Digit Recognizer (Kaggle)

Ritesh Kumar

2024WI_MS_DSP_422-DL_SEC61: Practical Machine Learning

Module 7 Assignment

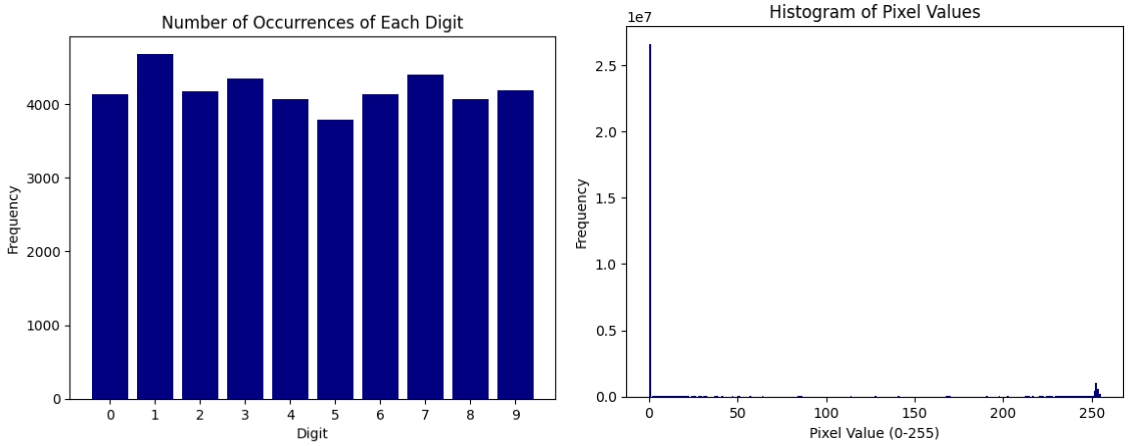
Digit Recognizer

Donald Wedding and Narayana Darapaneni

February 16, 2024

Exploration

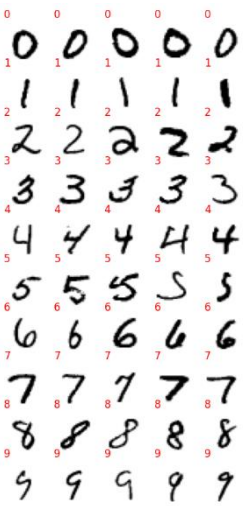
We start with a comprehensive Exploratory Data Analysis (EDA), commencing with the extraction of descriptive statistics for the ‘label column, serving as the target variable for prediction. We also plot the pixel-distribution for the 42000 records.



The distribution of digits is roughly uniform, suggesting a balanced dataset which is ideal for machine learning classification tasks, as it may prevent bias towards more frequent classes.

The pronounced peak at the left indicates an abundance of black pixels, while the peak at the right suggests a presence of white pixels. The scarcity of mid-range values implies these images have high contrast with predominantly black backgrounds and lighter features i.e. digits are typically white on a black background, contributing to the stark bimodal distribution observed.

We displayed a sample of the digits.



Model Development

Design of Experiments was conducted by development and evaluation of models using TensorFlow and Keras, encompassing a comprehensive approach to machine learning tasks, from data preparation to model optimization and evaluation. Various architectures were explored, including Dense Neural Networks (DNNs), Convolutional Neural Networks (CNNs), and Long Short-Term Memory (LSTM) networks, each tailored to the specific nature of the data. For instance, DNNs are optimized through grid searches focusing on hyperparameters like the number of layers and neurons, leveraging GridSearchCV for systematic tuning to improve accuracy. Similarly, CNNs are designed with considerations for the number of convolutional layers and filters, capitalizing on their ability to extract spatial features from image data. LSTMs, known for their efficiency in handling sequence data, are optimized by adjusting layers and units to capture temporal dependencies effectively.

The evaluation of these models involves not just examining training and validation accuracies but also employing techniques like early stopping to prevent overfitting. The results indicate that the optimal configurations vary by model type, emphasizing the importance of architectural design in achieving high performance. The process underscores the iterative nature of machine learning, where data exploration, model building, and rigorous evaluation converge to guide the optimization of neural network architectures for enhanced predictive capabilities.

Model Deployment

Dense Neural Network: The code outlined a method for optimizing a neural network's structure via grid search, focusing on hyperparameters like the number of layers and neurons. Using the build_model function, it builds models with different configurations, evaluated

against a dataset split into training and validation sets. This approach allows for exhaustive exploration of parameter combinations to identify the most effective neural network architecture. The process employs the Keras library for model construction and optimization, leveraging the GridSearchCV class from Scikit-Learn for systematic hyperparameter tuning, aiming to maximize accuracy. Results were collected and displayed, highlighting the impact of different configurations on model performance.

Layers	Nodes	Time	Training Accuracy	Validation Accuracy
2	14	93.63	0.953	0.943
2	28	90.61	0.956	0.942
2	56	91.84	0.945	0.934
5	14	92.83	0.948	0.937
5	28	91.05	0.953	0.944
5	56	92.11	0.957	0.946

The search tested combinations of 2 and 5 layers with 14, 28, and 56 nodes, assessing their impact on model performance over time, and evaluating training and validation accuracy. Models with 5 layers and 56 nodes achieved the highest validation accuracy (0.946) and training accuracy (0.957), indicating a potential sweet spot in complexity for this particular dataset. Conversely, models with 2 layers and 56 nodes showed lower validation accuracy (0.934), suggesting that simply increasing nodes without adjusting layers might not always yield better results.

Convolutional Neural Network (CNN): The code introduced an approach to optimize a Convolutional Neural Network (CNN) model for image classification tasks. It involves defining a function, build_cnn_model, which constructs a CNN with customizable parameters such as the number of convolutional layers, number of filters, kernel size, learning rate, and input shape. The model aims to maximize accuracy through iterative training and validation, utilizing Keras for model construction and Adam for optimization. A grid search strategy is employed to systematically explore different configurations of convolutional layers and

filters, facilitated by the GridSearchCV tool from Scikit-Learn, with the objective of finding the optimal model settings for enhanced performance on a given dataset.

Conv Layers	Filters	Time	Training Accuracy	Validation Accuracy
2	28	71.71	0.988	0.983
2	56	70.75	0.975	0.975
3	28	69.24	0.981	0.976
3	56	70.22	0.982	0.978

The summary details the outcomes of optimizing a Convolutional Neural Network (CNN) through grid search, varying the number of convolutional layers and filters. The configurations explored include 2 or 3 convolutional layers with 28 or 56 filters. The best model, with 2 convolutional layers and 28 filters, achieved the highest training accuracy of 98.8% and validation accuracy of 98.3%, indicating superior performance and generalizability. Models with more filters or layers did not necessarily perform better, highlighting the importance of finding the right balance between model complexity and efficiency for optimal performance.

Long Short-Term Memory (LSTM): We designed and optimized a Long Short-Term Memory (LSTM) neural network model for sequence data processing.
The build_lstm_model function is central to this process, enabling the creation of models with varying numbers of LSTM layers and units, tailored to the specific structure of the input data. By leveraging a grid search technique, the code aims to find the optimal combination of LSTM layers and units that maximizes model accuracy. This approach is facilitated by Keras for model construction and optimization, and the search for the best hyperparameters is conducted using Scikit-Learn's GridSearchCV, focusing on maximizing accuracy. The process not only involves model training and validation but also a thorough evaluation of the results to identify the configuration that offers the best performance on the given dataset.

LSTM Layers	Units	Time	Training Accuracy	Validation Accuracy
1	50	124.74	0.948	0.942
1	100	123.84	0.942	0.941
2	50	121.13	0.949	0.946
2	100	125.61	0.955	0.949

The summary showcases the results of optimizing a Long Short-Term Memory (LSTM) network through varying the number of layers and units. Models were assessed based on their training and validation accuracies. The configuration with 2 LSTM layers and 100 units emerged as the most effective, achieving the highest training accuracy of 95.5% and validation accuracy of 94.9%. This indicates that increasing both the number of layers and the complexity (units) of the LSTM model can lead to improved performance on the dataset, suggesting a better capability to capture complex patterns and dependencies in the data.

Conclusion

Analyzing the outcomes from experiments with traditional densely connected Neural Networks alongside those of Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks provides a comprehensive view of model optimization across different data types and architectural paradigms. The densely connected network optimization highlighted the effectiveness of deeper models with more neurons, particularly showing that a configuration with 5 layers and 56 neurons outperformed others in terms of validation accuracy. This finding aligns with the broader theme observed across the experiments: the importance of tailoring the network architecture to the task at hand for optimal performance.

The CNNs proved most adept at image processing tasks, optimizing at a relatively simpler architecture with 2 layers and 28 filters, emphasizing the value of spatial feature extraction in image data. In contrast, the LSTMs, with their ability to process sequential data, reached peak performance with 2 layers and 100 units, showcasing their strength in capturing temporal dependencies.

