CNN Implementation for MNIST Digit Recognition

Introduction:

Automatic handwritten text recognition is an essential area of research in digital image processing, with practical applications ranging from bank check processing to form data entry and historical document digitalization. Among these, recognizing handwritten digits is a fundamental job that acts as a baseline for assessing the effectiveness of image processing algorithms.

We aim implement a Convolutional Neural Network (CNN) that can correctly recognize handwritten digits from the MNIST dataset. This dataset, which contains thousands of labeled images of handwritten digits, is a benchmark for training and testing machine learning models in optical character recognition (OCR).

The project is crucial for its practical implications and as a learning exercise in understanding the complexities of neural network topologies, particularly CNNs, which are at the forefront of deep learning. CNNs can uniquely capture spatial hierarchies in picture data due to their deep design and convolution processes that maintain pixel associations.

Process Used:

Importing Necessary Libraries: We begin by importing essential libraries such as NumPy, Pandas, Matplotlib, Seaborn, and scikit-learn's datasets module, tensorflow modules, Conv2D, MaxPooling2D, Flatten, Dense, Dropout and sequential.

Loading the MNIST Dataset: The MNIST dataset is loaded using https://archive.ics.uci.edu/dataset/80/optical+recognition+of+handwritten+digits the following link.

Exploratory Data Analysis (EDA): Exploratory data analysis is performed to understand the dataset's structure and distribution. This includes printing dataset information, class distribution, and visualizing pairwise relationships between features using a pair plot.

Data Exploration and Preparation:

Reshaping the Data: The X_train and X_test arrays are reshaped from 4D (samples, 8, 8, 1) to 2D (samples, 64), converting the 8x8 pixel images into a single vector with 64 features per sample.

Creating DataFrames: Creates pandas data frames from these 2D arrays, identifying each column Pixel_0, Pixel_1,..., Pixel_63 to represent a pixel in the flattened image. Printing the data: Displays the first few rows of the training and testing data frames as a snapshot of the prepared data, ready for visual inspection or further processing in pandas.

This transformation provides a tabular representation of the data, which is better suited for certain sorts of analysis and visualization, but also discards the spatial correlations between pixels, which are critical for convolutional neural network processing.

Data Preprocessing:

- **Splitting the Dataset:** The dataset is split into training and testing sets using the train_test_split function from scikit-learn.
- **Scaling the Features:** Feature scaling is applied using Standard Scaler to ensure all features have the same scale, which is crucial for many Deep learning algorithms.

Dataset Download: The ucimlrepo package directly downloads the Optical Recognition of Handwritten Digits dataset.

Data Extraction: Extracts the dataset's features (X) and labels (Y).

Normalization: Use StandardScaler to normalize feature values, which is necessary for practical neural network training.

Reshaping: Transforms the data into a format suitable for input into a CNN, assuming that each image is 8x8 pixels in a single grayscale channel.

One-Hot Encoding: Converts integer labels into a binary class matrix that can be used with categorical cross-entropy loss during model training.

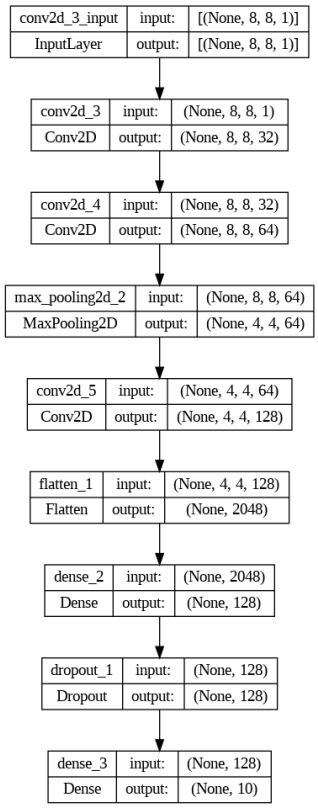
Splitting the Dataset: Separates the data into training and testing sets to assess the model's performance.

Implementation of CNN:

To build and implement the requested Convolutional Neural Network (CNN) architecture using TensorFlow/Keras, let us first define the architecture based on your requirement for 3 to 5 convolutional layers, each with ReLU activation functions. After that, I will show you how to create a diagram of this architecture.

