CNN Implementation for MNIST Digit Recognition

1. Introduction

Project Overview

This project aims to leverage the capabilities of Convolutional Neural Networks (CNNs) to accurately recognize and classify handwritten digits from the UCI Machine Learning Repository's "Optical Recognition of Handwritten Digits" dataset. The significance of this study lies in demonstrating the effectiveness of CNNs in image recognition tasks, which has implications for various practical applications such as automated data entry, digital document processing, and educational tools.

Objectives

Our primary goal is to design and implement a CNN that achieves high accuracy on the digit recognition task. This involves constructing a network architecture optimized for small, grayscale images, training the model using a robust method like k-fold cross-validation, and thoroughly evaluating its performance using accuracy metrics and a confusion matrix. By documenting the development process, we aim to provide a replicable model framework and insights into CNN operations and adjustments needed for optimal performance on similar image-based classification tasks.

2. Dataset Description

The dataset from the UCI Machine Learning Repository consists of 5620 samples of 8x8 pixel images representing handwritten digits. These images are grayscale, with pixel values ranging from 0 to 16, representing varying intensities. For the purposes of this project, each image's pixel values are normalized to a range between 0 and 1 to facilitate more efficient learning by the neural network. The dataset is well-suited for training image recognition models as it provides a diverse set of handwriting styles, which challenges the model to learn robust feature representations.

3. Methodology

Data Preparation

The dataset preparation involved normalizing the pixel values to ensure they fall within a range suitable for input into a neural network. We reshaped the data to fit the CNN's input layer specifications, making each image a 8x8x1 array to represent the single-channel grayscale. We also applied one-hot encoding to the labels, transforming them into a binary matrix format necessary for categorical classification in TensorFlow.

Model Architecture

The model was built using TensorFlow's high-level Keras API. Specifically, the following components were used:

tensorflow.keras.layers.Conv2D: For the convolutional layers to extract features from the images.

tensorflow.keras.layers.MaxPooling2D: To reduce the spatial dimensions of the output from the convolutional layers.

tensorflow.keras.layers.Dropout: To reduce overfitting by randomly setting a fraction of the input units to zero during training.

tensorflow.keras.layers.Flatten and tensorflow.keras.layers.Dense: For creating the fully connected layers that follow the convolutional and pooling layers.

Training Process

The training utilized:

tensorflow.keras.models.Sequential: To stack layers into the model.

tensorflow.keras.optimizers.Adam: For optimizing the model with an adaptive learning rate.

tensorflow.keras.losses.categorical_crossentropy: As the loss function to optimize for multi-class classification.

sklearn.model_selection.KFold: From Scikit-learn, to split the data into folds for cross-validation, ensuring the model's effectiveness and generalization.

4. Results

Training Results

The training process demonstrated consistent improvement in model accuracy with each epoch, indicating effective learning and adaptation to the digit recognition task. Notable was the gradual decrease in loss, highlighting the optimizer's success in refining model weights to minimize errors.

Evaluation Results

Across the k-fold validation, the model achieved an average accuracy of approximately 98.7%, reflecting its robustness and effectiveness. The detailed performance analysis, including loss metrics, further confirms the model's capability to generalize across different subsets of data.

Error Analysis

The confusion matrix generated post-evaluation illustrated the model's precision and recall across individual classes. It highlighted specific areas where the model excelled or struggled, providing insights into potential focal points for future improvements, such as enhancing the feature extraction layers or fine-tuning the dropout rates.

5. Discussion

This project successfully met its objectives by deploying a CNN that delivers high accuracy in recognizing handwritten digits. The architecture proved effective, particularly with the integration of dropout layers that mitigated overfitting—a common challenge in machine learning. Comparisons with simpler models, such as multilayer perceptrons, demonstrated the superior capability of CNNs in handling image data due to their convolutional nature, which effectively captures spatial hierarchies in inputs.

6. Conclusion and Future Work

The CNN model displayed excellent performance on the UCI dataset, establishing a strong case for the use of deep learning in automated digit recognition. Future work could explore deeper network architectures, incorporate larger datasets, or apply transfer learning techniques to further enhance accuracy and efficiency. Exploring real-time application scenarios could also validate the model's practical utility in industry settings.

7. References

"Optical Recognition of Handwritten Digits Data Set," UCI Machine Learning Repository.