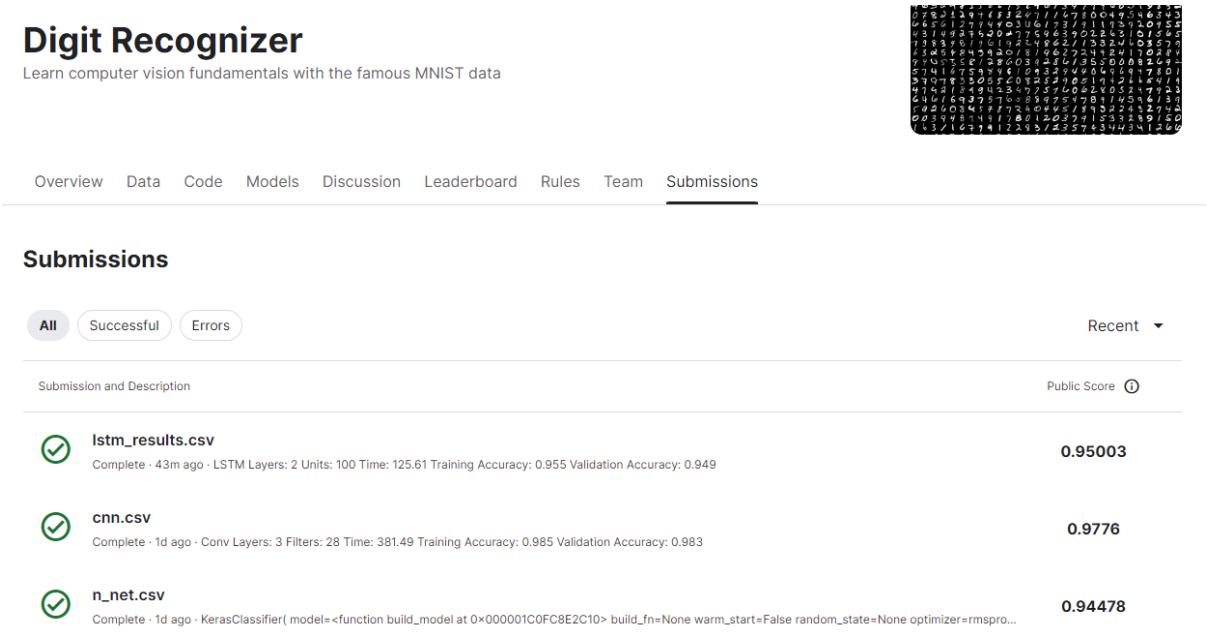
Across all models, the grid search strategy was essential in navigating the complex landscape of hyperparameters to identify configurations that maximize accuracy. This exploration underscores the critical role of architectural design in achieving high performance, demonstrating that while there is no universal solution, certain patterns—such as the depth of the network and the number of processing units—consistently influence outcomes. These insights reinforce the principle that effective model optimization is contingent on understanding the unique characteristics of the dataset and task, guiding the selection of the most suitable neural network architecture.

Kaggle Submission

My Kaggle username is riteshrk.

The notebook has been uploaded on Kaggle and can be accessed here.

The results of all the three models were submitted on Kaggle and can be accessed here.



For the test data on Kaggle, the best accuracy of 97.76% was returned for the Convolutional Neural Networks Model.

Code

1. Imports

In [2]:
```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from scikeras.wrappers import KerasRegressor, KerasClassifier
from scipy.stats import reciprocal
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split, RandomizedSearchCV, GridSearc
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from keras.optimizers import Adam
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import make_scorer
import time
import itertools
print(tf.__version__)
```

```
2.15.0
```

In [4]:
```
# Read the input file
df = pd.read_csv('train.csv')
print(df.shape)
```

```
(42000, 785)
```

2. Exploration

In [5]:
```
# Checking for missing values
df.isna().sum().max()
```

Out[5]:
```
0
```

In [6]:
```
# Checking for duplicates()
df.duplicated().sum()
```

Out[6]:
```
0
```

In [7]:
```
# Show the first 5 rows
df.head()
```

Out[7]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel77
0	1	0	0	0	0	0	0	0	0	0	...	0	
1	0	0	0	0	0	0	0	0	0	0	...	0	
2	1	0	0	0	0	0	0	0	0	0	...	0	
3	4	0	0	0	0	0	0	0	0	0	...	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	

5 rows × 785 columns

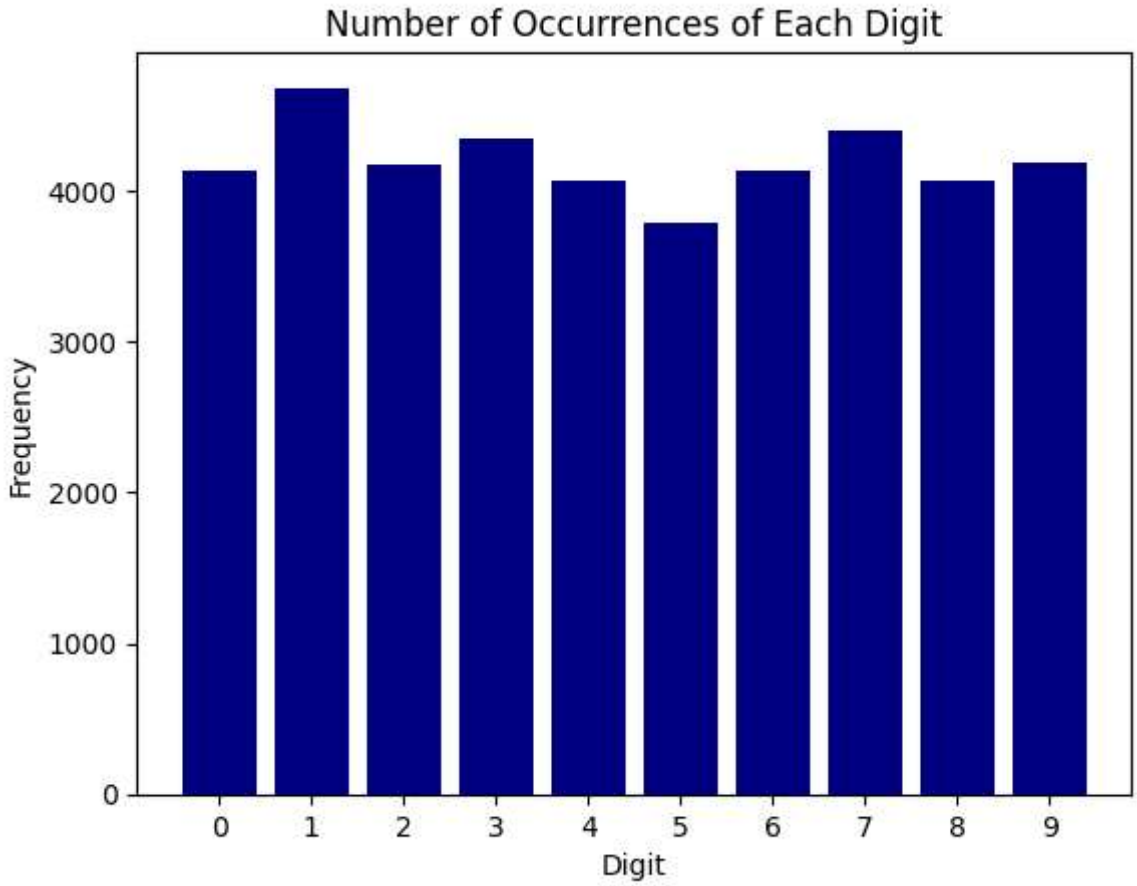
In [8]:
```
# Show the data-summary
df.describe()
```

Out[8]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel
count	42000.000000	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.
mean	4.456643	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
std	2.887730	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
min	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
25%	2.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
50%	4.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
75%	7.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
max	9.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.

