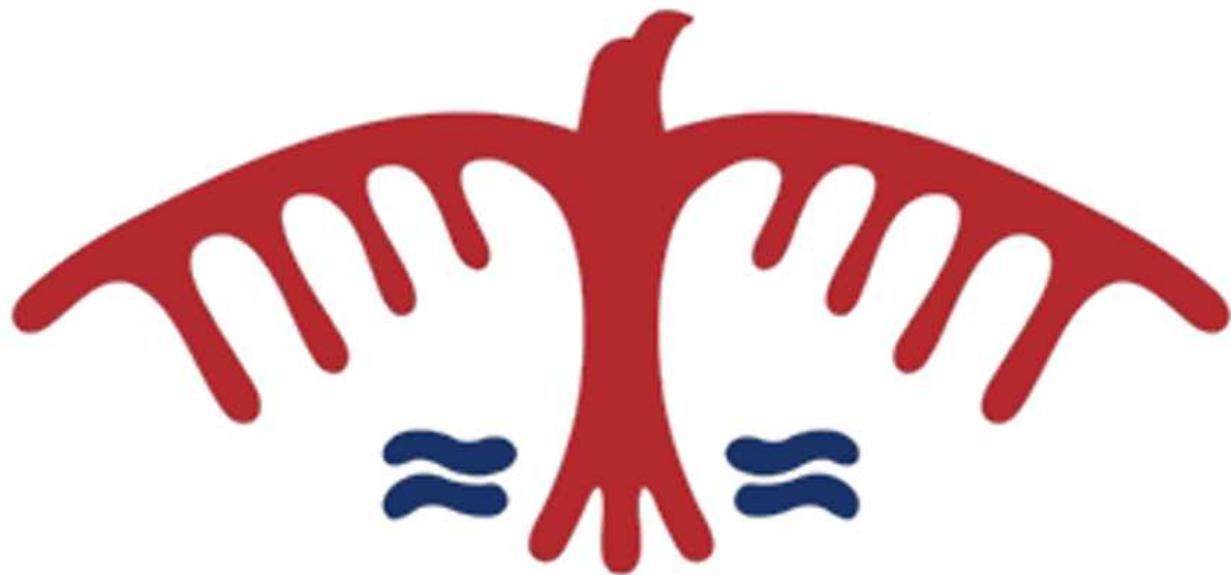


Project Final Report - Handwritten Digit Recognizer
COSC 3117-F05 - Artificial Intelligence



Algoma
UNIVERSITY

Faculty of
Computer Science
& Technology

Submitted By: Vikas Saahil

Student Number: 239408810

Colleague ID: 5143498

Submitted To: Dr. Zamilur Rahman

Faculty of Computer Science & Technology

Acknowledgement

I feel immense pleasure in acknowledging my ineptness and heartfelt sense of gratitude to my respected professor “Dr. Zamilur Rahman”, Faculty of Computer Science & Technology for his sustained encouragement, regular guidance, inspiration, valuable suggestion and great support throughout the semester.

I would like to thank and express my sincere appreciation to Dr. Zamilur Rahman and Faculty of Computer Science & Technology.

Vikas Saahil

In this project I built a simple handwritten digit recognizer using a convolutional neural network (CNN) trained on the MNIST dataset. The goal was to take 28×28 grayscale images of digits (0–9) and predict which digit is in the image. I've also created a small local interface where you can draw a digit with the mouse and see the model's prediction and class probabilities in real time.

I used Python with TensorFlow/Keras, NumPy, and Matplotlib. The model was trained on the standard MNIST training set and then tested on the separate test set. In my runs, the CNN reached about 98-99% test accuracy, which is in the normal range for this kind of architecture on MNIST.

The project has three main parts: training and saving the model, evaluating the model and visualizing some predictions, and an interactive UI that shows how the model behaves on user-drawn digits. The results are good on clean MNIST style digits, but the model struggles more on very messy or off center drawings. In the future, I could improve this with data augmentation, a deeper model, and a web based interface.