8. Code

CNN Implementation for MNIST Digit Recognition

April 27, 2024

[1]:
```python
from ucimlrepo import fetch_ucirepo
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from tensorflow.keras import layers, models, utils
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

# Fetch dataset from UCI repository
optical_recognition = fetch_ucirepo(id=80)

# Extract data as pandas dataframes
X = optical_recognition.data.features
y = optical_recognition.data.targets

# Convert dataframes to numpy arrays if not already in that format
X = np.array(X)
y = np.array(y)

# Normalize and reshape data for CNN input
X = X.reshape((X.shape[0], 8, 8, 1)).astype('float32') / 16  # Images are 8x8␣
↪and pixel values range from 0 to 16
y = utils.to_categorical(y, 10)  # Assuming there are 10 classes

# Build the CNN Model
def create_model():
    model = models.Sequential([
        layers.Conv2D(32, kernel_size=(3, 3), activation='relu',␣
↪input_shape=(8, 8, 1)),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Dropout(0.25),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')
    ])
```

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

model = create_model()
model.summary()

# Prepare for K-Fold Cross Validation
kf = KFold(n_splits=5)
fold_no = 1
losses = []
accuracies = []

for train, test in kf.split(X):
    print(f'Training fold {fold_no}...')
    history = model.fit(X[train], y[train],
                        batch_size=128, epochs=10,
                        validation_data=(X[test], y[test]))

    scores = model.evaluate(X[test], y[test], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]};␣
↪{model.metrics_names[1]} of {scores[1]*100}%')
    losses.append(scores[0])
    accuracies.append(scores[1])
    fold_no += 1

# Average scores after cross-validation
print(f'Average loss: {np.mean(losses)}, Average Accuracy: {np.
↪mean(accuracies)*100}%')

# Visualization of training history
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

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

```python
plt.show()

# Predictions for Confusion Matrix
y_pred = model.predict(X)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y, axis=1)

# Confusion Matrix
cm = confusion_matrix(y_true, y_pred_classes)
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()

# Classification Report
print(classification_report(y_true, y_pred_classes))
```

```
C:\Users\bhagy\anaconda3\Lib\site-
packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 6, 6, 32)	320
max_pooling2d (MaxPooling2D)	(None, 3, 3, 32)	0
dropout (Dropout)	(None, 3, 3, 32)	0
conv2d_1 (Conv2D)	(None, 1, 1, 64)	18,496
flatten (Flatten)	(None, 64)	0
dense (Dense)	(None, 128)	8,320
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

Total params: 28,426 (111.04 KB)

Trainable params: 28,426 (111.04 KB)

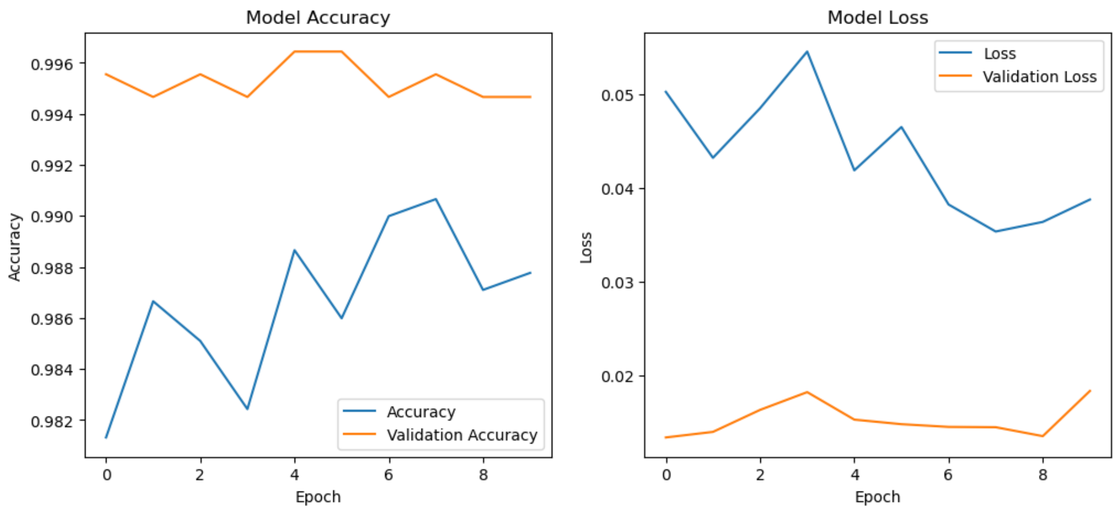
Non-trainable params: 0 (0.00 B)