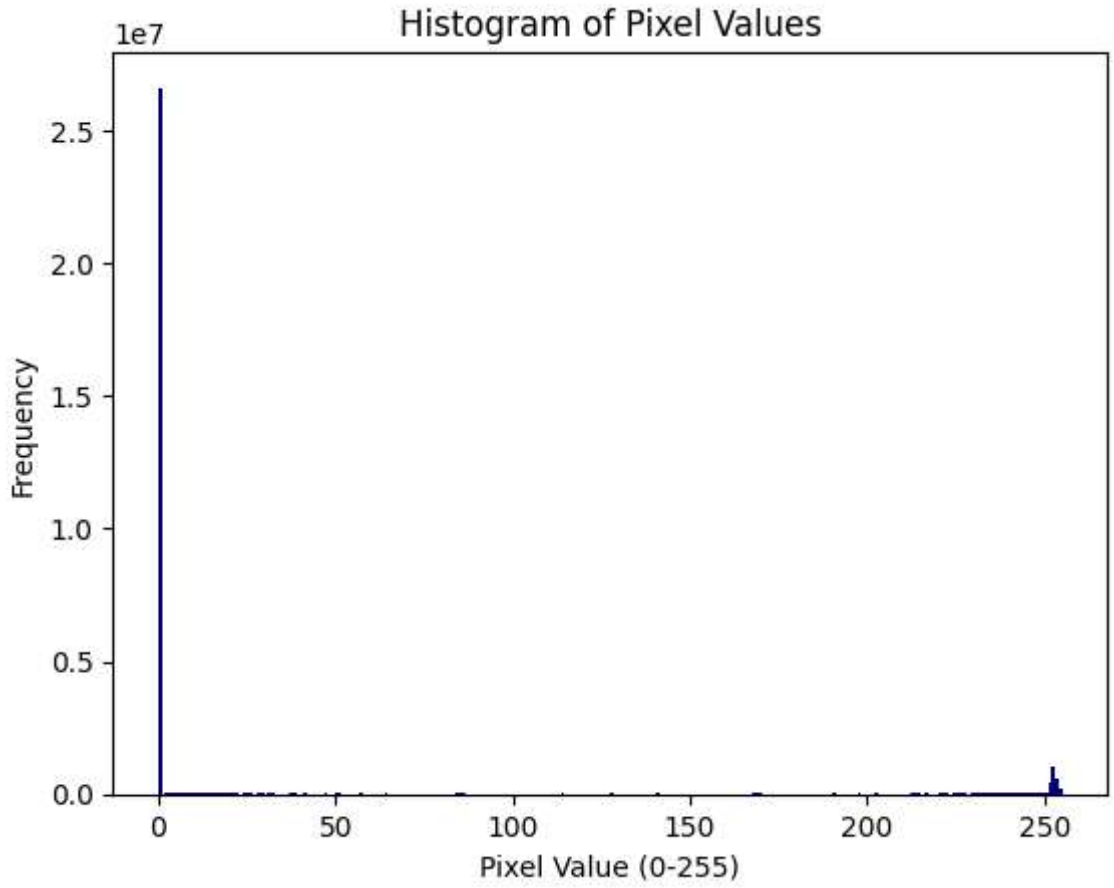
8 rows × 785 columns

In [9]:
```
# Count the occurrences of each label and sort by index (digit)
label_counts = df['label'].value_counts().sort_index()

# Ensure all digits are represented in the x-axis, even if their count is zero
all_digits = np.arange(0, 10)

# Get counts for each digit, fill missing ones with zero
counts = label_counts.reindex(all_digits, fill_value=0)

# Create bar plot
plt.bar(counts.index, counts.values, color='navy')

plt.title('Number of Occurrences of Each Digit')
plt.xlabel('Digit')
plt.ylabel('Frequency')

# Set x-ticks to be clearly labeled for each digit
plt.xticks(all_digits)

plt.show()
```



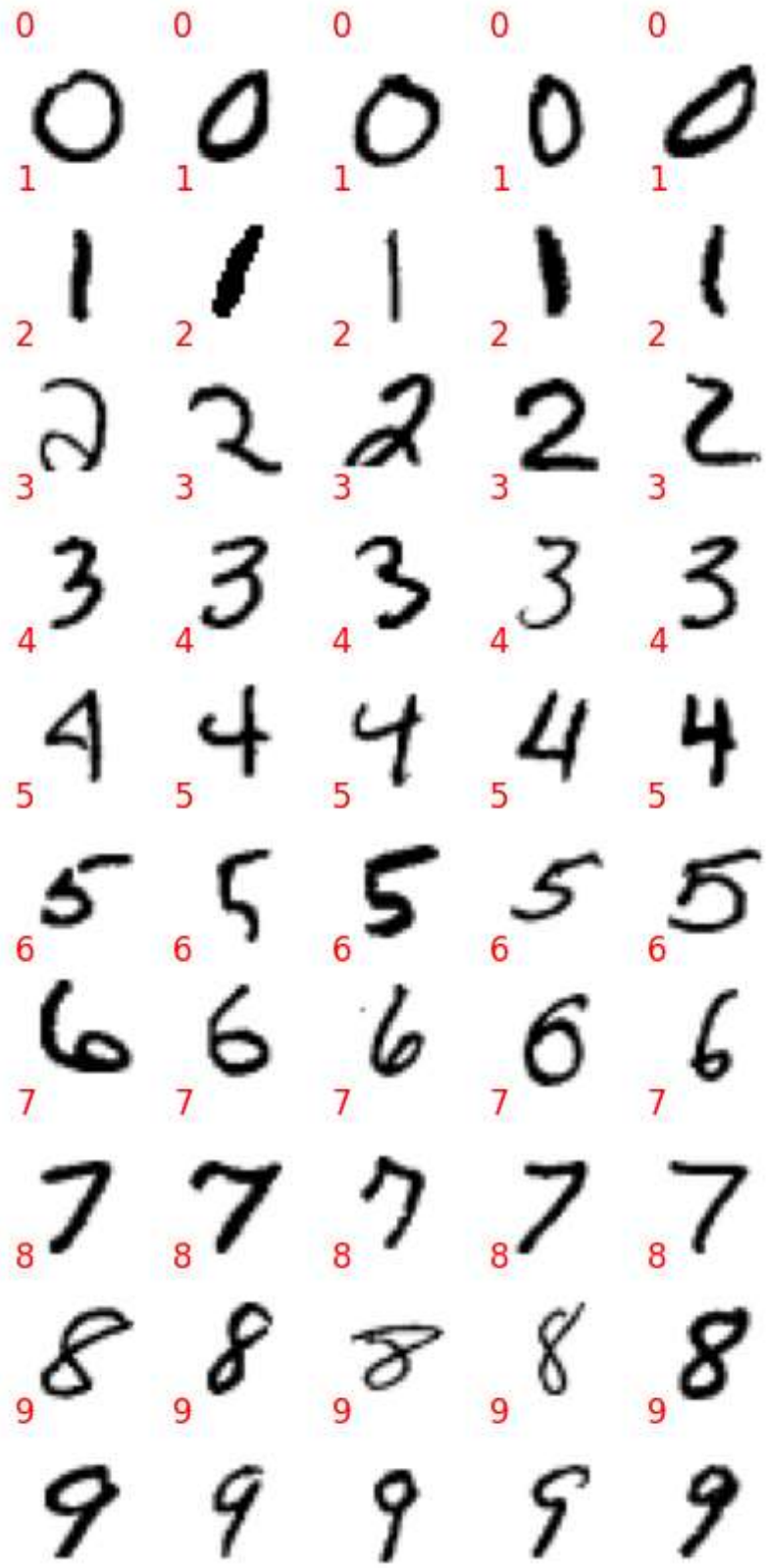
The distribution of digits is roughly uniform, suggesting a balanced dataset which is ideal for machine learning classification tasks, as it may prevent bias towards more frequent classes.

In [10]:
```
# Plot the histogram for pixel-distribution

# Drop the 'label' column and convert the data to a 1D array
pixel_values = df.drop('label', axis=1).values.flatten()

# Generate the histogram
plt.hist(pixel_values, bins=256, color='navy')

plt.title('Histogram of Pixel Values')
plt.xlabel('Pixel Value (0-255)')
plt.ylabel('Frequency')

plt.show()
```



The pronounced peak at the left indicates an abundance of black pixels, while the peak at the right suggests a presence of white pixels. The scarcity of mid-range values implies these images have high contrast with predominantly black backgrounds and lighter features i.e. digits are typically white on a black background, contributing to the stark bimodal distribution observed.

In [11]:
```
# Display Samples Digits

# Number of samples to display for each digit
num_samples = 5

# Calculate the number of unique digits in df
num_digits = df['label'].nunique()

# Plot the digits
fig, axs = plt.subplots(num_digits, num_samples, figsize=(num_samples, num_digits))
for i in range(num_digits):
    digits = df[df['label'] == i].sample(num_samples)
    for j in range(num_samples):
        img_data = digits.iloc[j, 1:].values.reshape(28, 28)
        axs[i, j].imshow(img_data, cmap=plt.cm.binary)
        # Place a text label inside each subplot
        axs[i, j].text(0, -4, str(i), color='red', fontsize=12)
        axs[i, j].axis('off')

plt.show()
```



3. Pre-modelling

In [12]:
```
# Define the features and target variable for Modelling
X = df.drop('label', axis=1).values
y = df['label'].values
```

In [13]:
```
X.shape
```

Out[13]:
```
(42000, 784)
```

In [14]:
```
# Fit and transform the data
X_scaled = X.astype(float) / 255
```

In [15]:
```
# Split the data
X_train, X_valid, y_train, y_valid = train_test_split(X_scaled, y, test_size=0.2, r
```

In [16]:
```
# Read the test_file
test_df = pd.read_csv('test.csv')
print(test_df.shape)
```

```
(28000, 784)
```

In [17]:
```
# Normalize the test values
X_test = test_df.values / 255
```

4. Modelling

4a. Dense Neural Network

The code outlines a method for optimizing a neural network's structure via grid search, focusing on hyperparameters like the number of layers and neurons. Using the `build_model` function, it builds models with different configurations, evaluated against a dataset split into training and validation sets. This approach allows for exhaustive exploration of parameter combinations to identify the most effective neural network architecture. The process employs the Keras library for model construction and optimization, leveraging the `GridSearchCV` class from Scikit-Learn for systematic hyperparameter tuning, aiming to maximize accuracy. Results are collected and displayed, highlighting the impact of different configurations on model performance.