1. Introduction

In this project I tried to build a basic AI system that recognizes handwritten digits from images. This is a classic problem in machine learning and is often used to practice image classification with neural networks. The dataset I used is MNIST, which contains thousands of small images of digits 0–9.

My goals were:

- Load and preprocess the MNIST dataset.
- Build and train a simple CNN to classify digits.
- Measure test accuracy and look at some example predictions.
- Build a local, interactive UI where I can draw digits and see the model's prediction and probability graph in real time.

I focused on a straightforward CNN and clear code, rather than very advanced tricks, so I could understand each step of the pipeline.

2. Dataset and Preprocessing

2.1 MNIST Dataset: I used the MNIST dataset, which is available directly in Keras. It has:

- 60,000 training images and 10,000 test images
- Each image is a 28×28 grayscale picture of a handwritten digit
- Each image has a label between 0 and 9

The data is loaded with `tf.keras.datasets.mnist.load_data()` and is already split into training and test sets.

2.2 Preprocessing: Before feeding the images to the CNN, I applied two simple preprocessing steps:

1. **Normalization:** convert pixel values from integers in [0, 255] to floats in [0, 1] by dividing by 255.
2. **Reshaping:** the original shape is (28, 28). I added a channel dimension so the shape becomes (28, 28, 1), which is what Keras convolutional layers expect.

After this, the training data has shape (60000, 28, 28, 1) and the test data has shape (10000, 28, 28, 1).

3. Model Architecture

I used one simple convolutional neural network for this project. The idea is to let convolutional layers learn useful patterns like edges and curves from the digit images.

The architecture is:

- Input layer: shape (28, 28, 1)
- Conv2D layer with 32 filters, kernel size 3×3 , activation ReLU
- MaxPooling2D layer with pool size 2×2
- Conv2D layer with 64 filters, kernel size 3×3 , activation ReLU
- MaxPooling2D layer with pool size 2×2
- Flatten layer to turn feature maps into a vector

- Dense layer with 64 units, activation ReLU
- Output Dense layer with 10 units, activation softmax (one score per digit)

For training, I used:

- Loss: sparse categorical cross-entropy
- Optimizer: Adam
- Metric: accuracy

This architecture is implemented in `train_mnist_cnn` and compiled with `model.compile(...)`.

4. Training Procedure

I trained the CNN on the 60,000 MNIST training images using the following setup:

- Epochs: 5
- Batch size: 128
- Validation split: 0.1 (10% of the training data used as validation)

In code, I used:

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

During training, Keras printed the loss and accuracy for both the training and validation sets at each epoch. This showed that the model was learning and not strongly overfitting in this short run.

After training, I evaluated the model on the test set using:

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
```

This gives a final test loss and test accuracy on the 10,000 test images.

5. Results and Evaluation

5.1 Test Accuracy: On the MNIST test set, the trained CNN achieved:

Test accuracy: 99.03%

This is in line with what is usually reported for a small CNN on MNIST, so the model seems to be working correctly.

```
Train shape: (60000, 28, 28, 1) Test shape: (10000, 28, 28, 1)

Model: "sequential_2"



| Layer (type)                   | Output Shape       | Param # |
|--------------------------------|--------------------|---------|
| conv2d_4 (Conv2D)              | (None, 26, 26, 32) | 320     |
| max_pooling2d_4 (MaxPooling2D) | (None, 13, 13, 32) | 0       |
| conv2d_5 (Conv2D)              | (None, 11, 11, 64) | 18,496  |
| max_pooling2d_5 (MaxPooling2D) | (None, 5, 5, 64)   | 0       |
| flatten_2 (Flatten)            | (None, 1600)       | 0       |
| dense_4 (Dense)                | (None, 64)         | 102,464 |
| dense_5 (Dense)                | (None, 10)         | 650     |



Total params: 121,930 (476.29 KB)

Trainable params: 121,930 (476.29 KB)

Non-trainable params: 0 (0.00 B)

Epoch 1/5
422/422 21s 44ms/step - accuracy: 0.9300 - loss: 0.2439 - val_accuracy: 0.9827 - val_loss: 0.0664
Epoch 2/5
422/422 18s 42ms/step - accuracy: 0.9807 - loss: 0.0633 - val_accuracy: 0.9863 - val_loss: 0.0462
Epoch 3/5
422/422 18s 43ms/step - accuracy: 0.9855 - loss: 0.0455 - val_accuracy: 0.9820 - val_loss: 0.0659
Epoch 4/5
422/422 20s 41ms/step - accuracy: 0.9891 - loss: 0.0347 - val_accuracy: 0.9865 - val_loss: 0.0513
Epoch 5/5
422/422 16s 39ms/step - accuracy: 0.9911 - loss: 0.0287 - val_accuracy: 0.9885 - val_loss: 0.0387
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
Test accuracy: 99.03%
Model saved as mnist_cnn_model.h5
```

5.2 Example Predictions: To get a better feel for how the model behaves, I wrote a small script/notebook (evaluate_and_visualize) that:

- Loads the saved model mnist_cnn_model.h5
- Takes a batch of test images (for example, 16 images)
- Gets predictions from the model
- Plots the 16 images in a 4×4 grid
- Shows for each image: T: true_label and P: predicted_label

6. Interactive User Interface

6.1 Design:

Besides training the model, I wanted a simple way to interact with it. For this, I implemented `digit_drawer.py`. It is a small local UI built with Matplotlib, not a full website, but it is enough to demo the model.

The window is split into three main parts:

- Left panel (canvas): a 28×28 drawing area. I draw with the mouse by holding down the left button.
- Middle panel (text): shows the predicted digit and the model's confidence (as a percentage).
- Right panel (bar chart): shows a horizontal bar graph with 10 bars, one for each digit from 0 to 9. The length of each bar is the predicted probability for that digit.

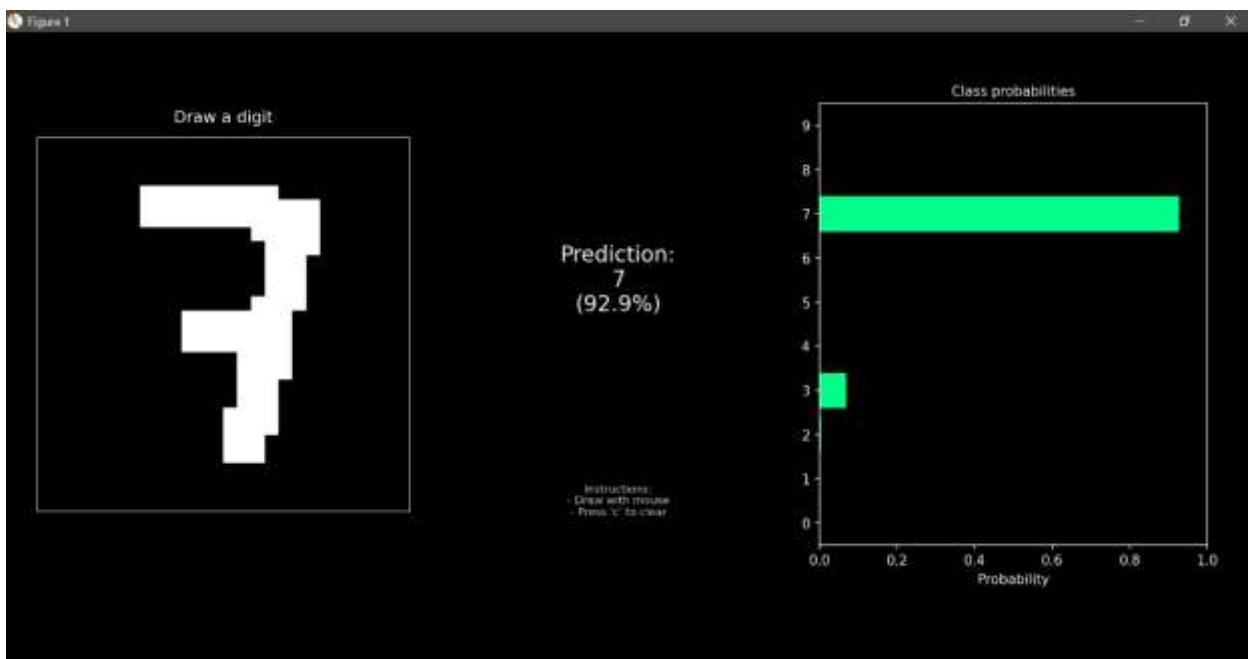
The UI works like this:

- The current canvas is stored as a 28×28 array of pixel values in $[0, 1]$.
- A timer calls a function every 500 milliseconds.
- Each time, if the canvas is not empty, the function reshapes the canvas to $(1, 28, 28, 1)$, feeds it to the trained model, and gets a probability vector of length 10.
- The middle text is updated with the top predicted digit and its confidence.
- The right bar chart is updated so each bar shows the model's probability for that digit.
- Pressing the key 'c' clears the canvas and resets the prediction.

6.2 Behavior and Observations: When I draw a clear, centered digit, the model usually predicts it correctly with high confidence

(for example, over 95%). The bar chart makes it obvious, because the bar for the true digit is much longer than the others.

