```
# Project Title: CNN Implementation for MNIST Digit Recognition

# SAINATH VADDI     - 101179915
# SONY REDDY GURRAM - 101179182
```

```
!pip install ucimlrepo
from ucimlrepo import fetch_ucirepo
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.model_selection import KFold
from sklearn.preprocessing import MinMaxScaler
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import seaborn as sns
from tensorflow.keras.utils import plot_model
```

```
Requirement already satisfied: ucimlrepo in /usr/local/lib/python3.10/dist-packages (0.0.6)
```

```
# 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_url': 'https://archive.ics.uci.edu/dataset/80/optical+re
           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]
```

```
# Shapes of X and y
print(X.shape)
print(y.shape)
```

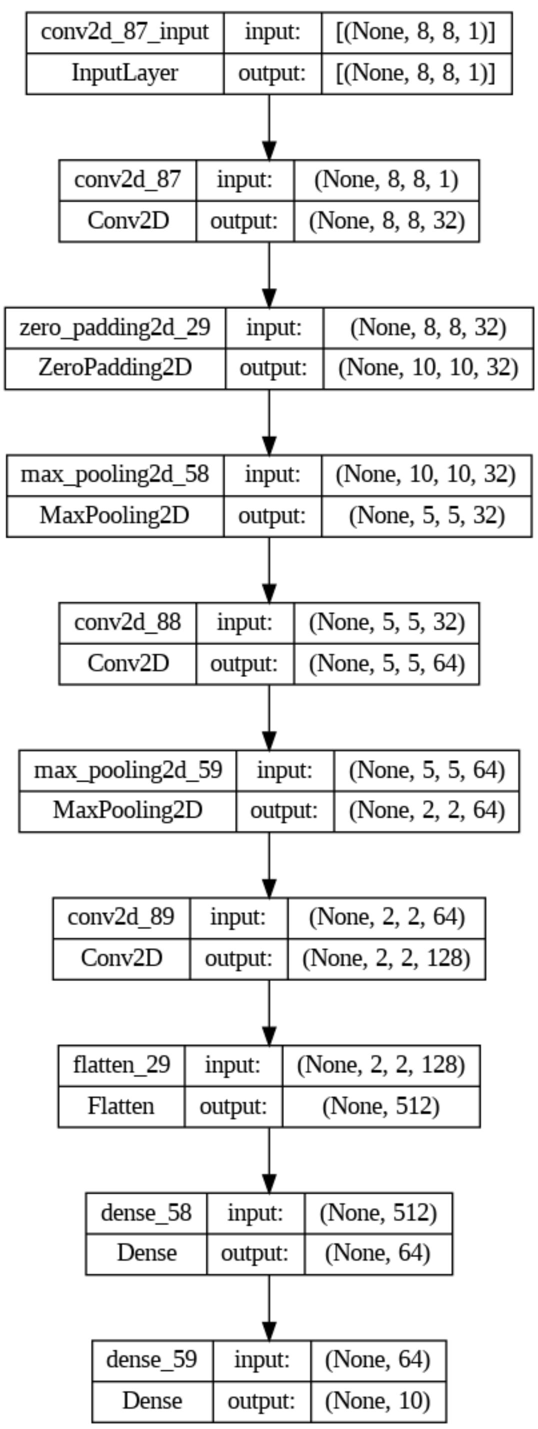
```
(5620, 64)
(5620, 1)
```

```
X = X.values
y = y.values
```

```
# Reshape data for CNN
X = X.reshape(-1, 8, 8, 1)
```

```
# Normalization data for CNN
scaler = MinMaxScaler()
X_normalized = scaler.fit_transform(X.reshape(-1, 64))  # Reshape for MinMaxScaler
X_normalized = X_normalized.reshape(-1, 8, 8, 1)
```

```
# Define the CNN model
def create_model():
    model = models.Sequential([

        # Convolutional Layer 1
        layers.Conv2D(32, (3, 3), activation = 'relu', padding = 'same', input_shape = (8, 8, 1)), # Parameters: 32 filters, kernel size (3,

        # Add padding to increase spatial dimensions
        layers.ZeroPadding2D((1, 1)), # Input Dimension: (8, 8, 32), Output Dimesnion: (10, 10, 32)

        # Max Pooling Layer 1
        layers.MaxPooling2D((2, 2)),  # Pool Size: (2, 2), Strides: (2, 2), Input Dimension: (10, 10, 32), Output Dimension: (5, 5, 32)

        # Convolutional Layer 2
        layers.Conv2D(64, (3, 3), activation='relu', padding = 'same',), # Parameters: 64 filters, kernel size (3, 3), Input Dimension: (5,

        # Max Pooling Layer 2
        layers.MaxPooling2D((2, 2)), # Pool Size: (2, 2), Strides: (2, 2), Input Dimension: (5, 5, 64), Output Dimension: (2, 2, 64)

        # Convolutional Layer 3
        layers.Conv2D(128, (3, 3), activation='relu', padding = 'same',), # Parameters: 128 filters, kernel size (3, 3), Input Dimension: (2

        # Flattening Layer
        layers.Flatten(),  # Flattening the output of the last convolutional layer, Input Dimension: (2, 2, 128), Output Dimension: (512,)

        # Fully Connected Layer 1
        layers.Dense(64, activation='relu'),  # Fully Connected Layer with 64 neurons and ReLU activation, Input Dimension: (512,), Output D

