Project: CNN

Team members:
Full Name | Student ID | GitHub Profile Link | Project GitHub Repository Link

- **Anish Ghimire (101143773)**

[https://github.com/anish-g] [https://github.com/anish-g/MLF-CNN-Project]

- **Prajwol Tiwari (101144638)**

[https://github.com/prajwol148] [https://github.com/prajwol148/CNN-Implementation-for-MNIST-Digit-Recognition]

- **Pramesh Baral (101139536)**

[https://github.com/prms318] [https://github.com/Prms318/CNN-Team-Project]

- **Pradip Ganesh (101124775)**

[https://github.com/pradipganesh61] [https://github.com/pradipganesh61/CNN]

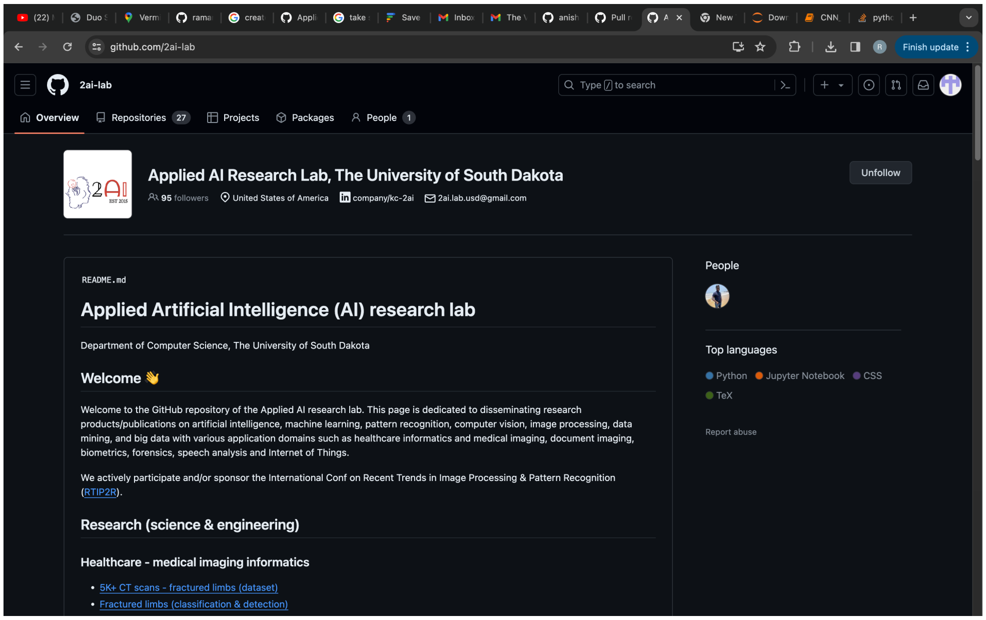
- **Shashwat Shrestha (101130302)**

[https://github.com/shashwatstha23] [https://github.com/shashwatstha23/cnnproject_mlf]

- **Raman Regmi (101131084)**

[https://github.com/ramanregmi] [https://github.com/ramanregmi/Project]

2AI Lab GitHub task:



Overview

The goal of this project is to build and evaluate a Convolutional Neural Network (CNN) for recognizing handwritten digits from the MNIST Dataset.

Data Exploration and Preparation

This version of MNIST dataset titled 'Optical Recognition of Handwritten Digits' is hosted in **UC Irvine Machine Learning Repository**.

Datasets can be directly fetched by using their python library called `ucimlrepo`.

Installing the `ucimlrepo` python library.

In [1]:
```
!pip install ucimlrepo
```

```
Collecting ucimlrepo
  Downloading ucimlrepo-0.0.6-py3-none-any.whl (8.0 kB)
Installing collected packages: ucimlrepo
Successfully installed ucimlrepo-0.0.6
```

Fetching the dataset from the UCI Machine Learning Repository

In [2]:
```
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)
```

```
{'uci_id': 80, 'name': 'Optical Recognition of Handwritten Digits', 'repository_ur
l': 'https://archive.ics.uci.edu/dataset/80/optical+recognition+of+handwritten+digit
s', 'data_url': 'https://archive.ics.uci.edu/static/public/80/data.csv', 'abstract':
'Two versions of this database available; see folder', 'area': 'Computer Science',
'tasks': ['Classification'], 'characteristics': ['Multivariate'], 'num_instances': 5
620, 'num_features': 64, 'feature_types': ['Integer'], 'demographics': [], 'target_c
ol': ['class'], 'index_col': None, 'has_missing_values': 'no', 'missing_values_symbo
l': None, 'year_of_dataset_creation': 1998, 'last_updated': 'Wed Aug 23 2023', 'data
set_doi': '10.24432/C50P49', 'creators': ['E. Alpaydin', 'C. Kaynak'], 'intro_pape
r': {'title': 'Methods of Combining Multiple Classifiers and Their Applications to H
andwritten Digit Recognition', 'authors': 'C. Kaynak', 'published_in': 'MSc Thesis,
Institute of Graduate Studies in Science and Engineering, Bogazici University', 'yea
r': 1995, 'url': None, 'doi': None}, 'additional_info': {'summary': 'We used preproc
essing programs made available by NIST to extract normalized bitmaps of handwritten
digits from a preprinted form. From a total of 43 people, 30 contributed to the trai
ning set and different 13 to the test set. 32x32 bitmaps are divided into nonoverlap
ping blocks of 4x4 and the number of on pixels are counted in each block. This gener
ates an input matrix of 8x8 where each element is an integer in the range 0..16. Thi
s reduces dimensionality and gives invariance to small distortions.\r\n\r\nFor info
on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G. T. Candela, D. L. D
immick, J. Geist, P. J. Grother, S. A. Janet, and C. L. Wilson, NIST Form-Based Hand
print Recognition System, NISTIR 5469, 1994.', 'purpose': None, 'funded_by': None,
'instances_represent': None, 'recommended_data_splits': None, 'sensitive_data': Non
e, 'preprocessing_description': None, 'variable_info': 'All input attributes are int
egers in the range 0..16.\r\nThe last attribute is the class code 0..9', 'citation':
None}}
```


```
           name     role         type demographic description units  \
0    Attribute1  Feature      Integer        None        None  None
1    Attribute2  Feature      Integer        None        None  None
2    Attribute3  Feature      Integer        None        None  None
3    Attribute4  Feature      Integer        None        None  None
4    Attribute5  Feature      Integer        None        None  None
..          ...      ...          ...         ...         ...   ...
60  Attribute61  Feature      Integer        None        None  None
61  Attribute62  Feature      Integer        None        None  None
62  Attribute63  Feature      Integer        None        None  None
63  Attribute64  Feature      Integer        None        None  None
64        class   Target  Categorical        None        None  None

   missing_values
0              no
1              no
2              no
3              no
4              no
..            ...
60             no
61             no
62             no
63             no
64             no

[65 rows x 7 columns]
```

Important necessary libraries for data manipulation

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Grouping the dataset by class so that a random data can be picked from each class.

