Detailed Analysis and Implementation of a CNN on MNIST Dataset

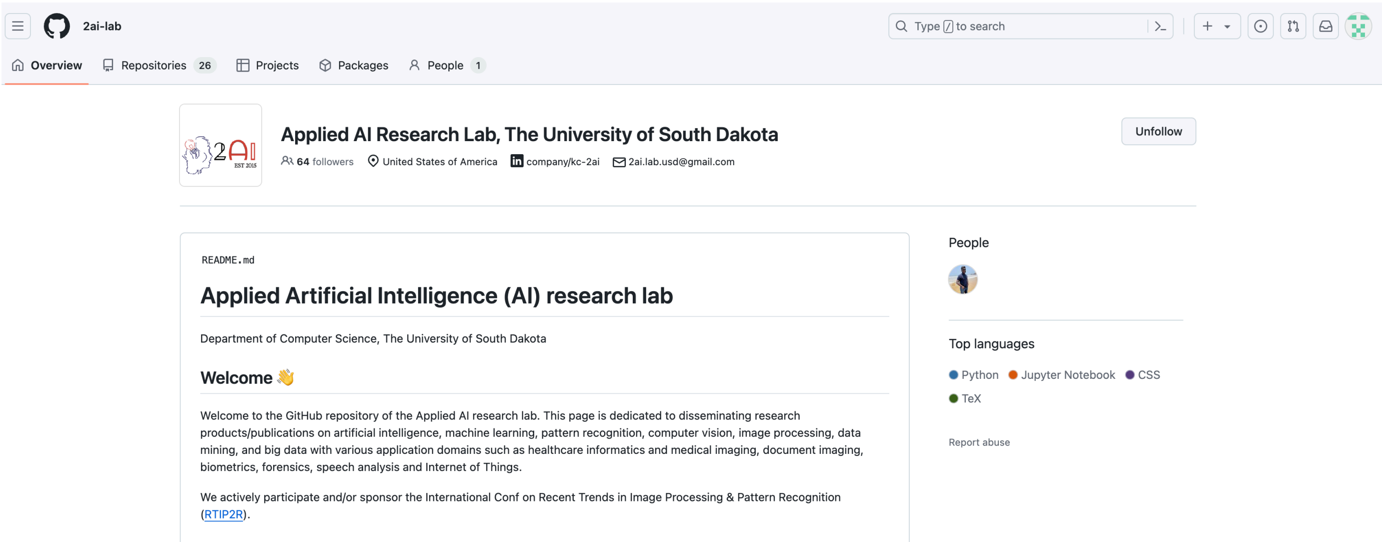
Team Members

Rahul Bhandari (Student ID: 101145564)

Abhaya Rawal (Student ID: 101145746)

Praveen Raghubanshi (Student ID: 101141342)

Screenshot of GitHub Following Applied AI



Data Loading and Initial Inspection

This section of the code handles the initial steps necessary to load and inspect the MNIST dataset specifically designed for optical recognition of handwritten digits. Here's a breakdown of what each line accomplishes:

Importing the Library: The fetch_ucirepo function from the ucimlrepo library is imported. This function is used to fetch datasets from the UCI repository directly.

Fetching the Dataset: The dataset with ID 80, which corresponds to the optical recognition of handwritten digits, is loaded from the UCI repository. This dataset includes images of handwritten digits along with their corresponding labels.

Data Structure: The fetched data is stored in two parts:

features: This contains the image data (handwritten digits), typically stored as pandas dataframes where each row represents an image and each column represents a pixel in the image.

targets: This contains the labels for the images, where each label denotes the digit that the image represents.

Printing Metadata and Variables: The metadata associated with the dataset is printed, providing details such as the number of samples, feature names, and other relevant information. Additionally, the structure and type of data variables are displayed to understand the dataset's layout better.

In [1]:
```
from ucimlrepo import fetch_ucirepo

# Fetch dataset
handwritten_digits = fetch_ucirepo(id=80)

# Data (as pandas dataframes)
features = handwritten_digits.data.features
targets = handwritten_digits.data.targets

# Variable information
print(handwritten_digits.metadata)
print(handwritten_digits.variables)

# Fetch the MNIST handwritten digits dataset from UCI repository and load it into variables for further processing.
```

{'uci_id': 80, 'name': 'Optical Recognition of Handwritten Digits', 'repository_url': 'https://archive.ics.uci.edu/dataset/80/optical+recognition+of+handwritten+digits', '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': 5620, 'num_features': 64, 'feature_types': ['Integer'], 'demographics': [], 'target_col': ['class'], 'index_col': None, 'has_missing_values': 'no', 'missing_values_symbol': None, 'year_of_dataset_creation': 1998, 'last_updated': 'Wed Aug 23 2023', 'dataset_doi': '10.24432/C50P49', 'creators': ['E. Alpaydin', 'C. Kaynak'], 'intro_paper': {'title': 'Methods of Combining Multiple Classifiers and Their Applications to Handwritten Digit Recognition', 'authors': 'C. Kaynak', 'published_in': 'MSc Thesis, Institute of Graduate Studies in Science and Engineering, Bogazici University', 'year': 1995, 'url': None, 'doi': None}, 'additional_info': {'summary': 'We used preprocessing 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 training set and different 13 to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of 4x4 and the number of on pixels are counted in each block. This generates an input matrix of 8x8 where each element is an integer in the range 0..16. This 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. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C. L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469, 1994.', 'purpose': None, 'funded_by': None, 'instances_represent': None, 'recommended_data_splits': None, 'sensitive_data': None, 'preprocessing_description': None, 'variable_info': 'All input attributes are integers 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]
```

Data Preprocessing and Splitting

This code segment reshapes the digit images into a consistent format and splits the dataset into training and testing subsets to ensure robust model evaluation.

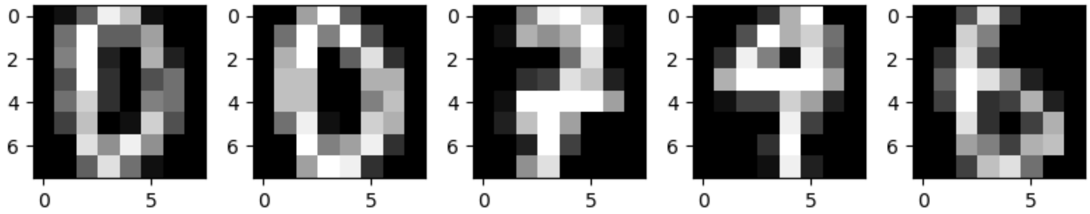
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

def get_index_sample_each_class(y_train):
    unique_classes = np.unique(y_train)
    sample_indices = []
    for cls in unique_classes:
        class_indices = np.where(y_train == cls)[0]
        sample_index = np.random.choice(class_indices)
        sample_indices.append(sample_index)
    return sample_indices

from sklearn.model_selection import train_test_split
X = features.to_numpy().reshape(-1, 8, 8)
# Split data
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    targets,
                                                    test_size=0.25,  # Increased test size for more robust testing
                                                    random_state=100,  # Changed random state
                                                    stratify=targets)  # Stratify to maintain distribution

print(f'Train and test data shapes after split: X_train: {X_train.shape}, X_test: {X_test.shape}')


# Split the data into training and test sets to prepare for model training. A test size of 25% provides a substantial amount of 
```

```
Train and test data shapes after split: X_train: (4215, 8, 8), X_test: (1405, 8, 8)
```