        # Output Layer
        layers.Dense(10, activation='softmax')  # Output Layer with 10 neurons for classification and softmax activation, Input Dimension: (
    ])
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model


# Plot the model architecture
plot_model(model, to_file='cnn_model.png', show_shapes=True, show_layer_names=True)
```



```
# Define k-fold cross-validation
k = 5
kf = KFold(n_splits=k, shuffle=True, random_state=42)
# Initialize lists to store results
fold_accuracy = []
fold_loss = []
all_y_true = []
all_y_pred = []


# Perform k-fold cross-validation
fold_accuracy = []
for train_index, val_index in kf.split(X):
    X_train, X_val = X_normalized[train_index], X_normalized[val_index]
    y_train, y_val = y[train_index], y[val_index]

    model = create_model()
    history = model.fit(X_train, y_train, epochs=8, batch_size=32, validation_data=(X_val, y_val))
```

```
Epoch 4/8
141/141 [==============================] - 3s 20ms/step - loss: 0.0725 - accuracy: 0.9780 - val_loss: 0.0753 - val_accuracy: 0.9795
Epoch 5/8
141/141 [==============================] - 2s 17ms/step - loss: 0.0578 - accuracy: 0.9818 - val_loss: 0.0654 - val_accuracy: 0.9831
Epoch 6/8
141/141 [==============================] - 2s 11ms/step - loss: 0.0503 - accuracy: 0.9851 - val_loss: 0.0759 - val_accuracy: 0.9751
Epoch 7/8
141/141 [==============================] - 2s 11ms/step - loss: 0.0341 - accuracy: 0.9907 - val_loss: 0.1948 - val_accuracy: 0.9359
Epoch 8/8
141/141 [==============================] - 2s 12ms/step - loss: 0.0293 - accuracy: 0.9911 - val_loss: 0.0612 - val_accuracy: 0.9786
Epoch 1/8
141/141 [==============================] - 3s 13ms/step - loss: 0.8960 - accuracy: 0.7226 - val_loss: 0.2327 - val_accuracy: 0.9244
Epoch 2/8
141/141 [==============================] - 2s 18ms/step - loss: 0.1595 - accuracy: 0.9520 - val_loss: 0.1797 - val_accuracy: 0.9386
Epoch 3/8
141/141 [==============================] - 2s 12ms/step - loss: 0.0941 - accuracy: 0.9718 - val_loss: 0.0830 - val_accuracy: 0.9742
Epoch 4/8
141/141 [==============================] - 2s 11ms/step - loss: 0.0669 - accuracy: 0.9798 - val_loss: 0.0926 - val_accuracy: 0.9662
Epoch 5/8
141/141 [==============================] - 2s 12ms/step - loss: 0.0526 - accuracy: 0.9847 - val_loss: 0.0795 - val_accuracy: 0.9760
Epoch 6/8
141/141 [==============================] - 2s 11ms/step - loss: 0.0484 - accuracy: 0.9851 - val_loss: 0.0530 - val_accuracy: 0.9831
Epoch 7/8
141/141 [==============================] - 2s 11ms/step - loss: 0.0261 - accuracy: 0.9924 - val_loss: 0.0426 - val_accuracy: 0.9884