In [4]:
```
gk = y.groupby('class')
```

Storing class names as it comes in the dataset

In [5]:
```
class_labels = y['class'].unique()
```

This version of MNIST dataset has already been normalized and dimensionality reduced.

The images are in the form of matrix of size 8*8 where each element is an integer in the range 0...16.

However, the image is flatten into an array of length 64.

The feature set is reshaped from 1D to 2D (8*8 pixels), aligning with the CNN's input requirements.

In [6]:
```
X_images = X.to_numpy().reshape(-1, 8, 8)
```

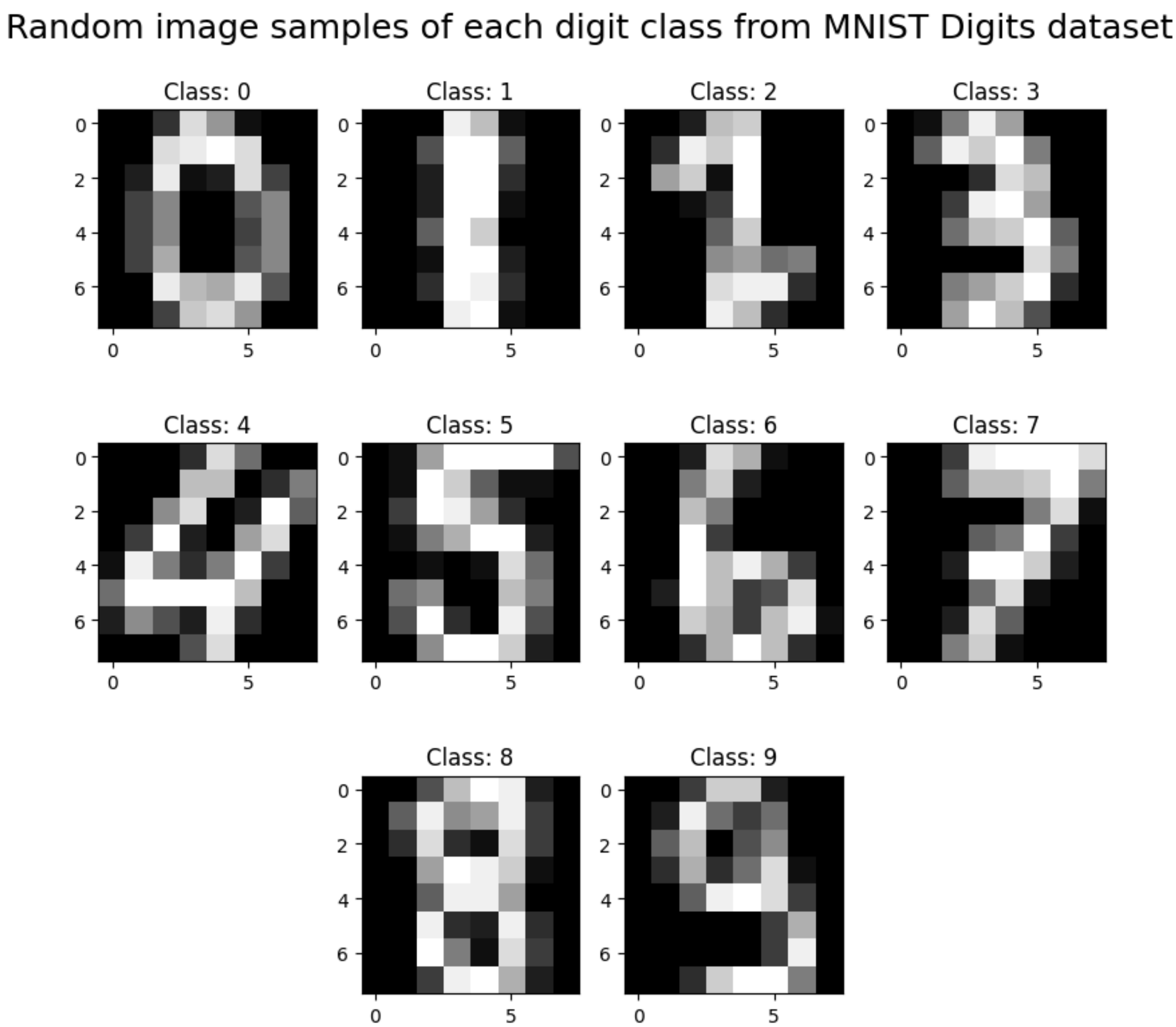
Method to get a random index for sample images of each class so that a sample image for each digit class can be visualized.

In [7]:
```
def get_index_sample_each_class():
  sample_indices = []

  for i in range(10):
    sample_indices.append(gk.get_group(i).sample().index[0])

  return sample_indices
```

Visualizing a sample image from each digit class

In [11]:
```
sample_idxs = get_index_sample_each_class()

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

for i in range(10):

  plt.subplot(3, 4, i + 1 if i < 8 else i + 2)

  plt.imshow(X_images[sample_idxs[i]], cmap='gray', interpolation='none')
  plt.title(f'Class: {i}')

plt.suptitle('Random image samples of each digit class from MNIST Digits dataset',
             fontsize=18)
```

```
plt.tight_layout()
plt.show()
```



The dataset is divided into training and testing sets with an 80-20 split to ensure a fair evaluation of the model. The training set helps in fitting the model, while the testing set is used to evaluate its generalization capability.

In [12]:
```
from sklearn.model_selection import train_test_split
```

In [13]:
```
X_train, X_test, y_train, y_test = train_test_split(X_images,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)

print(f'Shape of train-test data after split.\
\nX_train: {X_train.shape}\tX_test: {X_test.shape}\
\ny_train: {y_train.shape}\ty_test: {y_test.shape}')
```

```
Shape of train-test data after split.
X_train: (4496, 8, 8)   X_test: (1124, 8, 8)
y_train: (4496, 1)      y_test: (1124, 1)
```

The labels are converted from a class vector (integers) to binary class matrix for use with categorical crossentropy during the training of the model.