Define the CNN Architecture: We will use Keras to define a CNN model suitable for classifying the Optical Recognition of Handwritten Digits dataset.

Visualizing the CNN Architecture: To visualize this CNN architecture, you can either use Keras' built-in plot_model function or draw a diagram yourself using tools such as diagrams.net, Lucidchart, or equivalent.



Running the Model:

After creating and visualizing the model, the following stages would be to train it using the prepared training data (X_train, y_train) and then evaluate it with the test data.

```
Epoch 1/10
127/127 [==============================] - 6s 34ms/step - loss: 0.6558 - accuracy: 0.7961 - val_loss: 0.0898 - val_accuracy: 0.9778
Epoch 2/10
127/127 [==============================] - 3s 22ms/step - loss: 0.1534 - accuracy: 0.9543 - val_loss: 0.0437 - val_accuracy: 0.9911
Epoch 3/10
127/127 [==============================] - 3s 21ms/step - loss: 0.0888 - accuracy: 0.9750 - val_loss: 0.0249 - val_accuracy: 0.9933
Epoch 4/10
127/127 [==============================] - 3s 21ms/step - loss: 0.0555 - accuracy: 0.9827 - val_loss: 0.0251 - val_accuracy: 0.9889
Epoch 5/10
127/127 [==============================] - 3s 27ms/step - loss: 0.0490 - accuracy: 0.9859 - val_loss: 0.0164 - val_accuracy: 0.9956
Epoch 6/10
127/127 [==============================] - 3s 25ms/step - loss: 0.0275 - accuracy: 0.9921 - val_loss: 0.0129 - val_accuracy: 0.9933
Epoch 7/10
127/127 [==============================] - 3s 21ms/step - loss: 0.0261 - accuracy: 0.9918 - val_loss: 0.0146 - val_accuracy: 0.9933
Epoch 8/10
127/127 [==============================] - 3s 21ms/step - loss: 0.0186 - accuracy: 0.9960 - val_loss: 0.0145 - val_accuracy: 0.9956
Epoch 9/10
127/127 [==============================] - 3s 22ms/step - loss: 0.0189 - accuracy: 0.9953 - val_loss: 0.0335 - val_accuracy: 0.9867
Epoch 10/10
127/127 [==============================] - 4s 30ms/step - loss: 0.0072 - accuracy: 0.9978 - val_loss: 0.0226 - val_accuracy: 0.9911
36/36 [==============================] - 0s 11ms/step - loss: 0.0386 - accuracy: 0.9902
Test Accuracy: 99.02%
```

Pooling:

We experience negative dimension size produced by subtracting 2 from 1, arises when the spatial dimensions of the feature map become too small for further max pooling operations after numerous layers of convolution and pooling. This is typical when the input image size is modest, such as 8x8 photos.

Before flattening the convolutional layer outputs, we must change the architecture to ensure that their spatial dimensions do not go below 1x1.

Padding: I added padding='same' to convolutional layers to ensure that each layer's output size is the same as its input size. This prevents the dimensions from shrinking too quickly.

Pooling Changes: Reduced the number of pooling layers or rearranged their placements. Because the input images are small, excessive pooling can result in a significant reduction in spatial dimensions.

These changes ensure that the model's architecture is better adapted to the small 8x8 input size, minimizing dimensionality reduction problems while efficiently using convolution and pooling layers.

To view the Convolutional Neural Network (CNN) architecture, utilize the TensorFlow/Keras plot_model function, which you've already included in your code. This function provides a diagram of the model's layers, including connections and tensor forms.

Plot_model is called using the model object, and to_file is used to save the diagram as an image file. show_shapes=True includes each layer's input and output shapes in the diagram, while show_layer_names=True displays each layer's name.

Max pooling is a downsampling approach often employed in CNNs to minimize the dimensionality of feature maps, which reduces computational complexity and parameter count. This procedure selects the greatest value from a set of pixels in a feature map, usually within a 2x2 window. This helps to make feature detection resistant to tiny shifts and distortions.

In this model, max pooling is already applied after the second convolutional layer. Here's how it is organized:

A 2x2 max pooling layer cuts the spatial dimensions of the feature map in half.

