Explanation

December 12, 2023

1 MNIST Digit Classification Application with Convolutional Neural Network

In the ever-evolving landscape of artificial intelligence and machine learning, the ability to effectively recognize and classify patterns in data is a fundamental aspect. One powerful approach for image classification tasks is the utilization of Convolutional Neural Networks (CNNs). This project embarks on a journey to build and train a sophisticated image classification model using the renowned deep learning library, **Keras**, complemented by the essential tools **Matplotlib** and **Numpy**.

1.1 Objective

The primary goal of this project is to construct a robust Convolutional Neural Network (CNN) specifically designed for the classification of MNIST handwritten numerical digits. Leveraging Keras, a high-level neural networks API, provides an intuitive and efficient platform for developing and training deep learning models tailored for this task. The inclusion of Matplotlib facilitates visualizations, enabling a comprehensive understanding of the model's performance on the MNIST dataset, while Numpy supports efficient numerical operations essential for data manipulation in the context of digit classification.

1.2 Key Components

1. **Keras:** As the core framework, Keras streamlines the implementation of neural networks, abstracting complexities and providing a user-friendly interface. The project delves into the sequential construction of a CNN, incorporating convolutional and pooling layers, flattening operations, and fully connected layers.
2. **Matplotlib:** Visualizing the model's training progress and performance is crucial for insights. Matplotlib comes into play for generating informative plots and graphs, allowing stakeholders to interpret accuracy trends, loss convergence, and other crucial metrics.
3. **Numpy:** A fundamental library for numerical operations, Numpy facilitates efficient data manipulation and array processing. It is instrumental in reshaping input data, handling mathematical computations, and ensuring seamless integration with Keras.

1.3 Project Workflow

1. **Data Loading and Exploration**
2. **Data Preprocessing**
3. **Model Architecture**

4. **Compilation**
5. **Training**
6. **Application**

1.4 Code Implementation

Now that we've outlined the objectives, key components, and workflow of our machine learning project, let's dive into the practical aspect. The following code implementation demonstrates the step-by-step process of constructing and training a Convolutional Neural Network (CNN) using the powerful tools at our disposal: **Keras** for model development, **Matplotlib** for insightful visualizations, and **Numpy** for efficient numerical operations.

In the subsequent sections, we'll walk through:

1.4.1 1.Data Loading and Exploration

Understanding the structure of our data is crucial. We'll explore the MNIST dataset to gain insights into its dimensions and characteristics.

Importing Essential Libraries

In this section, we import the necessary libraries essential for the implementation:

- **numpy:** A library for numerical operations in Python.
- **matplotlib.pyplot:** A library for creating visualizations in Python.
- **to_categorical:** A function from Keras utilities to convert integer labels to one-hot encoding.
- **Sequential:** A Keras class for creating a linear stack of layers in a neural network.
- **Conv2D, MaxPooling2D, Flatten, Dense, Dropout:** Different layers used in building a Convolutional Neural Network (CNN).
- **EarlyStopping, ModelCheckpoint:** Callbacks from Keras for early stopping during training and model checkpointing.

[14]:
```
import numpy as np
import matplotlib.pyplot as plt
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D, MaxPool2D, Flatten, Dense, Dropout
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.datasets import mnist
from tensorflow.compat.v1.train import AdamOptimizer
```

1.4.2 2.Data Preprocessing:

Before feeding the data into the model, we'll perform necessary preprocessing steps, including converting labels to one-hot encoding and reshaping the input data.

[15]:
```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

This single line of code above fetches the MNIST dataset, splitting it into training and testing sets. It assigns the images of the training set to `X_train` and the corresponding labels to `y_train`.

Similarly, it assigns the images of the testing set to `X_test` and the corresponding labels to `y_test`.

To gain insights into the structure of the dataset, we print the shapes of the training and testing data arrays using the following code:

[16]:
```
print(f"Training data shape: {X_train.shape}, Labels shape: {y_train.shape}")
print(f"Testing data shape: {X_test.shape}, Labels shape: {y_test.shape}")
```