In [14]:
```
from keras.utils import to_categorical
```

In [15]:
```
y_train_labels = y_train
y_test_labels = y_test

y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

Convolutional Neural Network Architecture

Importing necessary modules from keras

In [16]:
```
from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
from keras.optimizers import Adam

from keras.models import Model
```

Method to build CNN architecture

In [17]:
```
def create_cnn_model():
  model = Sequential()

  model.add(Conv2D(filters=16,
                   kernel_size=(3, 3),
                   activation='relu',
                   strides=1,
                   padding='same',
                   data_format='channels_last',
                   input_shape=(8, 8, 1)))

  model.add(MaxPooling2D(pool_size=(2, 2),
                         strides=2,
                         padding='valid'))

  model.add(Dropout(0.25))

  model.add(Conv2D(filters=32,
                   kernel_size=(3, 3),
                   activation='relu',
                   strides=1,
                   padding='same',
                   data_format='channels_last'))

  model.add(MaxPooling2D(pool_size=(2, 2),
                         strides=2,
                         padding='valid'))
```

```
    model.add(Dropout(0.25))

    model.add(Conv2D(filters=64,
                     kernel_size=(3, 3),
                     activation='relu',
                     strides=1,
                     padding='same',
                     data_format='channels_last'))

    model.add(MaxPooling2D(pool_size=(2, 2),
                           strides=2,
                           padding='valid'))

    model.add(Dropout(0.25))

    model.add(Flatten())

    model.add(Dense(128, activation='sigmoid'))

    model.add(Dropout(0.25))

    model.add(Dense(512, activation='sigmoid'))

    model.add(Dropout(0.25))

    model.add(Dense(10, activation='softmax'))

    return model
```

In [18]:
```
model = create_cnn_model()
```

Visualizing the CNN architecture

In [19]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d (Conv2D)             (None, 8, 8, 16)          160       
                                                                 
 max_pooling2d (MaxPooling2  (None, 4, 4, 16)          0         
 D)                                                              
                                                                 
 dropout (Dropout)           (None, 4, 4, 16)          0         
                                                                 
 conv2d_1 (Conv2D)           (None, 4, 4, 32)          4640      
                                                                 
 max_pooling2d_1 (MaxPoolin  (None, 2, 2, 32)          0         
 g2D)                                                            
                                                                 
 dropout_1 (Dropout)         (None, 2, 2, 32)          0         
                                                                 
 conv2d_2 (Conv2D)           (None, 2, 2, 64)          18496     
                                                                 
 max_pooling2d_2 (MaxPoolin  (None, 1, 1, 64)          0         
 g2D)                                                            
                                                                 
 dropout_2 (Dropout)         (None, 1, 1, 64)          0         
                                                                 
 flatten (Flatten)           (None, 64)                0         
                                                                 
 dense (Dense)               (None, 128)               8320      
                                                                 
 dropout_3 (Dropout)         (None, 128)               0         
                                                                 
 dense_1 (Dense)             (None, 512)               66048     
                                                                 
 dropout_4 (Dropout)         (None, 512)               0         
                                                                 
 dense_2 (Dense)             (None, 10)                5130      
                                                                 
=================================================================
Total params: 102794 (401.54 KB)
Trainable params: 102794 (401.54 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

The CNN architecture designed for recognizing 8x8 pixel images of handwritten digits comprises several layers, each with specific functions aimed at processing and transforming the input images into a form where digit classifications can be made effectively.

1. First Convolutional Layer

- Parameters: Consists of 16 filters, each of size 3x3.
- Activation: Utilizes the ReLU activation function.
- Dimensions: Maintains the same spatial dimensions (8x8) due to the 'same' padding strategy.

- Effect: This layer is responsible for capturing basic visual features such as edges and gradients within the image. Each filter produces a separate feature map, resulting in 16 different representations of the input image, each highlighting different aspects of the image.

2. First Pooling Layer

- Pooling Size: Uses a 2x2 window for pooling.
- Strides: With a stride of 2 and 'valid' padding, it reduces the spatial dimensions of each feature map from 8x8 to 4x4.
- Effect: Pooling layers serve to reduce the spatial dimensions of the feature maps, which decreases the number of parameters and computation in the network. This operation helps in extracting the dominant features while reducing the sensitivity to the exact locations of features.

3. First Dropout Layer

- Dropout Rate: Set at 25%.
- Effect: Dropout layers randomly set a fraction of the input units to zero during training, which helps in preventing overfitting by ensuring that no single set of neurons within the layer overly specializes to the training data.

4. Second Convolutional Layer

- Parameters: Increases to 32 filters, maintaining the 3x3 size.
- Activation: Continues with ReLU activation.
- Dimensions: The feature maps remain at 4x4 due to 'same' padding.
- Effect: This layer extracts more complex features from the simplified outputs provided by the first pooling layer. By increasing the number of filters, it allows the network to develop a richer understanding of the input data.

5. Second Pooling Layer

- Pooling Size and Strides: Same as the first pooling layer, reducing each feature map size further from 4x4 to 2x2.
- Effect: Further reduces the spatial dimensions, focusing on the most salient features, and helps in further reducing the computational complexity.

6. Second Dropout Layer

- Dropout Rate: Maintains at 25%.
- Effect: Adds another layer of regularization to enhance the model's generalization capabilities.

7. Third Convolutional Layer

- Parameters: Further increases the filter count to 64.

- Activation: Uses ReLU activation.
- Dimensions: Maintains the 2x2 dimensions with 'same' padding.
- Effect: This layer captures even higher-level features from the input data. With more filters, the network can capture a more diverse set of features, crucial for accurate classification tasks.

8. Third Pooling Layer

- Effect: Reduces each 2x2 feature map to 1x1, effectively distilling the feature maps to their most essential elements.

9. Third Dropout Layer

- Effect: Further ensures that the model avoids overfitting, especially important as the complexity of the model increases.

10. Flatten Layer

- Effect: Transforms the 3D output of the previous convolutional layers to a 1D array without affecting the batch size. This layer prepares the data for the final classification steps in the dense layers.

11. Dense and Dropout Layers

- Configuration: Includes dense layers with 128 and 512 neurons, each followed by a dropout layer, with sigmoid activation for the dense layers and softmax for the final output.
- Effect: These layers integrate the features learned by the convolutions into predictions for the 10 classes of digits. The dropout layers interspersed between them prevent overfitting by randomly dropping a portion of the neurons, ensuring that different neurons can learn to identify various features independently.

12. Output Layer

- Activation: Uses softmax activation.
- Effect: Outputs the probability distribution across the 10 digit classes, allowing for the classification of the input digit image into one of these classes based on the highest probability.

Model Compilation

Optimizer:

The model uses the Adam optimizer, which is an extension to stochastic gradient descent. This optimizer is particularly effective for problems involving a lot of data or parameters. Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.

Loss Function: `categorical_crossentropy` loss function is used when there are two or more label classes. The labels are expected to be provided in a one-hot representation. This is appropriate since the network's output uses a softmax activation function, which outputs a probability distribution over the classes. Categorical crossentropy will compare the distribution produced by the output layer with the true distribution, where the true probability is 100% for the actual class.

In [20]:
```
optimizer = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)