Visualizing Sample Digits

This script visualizes the first five samples from the dataset, allowing for a quick check of data integrity and format directly within the images.

In [3]:
```
idx = [0,1,2,3,4]
# Plotting the samples
plt.figure(figsize=(8, 8))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.imshow(X[idx[i]], cmap='gray', interpolation='none')
#     plt.title(f'Class: {targets[idx[i]]}')
# plt.suptitle('Random image samples of each digit class from MNIST Digits dataset', fontsize=18)
plt.tight_layout()
plt.show()
# Visualize some samples from the dataset to confirm data integrity and understand the data's format.
```



CNN Model Definition and Compilation

This code defines and compiles a convolutional neural network (CNN) with layers for feature extraction and classification, optimized for digit recognition from 8x8 images.

In [4]:
```
from keras.models import Sequential
from keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense
from keras.optimizers import Adam
from keras.utils import to_categorical

def create_cnn_model():
    model = Sequential([
        Input(shape=(8, 8, 1)),  # Use Input layer for specifying input shape
        Conv2D(32, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(64, activation='relu'),
        Dense(10, activation='softmax')  # Output layer with 10 classes
    ])
    return model

model = create_cnn_model()
model.compile(optimizer=Adam(learning_rate=0.0005),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Define a sequential CNN model architecture with convolutional, pooling, and dense layers. Additionally, set up K-Fold cross-val
```

Encoding the target labels into a one-hot format and trains the CNN model on the training data, while also validating it against the test set to monitor performance across epochs.