In [18]:
```
# Define a function to build the neural network model
def build_model(n_layers=2, n_neurons=10, learning_rate=3e-3, input_shape=[784] ):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_layers):
        model.add(keras.layers.Dense(n_neurons, activation='relu'))
    model.add(keras.layers.Dense(10, activation='softmax'))
    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, metr
    return model

# Define the parameter grid for the grid search
param_grid = {
    'n_layers': [2, 5],
    'n_neurons': [14, 28, 56]
}

# Initialize the KerasClassifier with build_model function
keras_cls = KerasClassifier(build_model, n_layers=2, n_neurons=10, learning_rate=3e

# Initialize GridSearchCV with the parameter grid and accuracy scoring
grid_search = GridSearchCV(keras_cls, param_grid, cv=3, scoring='accuracy')  # Usir

# Fit the grid search to the data
results = []
for n_layers in param_grid['n_layers']:
```

```
    for n_neurons in param_grid['n_neurons']:
        keras_cls.set_params(n_layers=n_layers, n_neurons=n_neurons)
        start_time = time.time()
        grid_search.fit(X_train, y_train, validation_data=(X_valid, y_valid),
                        callbacks=[keras.callbacks.EarlyStopping(patience=10)])
        elapsed_time = time.time() - start_time

        # Get the best score and model for the current configuration
        best_score = grid_search.best_score_
        best_model_1 = grid_search.best_estimator_.model_

        # Evaluate on training and validation data
        train_accuracy = grid_search.score(X_train, y_train)
        valid_accuracy = grid_search.score(X_valid, y_valid)

        # Store the results
        results.append({
            'Layers': n_layers,
            'Nodes': n_neurons,
            'Time': round(elapsed_time, 2),
            'Training Accuracy': round(train_accuracy, 3),
            'Validation Accuracy': round(valid_accuracy, 3)
        })

# Print the results in a tabular format
print(f"{'Layers':<7} {'Nodes':<6} {'Time':<5} {'Training Accuracy':<17} {'Validati
for res in results:
    print(f"{res['Layers']:<7} {res['Nodes']:<6} {res['Time']:<5} {res['Training Ac
```

```
700/700 [==============================] - 4s 3ms/step - loss: 0.5782 - accuracy:
0.8232 - val_loss: 0.3150 - val_accuracy: 0.9108
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5577 - accuracy:
0.8285 - val_loss: 0.3402 - val_accuracy: 0.9023
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5505 - accuracy:
0.8346 - val_loss: 0.3712 - val_accuracy: 0.8895
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3856 - accuracy:
0.8860 - val_loss: 0.2368 - val_accuracy: 0.9305
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4133 - accuracy:
0.8722 - val_loss: 0.2583 - val_accuracy: 0.9204
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3905 - accuracy:
0.8805 - val_loss: 0.2396 - val_accuracy: 0.9306
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3309 - accuracy:
0.9005 - val_loss: 0.1883 - val_accuracy: 0.9445
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3313 - accuracy:
0.8993 - val_loss: 0.2093 - val_accuracy: 0.9405
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3503 - accuracy:
0.8924 - val_loss: 0.2043 - val_accuracy: 0.9380
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.8428 - accuracy:
0.7213 - val_loss: 0.4570 - val_accuracy: 0.8679
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.7851 - accuracy:
0.7333 - val_loss: 0.4005 - val_accuracy: 0.8858
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.6609 - accuracy:
0.7851 - val_loss: 0.3642 - val_accuracy: 0.8942
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 5s 4ms/step - loss: 0.5975 - accuracy:
0.8068 - val_loss: 0.3478 - val_accuracy: 0.8939
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4927 - accuracy:
0.8511 - val_loss: 0.2874 - val_accuracy: 0.9157
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.5001 - accuracy:
0.8401 - val_loss: 0.2793 - val_accuracy: 0.9151
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4168 - accuracy:
0.8721 - val_loss: 0.2503 - val_accuracy: 0.9287
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3948 - accuracy:
0.8756 - val_loss: 0.2116 - val_accuracy: 0.9387
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4040 - accuracy:
0.8741 - val_loss: 0.2239 - val_accuracy: 0.9348
350/350 [==============================] - 1s 1ms/step
1050/1050 [==============================] - 4s 3ms/step - loss: 0.2890 - accurac
y: 0.9133 - val_loss: 0.1909 - val_accuracy: 0.9435
1050/1050 [==============================] - 1s 1ms/step
263/263 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5798 - accuracy:
0.8154 - val_loss: 0.2955 - val_accuracy: 0.9149
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5449 - accuracy:
0.8375 - val_loss: 0.3366 - val_accuracy: 0.9007
350/350 [==============================] - 0s 1ms/step
```

```
700/700 [==============================] - 3s 3ms/step - loss: 0.5082 - accuracy:
0.8421 - val_loss: 0.3050 - val_accuracy: 0.9094
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3947 - accuracy:
0.8819 - val_loss: 0.2509 - val_accuracy: 0.9273
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4098 - accuracy:
0.8771 - val_loss: 0.2828 - val_accuracy: 0.9170
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4115 - accuracy:
0.8761 - val_loss: 0.2600 - val_accuracy: 0.9212
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3332 - accuracy:
0.9004 - val_loss: 0.1821 - val_accuracy: 0.9433
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3417 - accuracy:
0.8951 - val_loss: 0.2007 - val_accuracy: 0.9408
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3433 - accuracy:
0.8962 - val_loss: 0.1991 - val_accuracy: 0.9429
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 5s 4ms/step - loss: 0.7428 - accuracy:
0.7620 - val_loss: 0.3755 - val_accuracy: 0.8880
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.8222 - accuracy:
0.7164 - val_loss: 0.4931 - val_accuracy: 0.8436
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.6736 - accuracy:
0.7850 - val_loss: 0.4484 - val_accuracy: 0.8686
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4763 - accuracy:
0.8548 - val_loss: 0.2725 - val_accuracy: 0.9210
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.5150 - accuracy:
0.8404 - val_loss: 0.2672 - val_accuracy: 0.9186
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4789 - accuracy:
0.8467 - val_loss: 0.3209 - val_accuracy: 0.9077
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3941 - accuracy:
0.8778 - val_loss: 0.2170 - val_accuracy: 0.9335
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4195 - accuracy:
0.8693 - val_loss: 0.2219 - val_accuracy: 0.9340
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4018 - accuracy:
0.8750 - val_loss: 0.2173 - val_accuracy: 0.9355
350/350 [==============================] - 1s 1ms/step
1050/1050 [==============================] - 4s 3ms/step - loss: 0.2926 - accurac
y: 0.9118 - val_loss: 0.1813 - val_accuracy: 0.9425
1050/1050 [==============================] - 1s 1ms/step
263/263 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4880 - accuracy:
0.8562 - val_loss: 0.3266 - val_accuracy: 0.9057
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5266 - accuracy:
0.8411 - val_loss: 0.3379 - val_accuracy: 0.9020
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5743 - accuracy:
0.8173 - val_loss: 0.3418 - val_accuracy: 0.8957
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4246 - accuracy:
0.8757 - val_loss: 0.2792 - val_accuracy: 0.9193
350/350 [==============================] - 0s 1ms/step
```

```
700/700 [==============================] - 3s 3ms/step - loss: 0.4094 - accuracy:
0.8775 - val_loss: 0.2737 - val_accuracy: 0.9201
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3984 - accuracy:
0.8793 - val_loss: 0.2631 - val_accuracy: 0.9214
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3553 - accuracy:
0.8931 - val_loss: 0.2203 - val_accuracy: 0.9319
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 4s 3ms/step - loss: 0.3356 - accuracy:
0.9013 - val_loss: 0.2241 - val_accuracy: 0.9310
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3379 - accuracy:
0.8979 - val_loss: 0.2319 - val_accuracy: 0.9306
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.6620 - accuracy:
0.7754 - val_loss: 0.3727 - val_accuracy: 0.8881
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.7963 - accuracy:
0.7385 - val_loss: 0.4536 - val_accuracy: 0.8685
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.7145 - accuracy:
0.7635 - val_loss: 0.4441 - val_accuracy: 0.8662
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.5566 - accuracy:
0.8142 - val_loss: 0.2869 - val_accuracy: 0.9099
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4967 - accuracy:
0.8470 - val_loss: 0.3000 - val_accuracy: 0.9075
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.5165 - accuracy:
0.8347 - val_loss: 0.2890 - val_accuracy: 0.9130
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4005 - accuracy:
0.8747 - val_loss: 0.2270 - val_accuracy: 0.9336
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3890 - accuracy:
0.8796 - val_loss: 0.2224 - val_accuracy: 0.9340
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4046 - accuracy:
0.8741 - val_loss: 0.3175 - val_accuracy: 0.8996
350/350 [==============================] - 1s 1ms/step
1050/1050 [==============================] - 4s 3ms/step - loss: 0.2922 - accurac
y: 0.9111 - val_loss: 0.2234 - val_accuracy: 0.9339
1050/1050 [==============================] - 1s 1ms/step
263/263 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5229 - accuracy:
0.8384 - val_loss: 0.3255 - val_accuracy: 0.9086
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5276 - accuracy:
0.8449 - val_loss: 0.3157 - val_accuracy: 0.9115
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5233 - accuracy:
0.8420 - val_loss: 0.3065 - val_accuracy: 0.9101
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4064 - accuracy:
0.8766 - val_loss: 0.2782 - val_accuracy: 0.9162
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3874 - accuracy:
0.8831 - val_loss: 0.2510 - val_accuracy: 0.9240
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4143 - accuracy:
0.8745 - val_loss: 0.2673 - val_accuracy: 0.9231
350/350 [==============================] - 0s 1ms/step
```

```
700/700 [==============================] - 3s 3ms/step - loss: 0.3501 - accuracy:
0.8942 - val_loss: 0.2700 - val_accuracy: 0.9144
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3414 - accuracy:
0.8975 - val_loss: 0.2433 - val_accuracy: 0.9233
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 4s 3ms/step - loss: 0.3340 - accuracy:
0.8975 - val_loss: 0.2213 - val_accuracy: 0.9352
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.7977 - accuracy:
0.7261 - val_loss: 0.4550 - val_accuracy: 0.8668
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.7371 - accuracy:
0.7544 - val_loss: 0.3906 - val_accuracy: 0.8831
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.6652 - accuracy:
0.7699 - val_loss: 0.3213 - val_accuracy: 0.9033
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4874 - accuracy:
0.8458 - val_loss: 0.3579 - val_accuracy: 0.8939
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4807 - accuracy:
0.8506 - val_loss: 0.3002 - val_accuracy: 0.9086
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.5409 - accuracy:
0.8242 - val_loss: 0.2763 - val_accuracy: 0.9199
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3959 - accuracy:
0.8797 - val_loss: 0.2097 - val_accuracy: 0.9376
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3909 - accuracy:
0.8786 - val_loss: 0.2590 - val_accuracy: 0.9239
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3935 - accuracy:
0.8770 - val_loss: 0.2261 - val_accuracy: 0.9311
350/350 [==============================] - 1s 1ms/step
1050/1050 [==============================] - 5s 4ms/step - loss: 0.3406 - accurac
y: 0.8947 - val_loss: 0.2160 - val_accuracy: 0.9368
1050/1050 [==============================] - 2s 1ms/step
263/263 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5578 - accuracy:
0.8313 - val_loss: 0.3335 - val_accuracy: 0.9032
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4833 - accuracy:
0.8528 - val_loss: 0.3043 - val_accuracy: 0.9101
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5138 - accuracy:
0.8396 - val_loss: 0.3153 - val_accuracy: 0.9096
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4052 - accuracy:
0.8778 - val_loss: 0.2397 - val_accuracy: 0.9315
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4016 - accuracy:
0.8781 - val_loss: 0.2417 - val_accuracy: 0.9258
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4159 - accuracy:
0.8729 - val_loss: 0.2418 - val_accuracy: 0.9262
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3421 - accuracy:
0.8950 - val_loss: 0.1901 - val_accuracy: 0.9436
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3292 - accuracy:
0.9010 - val_loss: 0.2014 - val_accuracy: 0.9393
350/350 [==============================] - 0s 1ms/step
```

```
700/700 [==============================] - 3s 3ms/step - loss: 0.3329 - accuracy:
0.8990 - val_loss: 0.2175 - val_accuracy: 0.9363
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.6881 - accuracy:
0.7767 - val_loss: 0.3584 - val_accuracy: 0.8942
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 5s 4ms/step - loss: 0.7496 - accuracy:
0.7452 - val_loss: 0.5137 - val_accuracy: 0.8432
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.7356 - accuracy:
0.7630 - val_loss: 0.4254 - val_accuracy: 0.8823
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.5336 - accuracy:
0.8285 - val_loss: 0.2831 - val_accuracy: 0.9175
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4975 - accuracy:
0.8461 - val_loss: 0.2694 - val_accuracy: 0.9224
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4776 - accuracy:
0.8487 - val_loss: 0.2987 - val_accuracy: 0.9115
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3966 - accuracy:
0.8783 - val_loss: 0.2674 - val_accuracy: 0.9227
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4024 - accuracy:
0.8741 - val_loss: 0.2260 - val_accuracy: 0.9317
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4165 - accuracy:
0.8713 - val_loss: 0.2247 - val_accuracy: 0.9352
350/350 [==============================] - 1s 1ms/step
1050/1050 [==============================] - 4s 3ms/step - loss: 0.2843 - accurac
y: 0.9139 - val_loss: 0.1887 - val_accuracy: 0.9439
1050/1050 [==============================] - 1s 1ms/step
263/263 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.4891 - accuracy:
0.8528 - val_loss: 0.3032 - val_accuracy: 0.9123
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5146 - accuracy:
0.8416 - val_loss: 0.2904 - val_accuracy: 0.9163
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.5143 - accuracy:
0.8414 - val_loss: 0.3155 - val_accuracy: 0.9067
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3834 - accuracy:
0.8859 - val_loss: 0.2759 - val_accuracy: 0.9157
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3987 - accuracy:
0.8807 - val_loss: 0.2341 - val_accuracy: 0.9315
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3990 - accuracy:
0.8775 - val_loss: 0.2448 - val_accuracy: 0.9264
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3388 - accuracy:
0.8956 - val_loss: 0.2108 - val_accuracy: 0.9348
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3450 - accuracy:
0.8958 - val_loss: 0.2198 - val_accuracy: 0.9336
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 3s 3ms/step - loss: 0.3472 - accuracy:
0.8938 - val_loss: 0.2027 - val_accuracy: 0.9395
350/350 [==============================] - 0s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.6822 - accuracy:
0.7850 - val_loss: 0.3941 - val_accuracy: 0.8840
350/350 [==============================] - 1s 1ms/step
```

```
700/700 [==============================] - 4s 4ms/step - loss: 0.6906 - accuracy:
0.7881 - val_loss: 0.4350 - val_accuracy: 0.8727
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.7254 - accuracy:
0.7702 - val_loss: 0.4576 - val_accuracy: 0.8694
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4960 - accuracy:
0.8417 - val_loss: 0.2751 - val_accuracy: 0.9189
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4988 - accuracy:
0.8421 - val_loss: 0.3411 - val_accuracy: 0.8965
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4915 - accuracy:
0.8481 - val_loss: 0.3908 - val_accuracy: 0.8840
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4007 - accuracy:
0.8736 - val_loss: 0.2435 - val_accuracy: 0.9306
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.4131 - accuracy:
0.8720 - val_loss: 0.2366 - val_accuracy: 0.9306
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3923 - accuracy:
0.8775 - val_loss: 0.2060 - val_accuracy: 0.9379
350/350 [==============================] - 1s 1ms/step
1050/1050 [==============================] - 4s 3ms/step - loss: 0.2876 - accurac
y: 0.9118 - val_loss: 0.1733 - val_accuracy: 0.9461
1050/1050 [==============================] - 1s 1ms/step
263/263 [==============================] - 0s 1ms/step
Layers  Nodes  Time  Training Accuracy Validation Accuracy
2       14     93.63 0.953             0.943
2       28     90.61 0.956             0.942
2       56     91.84 0.945             0.934
5       14     92.83 0.948             0.937
5       28     91.05 0.953             0.944
5       56     92.11 0.957             0.946
```