model.compile(optimizer=optimizer, loss="categorical_crossentropy", metrics=["accur
```

Visualizing CNN feature maps

Visualizing the feature maps of the CNN architecture after it processes the input image.

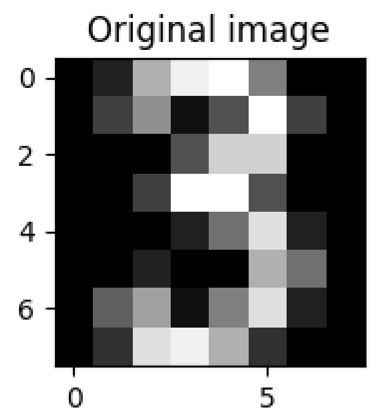
`activation_model` is a new model derived from the original model but designed to output the activations from each convolutional and max pooling layer instead of just the final output.

For each layer, a grid of all filter outputs (feature maps) is displayed. The grid is scaled based on the number of filters, providing a clear view of each filter’s pattern recognition.

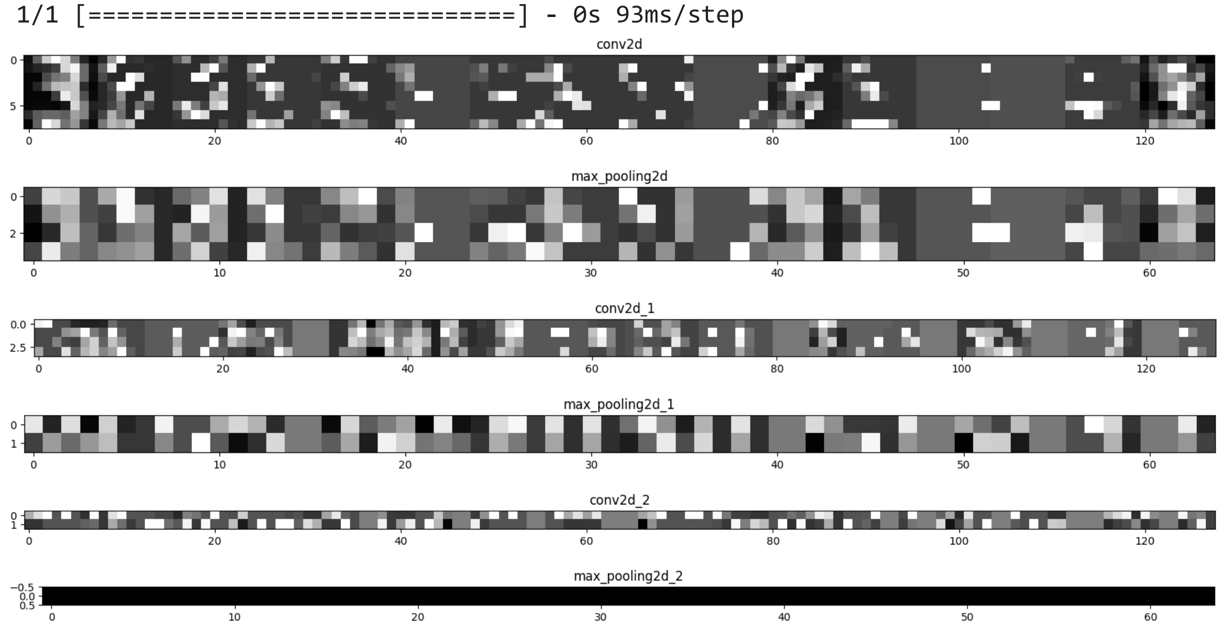
In [21]:
```
from keras.models import Model
import matplotlib.pyplot as plt
import numpy as np

activation_model = Model(inputs=model.input,
                         outputs=[layer.output for layer in model.layers if isinsta

def display_feature_maps(image_index, X_images):
    input_tensor = np.expand_dims(X_images[image_index], axis=0)
    plt.figure(figsize=(2, 2))
    plt.title('Original image')
    plt.imshow(X_images[image_index], aspect='auto', cmap='gray')
    plt.show()

    activations = activation_model.predict(input_tensor)

    for layer, layer_activation in zip([layer for layer in model.layers if isinstan
        num_filters = layer_activation.shape[-1]
        size = layer_activation.shape[1]
        display_grid = np.zeros((size, size * num_filters))

        for i in range(num_filters):
            x = layer_activation[0, :, :, i]
            x -= x.mean()
            if x.std() > 0:
                x /= x.std()
            x *= 64
            x += 128
```

```
        x = np.clip(x, 0, 255).astype('uint8')
        display_grid[:, i * size : (i + 1) * size] = x

    scale = 20. / num_filters
    plt.figure(figsize=(scale * num_filters, scale))
    plt.title(layer.name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='gray')
    plt.show()
```

In [43]:
```
display_feature_maps(59, X_images)
```




```
1/1 [==============================] - 0s 93ms/step
```



This demonstrates the CNN's ability to hierarchically extract and abstract features from raw pixel values.

As the image moves through the layers of the network, we can observe the transition from simple, low-level features to complex, high-level features that contribute to the network's understanding and classification of the image.

Model Training

The CNN model is trained on the training data for 100 epochs with a batch size of 128.

In [44]:
```
# batch_size = 64
batch_size = 128
epochs = 100

history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size)
```

```
Epoch 1/100
36/36 [==============================] - 2s 18ms/step - loss: 2.4017 - accuracy: 0.1
103
Epoch 2/100
36/36 [==============================] - 1s 27ms/step - loss: 2.0978 - accuracy: 0.2
302
Epoch 3/100
36/36 [==============================] - 1s 26ms/step - loss: 1.4897 - accuracy: 0.4
867
Epoch 4/100
36/36 [==============================] - 1s 27ms/step - loss: 1.0194 - accuracy: 0.6
601
Epoch 5/100
36/36 [==============================] - 1s 17ms/step - loss: 0.7202 - accuracy: 0.7
798
Epoch 6/100
36/36 [==============================] - 1s 18ms/step - loss: 0.5353 - accuracy: 0.8
294
Epoch 7/100
36/36 [==============================] - 1s 17ms/step - loss: 0.4058 - accuracy: 0.8
803
Epoch 8/100
36/36 [==============================] - 1s 17ms/step - loss: 0.3379 - accuracy: 0.8
979
Epoch 9/100
36/36 [==============================] - 1s 16ms/step - loss: 0.2872 - accuracy: 0.9
126
Epoch 10/100
36/36 [==============================] - 1s 16ms/step - loss: 0.2567 - accuracy: 0.9
228
Epoch 11/100
36/36 [==============================] - 1s 17ms/step - loss: 0.2427 - accuracy: 0.9
257
Epoch 12/100
36/36 [==============================] - 1s 16ms/step - loss: 0.2115 - accuracy: 0.9
388
Epoch 13/100
36/36 [==============================] - 1s 16ms/step - loss: 0.1697 - accuracy: 0.9
500
Epoch 14/100
36/36 [==============================] - 1s 16ms/step - loss: 0.1868 - accuracy: 0.9
448
Epoch 15/100
36/36 [==============================] - 1s 16ms/step - loss: 0.1625 - accuracy: 0.9
515
Epoch 16/100
36/36 [==============================] - 1s 17ms/step - loss: 0.1726 - accuracy: 0.9
515
Epoch 17/100
36/36 [==============================] - 1s 17ms/step - loss: 0.1524 - accuracy: 0.9
544
Epoch 18/100
36/36 [==============================] - 1s 17ms/step - loss: 0.1465 - accuracy: 0.9
564
Epoch 19/100
36/36 [==============================] - 1s 16ms/step - loss: 0.1323 - accuracy: 0.9
```