In [5]:
```
# Assuming y_train and y_test are initially integer labels for classes
y_train_encoded = to_categorical(y_train, num_classes=10)
y_test_encoded = to_categorical(y_test, num_classes=10)

# Fit model with one-hot encoded targets
history = model.fit(X_train, y_train_encoded, epochs=50, batch_size=32, validation_data=(X_test, y_test_encoded))
```

```
Epoch 1/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.4601 - loss: 2.0160 - val_accuracy: 0.8982 - val_loss: 0.3817
Epoch 2/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 884us/step - accuracy: 0.9271 - loss: 0.2820 - val_accuracy: 0.9395 - val_loss: 0.2053
Epoch 3/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 834us/step - accuracy: 0.9606 - loss: 0.1508 - val_accuracy: 0.9601 - val_loss: 0.1502
Epoch 4/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 831us/step - accuracy: 0.9695 - loss: 0.1188 - val_accuracy: 0.9680 - val_loss: 0.1183
Epoch 5/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 884us/step - accuracy: 0.9790 - loss: 0.0850 - val_accuracy: 0.9673 - val_loss: 0.1086
Epoch 6/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 889us/step - accuracy: 0.9794 - loss: 0.0761 - val_accuracy: 0.9744 - val_loss: 0.0953
Epoch 7/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 894us/step - accuracy: 0.9850 - loss: 0.0603 - val_accuracy: 0.9772 - val_loss: 0.0853
Epoch 8/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 873us/step - accuracy: 0.9807 - loss: 0.0616 - val_accuracy: 0.9680 - val_loss: 0.1037
Epoch 9/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 880us/step - accuracy: 0.9841 - loss: 0.0536 - val_accuracy: 0.9801 - val_loss: 0.0752
Epoch 10/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 842us/step - accuracy: 0.9848 - loss: 0.0480 - val_accuracy: 0.9843 - val_loss: 0.0645
Epoch 11/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 870us/step - accuracy: 0.9874 - loss: 0.0426 - val_accuracy: 0.9836 - val_loss: 0.0585
Epoch 12/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 922us/step - accuracy: 0.9892 - loss: 0.0339 - val_accuracy: 0.9701 - val_loss: 0.0825
Epoch 13/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 815us/step - accuracy: 0.9916 - loss: 0.0287 - val_accuracy: 0.9822 - val_loss: 0.0527
Epoch 14/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 851us/step - accuracy: 0.9947 - loss: 0.0246 - val_accuracy: 0.9786 - val_loss: 0.0799
Epoch 15/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 924us/step - accuracy: 0.9906 - loss: 0.0287 - val_accuracy: 0.9836 - val_loss: 0.0555
Epoch 16/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 885us/step - accuracy: 0.9942 - loss: 0.0207 - val_accuracy: 0.9851 - val_loss: 0.0562
Epoch 17/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9966 - loss: 0.0181 - val_accuracy: 0.9829 - val_loss: 0.0571
Epoch 18/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9939 - loss: 0.0223 - val_accuracy: 0.9765 - val_loss: 0.0665
Epoch 19/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9943 - loss: 0.0186 - val_accuracy: 0.9815 - val_loss: 0.0506
Epoch 20/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 963us/step - accuracy: 0.9971 - loss: 0.0132 - val_accuracy: 0.9843 - val_loss: 0.0474
Epoch 21/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 828us/step - accuracy: 0.9978 - loss: 0.0105 - val_accuracy: 0.9851 - val_loss: 0.0485
Epoch 22/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 849us/step - accuracy: 0.9958 - loss: 0.0142 - val_accuracy: 0.9843 - val_loss: 0.0552
Epoch 23/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 853us/step - accuracy: 0.9987 - loss: 0.0105 - val_accuracy: 0.9865 - val_loss: 0.0504
Epoch 24/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 854us/step - accuracy: 0.9987 - loss: 0.0103 - val_accuracy: 0.9836 - val_loss: 0.0533
Epoch 25/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 788us/step - accuracy: 0.9998 - loss: 0.0055 - val_accuracy: 0.9829 - val_loss: 0.0578
Epoch 26/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 826us/step - accuracy: 0.9990 - loss: 0.0076 - val_accuracy: 0.9865 - val_loss: 0.0489
Epoch 27/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 839us/step - accuracy: 0.9993 - loss: 0.0057 - val_accuracy: 0.9822 - val_loss: 0.0615
Epoch 28/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 825us/step - accuracy: 0.9949 - loss: 0.0159 - val_accuracy: 0.9829 - val_loss: 0.0561
Epoch 29/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 908us/step - accuracy: 0.9983 - loss: 0.0077 - val_accuracy: 0.9865 - val_loss: 0.0560
Epoch 30/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 908us/step - accuracy: 0.9995 - loss: 0.0056 - val_accuracy: 0.9865 - val_loss: 0.0532
Epoch 31/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 893us/step - accuracy: 0.9966 - loss: 0.0121 - val_accuracy: 0.9858 - val_loss: 0.0561
Epoch 32/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 899us/step - accuracy: 0.9992 - loss: 0.0044 - val_accuracy: 0.9879 - val_loss: 0.0456
Epoch 33/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 922us/step - accuracy: 0.9999 - loss: 0.0029 - val_accuracy: 0.9886 - val_loss: 0.0470
Epoch 34/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9990 - loss: 0.0054 - val_accuracy: 0.9794 - val_loss: 0.0680
Epoch 35/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 889us/step - accuracy: 0.9993 - loss: 0.0062 - val_accuracy: 0.9843 - val_loss: 0.0538
Epoch 36/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 906us/step - accuracy: 0.9983 - loss: 0.0041 - val_accuracy: 0.9865 - val_loss: 0.0530
Epoch 37/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 871us/step - accuracy: 0.9993 - loss: 0.0025 - val_accuracy: 0.9872 - val_loss: 0.0435
Epoch 38/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 911us/step - accuracy: 1.0000 - loss: 0.0017 - val_accuracy: 0.9851 - val_loss: 0.0555
Epoch 39/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 834us/step - accuracy: 1.0000 - loss: 0.0021 - val_accuracy: 0.9872 - val_loss: 0.0455
Epoch 40/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 878us/step - accuracy: 1.0000 - loss: 0.0014 - val_accuracy: 0.9893 - val_loss: 0.0426
Epoch 41/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 852us/step - accuracy: 1.0000 - loss: 0.0011 - val_accuracy: 0.9886 - val_loss: 0.0490
Epoch 42/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 952us/step - accuracy: 1.0000 - loss: 0.0014 - val_accuracy: 0.9815 - val_loss: 0.0607
Epoch 43/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 888us/step - accuracy: 0.9986 - loss: 0.0060 - val_accuracy: 0.9779 - val_loss: 0.0869
Epoch 44/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 876us/step - accuracy: 0.9959 - loss: 0.0140 - val_accuracy: 0.9822 - val_loss: 0.0636
Epoch 45/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 873us/step - accuracy: 0.9987 - loss: 0.0048 - val_accuracy: 0.9843 - val_loss: 0.0583
Epoch 46/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 905us/step - accuracy: 1.0000 - loss: 0.0014 - val_accuracy: 0.9886 - val_loss: 0.0457
Epoch 47/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 918us/step - accuracy: 1.0000 - loss: 0.0013 - val_accuracy: 0.9872 - val_loss: 0.0506
Epoch 48/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 873us/step - accuracy: 1.0000 - loss: 9.6284e-04 - val_accuracy: 0.9886 - val_loss: 0.0459
Epoch 49/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 893us/step - accuracy: 1.0000 - loss: 5.6427e-04 - val_accuracy: 0.9893 - val_loss: 0.0448
```

```
Epoch 50/50
132/132 ━━━━━━━━━━━━━━━━━━━━ 0s 917us/step - accuracy: 1.0000 - loss: 5.1256e-04 - val_accuracy: 0.9886 - val_loss: 0.0451
```

In [6]:
```
X_train = X_train.reshape((-1, 8, 8, 1))
X_test = X_test.reshape((-1, 8, 8, 1))
```

Implementing K-Fold Cross-Validation in CNN Training

In this section of the code, we implement a rigorous approach to validating our convolutional neural network (CNN) model using K-fold cross-validation. This method is particularly effective in ensuring that our model's performance is not only good on a single test/train split but generalizes well across various subsets of the data.

Key Aspects of the Implementation:

Data Preparation: Before proceeding with cross-validation, we ensure that both X_train and X_test are reshaped correctly to fit the model's input requirements. Also, y_train and y_test are converted from categorical labels into a one-hot encoded format to match the output layer of the CNN.

K-Fold Setup: We utilize the KFold class from sklearn.model_selection to set up a 5-fold cross-validation. This splits the training data into 5 subsets, where each subset gets a turn at being the validation set.

Model Training and Validation: For each fold, a new instance of the CNN model is created and compiled. The model is then trained on the training subset and validated on the validation subset. This process is repeated for each fold, ensuring that each subset of the data is used for validation exactly once.

Evaluation: After training, the model's performance on the validation set is assessed using the loss and accuracy metrics, which provide insights into how well the model is likely to perform on unseen data.

