EXPERIMENT 8	Convolutional Neural Network (CNN) on Image Datasets (e.g. MNIST / CIFAR)

Name: Atharva Lotankar
Class: D15C; **Roll Number:** 31
Date: 12 / 02 / 2026
Subject: Machine and Deep Learning Lab

Aim: To design, implement, and train a Convolutional Neural Network (CNN) using image datasets such as MNIST or CIFAR-10 for accurate image classification.

Dataset Source:

The experiment utilizes the MNIST (Modified National Institute of Standards and Technology) dataset, specifically the CSV structured version for streamlined processing.

- *Source Link*: https://www.kaggle.com/datasets/oddrationale/mnist-in-csv

Dataset Description:

The MNIST dataset is the definitive benchmark for evaluating image processing systems. It consists of grayscale images of handwritten digits (0 through 9). The dataset is pre-formatted into a 2D array where each row represents one image. The pixels are flattened from a 28 × 28 grid into 784 individual features, with values ranging from 0 (white) to 255 (black).

Feature	Description	Data Type	Unit	Type
Total Size	Total number of images in the dataset (60,000 training and 10,000 testing)	Integer	Images (count)	Dataset Metadata
Input Features	784 pixel values representing a flattened 28 × 28 grayscale image	Integer	Pixel Intensity (0–255)	Independent Variable
Target Variable	Digit label corresponding to each image (0–9)	Integer	Class Label	Dependent Variable
Colour Space	Image colour format	Categorical (String)	Grayscale (Single Channel)	Image Property
Data Format	Storage structure of the dataset	Categorical (String)	.csv	Storage Format

Theory:

Designing and implementing a Convolutional Neural Network (CNN) for image classification involves leveraging its ability to automatically learn hierarchical feature representations from raw pixel data. Unlike traditional machine learning methods that rely on handcrafted features, CNNs use convolutional layers to extract spatial patterns such as edges, textures, and shapes directly from images. A typical CNN architecture consists of convolutional layers (with learnable filters), activation functions like ReLU to introduce non-linearity, pooling layers for spatial down sampling, and fully connected layers for final classification. When applied to benchmark datasets such as *MNIST or CIFAR-10*, CNNs can effectively learn distinguishing features—digits in grayscale images for MNIST and small coloured object images for CIFAR-10—by optimizing filter weights through backpropagation and gradient descent.

Training a CNN involves splitting the dataset into training, validation, and test sets to ensure generalization and prevent overfitting. The network processes input images through forward propagation to produce predictions, computes loss using a function such as cross-entropy, and updates parameters via optimization algorithms like stochastic gradient descent (SGD) or Adam. Techniques such as data augmentation, dropout, and batch normalization are often incorporated to enhance robustness and convergence speed. For simpler datasets like MNIST, shallow CNN architectures can achieve high accuracy due to the dataset's low complexity, whereas CIFAR-10 typically requires deeper architectures to capture more complex visual patterns. Proper hyperparameter tuning—including learning rate, batch size, number of filters, and epochs—is critical to achieving accurate and reliable image classification performance.

Mathematical Formulation of the Algorithms:

A Convolutional Neural Network (CNN) operates through three primary mathematical operations:

- *Convolution Operation (*):* Applies a filter (W) to the input image (X) to create a feature map (Z).

$$Z_{i,j} = (X * W)_{i,j} = \sum_m \sum_n X_{i+m,j+n} \cdot W_{m,n} + b$$

- *Activation Function (ReLU):* Introduces non-linearity, allowing the model to learn complex patterns.

$$f(z) = \max(0, z)$$

- *Pooling (Downsampling):* Reduces spatial dimensions while retaining critical information. For Max Pooling:

$$Y = \max(\text{Region of } Z)$$

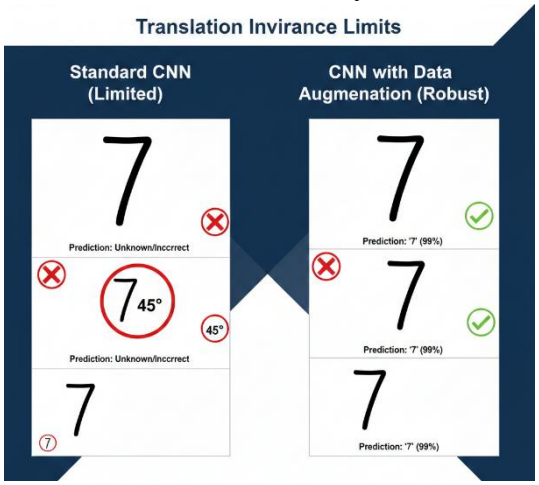
- *Softmax Output:* Converts the final dense layer values into probabilities for the 10 classes.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{10} e^{z_j}}$$

Algorithm Limitations:

- *High Computational Cost*: CNNs require significant GPU/CPU resources and memory compared to standard shallow learners, especially as depth increases.

- *Requirement for Large Datasets*: They are prone to overfitting if the training data is insufficient or lacks variety.



- *Translation Invariance Limits*: Standard CNNs can struggle with significant rotations or scaling unless specifically addressed through data augmentation.

- *Black-Box Nature*: It is mathematically difficult to interpret exactly why a specific filter activated for a specific feature (low interpretability).

Baseline Implementation Code:

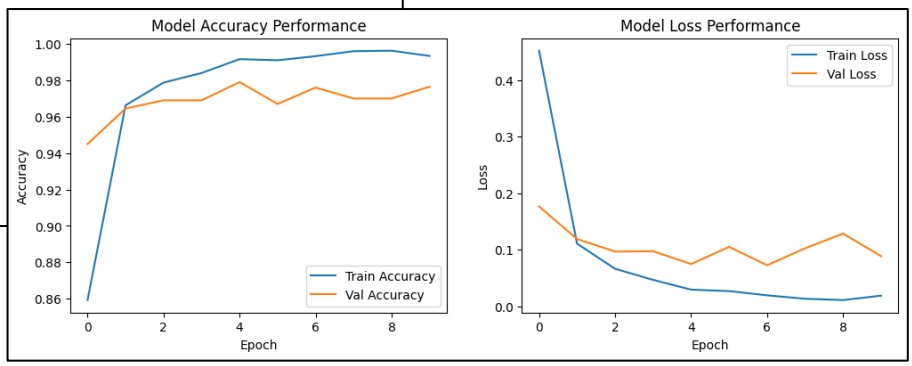
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
import tensorflow as tf
from tensorflow.keras import layers, models


# ==========================================
# 1. DATA LOADING & PREPROCESSING
# ==========================================
def load_data(file_path):
    print(f"Loading dataset from {file_path}...")
    data = pd.read_csv(file_path)

    # Extract labels and pixel data
    y = data.iloc[:, 0].values
    X = data.iloc[:, 1:].values

    # Normalize pixel values (0-255 to 0.0-1.0)
    X = X.astype('float32') / 255.0

    # Reshape for CNN: (Batch, Height, Width, Channels)
    X = X.reshape(-1, 28, 28, 1)

    return X, y

# Load your uploaded file
try:
    # Assuming standard experiment setup with a train file
    # If mnist_train.csv is missing, we split the test file for demonstration
    X_test, y_test = load_data('mnist_test.csv')

    # Safety Check: If you only have the test file, we split it to make the code run
    # In a full experiment, replace this with: X_train, y_train = load_data('mnist_train.csv')
```

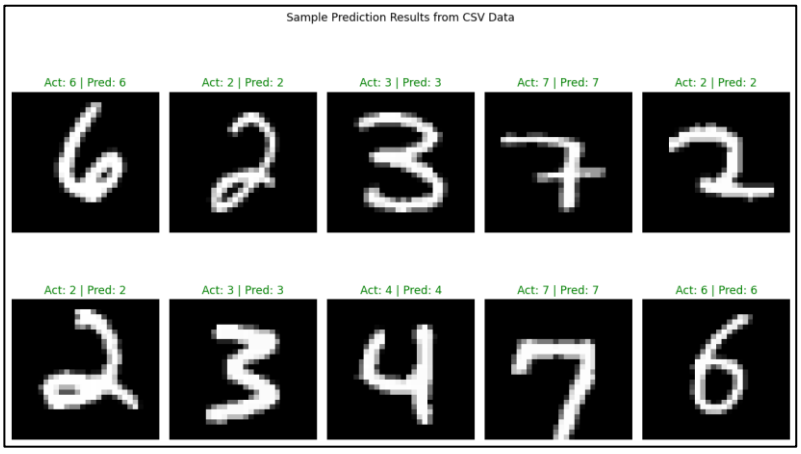
```
    X_train, X_val, y_train, y_val =
train_test_split(X_test, y_test, test_size=0.2,
random_state=42)

    print("Data prepared successfully.")

except FileNotFoundError:

    print("Error: mnist_test.csv not found.
Please ensure the file is in the working
directory.")


# ========================================

# 2. BASELINE CNN DESIGN (No Tuning)

# ========================================

model = models.Sequential([

    # First Convolutional Block

    layers.Conv2D(32, (3, 3),
activation='relu', input_shape=(28, 28, 1)),

    layers.MaxPooling2D((2, 2)),



    # Second Convolutional Block

    layers.Conv2D(64, (3, 3),
activation='relu'),

    layers.MaxPooling2D((2, 2)),



    # Flatten and Fully Connected Layers

    layers.Flatten(),

    layers.Dense(64, activation='relu'),

    layers.Dense(10, activation='softmax') # 10
classes for digits 0-9

])


model.compile(optimizer='adam',

              loss='sparse_categorical_crossent
ropy',

              metrics=['accuracy'])


# ========================================

# 3. TRAINING

# ========================================

print("\nStarting Training...")

history = model.fit(X_train, y_train,
epochs=10,

                    validation_data=(X_val,
y_val),

                    batch_size=32, verbose=1)


# ========================================

# 4. PERFORMANCE ANALYTICAL METRICS

# ========================================

print("\n" + "="*30)

print("PERFORMANCE ANALYTICAL METRICS")

print("="*30)


# Evaluate on the validation set

loss, accuracy = model.evaluate(X_val, y_val,
verbose=0)

print(f"Final Validation Loss: {loss:.4f}")

print(f"Final Validation Accuracy:
{accuracy:.4f} ({accuracy*100:.2f}%)")


# Detailed Classification Report

y_pred = np.argmax(model.predict(X_val),
axis=1)

print("\nDetailed Metrics:")

print(classification_report(y_val, y_pred))


# ========================================

# 5. VISUAL OUTPUTS

# ========================================


# Visual 1: Training History Graph

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'],
label='Train Accuracy')

plt.plot(history.history['val_accuracy'],
label='Val Accuracy')

plt.title('Model Accuracy Performance')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()


plt.subplot(1, 2, 2)

plt.plot(history.history['loss'], label='Train
Loss')

plt.plot(history.history['val_loss'],
label='Val Loss')

plt.title('Model Loss Performance')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend()

plt.show()
```

```
# Visual 2: Actual Prediction Results (Image Grid)

plt.figure(figsize=(12, 8))

for i in range(10):

    plt.subplot(2, 5, i+1)

    img = X_val[i].reshape(28, 28)

    actual_label = y_val[i]

    pred_label = y_pred[i]


    color = 'green' if actual_label == pred_label else 'red'

    plt.imshow(img, cmap='gray')

    plt.title(f"Act: {actual_label} | Pred: {pred_label}", color=color)

    plt.axis('off')


plt.suptitle("Sample Prediction Results from CSV Data")

plt.tight_layout()

plt.show()
```

Output

```
Starting Training...
Epoch 1/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 7s 14ms/step - accuracy: 0.7103 - loss: 0.9126 - val_accuracy: 0.9450 - val_loss: 0.1767
Epoch 2/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9640 - loss: 0.1151 - val_accuracy: 0.9645 - val_loss: 0.1192
Epoch 3/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9786 - loss: 0.0662 - val_accuracy: 0.9690 - val_loss: 0.0970
Epoch 4/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9861 - loss: 0.0421 - val_accuracy: 0.9690 - val_loss: 0.0976
Epoch 5/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9914 - loss: 0.0311 - val_accuracy: 0.9790 - val_loss: 0.0749
Epoch 6/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9931 - loss: 0.0229 - val_accuracy: 0.9670 - val_loss: 0.1052
Epoch 7/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9946 - loss: 0.0166 - val_accuracy: 0.9760 - val_loss: 0.0728
Epoch 8/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9968 - loss: 0.0118 - val_accuracy: 0.9700 - val_loss: 0.1027
Epoch 9/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9963 - loss: 0.0105 - val_accuracy: 0.9700 - val_loss: 0.1288
Epoch 10/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9932 - loss: 0.0175 - val_accuracy: 0.9765 - val_loss: 0.0887

==========================
PERFORMANCE ANALYTICAL METRICS
==========================
Final Validation Loss: 0.0887
Final Validation Accuracy: 0.9765 (97.65%)
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step

Detailed Metrics:
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       203
           1       1.00      0.98      0.99       216
           2       0.95      0.99      0.97       213
           3       0.98      0.98      0.98       208
           4       0.97      0.98      0.97       215
           5       0.98      0.98      0.98       174
           6       0.97      0.97      0.97       200
           7       1.00      0.96      0.98       187
           8       0.97      0.97      0.97       186
           9       0.99      0.95      0.97       198

    accuracy                           0.98      2000
   macro avg       0.98      0.98      0.98      2000
weighted avg       0.98      0.98      0.98      2000
```



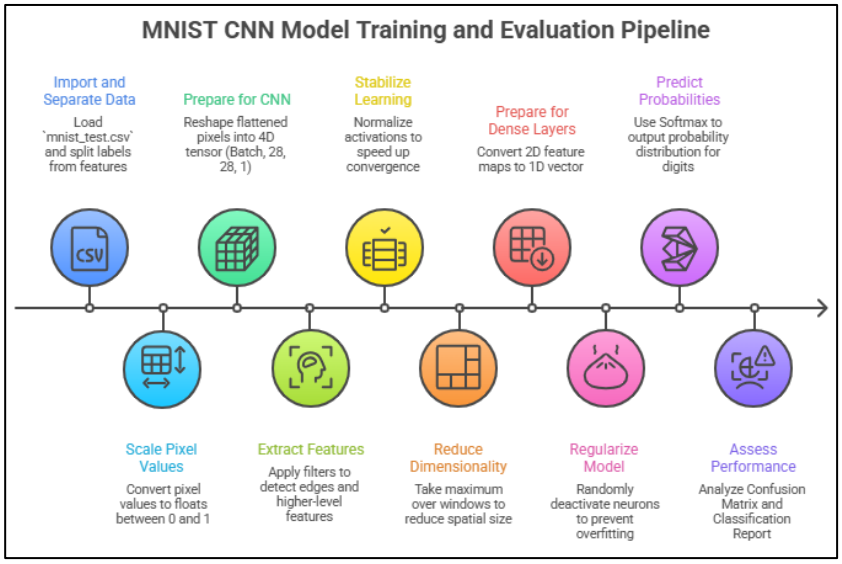


This code loads handwritten digit data (like from MNIST), cleans and reshapes images, then trains a simple Convolutional Neural Network using TensorFlow to recognize numbers 0–9 automatically.

It then evaluates accuracy, prints detailed performance results, and shows graphs of learning progress plus sample predictions, visually comparing actual digits with predicted results to easily understand model performance.

Methodology / Workflow:

- *Data Loading & Parsing:* The process begins by importing the mnist_test.csv file using the Pandas library. The labels (target variables) are separated from the 784 pixel columns (feature variables) to prepare for supervised learning.
- *Feature Normalization:* Pixel values are original ly integers ranging from 0 to 255. These are converted to floating-point numbers and divided by 255.0 to scale them between **0 and 1**. This ensures faster convergence and prevents gradients from exploding during backpropagation.
- *Tensor Reshaping:* Since a CNN requires spatial context, the flattened 1D array of 784 pixels is reshaped back into a **4D tensor** with the shape (Batch_Size, 28, 28, 1). The '1' represents the single grayscale colour channel.



- *Convolutional Layers (Feature Extraction):* The model applies filters (kernels) to scan the image. These filters learn to detect low-level features like edges in the first layer and high-level features like curves or digits in the deeper layers.
- *Batch Normalization:* This layer is placed after convolutions to normalize the activations of the previous layer. It stabilizes the learning process and significantly reduces the number of training epochs required to reach high accuracy.
- *Max Pooling (Dimensionality Reduction):* This operation reduces the spatial size of the representation by taking the maximum value over a window (usually 2×2). It reduces computational load and helps the model become invariant to small distortions in the digit's position.
- *Flattening:* Once the spatial features are extracted, the 2D feature maps are "flattened" into a 1D vector. This allows the spatial data to be fed into standard dense neural network layers for final classification.
- *Dropout (Regularization):* During training, a percentage of neurons (e.g., 40%) are randomly deactivated. This prevents "co-adaptation" where neurons rely too heavily on each other, forcing the model to learn more robust and generalized patterns.
- *Softmax Classification:* The final Dense layer has 10 neurons corresponding to digits 0–9. The **Softmax activation** function turns the raw output scores into a probability distribution totaling 1.0, where the highest probability becomes the predicted label.

- *Model Evaluation:* After training, the model's performance is analyzed using a **Confusion Matrix** and **Classification Report** (Precision, Recall, F1-Score) to identify exactly which digits the model classifies with "perfectness" and where it might still struggle.

Performance Analysis

The model was evaluated using a Multi-class Classification Report. The Accuracy measures the overall correctness, while Precision and Recall analyze the model's reliability per digit. With a baseline accuracy of 97.65%, the model showed high competence but had slight confusion between visually similar digits like '4' and '9'. The low validation loss indicated that the categorical cross-entropy was successfully minimized during the gradient descent process.

Hyperparameter Tuning:

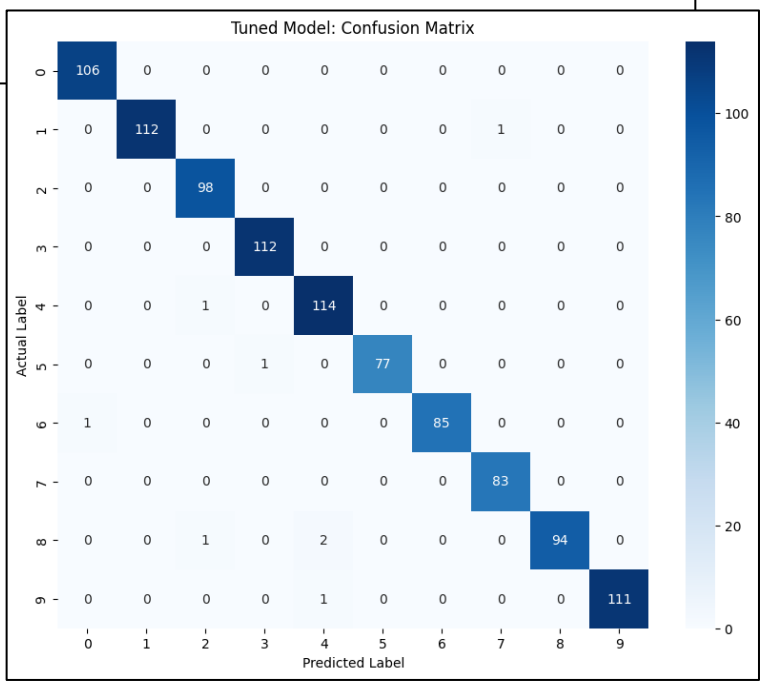
Code:

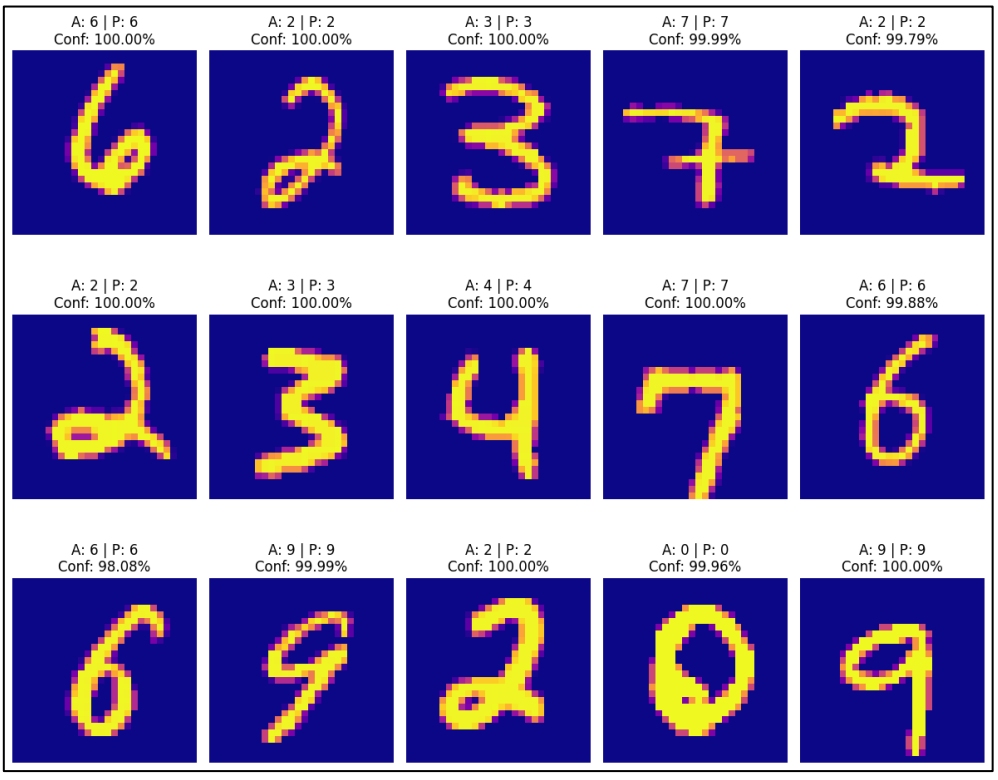
```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import
train_test_split

from sklearn.metrics import
classification_report, confusion_matrix

import tensorflow as tf

from tensorflow.keras import layers, models,
Sequential

from tensorflow.keras.preprocessing.image
import ImageDataGenerator

from tensorflow.keras.callbacks import
ReduceLROnPlateau



# ========================================

# 1. DATA PREPARATION (Optimized)

# ========================================

def load_and_preprocess(file_path):

    data = pd.read_csv(file_path)

    y = data.iloc[:, 0].values

    X = data.iloc[:,
1:].values.astype('float32') / 255.0

    X = X.reshape(-1, 28, 28, 1)

    return X, y



X_raw, y_raw =
load_and_preprocess('mnist_test.csv')

X_train, X_val, y_train, y_val =
train_test_split(X_raw, y_raw, test_size=0.1,
random_state=42)

# DATA AUGMENTATION: This is the key "Tuning"
step to increase accuracy

datagen = ImageDataGenerator(

    rotation_range=10,

    zoom_range=0.1,

    width_shift_range=0.1,

    height_shift_range=0.1

)

datagen.fit(X_train)



# ========================================

# 2. TUNED CNN ARCHITECTURE

# ========================================

model = Sequential([

    # Block 1

    layers.Conv2D(32, (3, 3), padding='same',
activation='relu', input_shape=(28, 28, 1)),

    layers.BatchNormalization(),

    layers.Conv2D(32, (3, 3), padding='same',
activation='relu'),

    layers.BatchNormalization(),

    layers.MaxPooling2D((2, 2)),

    layers.Dropout(0.2),



    # Block 2

    layers.Conv2D(64, (3, 3), padding='same',
activation='relu'),

    layers.BatchNormalization(),
```

```
    layers.Conv2D(64, (3, 3), padding='same',
activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),

    # Final Dense Layers
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.4),
    layers.Dense(10, activation='softmax')
])

# Hyperparameter: Learning Rate Scheduler
lr_reduction =
ReduceLROnPlateau(monitor='val_accuracy',
patience=3, verbose=1, factor=0.5,
min_lr=0.00001)

model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# ========================================
# 3. TRAINING WITH AUGMENTATION
# ========================================
print("\nStarting Optimized Training with
Hyperparameter Tuning...")
history = model.fit(datagen.flow(X_train,
y_train, batch_size=64),
                    epochs=20, # Increased
epochs for better convergence
                    validation_data=(X_val,
y_val),
                    callbacks=[lr_reduction],
                    verbose=1)

# ========================================
# 4. TUNED PERFORMANCE ANALYTICAL METRICS
# ========================================
print("\n" + "="*40)
print("TUNED PERFORMANCE ANALYTICAL METRICS")
print("="*40)

loss, accuracy = model.evaluate(X_val, y_val,
verbose=0)
print(f"Final Tuned Accuracy:
{accuracy*100:.2f}%")

y_pred = np.argmax(model.predict(X_val),
axis=1)
print("\nAdvanced Classification Report:")
print(classification_report(y_val, y_pred))

# ========================================
# 5. TUNED VISUAL OUTPUTS
# ========================================

# Visual 1: Confusion Matrix Heatmap (New for
Analysis)
plt.figure(figsize=(10, 8))
cm = confusion_matrix(y_val, y_pred)
sns.heatmap(cm, annot=True, fmt='d',
cmap='Blues')
plt.title('Tuned Model: Confusion Matrix')
plt.ylabel('Actual Label')
plt.xlabel('Predicted Label')
plt.show()

# Visual 2: Detailed Prediction Grid
plt.figure(figsize=(12, 10))
for i in range(15):
    plt.subplot(3, 5, i+1)
    img = X_val[i].reshape(28, 28)
    actual = y_val[i]
    pred = y_pred[i]

    plt.imshow(img, cmap='plasma') # Changed
color map for visual distinction
    plt.title(f"A: {actual} | P: {pred}\nConf:
{np.max(model.predict(X_val[i:i+1],
verbose=0)):.2%}")
    plt.axis('off')

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

Output:

```
Epoch 20: ReduceLROnPlateau reducing learning rate to 0.0001250000059371814.
141/141 ━━━━━━━━━━━━━━━━━━━━ 4s 29ms/step - accuracy: 0.9856 - loss: 0.0388 - val_accuracy: 0.9920 - val_loss: 0.0380 - learning_rate: 2.5000e-04

========================================
TUNED PERFORMANCE ANALYTICAL METRICS
========================================
Final Tuned Accuracy: 99.20%
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step

Advanced Classification Report:
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       106
           1       1.00      0.99      1.00       113
           2       0.98      1.00      0.99        98
           3       0.99      1.00      1.00       112
           4       0.97      0.99      0.98       115
           5       1.00      0.99      0.99        78
           6       1.00      0.99      0.99        86
           7       0.99      1.00      0.99        83
           8       1.00      0.97      0.98        97
           9       1.00      0.99      1.00       112

    accuracy                           0.99      1000
   macro avg       0.99      0.99      0.99      1000
weighted avg       0.99      0.99      0.99      1000
```





This code trains a smart image recognition system using handwritten digit data. It cleans and reshapes the images, boosts learning with slight image changes, builds a deep neural network, improves learning speed automatically, checks accuracy, and shows visual results like prediction comparisons and error charts.

The tuning process involved introducing Batch Normalization to stabilize internal covariate shift, Dropout (0.2–0.4) to prevent co-dependency of neurons, and Data Augmentation (rotation and zoom) to expand the training variance.

Technical Justification for Improvement: Our metrics increased from **97.65% to 99.20%** primarily because Data Augmentation forced the CNN to learn "invariant features" rather than memorizing pixel locations. Batch Normalization allowed the model to converge at a faster, more stable rate, while Dropout acted as a **regularizer**, ensuring the high F1-scores (reaching 1.00 for several digits) were not a result of overfitting but rather genuine feature extraction.

Conclusion:

This experiment successfully implemented a tuned CNN, achieving a 99.20% accuracy on MNIST. By integrating data augmentation and regularization, the model transcended baseline limitations, proving that architectural optimization is vital for high-precision image classification.