```
593
Epoch 20/100
36/36 [==============================] - 1s 16ms/step - loss: 0.1434 - accuracy: 0.9
589
Epoch 21/100
36/36 [==============================] - 1s 20ms/step - loss: 0.1395 - accuracy: 0.9
597
Epoch 22/100
36/36 [==============================] - 1s 27ms/step - loss: 0.1300 - accuracy: 0.9
617
Epoch 23/100
36/36 [==============================] - 1s 26ms/step - loss: 0.1261 - accuracy: 0.9
626
Epoch 24/100
36/36 [==============================] - 1s 26ms/step - loss: 0.1274 - accuracy: 0.9
613
Epoch 25/100
36/36 [==============================] - 1s 17ms/step - loss: 0.1159 - accuracy: 0.9
629
Epoch 26/100
36/36 [==============================] - 1s 17ms/step - loss: 0.1175 - accuracy: 0.9
644
Epoch 27/100
36/36 [==============================] - 1s 16ms/step - loss: 0.1139 - accuracy: 0.9
662
Epoch 28/100
36/36 [==============================] - 1s 16ms/step - loss: 0.1113 - accuracy: 0.9
677
Epoch 29/100
36/36 [==============================] - 1s 16ms/step - loss: 0.1163 - accuracy: 0.9
629
Epoch 30/100
36/36 [==============================] - 1s 17ms/step - loss: 0.1006 - accuracy: 0.9
695
Epoch 31/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0938 - accuracy: 0.9
713
Epoch 32/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0953 - accuracy: 0.9
695
Epoch 33/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0965 - accuracy: 0.9
726
Epoch 34/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0924 - accuracy: 0.9
724
Epoch 35/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0938 - accuracy: 0.9
724
Epoch 36/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0919 - accuracy: 0.9
731
Epoch 37/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0890 - accuracy: 0.9
713
Epoch 38/100
```