```
Training data shape: (60000, 28, 28), Labels shape: (60000,)
Testing data shape: (10000, 28, 28), Labels shape: (10000,)
```

[17]:
```
y_train_one_hot = to_categorical(y_train, num_classes=10)
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], X_train.
 ↪shape[2], 1))
```

Here, the integer labels in y_train are converted to one-hot encoding using the to_categorical function. Additionally, the input data (X_train) is reshaped to include a channel dimension, which is necessary for CNNs.

1.4.3 3.Model Architecture

The heart of our project lies in designing a robust CNN. We'll sequentially add convolutional layers, pooling, flattening operations, and dense layers to form a powerful architecture.

[18]:
```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=(28, 28, 1),␣
 ↪activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPool2D((2, 2)))
model.add(Flatten())
model.add(Dropout(0.25))
model.add(Dense(10, activation='softmax'))
```

The architecture includes the following layers:

1. **Conv2D:** Convolutional layer with 32 filters, each filter scanning the input image with a kernel size of (3, 3). The ReLU activation function is applied, introducing non-linearity to the model. This layer is responsible for capturing local patterns in the input images.
2. **MaxPooling2D:** Max-pooling layer with a pool size of (2, 2). It reduces the spatial dimensions of the representation and retains the most important information, providing translation invariance. This helps in reducing computation and controlling overfitting.
3. **Conv2D:** Another convolutional layer with 64 filters, similar to the first Conv2D layer. This layer further refines the learned features from the previous layer, extracting more complex and abstract features.
4. **MaxPooling2D:** Another max-pooling layer to downsample the spatial dimensions further. It maintains the most relevant information while reducing computational complexity.
5. **Flatten:** Flattening layer that converts the 2D matrix data to a vector. It prepares the data for input into the fully connected layers.

6. **Dropout:** Dropout layer with a rate of 0.25. It randomly drops a fraction of input units during training, acting as a regularization technique to prevent overfitting.

7. **Dense:** Fully connected layer with 10 units, corresponding to the 10 classes of digits (0-9). The softmax activation function is applied, providing probabilities for each class. This layer produces the final output for digit classification.

This architecture is specifically tailored for the MNIST digit classification task. Each layer plays a crucial role in capturing hierarchical features from the input images, reducing spatial dimensions, and making predictions. The Convolutional Neural Network's ability to automatically learn and extract relevant features makes it well-suited for image classification tasks. Network.

1.4.4 4.Compilation

Configuring the model involves specifying an optimizer, a loss function, and evaluation metrics. We'll compile our CNN to prepare it for training.

[19]:
```
model.compile(optimizer=AdamOptimizer(), loss='categorical_crossentropy',␣
↪metrics=['accuracy'])
```

In this compilation step:

- **Optimizer ('adam'):** The Adam optimizer is employed, representing a widely used and efficient optimization algorithm for training neural networks. It adapts learning rates for each parameter individually, enhancing the efficiency of the training process. Adam is known for its robustness and effectiveness in a variety of scenarios.

- **Loss Function ('categorical_crossentropy'):** This is the chosen loss function for multi-class classification tasks, such as MNIST digit classification. Categorical crossentropy measures the difference between the predicted probability distribution and the true distribution of the labels. It quantifies how well the model is aligning its predictions with the actual class labels.

- **Metric to Monitor ('accuracy'):** The accuracy metric is selected to monitor the model's performance during training. Accuracy provides a straightforward measure of how well the model is classifying the input images, representing the ratio of correctly predicted samples to the total number of samples. Monitoring accuracy helps ensure that the model is making meaningful progress in learning and generalizing patterns from the training data.

[26]:
```
es = EarlyStopping(monitor='val_accuracy', min_delta=0.01, patience=4,␣
↪verbose=1, restore_best_weights=True)
mc = ModelCheckpoint("best.keras", monitor="val_accuracy", verbose=1,␣
↪save_best_only=True)
```

Callbacks are defined to enhance training:

EarlyStopping: - **Purpose:** Monitors the validation accuracy during training. - **Functionality:** Stops training if there's no improvement after a certain number of epochs (patience). - **Action on Stop:** Restores the model weights to the best weights recorded during training.

ModelCheckpoint: - **Purpose:** Monitors the validation accuracy during training. - **Functionality:** Saves the best model based on validation accuracy. ccuracy. ccuracy. cy. ng.

1.4.5 5.Training

The hands-on training process begins, with a focus on performance metrics. We'll monitor key indicators and use early stopping and model checkpointing for efficiency.

[22]:
```
history = model.fit(X_train, y_train_one_hot, epochs=50, validation_split=0.3)
```

```
Epoch 1/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0922 -
accuracy: 0.9720 - val_loss: 0.0638 - val_accuracy: 0.9810
Epoch 2/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0758 -
accuracy: 0.9776 - val_loss: 0.0595 - val_accuracy: 0.9819
Epoch 3/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0692 -
accuracy: 0.9791 - val_loss: 0.0753 - val_accuracy: 0.9809
Epoch 4/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0639 -
accuracy: 0.9798 - val_loss: 0.0859 - val_accuracy: 0.9772
Epoch 5/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0594 -
accuracy: 0.9824 - val_loss: 0.0574 - val_accuracy: 0.9839
Epoch 6/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0561 -
accuracy: 0.9831 - val_loss: 0.0607 - val_accuracy: 0.9850
Epoch 7/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0498 -
accuracy: 0.9849 - val_loss: 0.0589 - val_accuracy: 0.9841
Epoch 8/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0453 -
accuracy: 0.9868 - val_loss: 0.0703 - val_accuracy: 0.9831
Epoch 9/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0471 -
accuracy: 0.9860 - val_loss: 0.0656 - val_accuracy: 0.9844
Epoch 10/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0434 -
accuracy: 0.9874 - val_loss: 0.0597 - val_accuracy: 0.9862
Epoch 11/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0411 -
accuracy: 0.9875 - val_loss: 0.0676 - val_accuracy: 0.9843
Epoch 12/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0396 -
accuracy: 0.9891 - val_loss: 0.0714 - val_accuracy: 0.9853
Epoch 13/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0432 -
accuracy: 0.9885 - val_loss: 0.0639 - val_accuracy: 0.9867
Epoch 14/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0376 -
accuracy: 0.9899 - val_loss: 0.0890 - val_accuracy: 0.9839
```

```
Epoch 15/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0457 -
accuracy: 0.9885 - val_loss: 0.0828 - val_accuracy: 0.9867
Epoch 16/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0387 -
accuracy: 0.9903 - val_loss: 0.0838 - val_accuracy: 0.9858
Epoch 17/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0405 -
accuracy: 0.9894 - val_loss: 0.0842 - val_accuracy: 0.9858
Epoch 18/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0385 -
accuracy: 0.9906 - val_loss: 0.0904 - val_accuracy: 0.9849
Epoch 19/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0425 -
accuracy: 0.9897 - val_loss: 0.0786 - val_accuracy: 0.9867
Epoch 20/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0354 -
accuracy: 0.9916 - val_loss: 0.0949 - val_accuracy: 0.9854
Epoch 21/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0423 -
accuracy: 0.9902 - val_loss: 0.1025 - val_accuracy: 0.9827
Epoch 22/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0349 -
accuracy: 0.9920 - val_loss: 0.1263 - val_accuracy: 0.9820
Epoch 23/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0416 -
accuracy: 0.9906 - val_loss: 0.0903 - val_accuracy: 0.9880
Epoch 24/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0361 -
accuracy: 0.9923 - val_loss: 0.1160 - val_accuracy: 0.9870
Epoch 25/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0437 -
accuracy: 0.9914 - val_loss: 0.1051 - val_accuracy: 0.9859
Epoch 26/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0331 -
accuracy: 0.9922 - val_loss: 0.1127 - val_accuracy: 0.9878
Epoch 27/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0396 -
accuracy: 0.9928 - val_loss: 0.1151 - val_accuracy: 0.9867
Epoch 28/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0405 -
accuracy: 0.9915 - val_loss: 0.1072 - val_accuracy: 0.9876
Epoch 29/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0396 -
accuracy: 0.9925 - val_loss: 0.1341 - val_accuracy: 0.9863
Epoch 30/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0399 -
accuracy: 0.9925 - val_loss: 0.1115 - val_accuracy: 0.9864
```