When I draw a messy digit, draw too small, or draw off-center, the model can become less confident or even wrong. In that case, the bar chart often shows several bars with similar heights, which means the model is uncertain between multiple digits. This makes the UI a good way to see the strengths and weaknesses of the model.



7. Limitations

Even though the model works well on MNIST, there are some clear limitations:

- It is trained only on MNIST, which has relatively clean and standardized digits. Real-world handwriting in notebooks or photos might look very different.
- The UI expects digits to be drawn fairly large and close to the center of the canvas. If they are too small or off-center, the model performance drops.

- I used only one simple CNN architecture and a small number of epochs. I did not experiment with deeper models, regularization, or data augmentation in this version.
- The interface is a local Matplotlib window, not a full web or mobile app, so it is mainly for demonstration on my own computer.

8. Future Work

If I continue this project, there are several possible improvements:

- Add data augmentation (random rotations, shifts, small amounts of noise) to make the model more robust to different handwriting styles.
- Try more advanced architectures or ensembles to see if I can push the accuracy higher or improve robustness.
- Add explainability tools such as saliency maps to show which parts of the image the model is focusing on.
- Turn the current local UI into a small web app using Flask or another framework, so it can be accessed in a browser.
- Extend the problem from digits to letters (A–Z) or to multiple languages.

9. GitHub Repository

All the code, the trained model, and this report are available on GitHub- https://github.com/VikSaahil/VS_DIGIT_RECOGNIZER

The repository contains:

- `train_mnist_cnn.ipynb` (or `.py`): trains the CNN on MNIST and saves `mnist_cnn_model.h5`.
- `evaluate_and_visualize.ipynb` (or `.py`): loads the model, evaluates it on the test set, and shows a grid of predictions.
- `digit_drawer.py` (or `.ipynb`): local interactive drawing UI with live class-probability graph.
- `mnist_cnn_model.h5`: the trained model file

10. References

- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324.
- MNIST handwritten digit database. Retrieved from <http://yann.lecun.com/exdb/mnist/>
- TensorFlow documentation. Retrieved from <https://www.tensorflow.org/>
- Sanderson, G. (3Blue1Brown). But what is a neural network? Retrieved from <https://www.3blue1brown.com/lessons/neural-networks>