In [9]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import numpy as np
from sklearn.model_selection import KFold
from keras.models import Sequential
from keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense
from keras.optimizers import Adam
from keras.utils import to_categorical
# Convert y_train and y_test to numpy arrays if they are pandas Series or DataFrames

y_train_encoded = to_categorical(y_train, num_classes=10)
y_test_encoded = to_categorical(y_test, num_classes=10)

# Ensure X_train and X_test are numpy arrays with the correct shape
X_train = X_train.reshape((-1, 8, 8, 1))
X_test = X_test.reshape((-1, 8, 8, 1))

# Convert to numpy arrays explicitly if still in DataFrame or Series format
if isinstance(y_train_encoded, pd.DataFrame) or isinstance(y_train_encoded, pd.Series):
    y_train_encoded = y_train_encoded.values
if isinstance(y_test_encoded, pd.DataFrame) or isinstance(y_test_encoded, pd.Series):
    y_test_encoded = y_test_encoded.values

# Proceed with k-fold cross-validation
n_folds = 5
kfold = KFold(n_splits=n_folds, shuffle=True, random_state=42)
fold_count = 1

for train_index, val_index in kfold.split(X_train):
    train_X, val_X = X_train[train_index], X_train[val_index]
    train_y, val_y = y_train_encoded[train_index], y_train_encoded[val_index]

    model_kfold = create_cnn_model()
    model_kfold.compile(optimizer=Adam(learning_rate=0.0005),
                        loss='categorical_crossentropy',
                        metrics=['accuracy'])

    print(f'Training Fold {fold_count}')
    model_kfold.fit(train_X, train_y, epochs=30, batch_size=40, verbose=0)

    val_loss, val_acc = model_kfold.evaluate(val_X, val_y, verbose=0)
    print(f'Validation results - Loss: {val_loss}, Accuracy: {val_acc}')

    fold_count += 1

# Define a sequential CNN model architecture with convolutional, pooling, and dense layers. Additionally, set up K-Fold cross-va
```

```
Training Fold 1
Validation results - Loss: 0.07176577299833298, Accuracy: 0.9810201525688171
Training Fold 2
Validation results - Loss: 0.07265524566173553, Accuracy: 0.9810201525688171
Training Fold 3
Validation results - Loss: 0.027115317061543465, Accuracy: 0.9928825497627258
Training Fold 4
Validation results - Loss: 0.029878173023462296, Accuracy: 0.9893238544464111
Training Fold 5
Validation results - Loss: 0.033021606504917145, Accuracy: 0.9893238544464111
```

This code evaluates a trained model on validation data by calculating key performance metrics: accuracy, precision, recall, and F1 score, helping to assess its effectiveness in classifying handwritten digits.

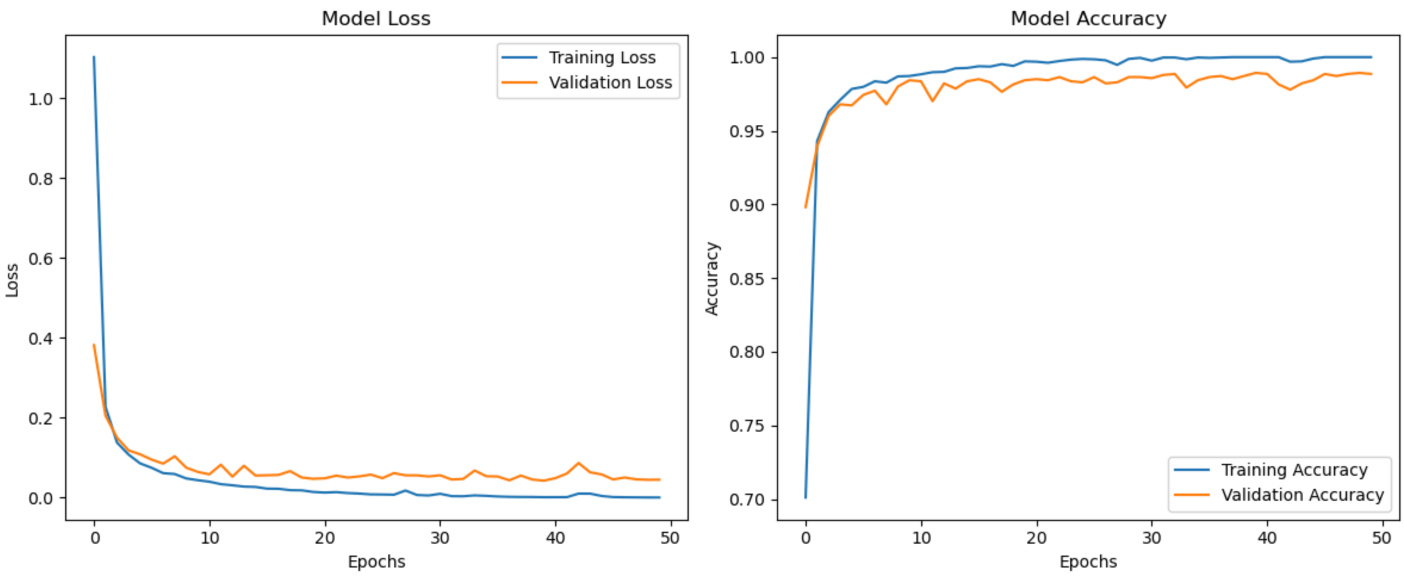
In [10]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report
# Assuming `val_X` and `val_y` are your validation datasets
```

```python
# Make predictions
predictions = model_kfold.predict(val_X)
predicted_classes = np.argmax(predictions, axis=1)  # Convert softmax outputs to class predictions
true_classes = np.argmax(val_y, axis=1)  # Assuming val_y is one-hot encoded

# Calculate metrics
accuracy = accuracy_score(true_classes, predicted_classes)
precision = precision_score(true_classes, predicted_classes, average='macro')  # Use 'micro' or 'weighted' based on needs
recall = recall_score(true_classes, predicted_classes, average='macro')
f1 = f1_score(true_classes, predicted_classes, average='macro')

# Print metrics
print(f'Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
```

```
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 895us/step
Accuracy: 0.9893238434163701
Precision: 0.9898802554598152
Recall: 0.9892815549045728
F1 Score: 0.989510243459686
```

This script visualizes the training and validation loss and accuracy over epochs, providing insights into the model's learning process and highlighting any trends in overfitting or underfitting.