Epoch 8/8
141/141 [==============================] - 2s 11ms/step - loss: 0.0277 - accuracy: 0.9913 - val_loss: 0.0570 - val_accuracy: 0.9849
Epoch 1/8
141/141 [==============================] - 4s 18ms/step - loss: 0.8897 - accuracy: 0.7160 - val_loss: 0.1756 - val_accuracy: 0.9555
Epoch 2/8
141/141 [==============================] - 2s 11ms/step - loss: 0.1714 - accuracy: 0.9513 - val_loss: 0.1002 - val_accuracy: 0.9733
Epoch 3/8
141/141 [==============================] - 2s 12ms/step - loss: 0.1002 - accuracy: 0.9722 - val_loss: 0.0899 - val_accuracy: 0.9733
Epoch 4/8
141/141 [==============================] - 2s 11ms/step - loss: 0.0826 - accuracy: 0.9740 - val_loss: 0.0592 - val_accuracy: 0.9875
Epoch 5/8
141/141 [==============================] - 2s 12ms/step - loss: 0.0487 - accuracy: 0.9847 - val_loss: 0.0527 - val_accuracy: 0.9840
Epoch 6/8
141/141 [==============================] - 2s 11ms/step - loss: 0.0424 - accuracy: 0.9875 - val_loss: 0.0548 - val_accuracy: 0.9867
Epoch 7/8
141/141 [==============================] - 2s 12ms/step - loss: 0.0537 - accuracy: 0.9820 - val_loss: 0.0772 - val_accuracy: 0.9769
Epoch 8/8
141/141 [==============================] - 3s 19ms/step - loss: 0.0226 - accuracy: 0.9929 - val_loss: 0.0413 - val_accuracy: 0.9875
Epoch 1/8
141/141 [==============================] - 3s 13ms/step - loss: 0.9456 - accuracy: 0.6993 - val_loss: 0.2388 - val_accuracy: 0.9315
Epoch 2/8
141/141 [==============================] - 2s 12ms/step - loss: 0.1962 - accuracy: 0.9393 - val_loss: 0.1035 - val_accuracy: 0.9671
Epoch 3/8
141/141 [==============================] - 2s 12ms/step - loss: 0.1194 - accuracy: 0.9653 - val_loss: 0.1022 - val_accuracy: 0.9689
Epoch 4/8
141/141 [==============================] - 2s 11ms/step - loss: 0.0820 - accuracy: 0.9746 - val_loss: 0.0626 - val_accuracy: 0.9831
Epoch 5/8
141/141 [==============================] - 2s 11ms/step - loss: 0.0622 - accuracy: 0.9804 - val_loss: 0.0463 - val_accuracy: 0.9902
Epoch 6/8
141/141 [==============================] - 2s 17ms/step - loss: 0.0414 - accuracy: 0.9884 - val_loss: 0.0426 - val_accuracy: 0.9884
Epoch 7/8
141/141 [==============================] - 2s 15ms/step - loss: 0.0381 - accuracy: 0.9889 - val_loss: 0.0534 - val_accuracy: 0.9813
Epoch 8/8
141/141 [==============================] - 2s 11ms/step - loss: 0.0323 - accuracy: 0.9900 - val_loss: 0.0379 - val_accuracy: 0.9893
```

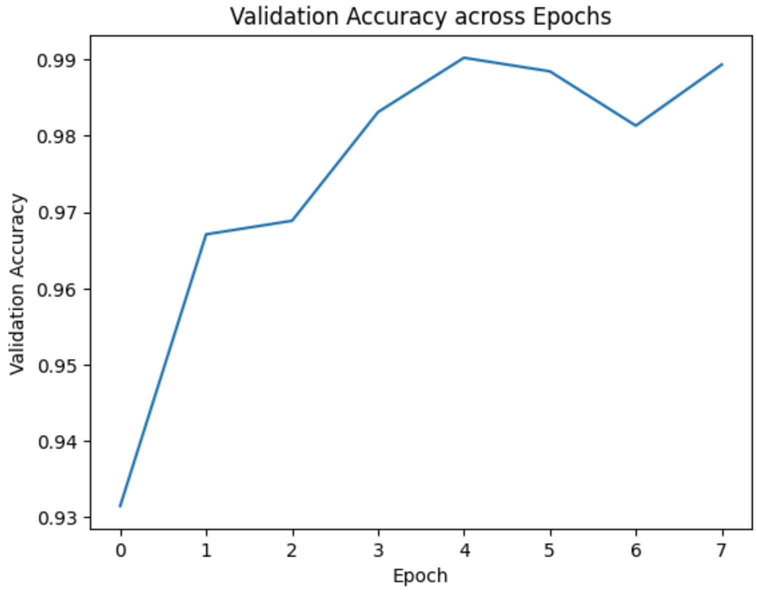
```
# Record accuracy and loss
fold_accuracy.append(history.history['val_accuracy'])
fold_loss.append(history.history['val_loss'])


# Predictions
y_pred = np.argmax(model.predict(X_val), axis=1)
all_y_true.extend(y_val)
all_y_pred.extend(y_pred)
```

```
36/36 [==============================] - 0s 4ms/step
```

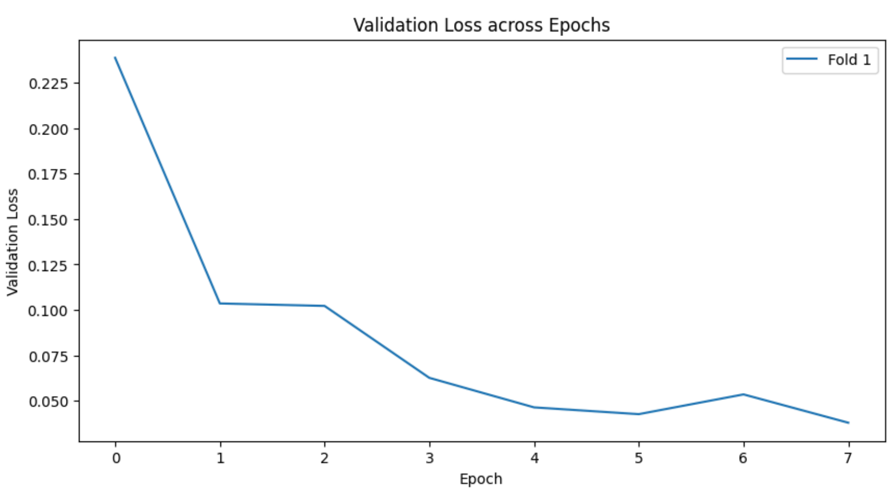
```
# Calculate and print average validation accuracy across folds
avg_val_accuracy = np.mean(fold_accuracy, axis=0)
print('Average validation accuracy across folds:', avg_val_accuracy)
```

```
Average validation accuracy across folds: [0.93149465 0.96708184 0.96886122 0.98309606 0.99021351 0.98843414
 0.98131675 0.98932385]
```

```
# Plot the validation accuracy across epochs
plt.plot(avg_val_accuracy)
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.title('Validation Accuracy across Epochs')
plt.show()
```




```
# Plot the validation loss across epochs
plt.figure(figsize=(10, 5))
for i in range(len(fold_loss)):
    plt.plot(history.epoch, fold_loss[i], label=f'Fold {i+1}')
plt.xlabel('Epoch')
plt.ylabel('Validation Loss')
plt.title('Validation Loss across Epochs')
plt.legend()
plt.show()
```



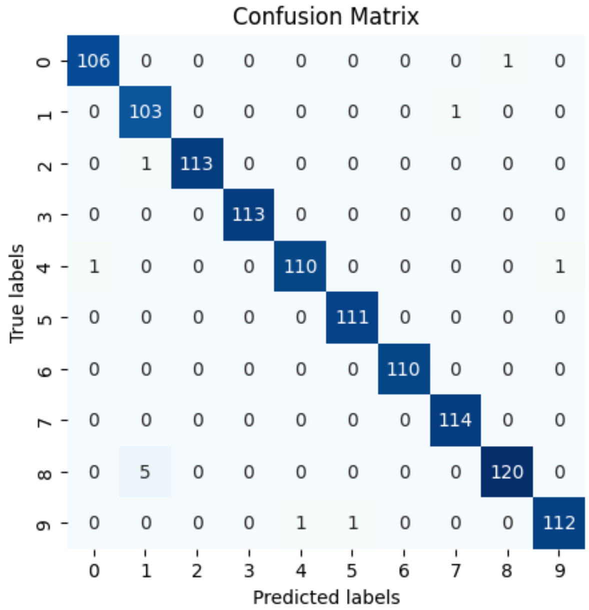

```
# Calculate overall accuracy
overall_accuracy = accuracy_score(all_y_true, all_y_pred)
print('Overall accuracy:', overall_accuracy)
```

```
Overall accuracy: 0.9893238434163701
```

```
classification_report = classification_report(y_val, y_pred)
print(classification_report)
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       107
           1       0.94      0.99      0.97       104
           2       1.00      0.99      1.00       114
           3       1.00      1.00      1.00       113
           4       0.99      0.98      0.99       112
           5       0.99      1.00      1.00       111
           6       1.00      1.00      1.00       110
           7       0.99      1.00      1.00       114
           8       0.99      0.96      0.98       125
           9       0.99      0.98      0.99       114

    accuracy                           0.99      1124
   macro avg       0.99      0.99      0.99      1124
weighted avg       0.99      0.99      0.99      1124
```

```
# Confusion matrix
conf_matrix = confusion_matrix(all_y_true, all_y_pred)
plt.figure(figsize=(5, 5))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
plt.show()
```


Confusion Matrix
True labels
Predicted labels
0 1 2 3 4 5 6 7 8 9
106 0 0 0 0 0 0 0 1 0
0 103 0 0 0 0 0 1 0 0
0 1 113 0 0 0 0 0 0 0
0 0 0 113 0 0 0 0 0 0
1 0 0 0 110 0 0 0 0 1
0 0 0 0 0 111 0 0 0 0
0 0 0 0 0 0 110 0 0 0
0 0 0 0 0 0 0 114 0 0
0 5 0 0 0 0 0 0 120 0
0 0 0 0 1 1 0 0 0 112