The search tested combinations of 2 and 5 layers with 14, 28, and 56 nodes, assessing their impact on model performance over time, and evaluating training and validation accuracy. Models with 5 layers and 56 nodes achieved the highest validation accuracy (0.946) and training accuracy (0.957), indicating a potential sweet spot in complexity for this particular dataset. Conversely, models with 2 layers and 56 nodes showed lower validation accuracy (0.934), suggesting that simply increasing nodes without adjusting layers might not always yield better results.

In [19]:
```python
# Identify the model with the highest validation accuracy
best_model_1_results = max(results, key=lambda x: x['Validation Accuracy'])

print("\nBest Model:")
print(f"Layers: {best_model_1_results['Layers']}")
print(f"Nodes: {best_model_1_results['Nodes']}")
print(f"Time: {best_model_1_results['Time']}")
print(f"Training Accuracy: {best_model_1_results['Training Accuracy']}")
print(f"Validation Accuracy: {best_model_1_results['Validation Accuracy']}")
```

```
Best Model:
Layers: 5
Nodes: 56
Time: 92.11
Training Accuracy: 0.957
Validation Accuracy: 0.946
```

The best model, identified through grid search, consists of 5 layers with 56 nodes each, taking 92.11 seconds to train. It achieved a high training accuracy of 95.7% and a validation accuracy of 94.6%, indicating excellent performance and generalization ability on unseen data, making it the optimal configuration tested.