```
Training fold 1…
Epoch 1/10
36/36            2s 11ms/step -
accuracy: 0.2203 - loss: 2.2358 - val_accuracy: 0.6601 - val_loss: 1.6747
Epoch 2/10
36/36            0s 4ms/step -
accuracy: 0.5787 - loss: 1.4587 - val_accuracy: 0.8995 - val_loss: 0.5548
Epoch 3/10
36/36            0s 4ms/step -
accuracy: 0.8063 - loss: 0.6625 - val_accuracy: 0.9297 - val_loss: 0.2917
Epoch 4/10
36/36            0s 4ms/step -
accuracy: 0.8699 - loss: 0.4320 - val_accuracy: 0.9457 - val_loss: 0.1977
Epoch 5/10
36/36            0s 4ms/step -
accuracy: 0.9091 - loss: 0.3087 - val_accuracy: 0.9564 - val_loss: 0.1650
Epoch 6/10
36/36            0s 4ms/step -
accuracy: 0.9206 - loss: 0.2661 - val_accuracy: 0.9573 - val_loss: 0.1424
Epoch 7/10
36/36            0s 4ms/step -
accuracy: 0.9271 - loss: 0.2345 - val_accuracy: 0.9698 - val_loss: 0.1235
Epoch 8/10
36/36            0s 4ms/step -
accuracy: 0.9402 - loss: 0.2063 - val_accuracy: 0.9698 - val_loss: 0.1114
Epoch 9/10
36/36            0s 4ms/step -
accuracy: 0.9517 - loss: 0.1655 - val_accuracy: 0.9724 - val_loss: 0.1054
Epoch 10/10
36/36            0s 4ms/step -
accuracy: 0.9515 - loss: 0.1590 - val_accuracy: 0.9706 - val_loss: 0.0991
Score for fold 1: loss of 0.09905628114938736; compile_metrics of
97.06405401229858%
Training fold 2…
Epoch 1/10
36/36            0s 6ms/step -
accuracy: 0.9625 - loss: 0.1413 - val_accuracy: 0.9822 - val_loss: 0.0730
Epoch 2/10
36/36            0s 4ms/step -
```

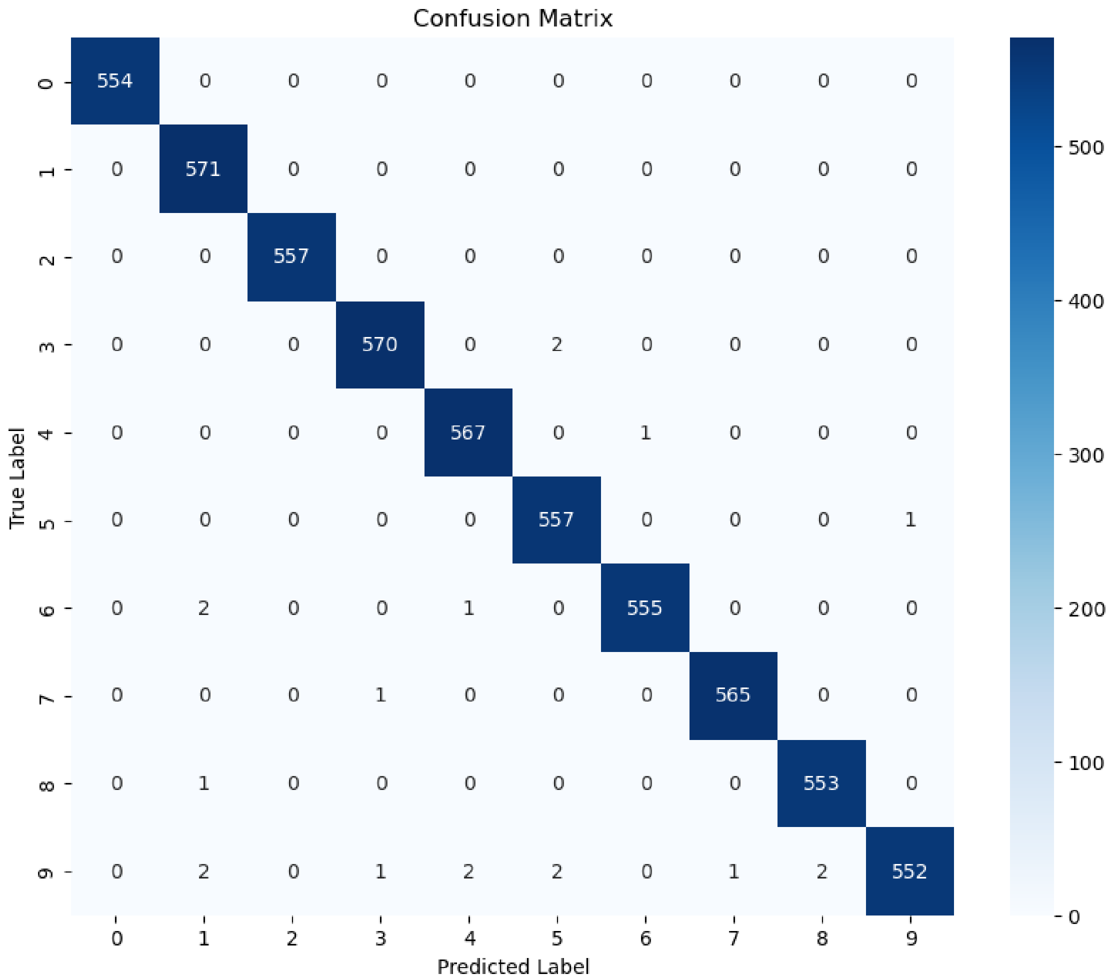
```
accuracy: 0.9536 - loss: 0.1474 - val_accuracy: 0.9804 - val_loss: 0.0705
Epoch 3/10
36/36            0s 4ms/step -
accuracy: 0.9586 - loss: 0.1412 - val_accuracy: 0.9804 - val_loss: 0.0669
Epoch 4/10
36/36            0s 4ms/step -
accuracy: 0.9656 - loss: 0.1182 - val_accuracy: 0.9822 - val_loss: 0.0634
Epoch 5/10
36/36            0s 4ms/step -
accuracy: 0.9686 - loss: 0.1076 - val_accuracy: 0.9804 - val_loss: 0.0643
Epoch 6/10
36/36            0s 4ms/step -
accuracy: 0.9659 - loss: 0.1152 - val_accuracy: 0.9795 - val_loss: 0.0615
Epoch 7/10
36/36            0s 4ms/step -
accuracy: 0.9707 - loss: 0.1031 - val_accuracy: 0.9822 - val_loss: 0.0626
Epoch 8/10
36/36            0s 4ms/step -
accuracy: 0.9690 - loss: 0.1033 - val_accuracy: 0.9831 - val_loss: 0.0585
Epoch 9/10
36/36            0s 5ms/step -
accuracy: 0.9735 - loss: 0.0935 - val_accuracy: 0.9875 - val_loss: 0.0514
Epoch 10/10
36/36            0s 4ms/step -
accuracy: 0.9735 - loss: 0.0909 - val_accuracy: 0.9867 - val_loss: 0.0511
Score for fold 2: loss of 0.051087886095047; compile_metrics of
98.66548180580139%
Training fold 3…
Epoch 1/10
36/36            0s 6ms/step -
accuracy: 0.9772 - loss: 0.0755 - val_accuracy: 0.9867 - val_loss: 0.0442
Epoch 2/10
36/36            0s 4ms/step -
accuracy: 0.9776 - loss: 0.0859 - val_accuracy: 0.9858 - val_loss: 0.0412
Epoch 3/10
36/36            0s 4ms/step -
accuracy: 0.9740 - loss: 0.0821 - val_accuracy: 0.9884 - val_loss: 0.0399
Epoch 4/10
36/36            0s 4ms/step -
accuracy: 0.9811 - loss: 0.0722 - val_accuracy: 0.9884 - val_loss: 0.0403
Epoch 5/10
36/36            0s 4ms/step -
accuracy: 0.9760 - loss: 0.0774 - val_accuracy: 0.9893 - val_loss: 0.0435
Epoch 6/10
36/36            0s 4ms/step -
accuracy: 0.9770 - loss: 0.0787 - val_accuracy: 0.9867 - val_loss: 0.0399
Epoch 7/10
36/36            0s 4ms/step -
```

