

Introduction to Handwritten Text Recognition

MNIST: The Starting Point for Handwritten Digit Recognition

MNIST

MNIST stands for Modified National Institute of Standards and Technology dataset.

It is a modified version of the original NIST dataset, which contained handwritten digits from US Census Bureau employees and high school students. The modification involved normalizing the images and making them more suitable for machine learning tasks.

It's a dataset of 70,000 grayscale images (28x28 pixels) of handwritten digits (0-9) where 60,000 are images for training, and 10,000 images for for testing.

Why is it a Good Starting Point?

- Simple & Structured - Small images, easy to process.

Benchmark Dataset - Standard for evaluating machine learning models.

- Fast Training - Works well even on CPUs.
- Foundation for Deep Learning - Prepares for advanced OCR tasks.

Real-World Applications

- OCR - Scanning documents, bank checks, and IDs.
- Digitizing handwritten forms - Surveys, exam papers, receipts.
- Postal Automation - Recognizing handwritten addresses.

Minimum System Requirements

- OS: Windows, macOS, or Linux
- CPU: Intel Core i3 (or equivalent)
- RAM: 4GB
- Storage: 2GB free space (for dependencies and dataset)
- Python Version: 3.7 or higher
- Libraries: TensorFlow/Keras, NumPy, Matplotlib
- GPU: Not required (CPU training works fine but slower)

Required Libraries

- NumPy: Numerical computing library
- Matplotlib: Data visualization tool
- TensorFlow: Deep learning framework
- Keras: High-level neural network API

Therefore...

```
pip install numpy matplotlib tensorflow keras
```

Step 1: Load and Explore MNIST

This step helps us understand the dataset before feeding it into a model.

```
import tensorflow as tf
import matplotlib.pyplot as plt

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

# Display dataset shape
print(f"Training Data Shape: {x_train.shape}, Labels: {y_train.shape}")
print(f"Testing Data Shape: {x_test.shape}, Labels: {y_test.shape}")

# Visualize some sample digits
fig, axes = plt.subplots(1, 5, figsize=(10, 3))
for i in range(5):
    axes[i].imshow(x_train[i], cmap='gray')
    axes[i].set_title(f"Label: {y_train[i]}")
    axes[i].axis('off')
plt.show()
```

Step 2: Preprocess the Data

```
# Import necessary libraries
import tensorflow as tf

# Load MNIST dataset (data comes raw and unprocessed)
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Print dataset shape before preprocessing ⚡ ADDED
print(f"Original Training Data Shape: {x_train.shape}")
print(f"Original Testing Data Shape: {x_test.shape}")

# Normalize pixel values (0-255 → 0-1) ⚡ ADDED
x_train, x_test = x_train / 255.0, x_test / 255.0

# Reshape images to add a single channel (for CNN input) ⚡ ADDED
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

# Print dataset shape after preprocessing ⚡ ADDED
print(f"Processed Training Data Shape: {x_train.shape}")
print(f"Processed Testing Data Shape: {x_test.shape}")

# Print sample labels ⚡ ADDED
print("Sample Labels (First 10 Training Labels):", y_train[:10])
```

Normalization in an 8-bit grayscale image explained

- ✓ Each pixel ranges from 0-255 (8-bit depth).
- ✓ To bring it to $[0,1]$, divide by 255.
- ✓ This keeps the relative intensity but makes computation easier.

Why Reshaping?

In the raw MNIST dataset, each image is stored as (28, 28) a # A 2D array (height \times width). This means every image is just a grid of numbers (28 rows \times 28 columns).

Convolutional Neural Networks (CNNs) expect 3D input: (height, width, color channel). Since MNIST images don't have a color channel (they're grayscale), we reshape them to (28, 28, 1), adding a "1" for the grayscale channel.

Step 3: Create Convolutional Neural Network (CNN) model for MNIST digit classification

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Define the CNN model
model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(28, 28, 1)), # ⚡ ADDED (First Convolution Layer)
    MaxPooling2D(2,2), # ⚡ ADDED (First Pooling Layer)
    Conv2D(64, (3,3), activation='relu'), # ⚡ ADDED (Second Convolution Layer)
    MaxPooling2D(2,2), # ⚡ ADDED (Second Pooling Layer)
    Flatten(), # ⚡ ADDED (Flattening for Dense Layer)
    Dense(128, activation='relu'), # ⚡ ADDED (Fully Connected Layer)
    Dense(10, activation='softmax') # ⚡ ADDED (Output Layer with 10 Classes)
])