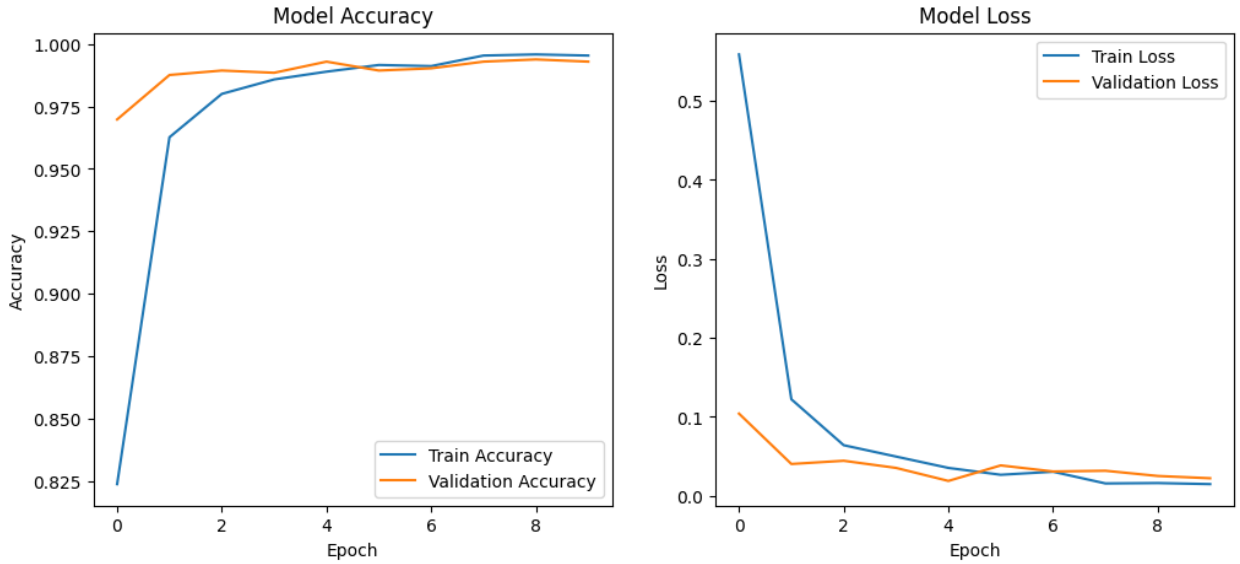
This improves not only computing efficiency but also allows the network to focus on the most important elements.