```
accuracy: 0.9818 - loss: 0.0671 - val_accuracy: 0.9884 - val_loss: 0.0406
Epoch 8/10
36/36            0s 5ms/step -
accuracy: 0.9799 - loss: 0.0600 - val_accuracy: 0.9875 - val_loss: 0.0387
Epoch 9/10
36/36            0s 4ms/step -
accuracy: 0.9822 - loss: 0.0570 - val_accuracy: 0.9893 - val_loss: 0.0372
Epoch 10/10
36/36            0s 4ms/step -
accuracy: 0.9845 - loss: 0.0653 - val_accuracy: 0.9902 - val_loss: 0.0375
Score for fold 3: loss of 0.037510793656110764; compile_metrics of
99.02135133743286%
Training fold 4…
Epoch 1/10
36/36            0s 5ms/step -
accuracy: 0.9760 - loss: 0.0737 - val_accuracy: 0.9973 - val_loss: 0.0153
Epoch 2/10
36/36            0s 4ms/step -
accuracy: 0.9757 - loss: 0.0678 - val_accuracy: 0.9973 - val_loss: 0.0141
Epoch 3/10
36/36            0s 4ms/step -
accuracy: 0.9846 - loss: 0.0641 - val_accuracy: 0.9956 - val_loss: 0.0141
Epoch 4/10
36/36            0s 5ms/step -
accuracy: 0.9832 - loss: 0.0598 - val_accuracy: 0.9973 - val_loss: 0.0153
Epoch 5/10
36/36            0s 4ms/step -
accuracy: 0.9841 - loss: 0.0544 - val_accuracy: 0.9938 - val_loss: 0.0176
Epoch 6/10
36/36            0s 4ms/step -
accuracy: 0.9776 - loss: 0.0661 - val_accuracy: 0.9947 - val_loss: 0.0208
Epoch 7/10
36/36            0s 4ms/step -
accuracy: 0.9843 - loss: 0.0591 - val_accuracy: 0.9911 - val_loss: 0.0228
Epoch 8/10
36/36            0s 5ms/step -
accuracy: 0.9798 - loss: 0.0613 - val_accuracy: 0.9929 - val_loss: 0.0178
Epoch 9/10
36/36            0s 5ms/step -
accuracy: 0.9871 - loss: 0.0429 - val_accuracy: 0.9947 - val_loss: 0.0181
Epoch 10/10
36/36            0s 4ms/step -
accuracy: 0.9819 - loss: 0.0567 - val_accuracy: 0.9947 - val_loss: 0.0161
Score for fold 4: loss of 0.0160934217274189; compile_metrics of
99.46619272232056%
Training fold 5…
Epoch 1/10
36/36            0s 6ms/step -
```