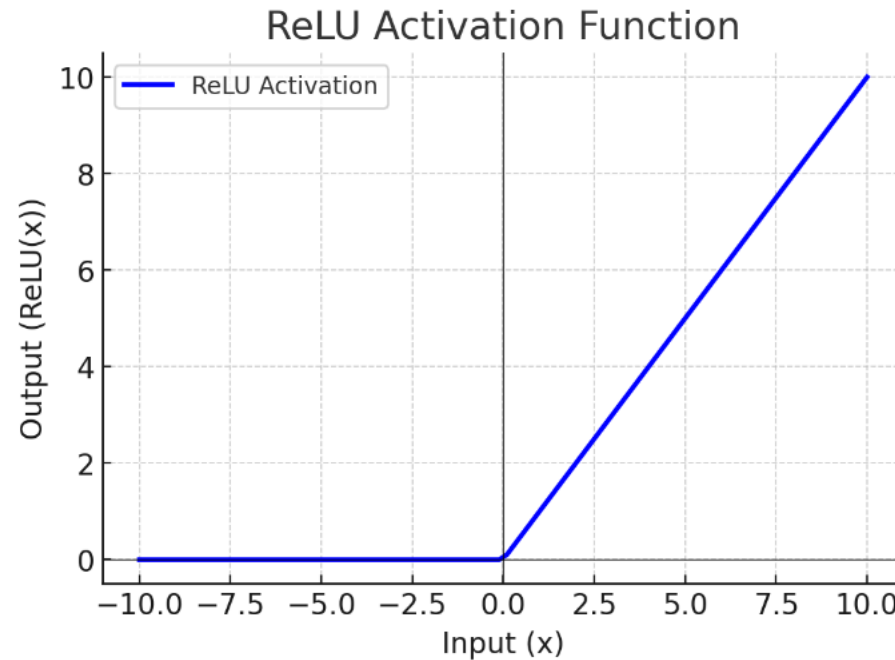
# Compile the model ⚡ ADDED
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Display model summary ⚡ ADDED
model.summary()
```

Rectified Linear Unit (ReLU)

ReLU works like a diode rectifier in electronics. It blocks negative values and let pass positive values unchanged.

ReLU Activation Function



- For $x < 0$: Output is 0 (negative values are removed).
 - For $x \geq 0$: Output is x (positive values pass unchanged).
 - In neural networks, negative values can confuse the learning process.
 - If a neuron gets a negative weighted sum, it might send negative values forward, which can lead to instability.
- In the separate lesson, we'll compare ReLU to Sigmoid to justify why we have chosen ReLU over Sigmoid.

Softmax: Making Predictions Easy

📌 What it does:

- ✓ Converts raw model outputs (logits) into probabilities.
- ✓ Ensures all probabilities sum to 1.
- ✓ Makes prediction simple → Pick the highest probability.

📌 Example:

Logits: [2.0, 1.0, 0.1]

Softmax → [0.66, 0.24, 0.10]

Highest = Prediction! ✓

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum e^{x_j}}$$

Adam Optimizer

Adam (Adaptive Moment Estimation) is an optimization algorithm that helps neural networks learn efficiently by combining the benefits of Momentum and RMSprop (Root Mean Square Propagation).

Momentum prevents the model from getting stuck or slowing down too much.

RMSprop prevents learning from being too fast (skipping important details) or too slow (lazy learning).

loss='sparse_categorical_crossentropy'

Entropy measures uncertainty:

- Fair coin (50/50) → High entropy (uncertain outcome).
- Biased coin (always heads) → Low entropy (certain outcome).
- In machine learning, entropy quantifies uncertainty in a probability distribution.

Crossentropy cross-checks:

Predicted probabilities (model's guess) against True labels (correct answer).

- Low crossentropy → Good prediction (model is confident and correct).
- High crossentropy → Bad prediction (model is uncertain or wrong).

Categorical crossentropy (as opposed to binary crossentropy for only two classes) is used for multi-class classification problems, where each input belongs to one of many possible categories, and the loss function helps the model improve by minimizing prediction errors.

- "Sparse" means using integer labels (like 5) instead of one-hot vectors (like [0,0,0,0,0,1,0,0,0,0]).
- One-hot encoding explicitly marks all categories, while sparse encoding simply states the correct answer.
- Sparse encoding is faster and more memory-efficient.

metrics=['accuracy']

Accuracy fits well for standard classification, where:

- ✓ Each image contains only one object (e.g., a single handwritten digit).
- ✓ Each image belongs to exactly one class (e.g., digit “5” in MNIST).
- ✓ The goal is to assign one correct label per image (e.g., cat, dog, car).

We use precision, recall, or F1-score when some mistakes are more important than others, and we use mAP when detecting multiple objects in an image.

For example, while in disease detection, a false negative (missing a real case) is worse than a false positive (a false alarm), in MNIST digit classification, all mistakes are equally bad—misclassifying an “8” as a “3” is just as bad as misclassifying a “5” as a “7”.

Step 4: Training the model