In [21]:
```
# Fit test data to the Best Neural Net Model and write the output file
predictions = best_model_1.predict(X_test).argmax(axis=1)
n_net = pd.DataFrame({'ImageId': range(1, len(predictions)+1), 'Label': predictions
n_net.to_csv('n_net.csv', index=False)
```

```
875/875 [==============================] - 1s 1ms/step
```

In [22]:
```
# Assuming X_train and X_valid are your original data with shapes (num_samples, 784
X_train = X_train.reshape((-1, 28, 28, 1))  # Reshape to (num_samples, 28, 28, 1)
X_valid = X_valid.reshape((-1, 28, 28, 1))  # Reshape to (num_samples, 28, 28, 1)
```

4b. Convolutional Neural Network (CNN)

This code introduces an approach to optimize a Convolutional Neural Network (CNN) model for image classification tasks. It involves defining a function, `build_cnn_model`, which constructs a CNN with customizable parameters such as the number of convolutional layers, number of filters, kernel size, learning rate, and input shape. The model aims to maximize accuracy through iterative training and validation, utilizing Keras for model construction and Adam for optimization. A grid search strategy is employed to systematically explore different configurations of convolutional layers and filters, facilitated by the `GridSearchCV` tool from Scikit-Learn, with the objective of finding the optimal model settings for enhanced performance on a given dataset.

In [23]:
```
# Define a function to build the CNN model
def build_cnn_model(n_conv_layers=2, n_filters=32, kernel_size=3, learning_rate=1e-
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_conv_layers):
        model.add(keras.layers.Conv2D(n_filters, kernel_size, activation='relu', pa
        model.add(keras.layers.MaxPooling2D(2))
    model.add(keras.layers.Flatten())
    model.add(keras.layers.Dense(128, activation='relu'))
    model.add(keras.layers.Dense(10, activation='softmax'))
    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, metr
    return model

# Define the parameter grid for the grid search
param_grid = {
    'n_conv_layers': [2, 3],
    'n_filters': [28, 56]
}

# Initialize the KerasClassifier with build_cnn_model function
cnn_cls = KerasClassifier(build_cnn_model, n_conv_layers=2, n_filters=32, kernel_si

# Initialize GridSearchCV with the parameter grid and accuracy scoring
grid_search = GridSearchCV(estimator=cnn_cls, param_grid=param_grid, cv=3, scoring=

# Fit the grid search to the data
results = []
for n_conv_layers in param_grid['n_conv_layers']:
```

```
    for n_filters in param_grid['n_filters']:
        cnn_cls.set_params(n_conv_layers=n_conv_layers, n_filters=n_filters)
        start_time = time.time()
        grid_search.fit(X_train, y_train, validation_data=(X_valid, y_valid),
                        callbacks=[keras.callbacks.EarlyStopping(patience=10)])
        elapsed_time = time.time() - start_time

        # Get the best score and model for the current configuration
        best_score = grid_search.best_score_
        best_model_2 = grid_search.best_estimator_.model_

        # Evaluate on training and validation data
        train_accuracy = grid_search.score(X_train, y_train)
        valid_accuracy = grid_search.score(X_valid, y_valid)

        # Store the results
        results.append({
            'Conv Layers': n_conv_layers,
            'Filters': n_filters,
            'Time': round(elapsed_time, 2),
            'Training Accuracy': round(train_accuracy, 3),
            'Validation Accuracy': round(valid_accuracy, 3)
        })

# Print the results in a tabular format
print(f"{'Conv Layers':<12} {'Filters':<7} {'Time':<5} {'Training Accuracy':<17} {'
for res in results:
    print(f"{res['Conv Layers']:<12} {res['Filters']:<7} {res['Time']:<5} {res['Tra
```

```
700/700 [==============================] - 6s 4ms/step - loss: 0.2506 - accuracy:
0.9244 - val_loss: 0.0984 - val_accuracy: 0.9694
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2351 - accuracy:
0.9303 - val_loss: 0.1045 - val_accuracy: 0.9688
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2582 - accuracy:
0.9212 - val_loss: 0.1051 - val_accuracy: 0.9675
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2110 - accuracy:
0.9336 - val_loss: 0.0767 - val_accuracy: 0.9750
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2100 - accuracy:
0.9349 - val_loss: 0.0816 - val_accuracy: 0.9735
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2120 - accuracy:
0.9347 - val_loss: 0.0744 - val_accuracy: 0.9764
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3135 - accuracy:
0.8996 - val_loss: 0.1113 - val_accuracy: 0.9656
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3039 - accuracy:
0.9075 - val_loss: 0.1397 - val_accuracy: 0.9573
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3123 - accuracy:
0.9018 - val_loss: 0.1051 - val_accuracy: 0.9665
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 5s 4ms/step - loss: 0.2454 - accuracy:
0.9214 - val_loss: 0.1109 - val_accuracy: 0.9625
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2305 - accuracy:
0.9298 - val_loss: 0.0797 - val_accuracy: 0.9750
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2455 - accuracy:
0.9224 - val_loss: 0.0847 - val_accuracy: 0.9750
350/350 [==============================] - 1s 2ms/step
1050/1050 [==============================] - 5s 4ms/step - loss: 0.1657 - accurac
y: 0.9486 - val_loss: 0.0534 - val_accuracy: 0.9835
1050/1050 [==============================] - 2s 1ms/step
263/263 [==============================] - 0s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2641 - accuracy:
0.9206 - val_loss: 0.0993 - val_accuracy: 0.9706
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2406 - accuracy:
0.9283 - val_loss: 0.1151 - val_accuracy: 0.9651
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2508 - accuracy:
0.9251 - val_loss: 0.1329 - val_accuracy: 0.9585
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2158 - accuracy:
0.9340 - val_loss: 0.0794 - val_accuracy: 0.9761
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2032 - accuracy:
0.9362 - val_loss: 0.0783 - val_accuracy: 0.9749
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2113 - accuracy:
0.9334 - val_loss: 0.0796 - val_accuracy: 0.9745
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3250 - accuracy:
0.8979 - val_loss: 0.1081 - val_accuracy: 0.9656
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3022 - accuracy:
0.9069 - val_loss: 0.0933 - val_accuracy: 0.9736
350/350 [==============================] - 1s 2ms/step
```

```
700/700 [==============================] - 4s 4ms/step - loss: 0.3202 - accuracy:
0.8989 - val_loss: 0.1248 - val_accuracy: 0.9624
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2334 - accuracy:
0.9272 - val_loss: 0.0807 - val_accuracy: 0.9750
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2350 - accuracy:
0.9270 - val_loss: 0.0734 - val_accuracy: 0.9770
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 6s 4ms/step - loss: 0.2391 - accuracy:
0.9245 - val_loss: 0.0890 - val_accuracy: 0.9727
350/350 [==============================] - 1s 2ms/step
1050/1050 [==============================] - 6s 4ms/step - loss: 0.1990 - accurac
y: 0.9375 - val_loss: 0.0842 - val_accuracy: 0.9745
1050/1050 [==============================] - 2s 2ms/step
263/263 [==============================] - 0s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2612 - accuracy:
0.9226 - val_loss: 0.0981 - val_accuracy: 0.9689
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2539 - accuracy:
0.9237 - val_loss: 0.1027 - val_accuracy: 0.9681
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2507 - accuracy:
0.9250 - val_loss: 0.0935 - val_accuracy: 0.9724
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2105 - accuracy:
0.9332 - val_loss: 0.0845 - val_accuracy: 0.9740
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2087 - accuracy:
0.9361 - val_loss: 0.0832 - val_accuracy: 0.9742
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2101 - accuracy:
0.9326 - val_loss: 0.0815 - val_accuracy: 0.9758
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3040 - accuracy:
0.9053 - val_loss: 0.1130 - val_accuracy: 0.9667
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3072 - accuracy:
0.9033 - val_loss: 0.1166 - val_accuracy: 0.9648
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3036 - accuracy:
0.9026 - val_loss: 0.1026 - val_accuracy: 0.9675
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2426 - accuracy:
0.9225 - val_loss: 0.0896 - val_accuracy: 0.9731
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2425 - accuracy:
0.9233 - val_loss: 0.0879 - val_accuracy: 0.9711
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2420 - accuracy:
0.9214 - val_loss: 0.0803 - val_accuracy: 0.9758
350/350 [==============================] - 1s 2ms/step
1050/1050 [==============================] - 5s 4ms/step - loss: 0.1675 - accurac
y: 0.9495 - val_loss: 0.0751 - val_accuracy: 0.9762
1050/1050 [==============================] - 2s 2ms/step
263/263 [==============================] - 0s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2474 - accuracy:
0.9248 - val_loss: 0.0934 - val_accuracy: 0.9729
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2542 - accuracy:
0.9208 - val_loss: 0.0920 - val_accuracy: 0.9724
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2451 - accuracy:
0.9256 - val_loss: 0.0842 - val_accuracy: 0.9739
```

```
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2221 - accuracy:
0.9292 - val_loss: 0.0785 - val_accuracy: 0.9749
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2046 - accuracy:
0.9361 - val_loss: 0.0721 - val_accuracy: 0.9779
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2124 - accuracy:
0.9336 - val_loss: 0.0926 - val_accuracy: 0.9713
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 5ms/step - loss: 0.3152 - accuracy:
0.9009 - val_loss: 0.1308 - val_accuracy: 0.9582
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3132 - accuracy:
0.9027 - val_loss: 0.1020 - val_accuracy: 0.9715
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.3239 - accuracy:
0.8972 - val_loss: 0.1102 - val_accuracy: 0.9673
350/350 [==============================] - 1s 1ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2358 - accuracy:
0.9275 - val_loss: 0.0726 - val_accuracy: 0.9750
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2436 - accuracy:
0.9240 - val_loss: 0.1001 - val_accuracy: 0.9698
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 4s 4ms/step - loss: 0.2524 - accuracy:
0.9200 - val_loss: 0.1088 - val_accuracy: 0.9662
350/350 [==============================] - 1s 2ms/step
1050/1050 [==============================] - 5s 4ms/step - loss: 0.1699 - accurac
y: 0.9475 - val_loss: 0.0682 - val_accuracy: 0.9777
1050/1050 [==============================] - 2s 2ms/step
263/263 [==============================] - 0s 2ms/step
Conv Layers  Filters Time  Training Accuracy Validation Accuracy
2            28      71.71 0.988             0.983
2            56      70.75 0.975             0.975
3            28      69.24 0.981             0.976
3            56      70.22 0.982             0.978
```