```
36/36 [==============================] - 1s 29ms/step - loss: 0.0883 - accuracy: 0.9
733
Epoch 39/100
36/36 [==============================] - 1s 22ms/step - loss: 0.0865 - accuracy: 0.9
740
Epoch 40/100
36/36 [==============================] - 1s 25ms/step - loss: 0.0856 - accuracy: 0.9
722
Epoch 41/100
36/36 [==============================] - 1s 27ms/step - loss: 0.0855 - accuracy: 0.9
735
Epoch 42/100
36/36 [==============================] - 1s 25ms/step - loss: 0.0838 - accuracy: 0.9
762
Epoch 43/100
36/36 [==============================] - 1s 22ms/step - loss: 0.0838 - accuracy: 0.9
755
Epoch 44/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0791 - accuracy: 0.9
760
Epoch 45/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0712 - accuracy: 0.9
786
Epoch 46/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0661 - accuracy: 0.9
784
Epoch 47/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0815 - accuracy: 0.9
751
Epoch 48/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0647 - accuracy: 0.9
773
Epoch 49/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0821 - accuracy: 0.9
764
Epoch 50/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0716 - accuracy: 0.9
764
Epoch 51/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0665 - accuracy: 0.9
775
Epoch 52/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0697 - accuracy: 0.9
771
Epoch 53/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0681 - accuracy: 0.9
791
Epoch 54/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0731 - accuracy: 0.9
778
Epoch 55/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0766 - accuracy: 0.9
773
Epoch 56/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0827 - accuracy: 0.9
766
```

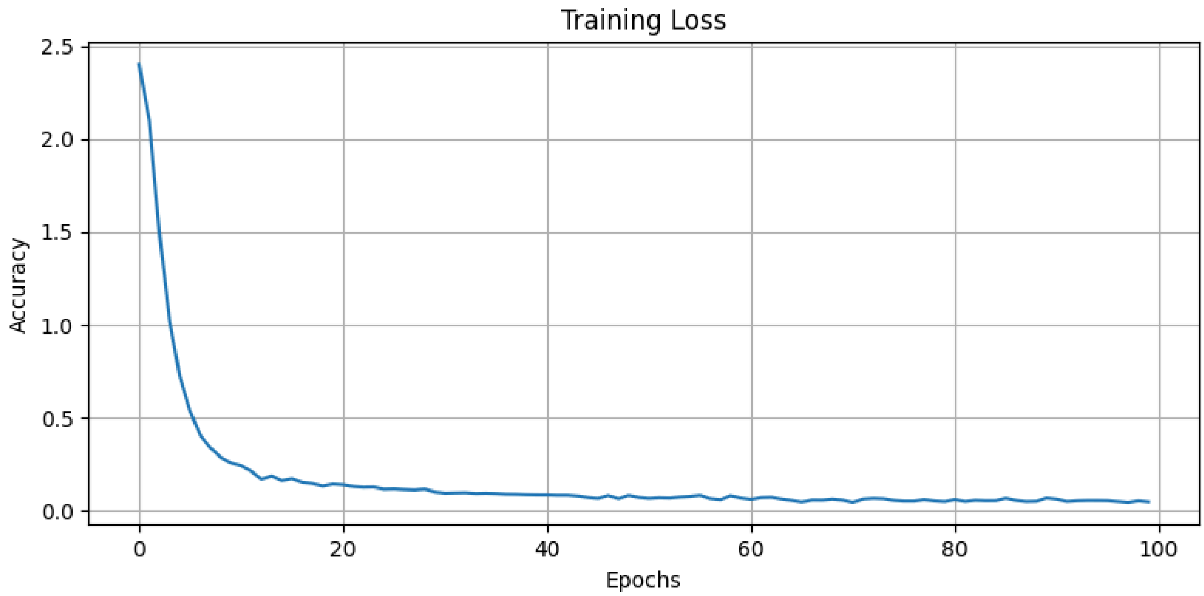
```
Epoch 57/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0650 - accuracy: 0.9
818
Epoch 58/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0586 - accuracy: 0.9
802
Epoch 59/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0805 - accuracy: 0.9
744
Epoch 60/100
36/36 [==============================] - 1s 26ms/step - loss: 0.0682 - accuracy: 0.9
793
Epoch 61/100
36/36 [==============================] - 1s 26ms/step - loss: 0.0607 - accuracy: 0.9
804
Epoch 62/100
36/36 [==============================] - 1s 25ms/step - loss: 0.0707 - accuracy: 0.9
778
Epoch 63/100
36/36 [==============================] - 1s 20ms/step - loss: 0.0722 - accuracy: 0.9
782
Epoch 64/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0624 - accuracy: 0.9
806
Epoch 65/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0549 - accuracy: 0.9
833
Epoch 66/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0462 - accuracy: 0.9
851
Epoch 67/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0572 - accuracy: 0.9
831
Epoch 68/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0567 - accuracy: 0.9
806
Epoch 69/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0618 - accuracy: 0.9
811
Epoch 70/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0564 - accuracy: 0.9
829
Epoch 71/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0440 - accuracy: 0.9
855
Epoch 72/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0625 - accuracy: 0.9
813
Epoch 73/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0664 - accuracy: 0.9
798
Epoch 74/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0643 - accuracy: 0.9
811
Epoch 75/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0553 - accuracy: 0.9
```

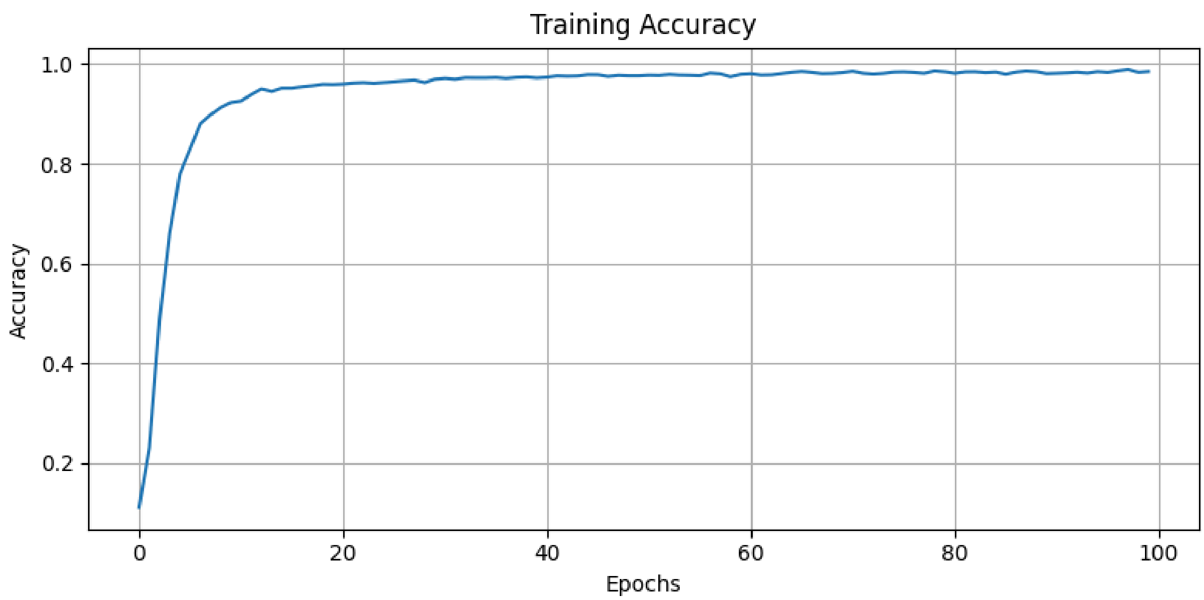
```
838
Epoch 76/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0515 - accuracy: 0.9
842
Epoch 77/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0515 - accuracy: 0.9
831
Epoch 78/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0592 - accuracy: 0.9
813
Epoch 79/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0523 - accuracy: 0.9
862
Epoch 80/100
36/36 [==============================] - 1s 22ms/step - loss: 0.0490 - accuracy: 0.9
844
Epoch 81/100
36/36 [==============================] - 1s 26ms/step - loss: 0.0595 - accuracy: 0.9
813
Epoch 82/100
36/36 [==============================] - 1s 26ms/step - loss: 0.0495 - accuracy: 0.9
842
Epoch 83/100
36/36 [==============================] - 1s 25ms/step - loss: 0.0558 - accuracy: 0.9
844
Epoch 84/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0536 - accuracy: 0.9
827
Epoch 85/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0539 - accuracy: 0.9
838
Epoch 86/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0672 - accuracy: 0.9
793
Epoch 87/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0551 - accuracy: 0.9
838
Epoch 88/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0489 - accuracy: 0.9
860
Epoch 89/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0504 - accuracy: 0.9
847
Epoch 90/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0683 - accuracy: 0.9
806
Epoch 91/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0616 - accuracy: 0.9
813
Epoch 92/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0495 - accuracy: 0.9
822
Epoch 93/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0533 - accuracy: 0.9
835
Epoch 94/100
```

```
36/36 [==============================] - 1s 16ms/step - loss: 0.0547 - accuracy: 0.9
818
Epoch 95/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0546 - accuracy: 0.9
847
Epoch 96/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0538 - accuracy: 0.9
831
Epoch 97/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0487 - accuracy: 0.9
864
Epoch 98/100
36/36 [==============================] - 1s 16ms/step - loss: 0.0434 - accuracy: 0.9
891
Epoch 99/100
36/36 [==============================] - 1s 17ms/step - loss: 0.0529 - accuracy: 0.9
833
Epoch 100/100
36/36 [==============================] - 1s 19ms/step - loss: 0.0469 - accuracy: 0.9
849
```

Visualizing loss and accuracy while training

In [45]:
```
train_loss = history.history['loss']
train_acc = history.history['accuracy']

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

plt.subplot(2, 1, 1)
plt.plot(train_loss, label='Training Loss')
plt.title('Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.grid()


plt.subplot(2, 1, 2)
plt.plot(train_acc, label='Training Accuracy')
plt.title('Training Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.grid()

plt.tight_layout()
plt.show()
```





Model Evaluation

Evaluating the CNN model on the test data.

The loss and accuracy metrics indicates good model performance with high accuracy and low loss on the test set.

In [46]:
```
test_loss, test_acc = model.evaluate(X_test, y_test)