```
16 # 4) Build CNN Model
17 model = keras.Sequential([
18     keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
19     keras.layers.MaxPooling2D(2,2),
20     keras.layers.Conv2D(64, (3,3), activation='relu'),
21     keras.layers.MaxPooling2D(2,2),
22     keras.layers.Flatten(),
23     keras.layers.Dense(128, activation='relu'),
24     keras.layers.Dense(10, activation='softmax') # Output layer (10 classes)
25 ])
26
27 # 5) Compile Model (Choosing optimizer, loss function, and metric)
28 model.compile(optimizer='adam',
29               loss='sparse_categorical_crossentropy',
30               metrics=['accuracy'])
31
32 # 6) Train Model
33 history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
34
```

What Happens During Training?

- 1) Forward Pass → The model takes an image, makes a prediction.
- 2) Calculate Loss → Compares prediction to the actual label.
- 3) Backpropagation → Adjusts weights using Adam optimizer to improve accuracy.
- 4) Repeat for all images → One full pass through all training images = 1 epoch.
- 5) Validation Step → After each epoch, the model is tested on unseen test data.

Weights: The Model's "Knowledge Pack"

Weights are the only output after training—this "knowledge pack" is what the model uses for predictions and can be reused for future training

What Are Weights?

- Weights are the only output of training—they store all the knowledge the model has learned.
- They determine which features matter most in an image.

How Weights Work?

- Before Training – The model starts with random weights (no knowledge).
- During Training – Weights are adjusted to learn patterns using backpropagation.
- After Training – The final weights are the only thing saved, storing the model's knowledge.

Weights in Prediction

- During prediction, the model uses saved weights—it doesn't relearn, it just applies its stored knowledge.
- The trained model can make decisions instantly using these weights.

Reusing Weights in the Next Training

- Weights can be saved and loaded to avoid retraining from scratch.
- Transfer learning allows a model trained on one dataset to help with another task.
- Weights are saved using `save_weights()` and loaded using `load_weights()`.

Step 5: Evaluating the Model

After the model is trained, its performance on unseen data has to be evaluated:

- Measure accuracy - Check how well the model predicts on test data.
- Compare training vs. testing performance - Detect overfitting.
- Analyze errors - Understand where the model makes mistakes.

```
35 # 5) Compile Model (Choosing optimizer, loss function, and metric)
36 model.compile(optimizer='adam',
37               loss='sparse_categorical_crossentropy',
38               metrics=['accuracy'])
39
40 # 6) Train Model
41 history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
42
43 # 7) Evaluate Model
44 test_loss, test_acc = model.evaluate(x_test, y_test)
45 print(f"Test Accuracy: {test_acc:.4f}")
```

`x_test` is a set of unseen images that the model has never trained on.

`y_test` contains the correct labels for `x_test` (unseen images).

Step 6: Saving the Model – Don't Lose Your Progress!

Why Save the Model?

- Avoid retraining – No need to start from scratch.
- Reuse the model – Load it anytime for predictions.
- Deploy it – Use in real-world applications like live digit recognition.

◆ Save the Model After Training: `model.save("mnist_model.keras")`

◆ Load the Model Later for Predictions:

```
import tensorflow as tf  
model = tf.keras.models.load_model("mnist_model.keras", compile=False)  
print("Model loaded successfully!")
```

Where is it saved?

By default, in the same directory as the script. You can specify a different path if needed.

Limitations of MNIST for Single-Digit Recognition

MNIST is great for learning digit recognition but is limited to single, well-separated, grayscale digits.


- 🚫 Only Recognizes One Digit at a Time: MNIST is designed for isolated digits (0-9), not multi-digit numbers like “456”.
- 📏 Fixed Image Size (28×28 Pixels): Handwritten digits must be resized, which can cause distortion or loss of detail.
- 🕒 Simple Grayscale Digits Only: MNIST does not support colored, stylized, or rotated handwritten text.
- 📖 Lacks Context Understanding: The model only classifies individual digits and does not recognize numbers as a whole.
- 🛑 No Support for Overlapping Digits: Touching or connected digits may be misclassified or ignored.

Final Takeaway


What We've Learned from MNIST

- ✓ Preprocessing – How to normalize, reshape, and visualize handwritten digits.
- ✓ Building & Training a Neural Network – Using CNNs with ReLU, Softmax, Adam optimizer, and loss functions.
- ✓ Model Evaluation – Checking accuracy, loss, and avoiding overfitting.
- ✓ Live Digit Recognition – Capturing real-time handwritten digits with a webcam.
- ✓ Handling Confidence Levels – Filtering low-confidence predictions for reliability.

How to Apply It

 Improve Real-World Handwriting Recognition: Extend to custom datasets beyond MNIST for better accuracy, and explore rotation-invariant and color-based recognition.

 Deploy the Model in Applications: Integrate into mobile/web apps for digit recognition, and Use in IoT projects with embedded systems (e.g., Raspberry Pi).

 Expand to More Complex Text Recognition: Move towards OCR (Optical Character Recognition) for reading entire words, and Train models on real-world handwriting datasets beyond single digits.