```
Epoch 31/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0351 -
accuracy: 0.9937 - val_loss: 0.1096 - val_accuracy: 0.9850
Epoch 32/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0382 -
accuracy: 0.9927 - val_loss: 0.1234 - val_accuracy: 0.9861
Epoch 33/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0378 -
accuracy: 0.9927 - val_loss: 0.1197 - val_accuracy: 0.9882
Epoch 34/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0450 -
accuracy: 0.9921 - val_loss: 0.1911 - val_accuracy: 0.9804
Epoch 35/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0402 -
accuracy: 0.9931 - val_loss: 0.1347 - val_accuracy: 0.9860
Epoch 36/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0374 -
accuracy: 0.9936 - val_loss: 0.1292 - val_accuracy: 0.9883
Epoch 37/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0421 -
accuracy: 0.9934 - val_loss: 0.1409 - val_accuracy: 0.9870
Epoch 38/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0439 -
accuracy: 0.9927 - val_loss: 0.1646 - val_accuracy: 0.9854
Epoch 39/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0353 -
accuracy: 0.9939 - val_loss: 0.1517 - val_accuracy: 0.9879
Epoch 40/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0359 -
accuracy: 0.9942 - val_loss: 0.1385 - val_accuracy: 0.9884
Epoch 41/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0399 -
accuracy: 0.9939 - val_loss: 0.1428 - val_accuracy: 0.9874
Epoch 42/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0428 -
accuracy: 0.9936 - val_loss: 0.1921 - val_accuracy: 0.9856
Epoch 43/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0436 -
accuracy: 0.9931 - val_loss: 0.1767 - val_accuracy: 0.9867
Epoch 44/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0365 -
accuracy: 0.9950 - val_loss: 0.1561 - val_accuracy: 0.9867
Epoch 45/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0365 -
accuracy: 0.9948 - val_loss: 0.2039 - val_accuracy: 0.9871
Epoch 46/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0402 -
accuracy: 0.9941 - val_loss: 0.2019 - val_accuracy: 0.9864
```

```
Epoch 47/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0354 -
accuracy: 0.9946 - val_loss: 0.1882 - val_accuracy: 0.9878
Epoch 48/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0412 -
accuracy: 0.9945 - val_loss: 0.2095 - val_accuracy: 0.9871
Epoch 49/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0440 -
accuracy: 0.9945 - val_loss: 0.2167 - val_accuracy: 0.9872
Epoch 50/50
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0435 -
accuracy: 0.9943 - val_loss: 0.1978 - val_accuracy: 0.9873
```

The `fit` function executes the training loop, during which the model's parameters are optimized based on the specified optimizer, and the defined loss function is minimized. Throughout each epoch, the training and validation performance metrics are monitored.

The training history, encapsulating valuable information about the model's learning progress, is stored in the `history` variable. This history includes metrics such as training loss, training accuracy, validation loss, and validation accuracy at each epoch. Storing this information allows for later analysis and visualization, enabling a comprehensive understanding of how well the model is learning from the training data and generalizing to unseen validation data.

To optimize the training process, two callbacks, EarlyStopping (`es`) and ModelCheckpoint (`mc`), are incorporated.

- **EarlyStopping (`es`):** Monitors the validation accuracy during training. If the improvement in accuracy is less than the specified `min_delta` for a certain number of `patience` epochs, training is halted. The model's weights are then restored to the best-performing epoch.
- **ModelCheckpoint (`mc`):** Monitors the validation accuracy. Whenever there is an improvement in accuracy, the model's weights are saved to a file named `bestmodel.h5`.

This approach ensures that the training stops early if the model's performance on the validation set does not show significant improvement. Additionally, it saves the weights of the best-performing model, allowing for later use without having to retrain the model from scratch.

[32]:
```
history = model.fit(X_train, y_train_one_hot, epochs=50, validation_split=0.3,␣
↪callbacks=[es, mc])
```

```
Epoch 1/50
1307/1313 [============================>.] - ETA: 0s - loss: 0.0414 - accuracy:
0.9948
Epoch 1: val_accuracy did not improve from 0.98811
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0420 -
accuracy: 0.9948 - val_loss: 0.2410 - val_accuracy: 0.9880
Epoch 2/50
1306/1313 [============================>.] - ETA: 0s - loss: 0.0398 - accuracy:
0.9950
Epoch 2: val_accuracy did not improve from 0.98811
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0396 -
```

```
accuracy: 0.9950 - val_loss: 0.2221 - val_accuracy: 0.9870
Epoch 3/50
1312/1313 [============================>.] - ETA: 0s - loss: 0.0433 - accuracy:
0.9949
Epoch 3: val_accuracy improved from 0.98811 to 0.98861, saving model to best.h5
WARNING:tensorflow:TensorFlow optimizers do not make it possible to access
optimizer attributes or optimizer state after instantiation. As a result, we
cannot save the optimizer as part of the model save file. You will have to
compile your model again after loading it. Prefer using a Keras optimizer
instead (see keras.io/optimizers).
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0433 -
accuracy: 0.9949 - val_loss: 0.2204 - val_accuracy: 0.9886
Epoch 4/50
1306/1313 [============================>.] - ETA: 0s - loss: 0.0527 - accuracy:
0.9945
Epoch 4: val_accuracy did not improve from 0.98861
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0525 -
accuracy: 0.9945 - val_loss: 0.2158 - val_accuracy: 0.9881
Epoch 5/50
1312/1313 [============================>.] - ETA: 0s - loss: 0.0482 - accuracy:
0.9942Restoring model weights from the end of the best epoch: 1.

Epoch 5: val_accuracy did not improve from 0.98861
1313/1313 [==============================] - 10s 7ms/step - loss: 0.0482 -
accuracy: 0.9942 - val_loss: 0.2275 - val_accuracy: 0.9870
Epoch 5: early stopping
```

The `bestmodel.h5` file serves a crucial purpose. It captures the weights of the model at the epoch where the validation accuracy is the highest. This allows for preserving the best-performing version of the model, avoiding overfitting to the training data. By saving these weights, we can later load the model with its optimal configuration for making predictions on new, unseen data without the need to retrain the entire model.

1.5 6. Testing

Now that the model has been trained and the best-performing configuration has been saved in `bestmodel.h5`, let's evaluate its performance on the test set.

[39]:
```
from keras.optimizers import Adam
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix

# Load the saved model
from keras.models import load_model

# Load the model in the native Keras format
loaded_model = load_model("C:\\Users\\tejus\\Desktop\\6th Sem␣
 ↪Project\\MNIST_Model\\best.h5")
```

```
# Compile the loaded model with the same configuration used during training
loaded_model.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])


# Preprocess the test data similar to the training data
X_test_processed = X_test.reshape((X_test.shape[0], X_test.shape[1], X_test.
 ↪shape[2], 1))
y_test_one_hot = to_categorical(y_test, num_classes=10)
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
```

[42]:
```
# Evaluate the model on the test set
test_results = saved_model.evaluate(X_test_processed, y_test_one_hot)

# Display the key evaluation metrics
print("Test Accuracy:", test_results[1])
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.2289 -
accuracy: 0.9886
Test Accuracy: 0.9886000156402588
```

1.5.1 8.Application:

This Python code creates an interactive application using the Pygame library. Users can draw numbers on a window with the mouse, and a pre-trained deep learning model ("bestmodel.h5") predicts the values in real-time. Pressing the "N" key clears the window for a new drawing.

[44]:
```
import pygame
import sys
import numpy as np
from tensorflow.keras.models import load_model
import cv2
# Press "N" to clear screen
WINDOWSIZEX = 640
WINDOWSIZEY = 480

WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)

MODEL = load_model("best.h5")

LABELS = {0: "Zero", 1: "One",
          2: "Two", 3: "Three",
          4: "Four", 5: "Five",
```

```
          6: "Six", 7: "Seven",
          8: "Eight", 9: "Nine"}

pygame.init()

FONT = pygame.font.Font("freesansbold.ttf", 18)
DISPLAYSURF = pygame.display.set_mode((WINDOWSIZEX, WINDOWSIZEY))

pygame.display.set_caption("Digit Board")

iswriting = False
image_cnt = 1
BOUNDARYINC = 5
PREDICT = True
IMAGESAVE = False

number_xcord = []
number_ycord = []
rect_min_x, rect_min_y, rect_max_x, rect_max_y = 0, 0, 0, 0  # Initialize␣
 ↪rectangle coordinates

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        if event.type == pygame.MOUSEMOTION and iswriting:
            xcord, ycord = event.pos
            pygame.draw.circle(DISPLAYSURF, WHITE, (xcord, ycord), 4, 0)

            number_xcord.append(xcord)
            number_ycord.append(ycord)

        if event.type == pygame.MOUSEBUTTONDOWN:
            iswriting = True

        if event.type == pygame.MOUSEBUTTONUP:
            iswriting = False
            if number_xcord and number_ycord:
                rect_min_x, rect_max_x = min(number_xcord) - BOUNDARYINC,␣
 ↪max(number_xcord) + BOUNDARYINC
                rect_min_y, rect_max_y = min(number_ycord) - BOUNDARYINC,␣
 ↪max(number_ycord) + BOUNDARYINC

                number_xcord = []
                number_ycord = []
```

```
                image_arr = np.array(pygame.PixelArray(DISPLAYSURF))[rect_min_x:
↪rect_max_x, rect_min_y:rect_max_y].T.astype(np.float32)

                if IMAGESAVE:
                    cv2.imwrite(f"image_{image_cnt}.png", image_arr)
                    image_cnt += 1

                if PREDICT:
                    image = cv2.resize(image_arr, (28, 28))
                    image = np.pad(image, ((10, 10), (10, 10)), 'constant',␣
↪constant_values=0)
                    image = cv2.resize(image, (28, 28)) / 255.0

                    label = str(LABELS[np.argmax(MODEL.predict(image.reshape(1,␣
↪28, 28, 1)))])
                    text_surface = FONT.render(label, True, RED, WHITE)
                    text_rect_obj = text_surface.get_rect()
                    text_rect_obj.left, text_rect_obj.bottom = rect_max_x + 5,␣
↪rect_min_y

                    DISPLAYSURF.blit(text_surface, text_rect_obj)

                # Draw a rectangular box around the written number
                pygame.draw.rect(DISPLAYSURF, RED, (rect_min_x, rect_min_y,␣
↪rect_max_x - rect_min_x, rect_max_y - rect_min_y), 2)

        if event.type == pygame.KEYDOWN:
            if event.unicode == 'n':
                DISPLAYSURF.fill(BLACK)

    pygame.display.update()
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 16ms/step
```

```
An exception has occurred, use %tb to see the full traceback.

SystemExit
```

1.6 Project Conclusion

In this project, we followed a structured workflow to build and train a Convolutional Neural Network (CNN) using Keras for the classification of MNIST handwritten numerical digits. Let's summarize our key findings and achievements based on the project workflow:

1.6.1 Project Workflow:

1. **Data Loading and Exploration:** We successfully loaded and explored the MNIST dataset, gaining insights into the structure of the training and testing data.
2. **Data Preprocessing:** Prior to feeding the data into the model, we performed essential preprocessing steps, including converting integer labels to one-hot encoding and reshaping the input data to meet the requirements of a CNN.
3. **Model Architecture:** The CNN architecture was meticulously designed, comprising convolutional layers, max-pooling layers, flattening, dropout for regularization, and a densely connected output layer tailored for digit classification.
4. **Compilation:** The model was compiled using the Adam optimizer, categorical crossentropy loss function, and accuracy as the monitoring metric.
5. **Training:** Training was executed for 50 epochs, with a validation split of 30%. EarlyStopping and ModelCheckpoint callbacks were employed to enhance training efficiency and preserve the best-performing model.
6. **Testing:** The testing phase ensures the loaded model's accurate performance on new data, addressing warnings by manually compiling, and evaluating key metrics for insights into its effectiveness.
7. **Application:** This Python code utilizes the Pygame library to create an interactive application where users can draw numbers on a window using the mouse. The drawn digits are then processed by a pre-trained deep learning model (loaded from "bestmodel.h5"), and the predicted values are displayed alongside the drawn numbers in real-time.

1.6.2 Future Considerations:

As we conclude this project, there are avenues for further exploration, such as hyperparameter tuning, experimenting with different architectures, or exploring transfer learning approaches for enhanced performance.

This project not only provided practical insights into building and training CNNs but also emphasized the significance of thoughtful model architecture, training strategies, and the use of callbacks to improve overall model performance. rall model performance.