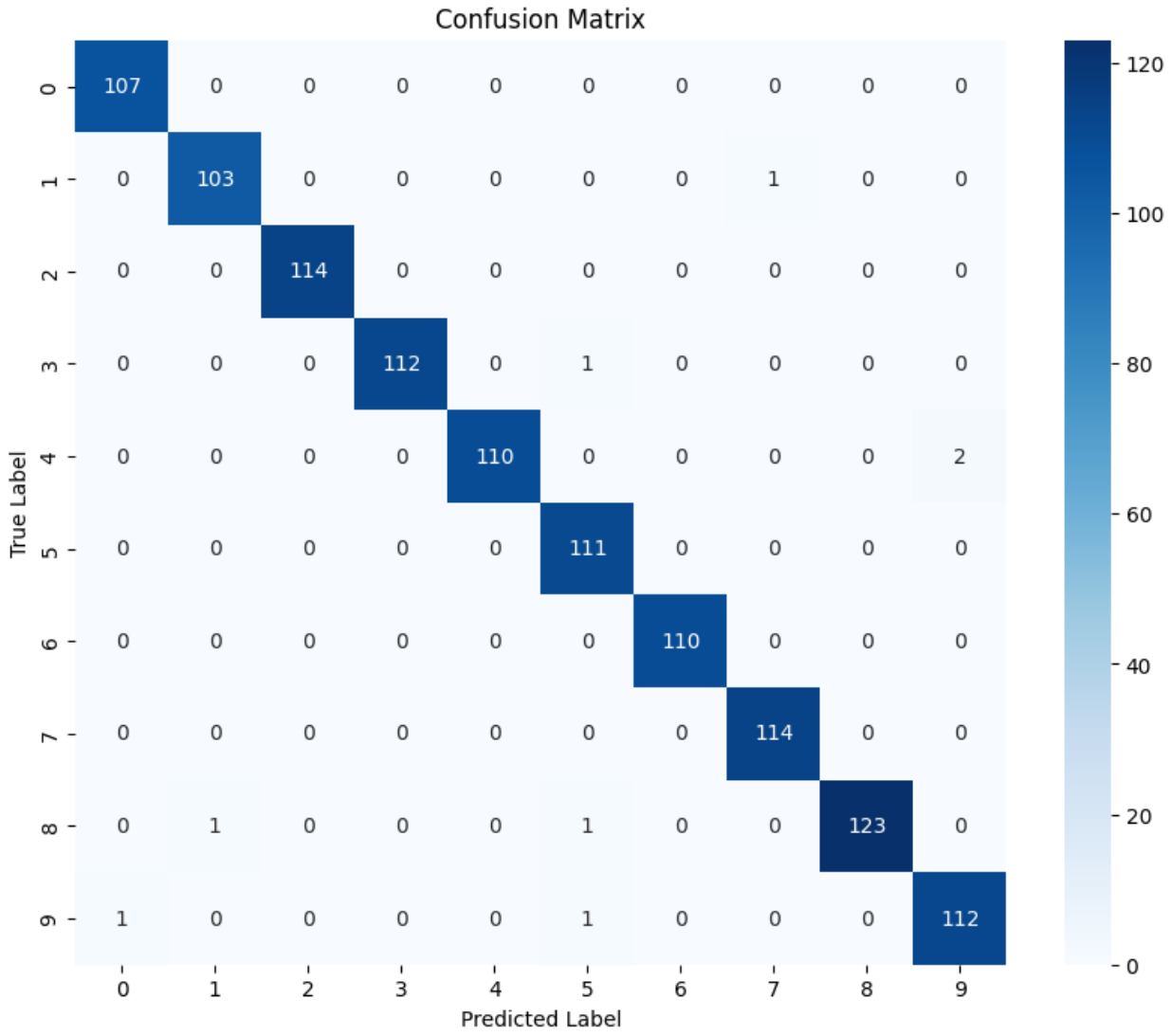
Cross-Validation:

K-fold cross-validation is a reliable method for evaluating a model's performance. It includes dividing the dataset into K subgroups (folds). For each iteration, one fold is utilized as the test set and the others as the training set. We will utilize Keras' KFold class from sklearn.model_selection.

Plot Loss and Accuracy Curves: After training, plot the training and validation loss/accuracy curves to see learning progress and performance.

A confusion matrix provides insight into the types of errors the model is making.





Conclusion:

We seek to use Convolutional Neural Networks to define a standard for handwritten digit recognition. By obtaining high accuracy in recognizing digits from the MNIST dataset, this research will demonstrate the usefulness of CNNs in handling complicated image recognition tasks, laying the groundwork for more advanced OCR tasks and real-world applications.

Link to my Git Hub:

https://github.com/nuthalapatideepika/ML-CNN-Project.git

This is my code:

```
Plotting the Confusion Matrix

from sklearn.metrics import confusion_matrix
import seaborn as sns

# Predict the values from the last fold's test data
y_pred = model.predict(X_reshaped[test])
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_categorical[test], axis=1)

# Compute the confusion matrix
cm = confusion_matrix(y_true, y_pred_classes)

# Plotting the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()
```

```
                    **CNN Implementation for MNIST Digit Recognition **

 Importing the Libraries

 import numpy as np
 import pandas as pd
 from sklearn.preprocessing import StandardScaler
 from sklearn.model_selection import train_test_split
 from tensorflow.keras.utils import to_categorical
 from ucimlrepo import fetch_ucirepo

 Loading The Dataset

 pip install ucimlrepo

 from ucimlrepo import fetch_ucirepo

 # fetch dataset
 optical_recognition_of_handwritten_digits = fetch_ucirepo(id=80)

 # data (as pandas dataframes)
 X = optical_recognition_of_handwritten_digits.data.features
 y = optical_recognition_of_handwritten_digits.data.targets
```

```
# metadata
print(optical_recognition_of_handwritten_digits.metadata)

# variable information
print(optical_recognition_of_handwritten_digits.variables)
```