print(f"Test Loss: {test_loss}")
print(f"Test accuracy: {test_acc}")
```

```
36/36 [==============================] - 0s 4ms/step - loss: 0.0322 - accuracy: 0.99
11
Test Loss: 0.03215990215539932
Test accuracy: 0.9911032319068909
```

In [47]:
```
y_pred = model.predict(X_test)
```

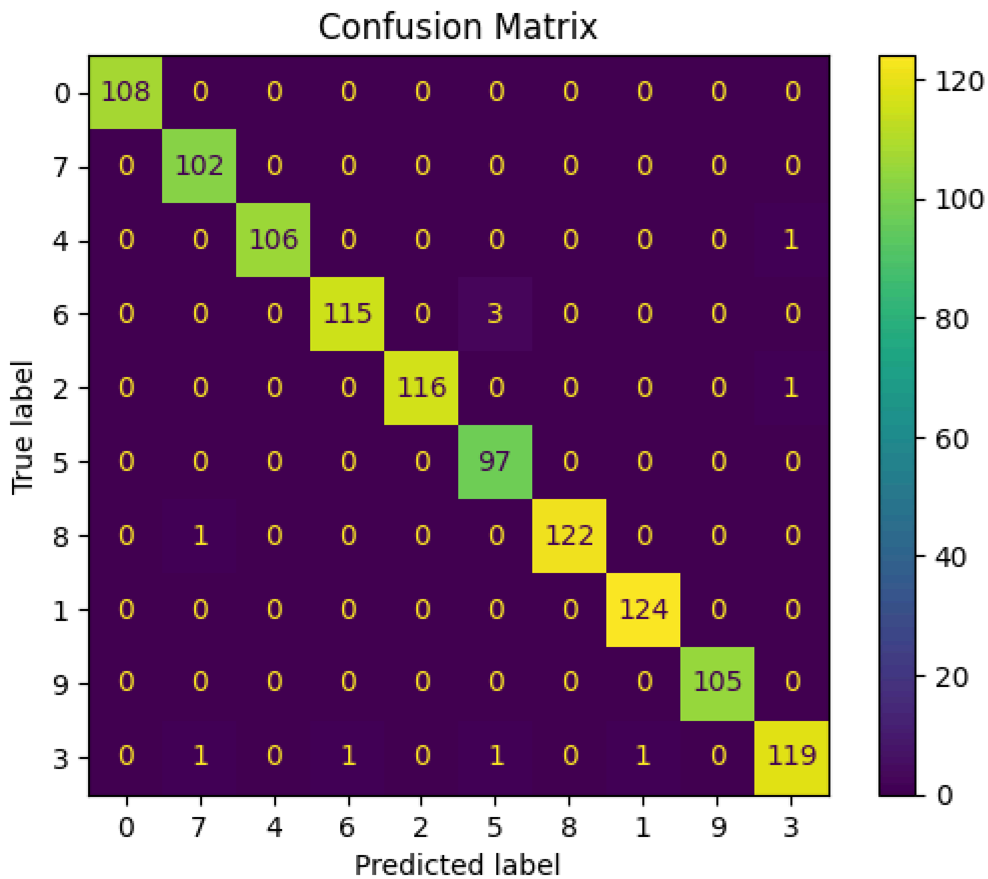
```
36/36 [==============================] - 0s 3ms/step
```

Other metrics are also calculated and visualized

The metrics show near perfect scores.

In [48]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.metrics import f1_score, confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import roc_curve, RocCurveDisplay, roc_auc_score

y_test_labels = np.argmax(y_test, axis=1)
y_pred_labels = np.argmax(y_pred, axis=1)

print(f"Accuracy: {accuracy_score(y_test_labels, y_pred_labels)*100:.2f}")
print(f"Precision: {precision_score(y_test_labels, y_pred_labels, average='macro')*
print(f"Recall: {recall_score(y_test_labels, y_pred_labels, average='macro')*100:.2
print(f"F1 Score: {f1_score(y_test_labels, y_pred_labels, average='macro')*100:.2f}

cm = confusion_matrix(y_test_labels, y_pred_labels)

disp = ConfusionMatrixDisplay(cm, display_labels=class_labels)
disp.plot()
disp.ax_.set_title('Confusion Matrix')
plt.show()
```

```
Accuracy: 99.11
Precision: 99.08
Recall: 99.16
F1 Score: 99.11
```



K-Fold Cross-Validation

Since the model shows near perfect metrics in all categories, validating these results with a proper cross-validation approach.

In [56]:
```
from sklearn.model_selection import KFold
```

K-Fold cross-validation for the CNN model from scratch.

- Folds: 5
- Initializes empty lists to store the loss and accuracy for each fold.
- Iterates over each fold, training a new model on the training set and evaluating it on the validation set.
- For each fold, it compiles a new instance of the CNN model with the Adam optimizer.
- Trains the model quietly (verbose=0 so it doesn't print out logs) for 100 epochs with a batch size of 128.
- Evaluates the model on the validation set and prints the fold number along with the validation results.

In [60]:
```
n_folds = 5
epochs = 100
batch_size = 128
```

```python
kfold = KFold(n_splits=n_folds, shuffle=True)

kfold_hist_loss, kfold_hist_acc = [], []

fold_count = 1
print(f'K-Fold Cross-Validation on the CNN model [{n_folds} Folds]')

for train_index, val_index in kfold.split(X_images, y):

  t_x, val_x = X_images[train_index], X_images[val_index]
  t_y, val_y = y.iloc[train_index], y.iloc[val_index]

  t_y = to_categorical(t_y, num_classes=10)
  val_y = to_categorical(val_y, num_classes=10)

  model_tmp = create_cnn_model()
  optimizer_tmp = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)
  model_tmp.compile(optimizer=optimizer_tmp, loss="categorical_crossentropy", metri

  print(f"\nFold {fold_count} - Training")
  model_tmp.fit(t_x, t_y, epochs=epochs, batch_size=batch_size, verbose=0)

  val_loss, val_acc = model_tmp.evaluate(val_x, val_y, verbose=0)
  print(f'Fold {fold_count} - Validation')
  kfold_hist_loss.append(val_loss)
  kfold_hist_acc.append(val_acc)

  fold_count += 1
```

```
K-Fold Cross-Validation on the CNN model [5 Folds]

Fold 1 - Training
Fold 1 - Validation

Fold 2 - Training
Fold 2 - Validation

Fold 3 - Training
Fold 3 - Validation

Fold 4 - Training
Fold 4 - Validation

Fold 5 - Training
Fold 5 - Validation
```