In [12]:
```python
import matplotlib.pyplot as plt

# Assuming `history` is the return value from model.fit()

# Plotting training and validation loss
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
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()

# Plotting training and validation accuracy
plt.subplot(1, 2, 2)
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()

plt.tight_layout()
plt.show()

# Plot training and validation metrics to assess the model's performance. This helps in visualizing overfitting and underfitting
```



CNN Model Training and Evaluation with K-Fold Cross-Validation

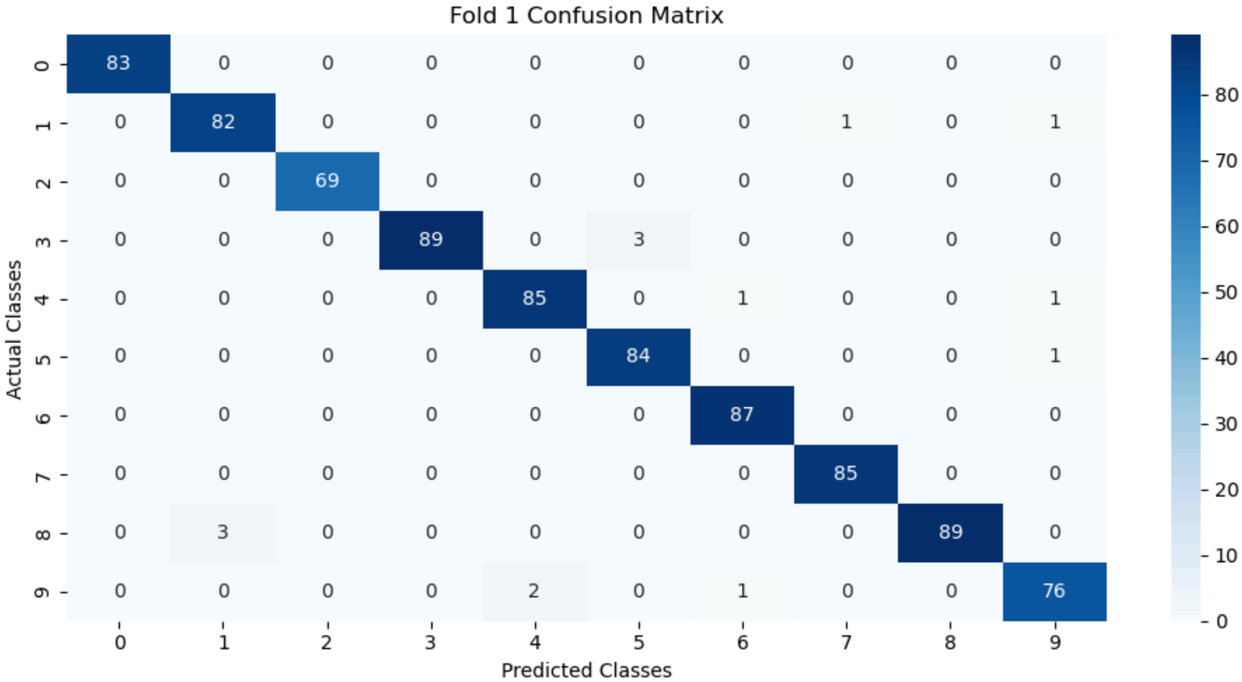
Model Training: This code defines and trains a convolutional neural network using 5-fold cross-validation to ensure robustness by evaluating model performance across different subsets of the training data.

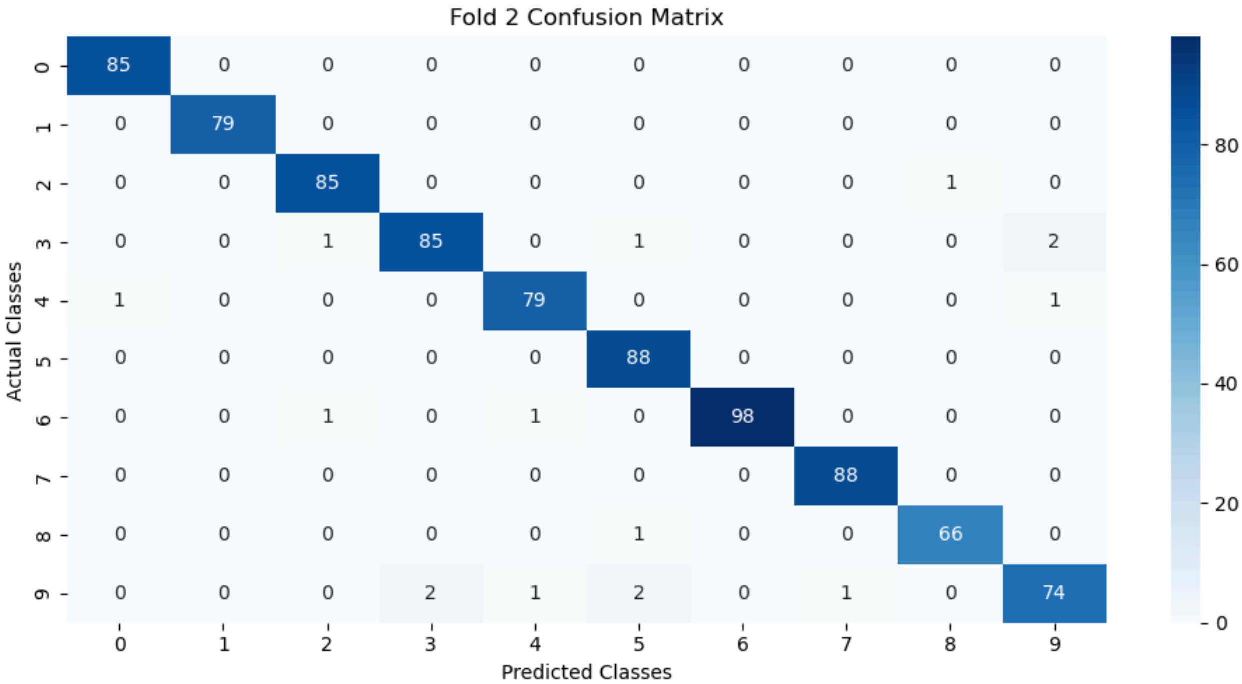
Confusion Matrix Visualization: After training, confusion matrices are plotted for each fold to visually assess the model's classification accuracy across different classes, highlighting potential areas for improvement.

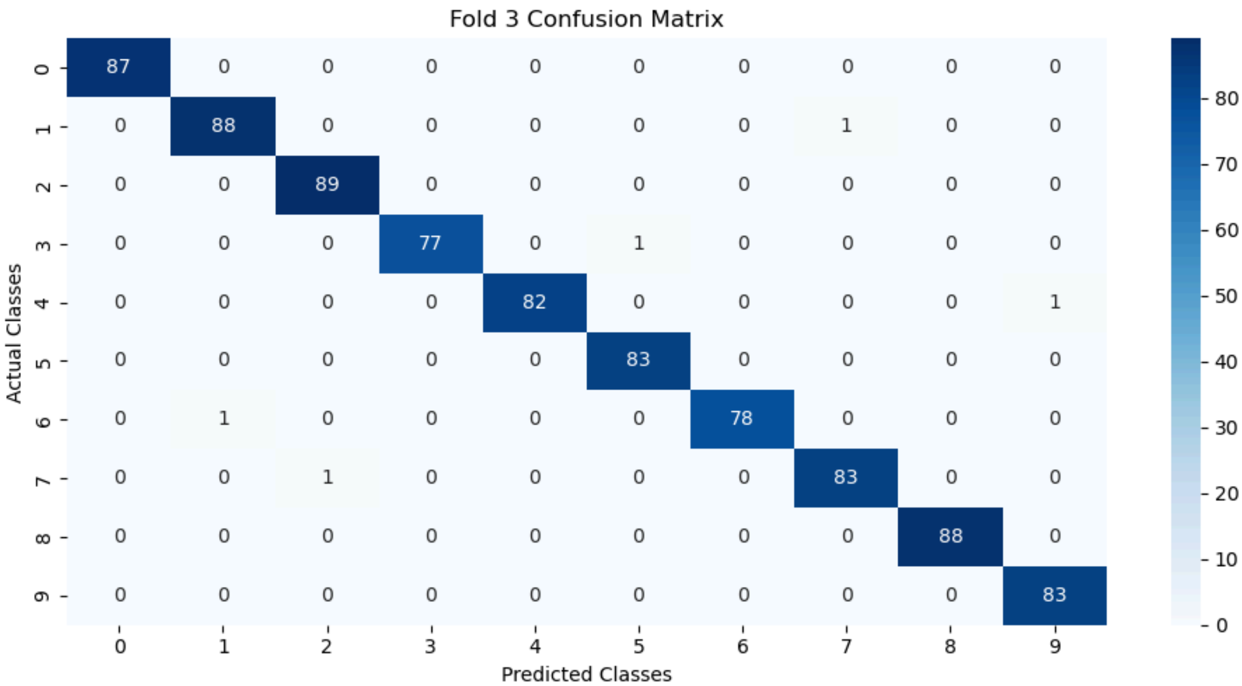
In [13]:
```python
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import numpy as np
from sklearn.model_selection import KFold
from keras.models import Sequential
from keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense
```

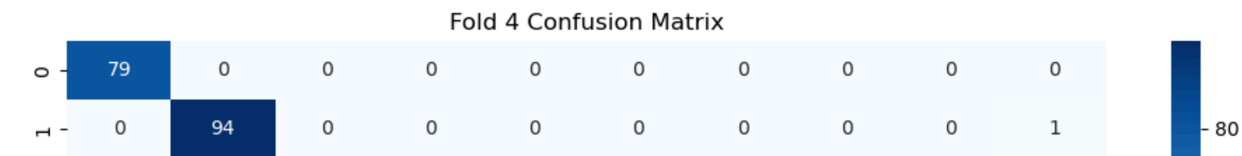
```python
from keras.optimizers import Adam
from keras.utils import to_categorical

def create_cnn_model():
    model = Sequential([
        Input(shape=(8, 8, 1)),
        Conv2D(32, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(64, activation='relu'),
        Dense(10, activation='softmax')
    ])
    return model

n_folds = 5
kfold = KFold(n_splits=n_folds, shuffle=True, random_state=42)
fold_no = 1

# Prepare figure for plotting
fig, axes = plt.subplots(n_folds, 1, figsize=(10, 5 * n_folds))

for train_index, val_index in kfold.split(X_train):
    train_X, val_X = X_train[train_index], X_train[val_index]
    train_y, val_y = y_train_encoded[train_index], y_train_encoded[val_index]

    model_kfold = create_cnn_model()
    model_kfold.compile(optimizer=Adam(learning_rate=0.0005),
                        loss='categorical_crossentropy',
                        metrics=['accuracy'])
    model_kfold.fit(train_X, train_y, epochs=30, batch_size=40, verbose=0)

    # Predictions and Confusion Matrix
    predictions = model_kfold.predict(val_X)
    predicted_classes = np.argmax(predictions, axis=1)
    true_classes = np.argmax(val_y, axis=1)
    cm = confusion_matrix(true_classes, predicted_classes)

    # Plotting
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=axes[fold_no - 1])
    axes[fold_no - 1].set_title(f'Fold {fold_no} Confusion Matrix')
    axes[fold_no - 1].set_xlabel('Predicted Classes')
    axes[fold_no - 1].set_ylabel('Actual Classes')

    fold_no += 1

plt.tight_layout()
plt.show()

# Define a sequential CNN model architecture with convolutional, pooling, and dense layers. Additionally, set up K-Fold cross-va
```

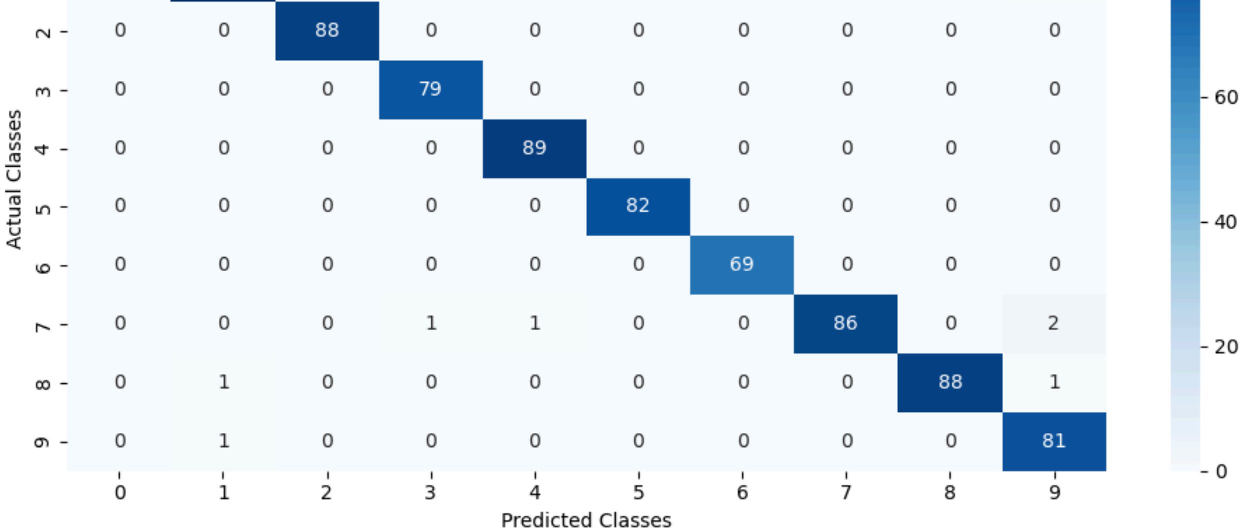
```
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 909us/step
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 889us/step
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 955us/step
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 888us/step
```

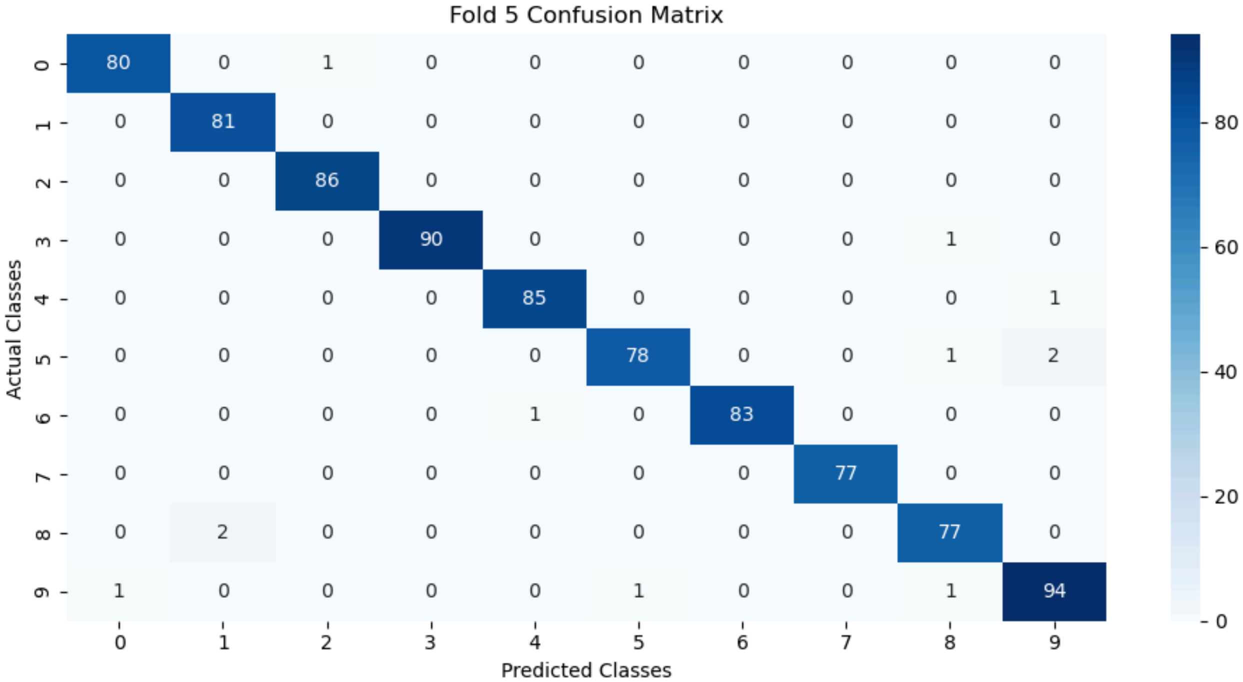

Fold 1 Confusion Matrix
Actual Classes
Predicted Classes
0: 83 0 0 0 0 0 0 0 0 0
1: 0 82 0 0 0 0 0 1 0 1
2: 0 0 69 0 0 0 0 0 0 0
3: 0 0 0 89 0 3 0 0 0 0
4: 0 0 0 0 85 0 1 0 0 1
5: 0 0 0 0 0 84 0 0 0 1
6: 0 0 0 0 0 0 87 0 0 0
7: 0 0 0 0 0 0 0 85 0 0
8: 0 3 0 0 0 0 0 0 89 0
9: 0 0 0 0 2 0 1 0 0 76
0 1 2 3 4 5 6 7 8 9
80 70 60 50 40 30 20 10 0


Fold 2 Confusion Matrix
Actual Classes
Predicted Classes
0: 85 0 0 0 0 0 0 0 0 0
1: 0 79 0 0 0 0 0 0 0 0
2: 0 0 85 0 0 0 0 0 1 0
3: 0 0 1 85 0 1 0 0 0 2
4: 1 0 0 0 79 0 0 0 0 1
5: 0 0 0 0 0 88 0 0 0 0
6: 0 0 1 0 1 0 98 0 0 0
7: 0 0 0 0 0 0 0 88 0 0
8: 0 0 0 0 0 1 0 0 66 0
9: 0 0 0 2 1 2 0 1 0 74
0 1 2 3 4 5 6 7 8 9
80 60 40 20 0


Fold 3 Confusion Matrix
Actual Classes
Predicted Classes
0: 87 0 0 0 0 0 0 0 0 0
1: 0 88 0 0 0 0 0 1 0 0
2: 0 0 89 0 0 0 0 0 0 0
3: 0 0 0 77 0 1 0 0 0 0
4: 0 0 0 0 82 0 0 0 0 1
5: 0 0 0 0 0 83 0 0 0 0
6: 0 1 0 0 0 0 78 0 0 0
7: 0 0 1 0 0 0 0 83 0 0
8: 0 0 0 0 0 0 0 0 88 0
9: 0 0 0 0 0 0 0 0 0 83
0 1 2 3 4 5 6 7 8 9
80 70 60 50 40 30 20 10 0


Fold 4 Confusion Matrix
0: 79 0 0 0 0 0 0 0 0 0
1: 0 94 0 0 0 0 0 0 0 1
80





Performance Tracking: It gathers key metrics such as accuracy and loss for both training and validation phases for each fold, providing comprehensive insight into the model's behavior and effectiveness.

In [14]:
```
from sklearn.model_selection import KFold
from keras.models import Sequential
from keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense
from keras.optimizers import Adam
from keras.utils import to_categorical

def create_cnn_model():
    model = Sequential([
        Input(shape=(8, 8, 1)),
        Conv2D(32, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(64, activation='relu'),
        Dense(10, activation='softmax')
    ])
    return model

n_folds = 5
kfold = KFold(n_splits=n_folds, shuffle=True, random_state=42)
fold_no = 1
fold_accuracies = []
fold_val_accuracies = []
fold_losses = []
fold_val_losses = []

for train_index, val_index in kfold.split(X_train):
    train_X, val_X = X_train[train_index], X_train[val_index]
    train_y, val_y = y_train_encoded[train_index], y_train_encoded[val_index]

    model_kfold = create_cnn_model()
    model_kfold.compile(optimizer=Adam(learning_rate=0.0005),
                        loss='categorical_crossentropy',
                        metrics=['accuracy'])
    history = model_kfold.fit(train_X, train_y, epochs=30, batch_size=40, verbose=0, validation_data=(val_X, val_y))

    # Collecting metrics
    fold_accuracies.append(history.history['accuracy'])
    fold_val_accuracies.append(history.history['val_accuracy'])
```

```
    fold_losses.append(history.history['loss'])
    fold_val_losses.append(history.history['val_loss'])

    fold_no += 1

# Define a sequential CNN model architecture with convolutional, pooling, and dense layers. Additionally, set up K-Fold cross-va
```

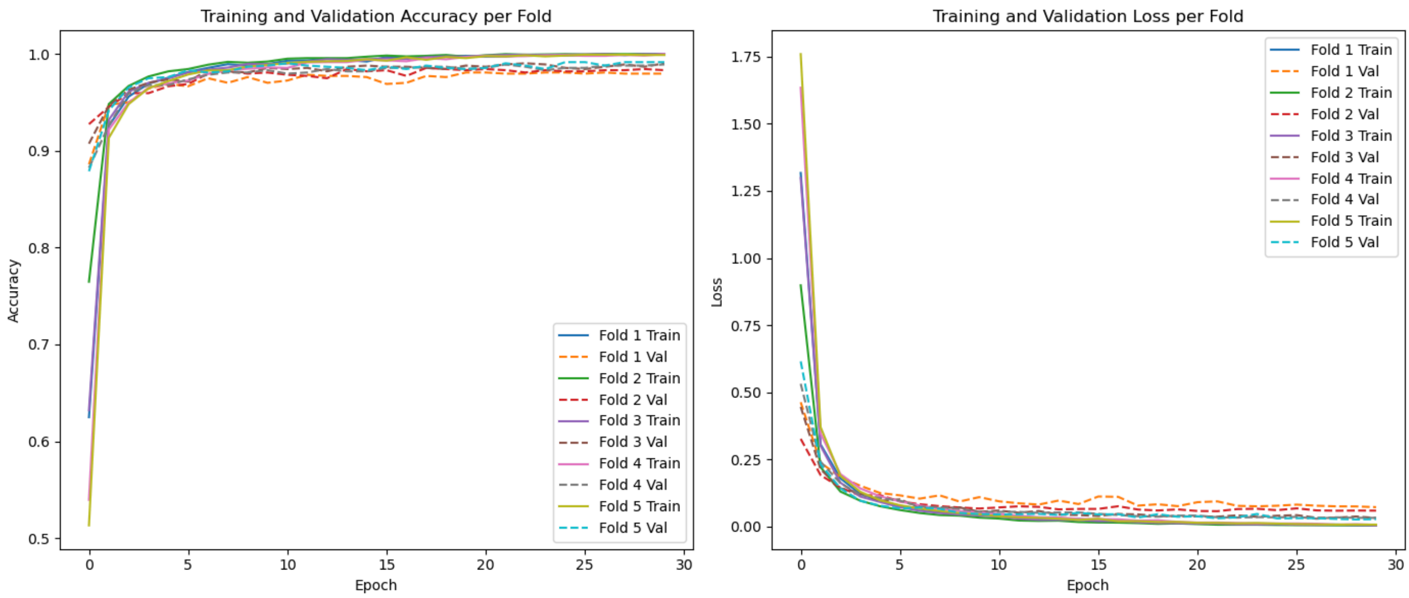
Visualizing Training and Validation Metrics

Accuracy and Loss Plots: This code visualizes the training and validation accuracy and loss for each fold of the cross-validation process, offering a clear view of model performance dynamics over training epochs.

Assessment of Model Fit: Through these plots, it's possible to identify trends in overfitting or underfitting, guiding potential adjustments in model training or architecture.

In [15]:

```
import matplotlib.pyplot as plt

# Plotting cross-validation metrics
plt.figure(figsize=(14, 6))

# Plot for training and validation accuracy
plt.subplot(1, 2, 1)
for i in range(n_folds):
    plt.plot(fold_accuracies[i], label=f'Fold {i+1} Train')
    plt.plot(fold_val_accuracies[i], label=f'Fold {i+1} Val', linestyle='--')
plt.title('Training and Validation Accuracy per Fold')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Plot for training and validation loss
plt.subplot(1, 2, 2)
for i in range(n_folds):
    plt.plot(fold_losses[i], label=f'Fold {i+1} Train')
    plt.plot(fold_val_losses[i], label=f'Fold {i+1} Val', linestyle='--')
plt.title('Training and Validation Loss per Fold')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()

# Plot training and validation metrics to assess the model's performance. This helps in visualizing overfitting and underfitting
```



In [ ]: