ABSTRACT

This project implements a Convolutional Neural Network (CNN) to recognize handwritten digits using the MNIST dataset. The proposed model consists of two convolutional layers with max-pooling, followed by a flatten layer and two fully connected layers. The network is trained on 60,000 images with a 90:10 train–validation split and evaluated on 10,000 test images, achieving about 98.9% test accuracy. A confidence-based rejection mechanism is applied during custom image prediction: if the model's maximum class probability is below a threshold (0.7), the system reports "Cannot be recognised" instead of forcing an unreliable prediction. The implementation uses TensorFlow/Keras, NumPy, Matplotlib, Pillow, and scikit-learn. Results show that CNNs can accurately classify handwritten digits and can be extended to practical applications such as form digitization and automated recognition systems.

The proposed CNN architecture consists of two convolutional layers with ReLU activation and max-pooling, followed by a flatten layer, a 64-neuron fully connected layer, and a 10-neuron softmax output layer. The model is trained using the Adam optimizer and sparse categorical cross-entropy loss for 5 epochs, with a 90:10 split between training and validation data. On the held-out test set, the network achieves approximately 98.9% accuracy, which demonstrates strong performance on the digit classification task. Evaluation includes training/validation curves, a confusion matrix, and qualitative inspection of predicted samples.

In addition, a custom prediction module allows users to input their own digit images. A confidence-based threshold is applied so that low-confidence predictions are reported as "Cannot be recognised" instead of forcing an answer. Overall, the system shows that CNNs are effective for handwritten digit recognition and can be extended toward real-world applications.

On the held-out test set, the network achieves approximately 98.9% accuracy, which demonstrates strong performance on the digit classification task. Evaluation includes training/validation curves, a confusion matrix, and qualitative inspection of predicted samples.

1. INTRODUCTION

Handwritten digit recognition is one of the fundamental problems in computer vision and machine learning. The task involves classifying an image of a handwritten digit (0–9) into the correct class. Historically, digit recognition has been solved using traditional machine learning methods like Support Vector Machines (SVMs) and k-Nearest Neighbors (k-NN). However, Convolutional Neural Networks (CNNs) have revolutionized image classification by automatically learning hierarchical features like edges, corners, and strokes.

1.1 Motivation

- Real-world applications: Postal automation, bank check processing, form digitization
- Benchmark problem: MNIST is the "Hello World" of deep learning
- Learning CNNs: Provides hands-on experience with modern deep learning frameworks

1.2 Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset is a collection of 70,000 grayscale images of handwritten digits:

- Training set: 60,000 images
- Test set: 10,000 images
- Image size: 28×28 pixels (grayscale, single channel)
- Classes: 10 (digits 0–9)
- Source: Keras built-in dataset (keras.datasets.mnist)

1.3 Project Objectives

1. Load and preprocess MNIST dataset
2. Design and implement a CNN architecture
3. Train the model and monitor accuracy/loss
4. Evaluate on test set and analyze results
5. Test on custom handwritten digit images
6. Implement confidence-based prediction with rejection threshold

2. METHODOLOGY

2.1 Data Preprocessing

Loading the dataset:

The MNIST dataset is loaded directly from Keras:

text

(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

Normalization:

Pixel values are scaled from [0, 255] to [0, 1] to improve training stability:

text

x_train = x_train.astype("float32") / 255.0

x_test = x_test.astype("float32") / 255.0

Reshaping:

A channel dimension is added for CNN input (height, width, channels):

text

x_train = np.expand_dims(x_train, -1) # Shape: (60000, 28, 28, 1)

x_test = np.expand_dims(x_test, -1)   # Shape: (10000, 28, 28, 1)

Data split:

- Training: 54,000 images (90%)
- Validation: 6,000 images (10%)
- Test: 10,000 images

2.2 CNN Architecture

The model follows a sequential architecture:

Layer	**Configuration**	**Output Shape**
Input	28×28×1 image	(28, 28, 1)
Conv2D	32 filters, 3×3 kernel, ReLU	(26, 26, 32)

Layer	Configuration	Output Shape
MaxPooling2D	2×2 pool size	(13, 13, 32)
Conv2D	64 filters, 3×3 kernel, ReLU	(11, 11, 64)
MaxPooling2D	2×2 pool size	(5, 5, 64)
Flatten	Convert to 1D vector	(1600,)
Dense	64 neurons, ReLU	(64,)
Dense	10 neurons, Softmax	(10,)

Architecture code:

python

```
model = keras.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28, 1)),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(64, (3, 3), activation="relu"),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(10, activation="softmax"),
])
```

2.3 Why Convolutional Neural Networks?

Advantages of CNN over fully connected networks:

1. Spatial feature learning: Convolution filters detect local patterns (edges, textures, shapes)
2. Parameter sharing: Same filter applied across the entire image → fewer parameters
3. Translation invariance: Features learned are robust to shifts in the image

4. Hierarchical learning: Early layers detect simple features; later layers combine them into complex patterns

2.4 Training Configuration

Compilation:

python

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

Training:

python

```
history = model.fit(
    x_train, y_train,
    epochs=5,
    batch_size=128,
    validation_split=0.1,
    verbose=1
)
```

Hyperparameters:

- Optimizer: Adam (adaptive learning rate)
- Loss function: Sparse categorical cross-entropy (for integer labels 0–9)
- Epochs: 5 (number of passes over training data)
- Batch size: 128 (images per gradient update)
- Validation split: 10% (6,000 images for validation)

3. RESULTS

3.1 Training and Validation Performance

Epoch-wise metrics:

Epoch	Train Accuracy	Val Accuracy	Train Loss	Val Loss
1	95.2%	97.1%	0.156	0.095
2	97.8%	98.5%	0.067	0.054
3	98.4%	98.8%	0.051	0.042
4	98.7%	98.9%	0.043	0.038
5	98.9%	98.95%	0.038	0.035

Key observations:

- Model converges quickly (good performance by epoch 2)
- Validation accuracy follows training accuracy (no significant overfitting)
- Final test set accuracy: 98.9%

3.2 Confusion Matrix Analysis

The confusion matrix reveals which digit pairs are most commonly misclassified:

Most confused pairs:

- 3 vs 8: 12 errors (similar rounded shapes)
- 4 vs 9: 8 errors (both have curved elements)
- 0 vs 6: 6 errors (both are circular)

Least confused:

- 1 (straight vertical line) – rarely confused
- 7 (distinctive angular shape) – rarely confused

This pattern is expected because digits with similar visual features are harder to distinguish.

3.3 Custom Image Prediction with Confidence Thresholding

Testing on handwritten digits:

When the model predicts a digit from a custom image, confidence varies based on how clearly the digit is written:

Example 1: Clear handwritten "7"

- Predicted digit: 7
- Confidence: 0.987 (98.7%)
- Status: ✓ Correct

Example 2: Ambiguous/Noisy image

- Predicted digit (raw model output): 3
- Confidence: 0.62 (62%)
- **Since confidence < 0.7 threshold, system output: "Cannot be recognised"**

Confidence threshold mechanism:

- Confidence threshold: 0.7 (70%)
- Predictions above 0.7 are accepted and displayed
- Predictions below 0.7 are marked as "Cannot be recognised"
- If model's maximum class probability is below 0.7, the system does not trust the prediction and shows 'Cannot be recognised' in both console and figure title

This rejection mechanism ensures the model only outputs predictions it is confident about, preventing overconfident false predictions.

4. IMPLEMENTATION DETAILS

4.1 Technologies Used

Component	Technology
Language	Python 3.8+
Deep Learning Framework	TensorFlow 2.x / Keras
Numerical Computing	NumPy
Image Processing	Pillow (PIL)
Visualization	Matplotlib
Evaluation	scikit-learn (confusion matrix)
Environment	Conda / venv
IDE	Visual Studio Code

4.2 File Structure

text

DigitCNN/

├── main.py # Data loading, training, saving model and plots

├── mnist_cnn.keras # Saved trained model

├── predict_own_image.py # Prediction on custom images with confidence threshold

├── confusion_matrix.py # Confusion matrix generation

├── training_curves.png # Accuracy/loss plots

├── predictions_example.png # Sample predictions

├── confusion_matrix.png # Confusion matrix heatmap

├── sample_train_images.png   # 25 MNIST examples

├── my_digit0.png to my_digit9.png   # Single-digit test images

├── my_digit_random.png   # Random/noisy test image

└── DigitCNN-Report.md   # Project report

4.3 Key Code Snippets

Loading and preprocessing:

python

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.astype("float32") / 255.0
x_train = np.expand_dims(x_train, -1)
print("Train shape:", x_train.shape)  # Output: (60000, 28, 28, 1)
```

Saving the model:

python

```
model.save("mnist_cnn.keras")
```

Loading and predicting:

python

```
model = keras.models.load_model("mnist_cnn.keras")
probs = model.predict(arr)  # arr shape: (1, 28, 28, 1)
pred_class = np.argmax(probs)
confidence = np.max(probs)
```

Confidence-based rejection:

python

```
if confidence < 0.7:
    print("Cannot be recognised")
else:
    print(f"Predicted digit: {pred_class}")
```

5. ANALYSIS AND DISCUSSION

5.1 Strengths of the Approach

1. High accuracy: 98.9% on test set demonstrates effective learning
2. Generalization: Model works on custom handwritten digits (not just MNIST)
3. Efficiency: CNN reduces parameters compared to fully connected networks
4. Robustness: Confidence thresholding prevents overconfident false predictions
5. Fast training: Converges in just 5 epochs

5.2 Limitations

1. Dataset limitation: MNIST digits are centered and normalized; real-world digits may vary
2. Model size: 98.9% accuracy suggests some digits are inherently ambiguous
3. Single-digit constraint: Model expects one digit per image; multiple digits are misclassified
4. No image segmentation: Custom images must be pre-cropped to single digits

5.3 Confusion Analysis

Why some digits are confused:

- **3 vs 8:** Both have rounded shapes; model sometimes misses the middle section
- **4 vs 9:** Both have vertical and curved elements; orientation determines the class
- **0 vs 6:** Both are circular; slight differences determine class

Potential improvements:

- Data augmentation (rotation, skewing, stretching)
- Deeper CNN architecture (ResNet, VGG-style)
- Ensemble of multiple models
- Regularization techniques (dropout, batch normalization)

6. CONCLUSIONS

This project successfully demonstrates the application of **Convolutional Neural Networks** to the handwritten digit recognition problem. The implemented CNN achieves:

- 98.9% test accuracy on MNIST
- Fast training (converges in 5 epochs).
- Custom image support with confidence-based rejection.
- Robust architecture with minimal overfitting.

Key Validations

1. CNNs are effective for image classification tasks
2. MNIST is an excellent benchmark for learning deep learning fundamentals
3. Simple architectures can achieve high performance without excessive complexity
4. Confidence thresholding prevents overconfident predictions, making the system more reliable

Future Enhancements

1. Deploy as web application: Allow users to draw digits and get predictions
2. Real-world digit recognition: Test on handwritten checks, forms, or license plates
3. Multi-digit segmentation: Implement image segmentation to handle pages of digits
4. Model compression: Convert to TensorFlow Lite for mobile deployment
5. Adversarial robustness: Test against adversarial examples and improve defenses
6. Ensemble methods: Combine multiple models for better predictions
7. Explainability: Implement attention mechanisms to visualize which regions the model focuses on

7. REFERENCES

[1] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324.

[2] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. Advances in Neural Information Processing Systems (NIPS), 25, 1097–1105.

[3] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

[4]TensorFlow Team. (2024). TensorFlow and Keras Documentation. https://www.tensorflow.org

[5]MNIST Database. (2024). The MNIST Database of Handwritten Digits. http://yann.lecun.com/exdb/mnist/

[6] Chollet, F. (2017). Deep Learning with Python. Manning Publications.

[7] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition.
[8] IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770–778.

[9] Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. International Conference on Learning Representations (ICLR).

[10] LeCun, Y., & Bengio, Y. (1995). Convolutional networks for images, speech, and time series. The Handbook of Brain Theory and Neural Networks, 3361(10), 1995.

[11] Keras. (2024). Keras Applications and Models. https://keras.io/api/models/

APPENDIX: SAMPLE CODE

A.1 Full Training Script (main.py)

python

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Load MNIST
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Preprocess
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)

print("Train shape:", x_train.shape)
print("Test shape:", x_test.shape)

# Define model
model = keras.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28, 1)),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(64, (3, 3), activation="relu"),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(10, activation="softmax"),
])
```

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

# Train
history = model.fit(
    x_train, y_train,
    epochs=5,
    batch_size=128,
    validation_split=0.1,
    verbose=1
)

# Evaluate
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {test_acc:.4f}")

# Save model
model.save("mnist_cnn.keras")
```

A.2 Prediction Script with Confidence Threshold (predict_own_image.py)

```
python
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import os

model = keras.models.load_model("mnist_cnn.keras")
```

```
def predict_own_image(path, threshold=0.7):
    if not os.path.exists(path):
        print("File not found:", path)
        return

    img = Image.open(path).convert("L")
    img = img.resize((28, 28))
    arr = np.array(img).astype("float32") / 255.0
    arr = np.expand_dims(arr, axis=(0, -1))

    probs = model.predict(arr)
    pred_class = int(np.argmax(probs, axis=1))
    confidence = float(np.max(probs))

    if confidence < threshold:
        print(f"Cannot be recognised (confidence = {confidence:.2f})")
        plt.imshow(arr[0, :, :, 0], cmap="gray")
        plt.title("Cannot be recognised")
    else:
        print(f"Predicted digit: {pred_class} (confidence = {confidence:.2f})")
        plt.imshow(arr[0, :, :, 0], cmap="gray")
        plt.title(f"Predicted: {pred_class} ({confidence:.2f})")

    plt.axis("off")
    plt.show()

# Usage
predict_own_image("my_digit_random.png")
```