The summary details the outcomes of optimizing a Convolutional Neural Network (CNN) through grid search, varying the number of convolutional layers and filters. The configurations explored include 2 or 3 convolutional layers with 28 or 56 filters. The best model, with 2 convolutional layers and 28 filters, achieved the highest training accuracy of 98.8% and validation accuracy of 98.3%, indicating superior performance and generalizability. Models with more filters or layers did not necessarily perform better, highlighting the importance of finding the right balance between model complexity and efficiency for optimal performance.

In [24]:
```python
# Identify the model with the highest validation accuracy
best_model_2_results = max(results, key=lambda x: x['Validation Accuracy'])

print("\nBest Model:")
print(f"Conv Layers: {best_model_2_results['Conv Layers']}")
print(f"Filters: {best_model_2_results['Filters']}")
print(f"Time: {best_model_2_results['Time']}")
print(f"Training Accuracy: {best_model_2_results['Training Accuracy']}")
print(f"Validation Accuracy: {best_model_2_results['Validation Accuracy']}")
```

```
Best Model:
Conv Layers: 2
Filters: 28
Time: 71.71
Training Accuracy: 0.988
Validation Accuracy: 0.983
```

The optimal CNN model, determined through grid search, features 2 convolutional layers with 28 filters each, trained in 71.71 seconds. It showcases outstanding efficiency and effectiveness, achieving a high training accuracy of 98.8% and a validation accuracy of 98.3%. This configuration excels in balancing complexity with performance, making it highly effective for the task.

In [26]:
```
# Reshape test data
X_test = X_test.reshape((-1, 28, 28, 1))

# Fit the test data to the best model
predictions = grid_search.best_estimator_.model_.predict(X_test)

# Convert the predictions from one-hot encoded class probabilities to class labels
labels = np.argmax(predictions, axis=-1)

# Now, create the DataFrame using the 1-dimensional labels
cnn = pd.DataFrame({'ImageId': range(1, len(labels)+1), 'Label': labels})

# Save the DataFrame to a CSV file
cnn.to_csv('cnn.csv', index=False)
```

```
875/875 [==============================] - 1s 2ms/step
```

4c. Long Short-Term Memory (LSTM)

We are designing and optimizing a Long Short-Term Memory (LSTM) neural network model for sequence data processing. The `build_lstm_model` function is central to this process, enabling the creation of models with varying numbers of LSTM layers and units, tailored to the specific structure of the input data. By leveraging a grid search technique, the code aims to find the optimal combination of LSTM layers and units that maximizes model accuracy. This approach is facilitated by Keras for model construction and optimization, and the search for the best hyperparameters is conducted using Scikit-Learn's `GridSearchCV`, focusing on maximizing accuracy. The process not only involves model training and validation but also a thorough evaluation of the results to identify the configuration that offers the best performance on the given dataset.

In [27]:
```
# Define a function to build the LSTM model
def build_lstm_model(n_lstm_layers=1, n_units=50, learning_rate=1e-3, input_shape=[
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_lstm_layers):
        # Return sequences for all but last LSTM layer
        return_sequences = layer < n_lstm_layers - 1
        model.add(keras.layers.LSTM(n_units, return_sequences=return_sequences))
    model.add(keras.layers.Flatten()) # Only necessary if you have more than 1 LSTM
    model.add(keras.layers.Dense(128, activation='relu'))
    model.add(keras.layers.Dense(10, activation='softmax'))
    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, metr
    return model
```

```
# Define the parameter grid for the grid search
param_grid = {
    'n_lstm_layers': [1, 2],
    'n_units': [50, 100]
}

# Initialize the KerasClassifier with build_lstm_model function
lstm_cls = KerasClassifier(build_lstm_model, n_lstm_layers=1, n_units=50, learning_

# Initialize GridSearchCV with the parameter grid and accuracy scoring
grid_search = GridSearchCV(estimator=lstm_cls, param_grid=param_grid, cv=3, scoring

# Fit the grid search to the data
results = []
for n_lstm_layers in param_grid['n_lstm_layers']:
    for n_units in param_grid['n_units']:
        lstm_cls.set_params(n_lstm_layers=n_lstm_layers, n_units=n_units)
        start_time = time.time()
        grid_search.fit(X_train, y_train, validation_data=(X_valid, y_valid),
                        callbacks=[keras.callbacks.EarlyStopping(patience=10)])
        elapsed_time = time.time() - start_time

        # Get the best score and model for the current configuration
        best_score = grid_search.best_score_
        best_model = grid_search.best_estimator_.model_

        # Evaluate on training and validation data
        train_accuracy = grid_search.score(X_train, y_train)
        valid_accuracy = grid_search.score(X_valid, y_valid)

        # Store the results
        results.append({
            'LSTM Layers': n_lstm_layers,
            'Units': n_units,
            'Time': round(elapsed_time, 2),
            'Training Accuracy': round(train_accuracy, 3),
            'Validation Accuracy': round(valid_accuracy, 3)
        })

# Print the results in a tabular format
print(f"{'LSTM Layers':<12} {'Units':<6} {'Time':<5} {'Training Accuracy':<17} {'Va
for res in results:
    print(f"{res['LSTM Layers']:<12} {res['Units']:<6} {res['Time']:<5} {res['Trair
```

```
700/700 [==============================] - 6s 6ms/step - loss: 0.7619 - accuracy:
0.7430 - val_loss: 0.3275 - val_accuracy: 0.8937
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 7s 6ms/step - loss: 0.7544 - accuracy:
0.7450 - val_loss: 0.3321 - val_accuracy: 0.8968
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 6s 5ms/step - loss: 0.7877 - accuracy:
0.7329 - val_loss: 0.3779 - val_accuracy: 0.8865
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 6s 6ms/step - loss: 0.6416 - accuracy:
0.7835 - val_loss: 0.2419 - val_accuracy: 0.9249
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 6s 5ms/step - loss: 0.6604 - accuracy:
0.7763 - val_loss: 0.2536 - val_accuracy: 0.9210
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 5s 5ms/step - loss: 0.6659 - accuracy:
0.7746 - val_loss: 0.2508 - val_accuracy: 0.9246
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.7353 - accuracy:
0.7503 - val_loss: 0.3038 - val_accuracy: 0.9080
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.7028 - accuracy:
0.7646 - val_loss: 0.2888 - val_accuracy: 0.9060
350/350 [==============================] - 1s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.7721 - accuracy:
0.7390 - val_loss: 0.3097 - val_accuracy: 0.9033
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.5652 - accuracy:
0.8112 - val_loss: 0.2012 - val_accuracy: 0.9398
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 10s 8ms/step - loss: 0.5733 - accuracy:
0.8069 - val_loss: 0.2370 - val_accuracy: 0.9252
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.6319 - accuracy:
0.7854 - val_loss: 0.2560 - val_accuracy: 0.9157
350/350 [==============================] - 2s 3ms/step
1050/1050 [==============================] - 10s 7ms/step - loss: 0.4771 - accurac
y: 0.8413 - val_loss: 0.1813 - val_accuracy: 0.9417
1050/1050 [==============================] - 3s 3ms/step
263/263 [==============================] - 1s 3ms/step
700/700 [==============================] - 6s 6ms/step - loss: 0.7847 - accuracy:
0.7334 - val_loss: 0.3896 - val_accuracy: 0.8731
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 5s 5ms/step - loss: 0.7609 - accuracy:
0.7481 - val_loss: 0.3110 - val_accuracy: 0.9031
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 6s 5ms/step - loss: 0.7575 - accuracy:
0.7466 - val_loss: 0.2999 - val_accuracy: 0.9068
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 6s 6ms/step - loss: 0.6537 - accuracy:
0.7821 - val_loss: 0.2382 - val_accuracy: 0.9249
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 6s 6ms/step - loss: 0.6224 - accuracy:
0.7904 - val_loss: 0.2388 - val_accuracy: 0.9258
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 6s 6ms/step - loss: 0.6214 - accuracy:
0.7911 - val_loss: 0.2997 - val_accuracy: 0.9040
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.7112 - accuracy:
0.7610 - val_loss: 0.2814 - val_accuracy: 0.9129
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 10s 11ms/step - loss: 0.7461 - accurac
y: 0.7454 - val_loss: 0.2936 - val_accuracy: 0.9068
350/350 [==============================] - 2s 3ms/step
```

```
700/700 [==============================] - 8s 8ms/step - loss: 0.7333 - accuracy:
0.7531 - val_loss: 0.4095 - val_accuracy: 0.8649
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.5763 - accuracy:
0.8050 - val_loss: 0.2179 - val_accuracy: 0.9325
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 9s 8ms/step - loss: 0.5558 - accuracy:
0.8163 - val_loss: 0.2575 - val_accuracy: 0.9207
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.5833 - accuracy:
0.8013 - val_loss: 0.2295 - val_accuracy: 0.9292
350/350 [==============================] - 2s 3ms/step
1050/1050 [==============================] - 11s 7ms/step - loss: 0.4434 - accurac
y: 0.8539 - val_loss: 0.1917 - val_accuracy: 0.9407
1050/1050 [==============================] - 4s 3ms/step
263/263 [==============================] - 1s 3ms/step
700/700 [==============================] - 6s 6ms/step - loss: 0.7725 - accuracy:
0.7411 - val_loss: 0.3581 - val_accuracy: 0.8860
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 5s 5ms/step - loss: 0.7538 - accuracy:
0.7498 - val_loss: 0.3197 - val_accuracy: 0.9006
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 6s 6ms/step - loss: 0.7955 - accuracy:
0.7311 - val_loss: 0.3381 - val_accuracy: 0.8960
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 5s 5ms/step - loss: 0.6332 - accuracy:
0.7866 - val_loss: 0.2882 - val_accuracy: 0.9088
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 5s 6ms/step - loss: 0.6917 - accuracy:
0.7636 - val_loss: 0.2674 - val_accuracy: 0.9173
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 5s 5ms/step - loss: 0.5909 - accuracy:
0.8048 - val_loss: 0.2293 - val_accuracy: 0.9290
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 10s 8ms/step - loss: 0.7609 - accuracy:
0.7425 - val_loss: 0.3390 - val_accuracy: 0.8914
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.6247 - accuracy:
0.7905 - val_loss: 0.2716 - val_accuracy: 0.9165
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.7118 - accuracy:
0.7604 - val_loss: 0.3341 - val_accuracy: 0.8888
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.5920 - accuracy:
0.8021 - val_loss: 0.2647 - val_accuracy: 0.9152
350/350 [==============================] - 1s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.6070 - accuracy:
0.7967 - val_loss: 0.2287 - val_accuracy: 0.9283
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.6141 - accuracy:
0.7921 - val_loss: 0.2144 - val_accuracy: 0.9321
350/350 [==============================] - 2s 3ms/step
1050/1050 [==============================] - 10s 7ms/step - loss: 0.4665 - accurac
y: 0.8450 - val_loss: 0.1763 - val_accuracy: 0.9462
1050/1050 [==============================] - 3s 3ms/step
263/263 [==============================] - 1s 3ms/step
700/700 [==============================] - 5s 5ms/step - loss: 0.8184 - accuracy:
0.7176 - val_loss: 0.3467 - val_accuracy: 0.8898
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 5s 5ms/step - loss: 0.7738 - accuracy:
0.7383 - val_loss: 0.3459 - val_accuracy: 0.8904
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 6s 6ms/step - loss: 0.8012 - accuracy:
0.7244 - val_loss: 0.3646 - val_accuracy: 0.8825
```

```
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 5s 5ms/step - loss: 0.6762 - accuracy:
0.7752 - val_loss: 0.2538 - val_accuracy: 0.9240
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 5s 5ms/step - loss: 0.6510 - accuracy:
0.7811 - val_loss: 0.2998 - val_accuracy: 0.9054
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 6s 6ms/step - loss: 0.6436 - accuracy:
0.7834 - val_loss: 0.2601 - val_accuracy: 0.9192
350/350 [==============================] - 1s 2ms/step
700/700 [==============================] - 9s 9ms/step - loss: 0.7260 - accuracy:
0.7542 - val_loss: 0.3114 - val_accuracy: 0.9012
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 9s 9ms/step - loss: 0.7219 - accuracy:
0.7559 - val_loss: 0.3162 - val_accuracy: 0.9001
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 9s 8ms/step - loss: 0.7455 - accuracy:
0.7468 - val_loss: 0.3084 - val_accuracy: 0.8963
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.6135 - accuracy:
0.7921 - val_loss: 0.2094 - val_accuracy: 0.9348
350/350 [==============================] - 1s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.6370 - accuracy:
0.7854 - val_loss: 0.2520 - val_accuracy: 0.9236
350/350 [==============================] - 2s 3ms/step
700/700 [==============================] - 8s 8ms/step - loss: 0.6001 - accuracy:
0.7993 - val_loss: 0.2188 - val_accuracy: 0.9293
350/350 [==============================] - 2s 3ms/step
1050/1050 [==============================] - 11s 8ms/step - loss: 0.4716 - accurac
y: 0.8438 - val_loss: 0.1598 - val_accuracy: 0.9494
1050/1050 [==============================] - 3s 3ms/step
263/263 [==============================] - 1s 3ms/step
LSTM Layers  Units  Time  Training Accuracy Validation Accuracy
1            50     124.74 0.948             0.942
1            100    123.84 0.942             0.941
2            50     121.13 0.949             0.946
2            100    125.61 0.955             0.949
```

The summary showcases the results of optimizing a Long Short-Term Memory (LSTM) network through varying the number of layers and units. Models were assessed based on their training and validation accuracies. The configuration with 2 LSTM layers and 100 units emerged as the most effective, achieving the highest training accuracy of 95.5% and validation accuracy of 94.9%. This indicates that increasing both the number of layers and the complexity (units) of the LSTM model can lead to improved performance on the dataset, suggesting a better capability to capture complex patterns and dependencies in the data.

In [28]:
```python
# Identify the model with the highest validation accuracy
best_lstm_results = max(results, key=lambda x: x['Validation Accuracy'])

print("\nBest LSTM Model:")
print(f"LSTM Layers: {best_lstm_results['LSTM Layers']}")
print(f"Units: {best_lstm_results['Units']}")
print(f"Time: {best_lstm_results['Time']}")
print(f"Training Accuracy: {best_lstm_results['Training Accuracy']}")
print(f"Validation Accuracy: {best_lstm_results['Validation Accuracy']}")
```

```
Best LSTM Model:
LSTM Layers: 2
Units: 100
Time: 125.61
Training Accuracy: 0.955
Validation Accuracy: 0.949
```

The optimal LSTM model, identified through systematic testing, comprises 2 layers and 100 units, requiring 125.61 seconds to train. It stands out with a high training accuracy of 95.5% and a validation accuracy of 94.9%, demonstrating its superior ability to learn and generalize from the sequence data efficiently and effectively.

In [30]:
```
# Fit the test data to the best LSTM model
predictions = grid_search.best_estimator_.model_.predict(X_test)  # Notice the char

# Convert the predictions from probabilities to class labels
labels = np.argmax(predictions, axis=1)

# Create a DataFrame using the 1-dimensional labels
lstm_results = pd.DataFrame({'ImageId': range(1, len(labels) + 1), 'Label': labels}

# Save the DataFrame to a CSV file
lstm_results.to_csv('lstm_results.csv', index=False)
```

```
875/875 [==============================] - 2s 3ms/step
```

5. Conclusion

Analyzing the outcomes from experiments with traditional densely connected Neural Networks alongside those of Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks provides a comprehensive view of model optimization across different data types and architectural paradigms. The densely connected network optimization highlighted the effectiveness of deeper models with more neurons, particularly showing that a configuration with 5 layers and 56 neurons outperformed others in terms of validation accuracy. This finding aligns with the broader theme observed across the experiments: the importance of tailoring the network architecture to the task at hand for optimal performance.

The CNNs proved most adept at image processing tasks, optimizing at a relatively simpler architecture with 2 layers and 28 filters, emphasizing the value of spatial feature extraction in image data. In contrast, the LSTMs, with their ability to process sequential data, reached peak performance with 2 layers and 100 units, showcasing their strength in capturing temporal dependencies.

Across all models, the grid search strategy was essential in navigating the complex landscape of hyperparameters to identify configurations that maximize accuracy. This exploration underscores the critical role of architectural design in achieving high performance, demonstrating that while there is no universal solution, certain patterns—such as the depth of the network and the number of processing units—consistently influence outcomes. These insights reinforce the principle that effective model optimization is contingent on understanding the unique characteristics of the dataset and task, guiding the selection of the most suitable neural network architecture.