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Reshape data for CNN input
# Assuming each image is 8x8 pixels and 1 channel (grayscale)
X_reshaped = X_scaled.reshape(-1, 8, 8, 1)

# Convert labels to one-hot encoding
y_categorical = to_categorical(y)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_reshaped, y_categorical, test_size=0.2,

# Convert X_train and X_test from 4D arrays back to 2D arrays for table display
X_train_flat = X_train.reshape(X_train.shape[0], -1)  # Reshape to (number of samples, 64)
X_test_flat = X_test.reshape(X_test.shape[0], -1)     # Reshape to (number of samples, 64)

# Create pandas DataFrames
train_df = pd.DataFrame(X_train_flat, columns=[f"Pixel_{i}" for i in range(64)])
test_df = pd.DataFrame(X_test_flat, columns=[f"Pixel_{i}" for i in range(64)])

# Display the first few rows of each DataFrame
print("Training Data:")
# print(train_df.head())
print("\nTesting Data:")
# print(test_df.head())
```

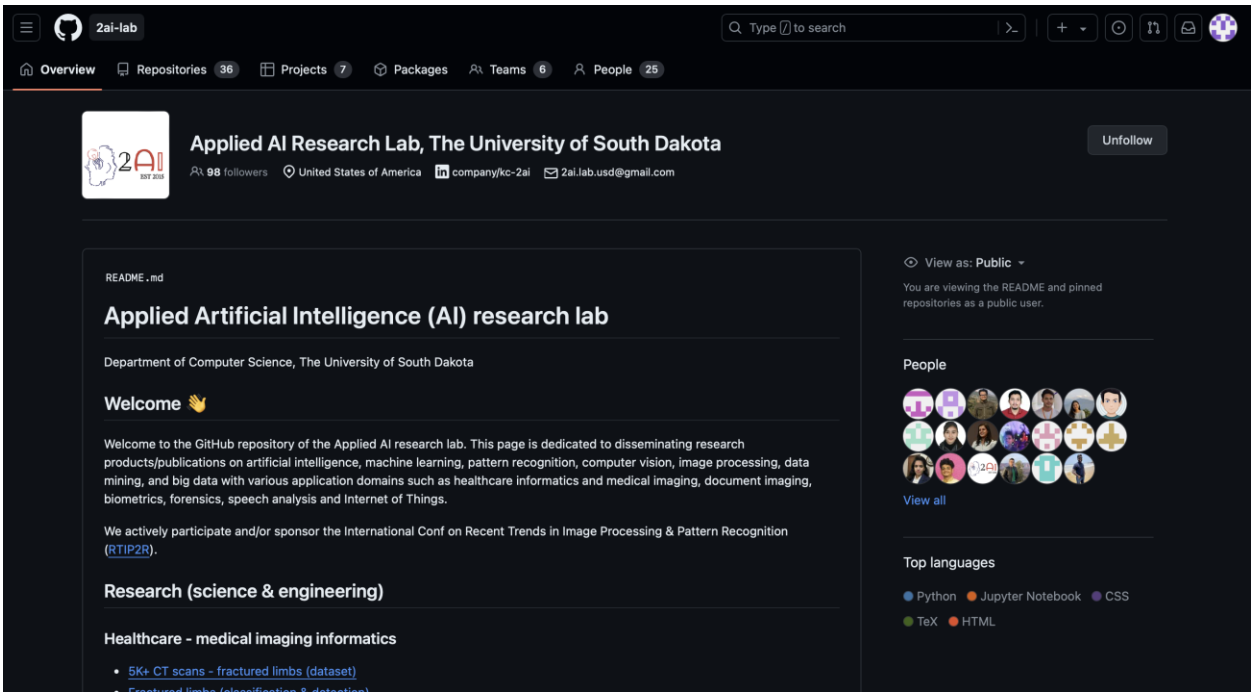
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# Define the CNN model
model = Sequential([
    # First Convolutional Layer
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(8, 8, 1), padding='same'),

    # Second Convolutional Layer (optional pooling)
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),  # Reduce dimension to 4x4

    # Third Convolutional Layer
    Conv2D(128, (3, 3), activation='relu', padding='same'),  # Using padding to maintain dimen

    # Optional: Adding pooling here if additional dimension reduction is needed (to 2x2)
    # MaxPooling2D(pool_size=(2, 2)),

    # Flattening the outputs from the convolutional layers to feed into a dense layer
    Flatten(),

    # Dense Layer
    Dense(128, activation='relu'),
    Dropout(0.5),
```

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

# Print model summary to review the architecture
model.summary()


from tensorflow.keras.utils import plot_model

# Generate a plot of the model
plot_model(model, to_file='model_architecture.png', show_shapes=True, show_layer_names=True)


Training the CNN Model

# Train the model
model.fit(X_train, y_train, epochs=10, validation_split=0.1, batch_size=32)

# Evaluate the model
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy*100:.2f}%")


from tensorflow.keras.utils import plot_model
```

```
from tensorflow.keras.utils import plot_model

# Assuming your model is named 'model'
# Generate a plot of the model
plot_model(model, to_file='model_architecture.png', show_shapes=True, show_layer_names=True, d

# Display the image in a Jupyter notebook (if you're using one)
from IPython.display import Image
Image(filename='model_architecture.png')


from tensorflow.keras.models import Model

# Define the model to output feature maps
layer_outputs = [model.layers[1].output,  # Output of the second convolutional layer (before p
                 model.layers[2].output]  # Output of the max pooling layer (after pooling)

feature_map_model = Model(inputs=model.input, outputs=layer_outputs)

# Use this model to predict on a sample input to get the feature maps
feature_maps = feature_map_model.predict(X_train[:1])  # Using the first image in the train se

# Feature maps before pooling
feature_maps_before_pooling = feature_maps[0]
# Feature maps after pooling
feature_maps_after_pooling = feature_maps[1]
```

```
Pooling the pictures

import matplotlib.pyplot as plt

def plot_feature_maps(feature_maps, title):
    num_maps = feature_maps.shape[-1]  # Get the number of feature maps
    fig, axes = plt.subplots(1, num_maps, figsize=(num_maps * 2.5, 3))  # Dynamic sizing of th
    fig.suptitle(title)

    if num_maps == 1:  # If there is only one feature map, axes is not an array
        axes = [axes]

    for i, ax in enumerate(axes):
        # Displaying the i-th feature map
        ax.imshow(feature_maps[:, :, :, i].squeeze(), cmap='gray', aspect='auto')
        ax.axis('off')
    plt.show()

# Now using the function to plot the feature maps
# Plot feature maps before and after pooling
plot_feature_maps(feature_maps_before_pooling, "Feature Maps Before Pooling")
plot_feature_maps(feature_maps_after_pooling, "Feature Maps After Pooling")


from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
```

```
model = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(8, 8, 1), padding='same'),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),  # Reduce dimension to 4x4
    Conv2D(128, (3, 3), activation='relu', padding='same'),  # Using padding to maintain dimen
    Flatten(),  # Flattening the outputs from the convolutional layers
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')  # Output layer with softmax activation for 10 classes
])

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


Using K-Fold

from sklearn.model_selection import KFold
import numpy as np
```

```
Using K-Fold

from sklearn.model_selection import KFold
import numpy as np

# Parameters
num_folds = 5
kfold = KFold(n_splits=num_folds, shuffle=True, random_state=42)

# K-fold Cross Validation model evaluation
fold_no = 1
accuracies = []
losses = []

for train, test in kfold.split(X_reshaped, y_categorical):
    model = Sequential([
        Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(8, 8, 1), padding='same
        Conv2D(64, (3, 3), activation='relu', padding='same'),
        MaxPooling2D(pool_size=(2, 2)),
        Conv2D(128, (3, 3), activation='relu', padding='same'),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
Plotting our Model

import matplotlib.pyplot as plt

# Average scores
print(f'Average Accuracy: {np.mean(accuracies)*100}% (+- {np.std(accuracies)})')
print(f'Average Loss: {np.mean(losses)}')

# Plotting training history from last fold for simplicity (you might want to average histories
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```