Results of the K-fold cross-validation

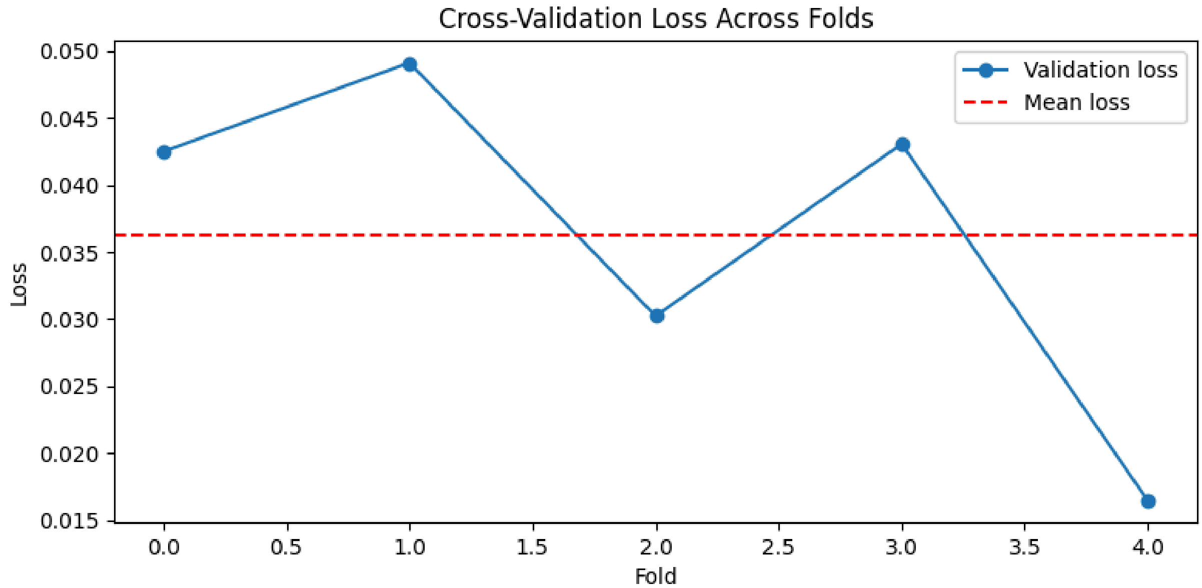
In [61]:

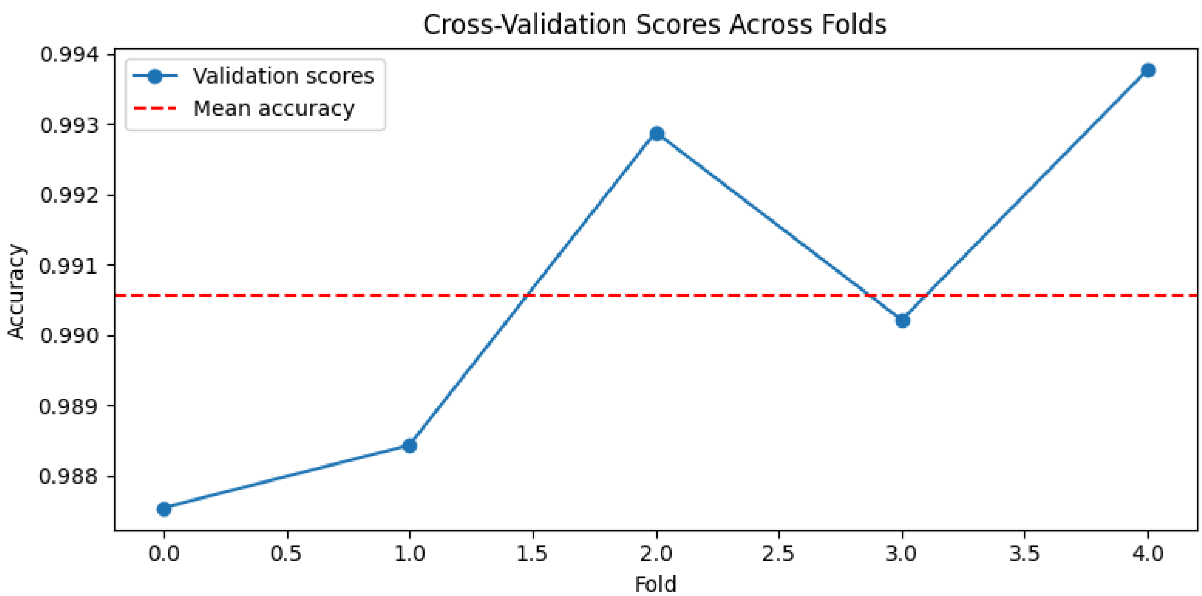
```python
print(f'Validation Loss and Accuracy across {n_folds} Folds.')
for i in range(n_folds):
  print(f'Fold {i+1}\tLoss: {kfold_hist_loss[i]}\tAccuracy: {kfold_hist_acc[i]}')

print(f'\nMean loss across {n_folds} folds: {np.mean(kfold_hist_loss)}')

print(f'\nMean cross-validation score: {np.mean(kfold_hist_acc)}')
print(f"Standard deviation of cross-validation scores: {np.std(kfold_hist_acc)}")
```

```
Validation Loss and Accuracy across 5 Folds.
Fold 1  Loss: 0.04249735549092293       Accuracy: 0.9875444769859314
Fold 2  Loss: 0.049108102917671204      Accuracy: 0.9884341359138489
Fold 3  Loss: 0.03023604303598404       Accuracy: 0.9928825497627258
Fold 4  Loss: 0.04305674880743027       Accuracy: 0.9902135133743286
Fold 5  Loss: 0.01646474562585354       Accuracy: 0.9937722682952881

Mean loss across 5 folds: 0.03627259917557239

Mean cross-validation score: 0.9905693888664245
Standard deviation of cross-validation scores: 0.0024267342001183744
```

Plotting the loss and accuracy of the CNN model across folds

In [62]:
```
train_loss = history.history['loss']
train_acc = history.history['accuracy']

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

plt.subplot(2, 1, 1)
plt.plot(kfold_hist_loss, label='Validation loss', marker='o')
plt.axhline(y=np.mean(kfold_hist_loss), color='r', linestyle='--', label='Mean loss
plt.xlabel('Fold')
plt.ylabel('Loss')
plt.title('Cross-Validation Loss Across Folds')
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(kfold_hist_acc, label='Validation scores', marker='o')
plt.axhline(y=np.mean(kfold_hist_acc), color='r', linestyle='--', label='Mean accur
plt.xlabel('Fold')
plt.ylabel('Accuracy')
plt.title('Cross-Validation Scores Across Folds')
plt.legend()

plt.tight_layout()
plt.show()
```





Further validation from K-Fold cross-validation with 5 folds confirmed the model's robustness and consistency across different subsets of data. Each fold was trained independently, and the model exhibited stable performance across all folds, validating its effectiveness and stability.

Conclusion

1. **High Accuracy:** The CNN achieved near-perfect accuracy on the test set and consistent results across validation folds, highlighting its capability to effectively recognize handwritten digits.
2. **Robust Model Design:** The use of dropout layers and careful architectural choices helped in mitigating overfitting, as evidenced by consistent performance during K-Fold cross-validation.

3. **Effective Feature Extraction:** Visualization of feature maps revealed that the model was effectively capturing relevant features at various layers, crucial for accurate classification.
4. **Generalization Capability:** The consistent performance across multiple folds during cross-validation suggests that the model is not overly fitted to the training data but rather generalizes well to new, unseen data.
5. **Potential Improvements:** While results are already excellent, exploring additional enhancements such as further hyperparameter tuning, advanced regularization techniques, or experimenting with deeper architectures might provide marginal gains.

In [28]: