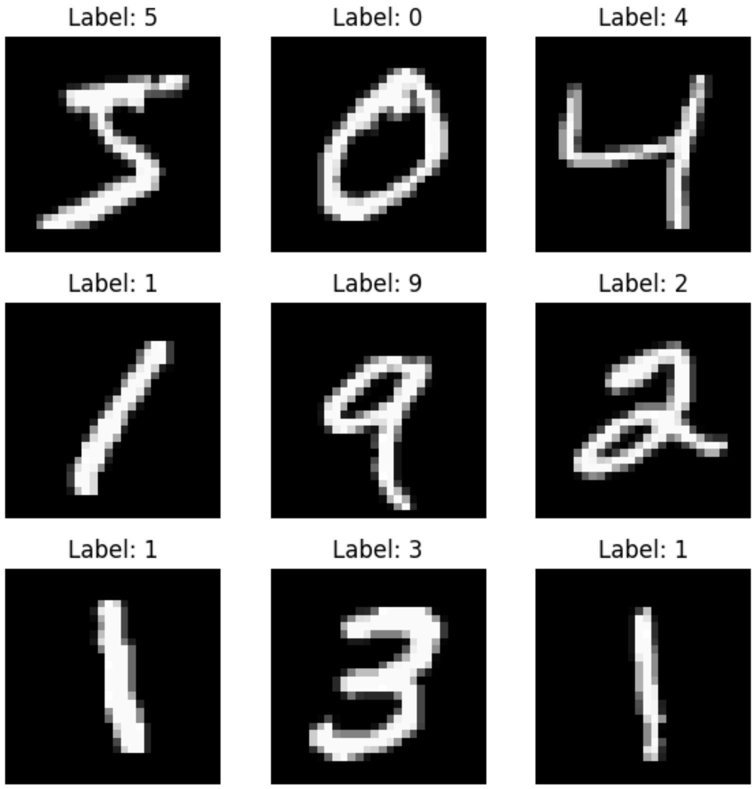
```
# Splitting into training and testing sets
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()

print("Training data shape: ", X_train.shape)
print("Test data shape: ", X_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Training data shape:  (60000, 28, 28)
Test data shape:  (10000, 28, 28)
```

3. Visualising sample images

Visualisation helps ensure the data looks correct, since the digits are handwritten and vary in shape, size, and thickness.

```
# Displaying the first 9 images from the training set
plt.figure(figsize=(6,6))
for i in range(9):
  plt.subplot(3, 3, i+1)
  plt.imshow(X_train[i], cmap='gray')
  plt.title(f"Label: {y_train[i]}")
  plt.axis('off')
plt.tight_layout()
plt.show()
```



4. Data preprocessing

- **Normalisation:** We convert the image pixel values (0-255) to float values (0-1), as this improves convergence during training.
- **Reshaping:** This makes the data compatible with Convolutional Neural Networks (CNNs). We reshape the data to include the channel dimension for CNN input, since it expects data as `(samples, height, width, channels)`.
- **One-hot encoding:** We convert the class vectors (0-9) to one-hot encoded format (i.e., binary matrices).

```
# Normalisation
X_train = X_train.astype('float32')/255.0
X_test = X_test.astype('float32')/255.0

# Reshaping
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))

# One-hot encoding
from tensorflow.keras.utils import to_categorical
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

print("Training data shape after preprocessing: ", X_train.shape)
print("Training label shape: ", y_train.shape)
```

```
Training data shape after preprocessing:  (60000, 28, 28, 1)
Training label shape:  (60000, 10)
```

5. Building the CNN model

- **Conv2D** layers extract spatial features from images.
- **MaxPooling2D** reduces the spatial features (downsampling).
- **Flatten** transforms the 2D output into a vector for dense layers.
- The final **softmax layer** outputs probability distributions across 10 classes.

```
model = models.Sequential([
    # First convolutional layer
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2,2)),

    # Second convolutional layer
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),

    # Flatten layer for converting 2D feature maps to 1D vector
    layers.Flatten(),

    # Fully connected (dense) layer
    layers.Dense(128, activation='relu'),
```

```
    # Output layer with 10 units (for digits 0-9)
    layers.Dense(10, activation='softmax')
])

model.summary()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input_shape`/`inp
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dense (Dense)	(None, 128)	204,928
dense_1 (Dense)	(None, 10)	1,290

```
 Total params: 225,034 (879.04 KB)
 Trainable params: 225,034 (879.04 KB)
 Non-trainable params: 0 (0.00 B)
```

6. Compiling the model

The model needs to be compiled before training.

- **Adam optimiser** adjusts the learning rate automatically.
- **Categorical cross-entropy** is ideal for multi-class classification problems.
- Tracking **accuracy** provides a clear performance metric.

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

7. Training the model

```
history = model.fit(
    X_train, y_train,
    epochs = 5,
    batch_size = 128,
    validation_split = 0.1,  # Using 10% of the training data for validation
    verbose = 1
)
```

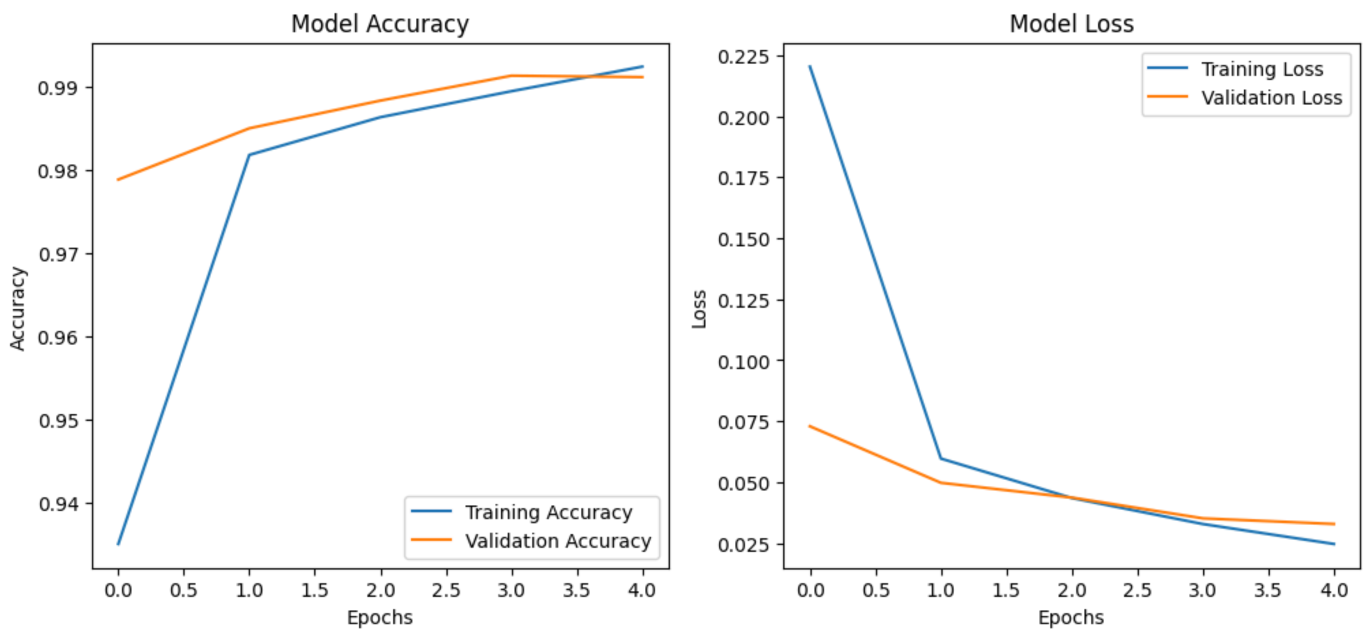
```
Epoch 1/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 40s 91ms/step - accuracy: 0.8500 - loss: 0.5082 - val_accuracy: 0.9788 - val_loss: 0.0729
Epoch 2/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 39s 93ms/step - accuracy: 0.9797 - loss: 0.0661 - val_accuracy: 0.9850 - val_loss: 0.0497
Epoch 3/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 38s 89ms/step - accuracy: 0.9858 - loss: 0.0457 - val_accuracy: 0.9883 - val_loss: 0.0436
Epoch 4/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 37s 89ms/step - accuracy: 0.9903 - loss: 0.0307 - val_accuracy: 0.9913 - val_loss: 0.0351
Epoch 5/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 37s 88ms/step - accuracy: 0.9931 - loss: 0.0224 - val_accuracy: 0.9912 - val_loss: 0.0328
```

8. Visualising training performance

The following plots show how the model's accuracy and loss evolve over epochs. Stable validation accuracy indicates good generalisation.

```
plt.figure(figsize=(12,5))

# Plotting training and validation accuracy
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title("Model Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()

# Plotting training and validation loss
plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
```

```
plt.title("Model Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()

plt.show()
```



9. Evaluating the model on test data

Evaluation on unseen test data ensures that the model is not overfitting.

```
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
print(f"Testing accuracy: {test_acc:.4f}")
print(f"Testing loss: {test_loss:.4f}")
```

```
Testing accuracy: 0.9902
Testing loss: 0.0305
```

10. Making predictions

```
predictions = model.predict(X_test)

# Converting one-hot encoded predictions to class labels
predicted_labels = np.argmax(predictions, axis=1)
true_labels = np.argmax(y_test, axis=1)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step
```

11. Visualising predictions

We now display a few test images along with predicted and true labels.This helps verify visually whether the model performs correctly.

```
plt.figure(figsize=(6,5))
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow(X_test[i].reshape(28,28), cmap='gray')
    plt.title(f"Predicted: {predicted_labels[i]} | True: {true_labels[i]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

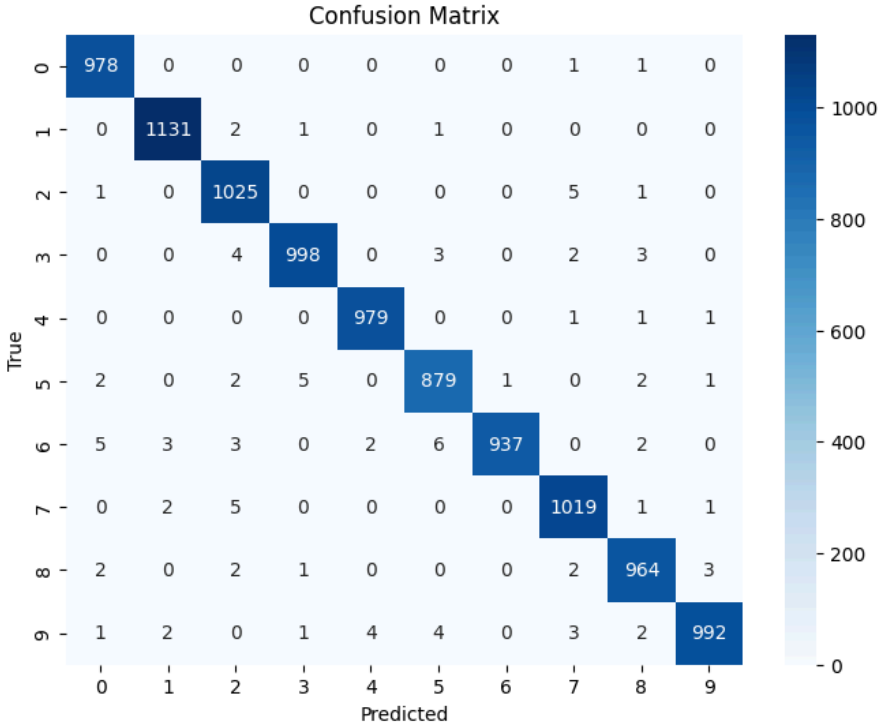
Show hidden output

12. Confusion matrix

The confusion matrix shows how well the model distinguishes between digits. Moreover, the classification report provides precision, recall, and F1-score for each class.

```
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

cm = confusion_matrix(true_labels, predicted_labels)

plt.figure(figsize=(8,6))
```

```python
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()

# For detailed classification metrics
print(classification_report(true_labels, predicted_labels))
```




```
              precision    recall  f1-score   support

           0       0.99      1.00      0.99       980
           1       0.99      1.00      1.00      1135
           2       0.98      0.99      0.99      1032
           3       0.99      0.99      0.99      1010
           4       0.99      1.00      1.00       982
           5       0.98      0.99      0.98       892
           6       1.00      0.98      0.99       958
           7       0.99      0.99      0.99      1028
           8       0.99      0.99      0.99       974
           9       0.99      0.98      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```