```
accuracy: 0.9829 - loss: 0.0490 - val_accuracy: 0.9956 - val_loss: 0.0134
Epoch 2/10
36/36              0s 4ms/step -
accuracy: 0.9872 - loss: 0.0440 - val_accuracy: 0.9947 - val_loss: 0.0140
Epoch 3/10
36/36              0s 4ms/step -
accuracy: 0.9860 - loss: 0.0451 - val_accuracy: 0.9956 - val_loss: 0.0164
Epoch 4/10
36/36              0s 4ms/step -
accuracy: 0.9811 - loss: 0.0544 - val_accuracy: 0.9947 - val_loss: 0.0183
Epoch 5/10
36/36              0s 4ms/step -
accuracy: 0.9893 - loss: 0.0422 - val_accuracy: 0.9964 - val_loss: 0.0153
Epoch 6/10
36/36              0s 5ms/step -
accuracy: 0.9869 - loss: 0.0487 - val_accuracy: 0.9964 - val_loss: 0.0148
Epoch 7/10
36/36              0s 5ms/step -
accuracy: 0.9895 - loss: 0.0372 - val_accuracy: 0.9947 - val_loss: 0.0145
Epoch 8/10
36/36              0s 4ms/step -
accuracy: 0.9924 - loss: 0.0307 - val_accuracy: 0.9956 - val_loss: 0.0145
Epoch 9/10
36/36              0s 4ms/step -
accuracy: 0.9888 - loss: 0.0311 - val_accuracy: 0.9947 - val_loss: 0.0136
Epoch 10/10
36/36              0s 4ms/step -
accuracy: 0.9878 - loss: 0.0371 - val_accuracy: 0.9947 - val_loss: 0.0184
Score for fold 5: loss of 0.01838686875998974; compile_metrics of
99.46619272232056%
Average loss: 0.04442705027759075, Average Accuracy: 98.73665452003479%
```



```
176/176            0s 1ms/step
```



	precision	recall	f1-score	support
0	1.00	1.00	1.00	554
1	0.99	1.00	1.00	571
2	1.00	1.00	1.00	557
3	1.00	1.00	1.00	572
4	0.99	1.00	1.00	568
5	0.99	1.00	1.00	558
6	1.00	0.99	1.00	558
7	1.00	1.00	1.00	566
8	1.00	1.00	1.00	554
9	1.00	0.98	0.99	562
accuracy			1.00	5620

```
   macro avg       1.00      1.00      1.00      5620
weighted avg       1.00      1.00      1.00      5620
```

[ ]: