Project Report

Handwritten Text Recognition

BA12-034 Nguyễn Duy Cương

BA12-015 Nguyễn Ngọc Anh

BA12-128 Vũ Ngọc Minh

BA12-126 Hà Tấn Minh

BA12-193 Nguyễn Lân Việt

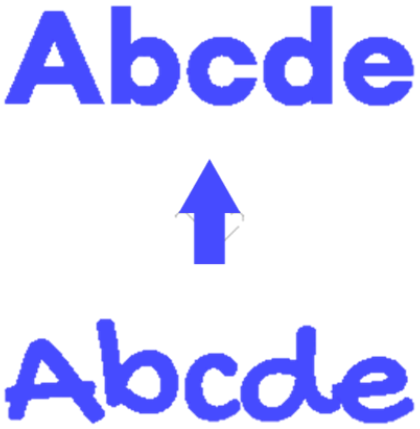
Contents

Abstract

In the field of Computer Vision, Optical Character Recognition (OCR) encompasses the automatic identification of characters from digital images and printed documents. This project is developed using Python and leverages the TensorFlow framework to construct and train the models. The process includes code for preprocessing input images, developing and training the AlexNet and LeNet-5 models, and evaluating their accuracy on a validation EMNIST dataset.

I. Introduction

Handwritten digit recognition refers to the process of converting handwritten text into a machine-readable format. This task plays a crucial role in applications such as digitizing historical documents, automating data entry from forms, and enabling secure handwriting-based authentication systems. However, it poses significant challenges due to the wide variability in individual writing styles and the difficulty in distinguishing visually similar characters.



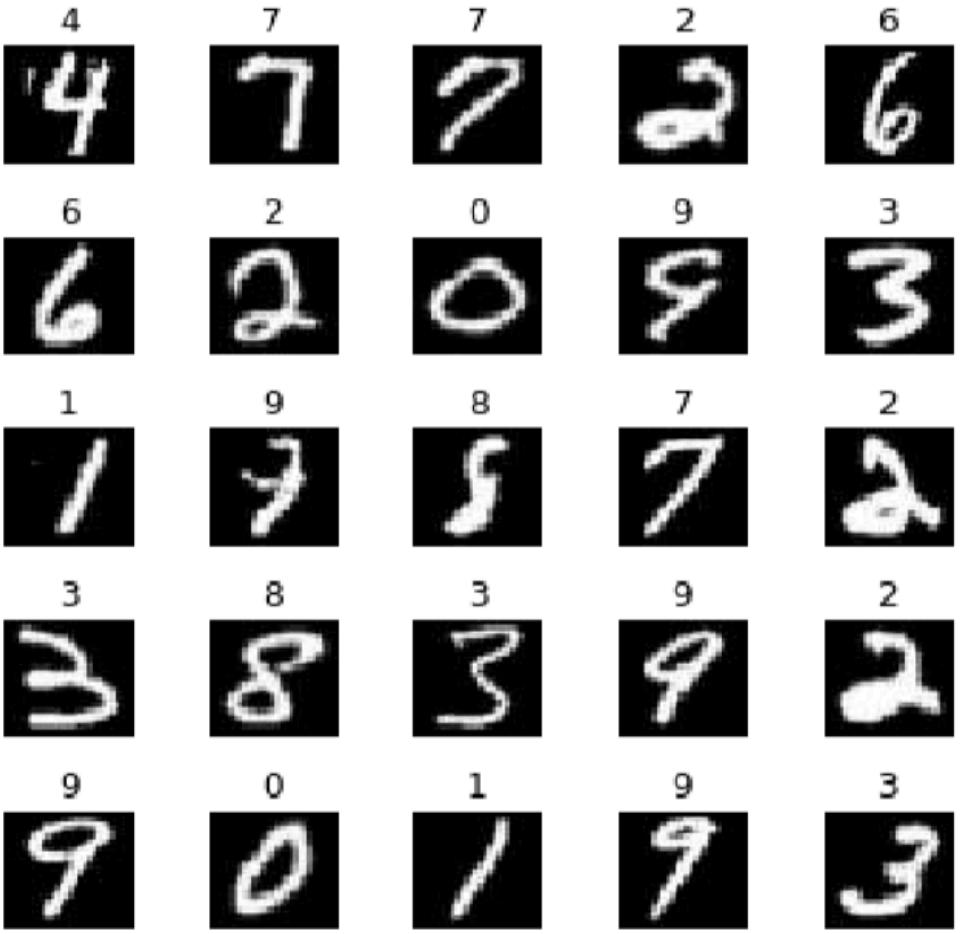
Recent advancements in deep learning and computer vision have enabled the development of efficient models capable of accurately recognizing handwritten digits. This project focuses on applying deep learning techniques to the task of handwritten digit recognition, with the goal of building a model that can reliably identify digits from 0 to 9 in various handwriting styles.

The primary objective of this project is to train a model that takes grayscale images of handwritten digits as input and outputs the correct digit label. This involves utilizing a deep learning model trained on a large dataset of digit samples and optimizing it to achieve high recognition accuracy.

A robust handwritten digit recognition system can greatly benefit applications such as automated form processing, postal code reading, and digit-based input systems. This study aims to contribute to the ongoing development of digit recognition technology by demonstrating the effectiveness of deep learning approaches in handling this specific and practical problem.

II. Methodology

1. Dataset

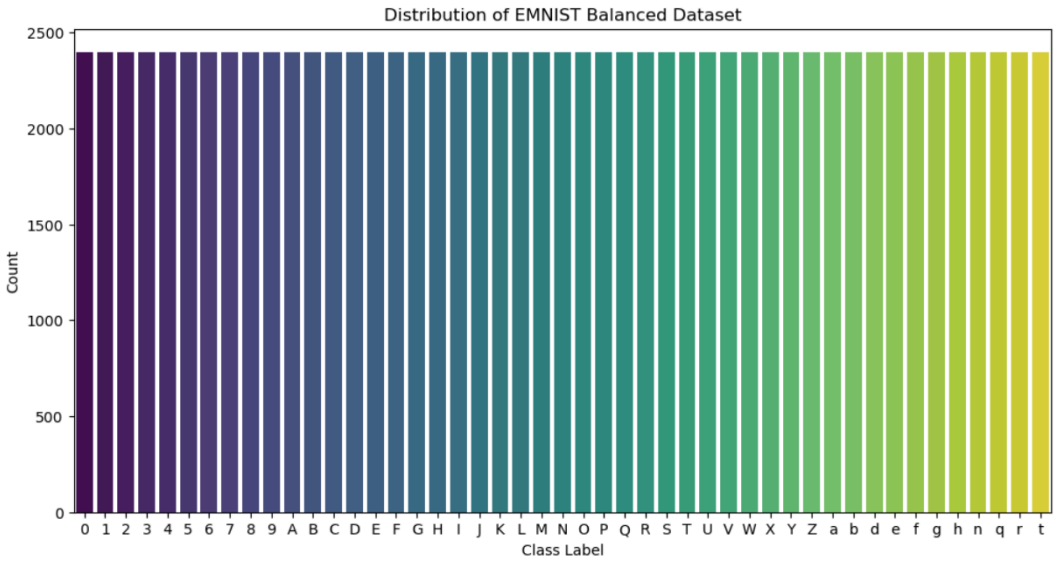


EMNIST

- The dataset comprises 3,000 data points per label, with a total of 10 labels representing the digits from 0 to 9. In total, the dataset contains 30,000 data points, structured in a shape of (30000, 785), indicating 30,000 rows and 785 columns. The first column denotes the label (the correct classification for each image), while the remaining 784 columns represent the pixel values of 28x28 grayscale images.
- The dataset consists of grayscale images of handwritten digits paired with their corresponding labels. Each image is represented as a 784-pixel value array (28x28), and the labels are integers ranging from 0 to 9, corresponding to the ten digit classes. This dataset is designed for training and evaluating machine learning models for handwritten digit recognition tasks.

2. Pre-processing phase

The pre-processing phase is vital in any project, as it significantly impacts a model's accuracy and performance. It ensures the data is clean and scalable, minimizing bias and reducing complexity. It is essential to determine whether the EMNIST dataset exhibits any imbalance in label distribution.



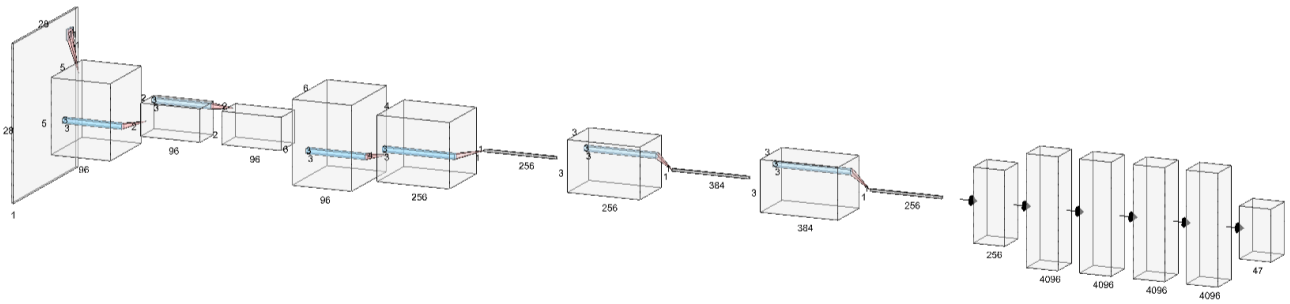
The chart above demonstrates that the data distribution across labels is perfectly balanced, ensuring that the model avoids biased predictions toward minority classes.

Observations indicate that the image content is misaligned in some cases. For example, the letter "W" in the lower right corner is misinterpreted as "3." To address this, all content is first vertically rotated and then flipped 90 degrees counter-clockwise to ensure proper alignment.

Finally, pixel values are normalized to the [0, 1] range, completing the pre-processing and ensuring accurate data representation.

3. Model

3.1. AlexNet



The input to the model is an image of shape (HEIGHT, WIDTH, 1), where HEIGHT and WIDTH are the height and width of the input image, and 1 is the number of channels.

The first layer is a convolutional layer with 96 filters of size 11x11 and a stride of 4. The layer is followed by a ReLU activation function and a max-pooling layer with a filter size of 3x3 and a stride of 2.

The second layer is a convolutional layer with 256 filters of size 5x5 and a stride of 1. This layer is also followed by a ReLU activation function and a max-pooling layer with a filter size of 3x3 and a stride of 2.

The third, fourth, and fifth layers are convolutional layers with 384, 384, and 256 filters of size 3x3, respectively, and a stride of 1. The fourth and fifth convolutional layers are not followed by pooling layers due to the fact that the dimension of the previous layer becomes so small, which is not enough for the next layer.

The output of the last convolutional layer is flattened and passed to two fully connected layers with 4096 units each. The first fully connected layer uses the ReLU activation function and a dropout rate of 0.5. The second fully connected layer also uses the ReLU activation function and a dropout rate of 0.5.

The output layer has softmax activation and n_outputs units, where n_outputs is the number of output classes.

AlexNet Training Time: 1 hours 57 minutes 29 seconds

Path to dataset files: /kaggle/input/mnist-in-csv

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 7, 7, 96)	11,712
max_pooling2d_2 (MaxPooling2D)	(None, 3, 3, 96)	0
batch_normalization_1 (BatchNormalization)	(None, 3, 3, 96)	384
conv2d_6 (Conv2D)	(None, 3, 3, 256)	614,656
max_pooling2d_3 (MaxPooling2D)	(None, 1, 1, 256)	0
conv2d_7 (Conv2D)	(None, 1, 1, 384)	885,120
conv2d_8 (Conv2D)	(None, 1, 1, 384)	1,327,488
conv2d_9 (Conv2D)	(None, 1, 1, 256)	884,992
flatten_1 (Flatten)	(None, 256)	0
dense_3 (Dense)	(None, 4096)	1,052,672
dropout_2 (Dropout)	(None, 4096)	0
dense_4 (Dense)	(None, 4096)	16,781,312
dropout_3 (Dropout)	(None, 4096)	0
dense_5 (Dense)	(None, 10)	40,970

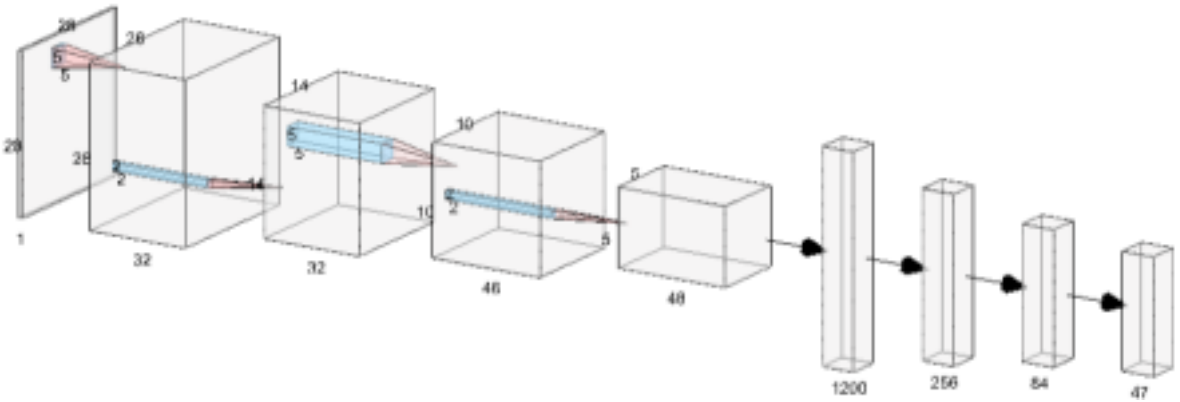
Total params: 21,599,306 (82.39 MB)
Trainable params: 21,599,114 (82.39 MB)
Non-trainable params: 192 (768.00 B)

```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.7528 - loss: 0.6669WARNING: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 recomm
469/469 ━━━━━━━━━━━━━━━━━━━━ 701s 1s/step - accuracy: 0.7531 - loss: 0.6661 - val_accuracy: 0.9593 - val_loss: 0.1467
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.9751 - loss: 0.0991WARNING: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 recomm
469/469 ━━━━━━━━━━━━━━━━━━━━ 724s 1s/step - accuracy: 0.9751 - loss: 0.0991 - val_accuracy: 0.9733 - val_loss: 0.1201
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.9819 - loss: 0.0745WARNING: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 recomm
469/469 ━━━━━━━━━━━━━━━━━━━━ 664s 1s/step - accuracy: 0.9819 - loss: 0.0745 - val_accuracy: 0.9774 - val_loss: 0.0924
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 691s 1s/step - accuracy: 0.9842 - loss: 0.0643 - val_accuracy: 0.9764 - val_loss: 0.1009
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.9890 - loss: 0.0437WARNING: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 recomm
469/469 ━━━━━━━━━━━━━━━━━━━━ 735s 1s/step - accuracy: 0.9890 - loss: 0.0437 - val_accuracy: 0.9847 - val_loss: 0.0699
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.9895 - loss: 0.0448WARNING: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 recomm
469/469 ━━━━━━━━━━━━━━━━━━━━ 680s 1s/step - accuracy: 0.9895 - loss: 0.0448 - val_accuracy: 0.9841 - val_loss: 0.0642
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 693s 1s/step - accuracy: 0.9910 - loss: 0.0375 - val_accuracy: 0.9826 - val_loss: 0.1043
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 679s 1s/step - accuracy: 0.9912 - loss: 0.0389 - val_accuracy: 0.9823 - val_loss: 0.0862
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 711s 2s/step - accuracy: 0.9924 - loss: 0.0292 - val_accuracy: 0.9853 - val_loss: 0.0801
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 716s 1s/step - accuracy: 0.9926 - loss: 0.0354 - val_accuracy: 0.9868 - val_loss: 0.0830

AlexNet Training Time: 1 hours 57 minutes 29 seconds
313/313 ━━━━━━━━━━━━━━━━━━━━ 17s 55ms/step

AlexNet Evaluation Metrics:
Accuracy: 0.98680
Precision: 0.98687
Recall: 0.98680
F1-score: 0.98680
```

3.2. LeNet-5



The input to the model is an image of shape (HEIGHT, WIDTH, 1), where HEIGHT and WIDTH are the height and width of the input image, and 1 is the number of channels.

The first layer is a convolutional layer with 32 filters of size 5x5, using the same padding and ReLU activation function. The layer is followed by a max-pooling layer with a filter size of 2x2 and a stride of 2.

The second layer is a convolutional layer with 48 filters of size 5x5, using valid padding and ReLU activation function. This layer is also followed by a max-pooling layer with a filter size of 2x2 and a stride of 2.

The output of the last convolutional layer is flattened and passed to two fully connected layers with 256 and 84 units, respectively. Both fully connected layers use the ReLU activation function

The output layer has softmax activation and n_outputs units, where n_outputs is the number of output classes.

LeNet-5 Training Time: 0 hours 19 minutes 3 seconds

```
Downloading from https://www.kaggle.com/api/v1/datasets/download/oddrationale/mnist-in-csv?dataset_version_number=2...
100%|██████████| 15.2M/15.2M [00:00<00:00, 16.4MB/s]Extracting files...

Path to dataset files: /root/.cache/kagglehub/datasets/oddrationale/mnist-in-csv/versions/2
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	832
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 10, 10, 48)	38,448
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 48)	0
flatten (Flatten)	(None, 1200)	0
dense (Dense)	(None, 256)	307,456
dense_1 (Dense)	(None, 84)	21,588
dense_2 (Dense)	(None, 10)	850

Total params: 369,174 (1.41 MB)
Trainable params: 369,174 (1.41 MB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 182ms/step - accuracy: 0.8706 - loss: 0.4343WARNING: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 re
469/469 ━━━━━━━━━━━━━━━━━━━━ 93s 195ms/step - accuracy: 0.8707 - loss: 0.4337 - val_accuracy: 0.9815 - val_loss: 0.0558
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 183ms/step - accuracy: 0.9848 - loss: 0.0505WARNING: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 re
469/469 ━━━━━━━━━━━━━━━━━━━━ 142s 194ms/step - accuracy: 0.9848 - loss: 0.0505 - val_accuracy: 0.9824 - val_loss: 0.0543
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 182ms/step - accuracy: 0.9897 - loss: 0.0343WARNING: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 re
469/469 ━━━━━━━━━━━━━━━━━━━━ 90s 193ms/step - accuracy: 0.9897 - loss: 0.0343 - val_accuracy: 0.9862 - val_loss: 0.0425
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 184ms/step - accuracy: 0.9917 - loss: 0.0266WARNING: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 re
469/469 ━━━━━━━━━━━━━━━━━━━━ 92s 196ms/step - accuracy: 0.9917 - loss: 0.0266 - val_accuracy: 0.9895 - val_loss: 0.0341
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 130s 169ms/step - accuracy: 0.9939 - loss: 0.0200 - val_accuracy: 0.9888 - val_loss: 0.0379
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 79s 169ms/step - accuracy: 0.9950 - loss: 0.0161 - val_accuracy: 0.9894 - val_loss: 0.0344
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 0s 171ms/step - accuracy: 0.9958 - loss: 0.0124WARNING: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 re
469/469 ━━━━━━━━━━━━━━━━━━━━ 88s 182ms/step - accuracy: 0.9958 - loss: 0.0124 - val_accuracy: 0.9916 - val_loss: 0.0259
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 147s 193ms/step - accuracy: 0.9970 - loss: 0.0096 - val_accuracy: 0.9905 - val_loss: 0.0329
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 147s 205ms/step - accuracy: 0.9970 - loss: 0.0089 - val_accuracy: 0.9885 - val_loss: 0.0428
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 135s 189ms/step - accuracy: 0.9963 - loss: 0.0101 - val_accuracy: 0.9915 - val_loss: 0.0334

LeNet-5 Training Time: 0 hours 19 minutes 3 seconds
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step

LeNet-5 Evaluation Metrics:
Accuracy: 0.99150
Precision: 0.99153
Recall: 0.99150
F1-score: 0.99149
```

3.3. Evaluation Model

Factor	AlexNet	LeNet-5	Impact on Speed
Model Complexity	21,599,306 parameters (82.39 MB)	369,174 parameters (1.41 MB)	AlexNet's higher parameter count increases computational requirements, slowing training.
Number of Layers	5 convolutional layers, 3 fully connected layers (2 with 4096 units)	2 convolutional layers, 3 fully connected layers (256 and 84 units)	AlexNet's deeper architecture requires more matrix operations, increasing training time.
Filter Sizes and Feature MapsFilter Sizes and Feature Maps	Feature Maps Larger filters (e.g., 11x11) and more filters (up to 384)	Smaller filters (5x5) and fewer filters (up to 48)	AlexNet's larger filters and feature maps demand more computations, slowing training.
Dropout Layers	Uses dropout (0.5) in fully connected layers	No dropout used	AlexNet's dropout adds computational overhead, further slowing training.
Input Processing	Processes 28x28 images with more intermediate feature maps due to depth	Processes 28x28 images with fewer feature maps due to simpler architecture	AlexNet's deeper structure generates more feature maps, increasing memory and time.
Generalization vs. Overfitting	Complex architecture may lead to overfitting, requiring more epochs to converge	Simpler architecture converges faster, generalizes better	AlexNet's slower convergence due to overfitting extends training time.

Conclusion: AlexNet is Slower than LeNet-5 in a general comparison

4. Evaluation Matrices

In the following table, we compare the performance of two different neural network models, AlexNet and LeNet-5, on a certain classification task. The task involved analyzing a dataset of images, and the objective was to accurately classify each image into one of several predefined categories. To measure the performance of the models, we used four different metrics: accuracy, precision, recall, and F1-score.

➔ Accuracy: The proportion of correctly classified samples out of the total number of samples. It is computed as
(TP + TN) / (TP + FP + TN + FN)

➔ Precision: The proportion of correctly classified positive samples out of the total number of samples that were predicted as positive. It is computed as
TP / (TP + FP)

➔ Recall: The proportion of correctly classified positive samples out of the total number of positive samples in the test set. It is computed as
TP / (TP + FN)

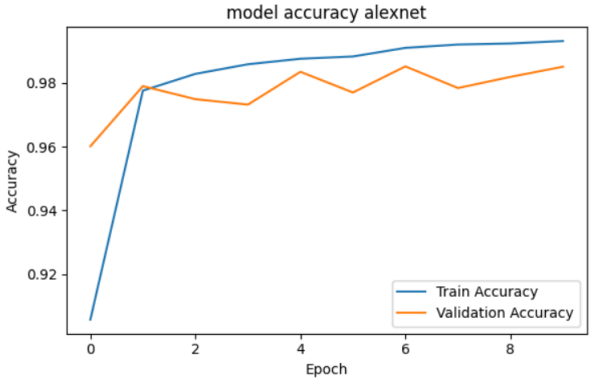
➔ F1-score: The harmonic mean of precision and recall. It is computed as
*2 * (precision * recall) / (precision + recall)*

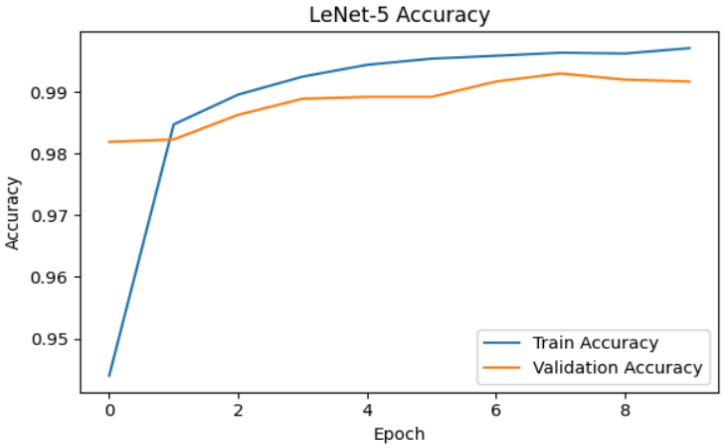
Model	**Accuracy**	**Precision Recall F1-score**
AlexNet	0.98680	0.98687 0.98680 0.98680
LeNet-5	0.99150	0.99153 0.99150 0.99149

Both models perform well, with LeNet-5 slightly outperforming AlexNet in terms of overall performance, as it has higher values for all four metrics. One probable explanation is that LeNet-5 has a simpler architecture than AlexNet, making it less susceptible to overfitting on training data and better at generalizing to new data.

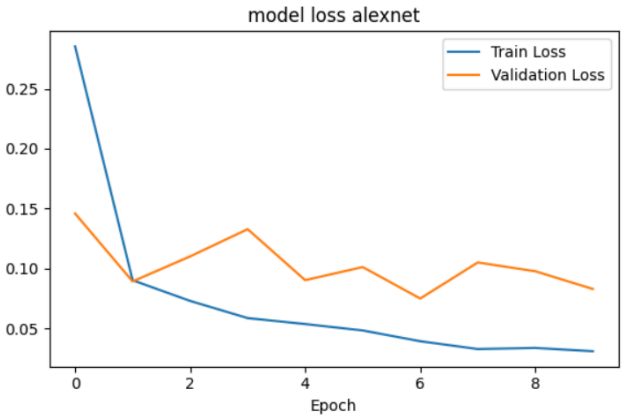
Training Metrics:

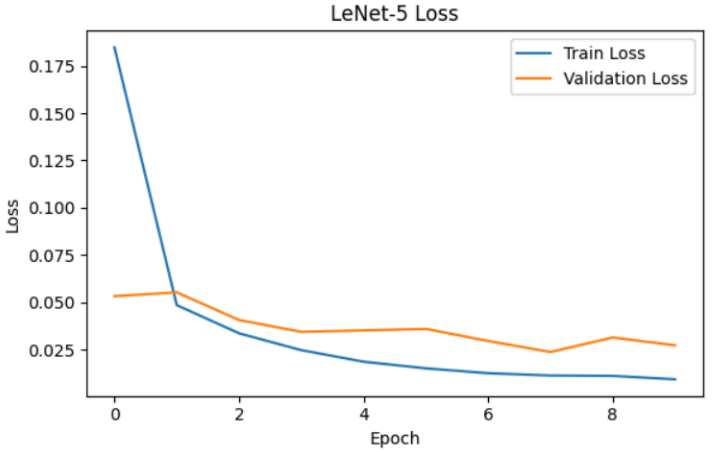
The validation data in the training process is used to evaluate the model with the data that it is not taught, in order to know whether the model is overfitting or not. As can be seen, the 2 lines are close together and both show good performance.





Whenever save_best_only is set to True, the callback will only store model weights that have the highest value for the monotored quantity, preserving the greatest validation accuracy throughout training.





The callback will only save model weights that provide the best value for the monitored quantity if save_best_only is set to True (in this case, validation loss). The callback will preserve the weights of the model with the lowest validation loss during training, for instance, if the monitored quantity is val_loss.

5. Characters Detection

For detecting handwritten characters in an image, the first task is to convert the image into grayscale. Then we will clean the image by reducing the noise of the image using Gaussian blur then we will detect the contours in the edged image.

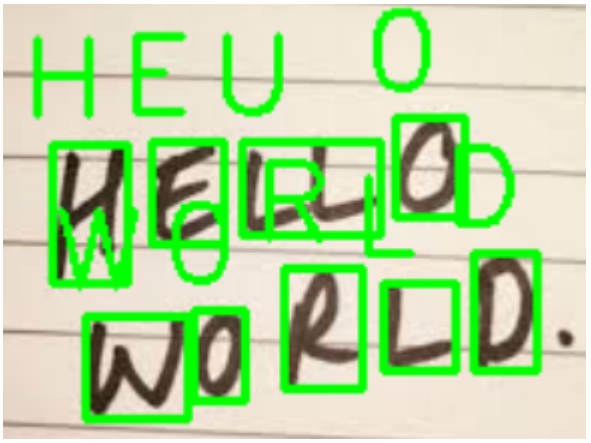
After that, we will take all coordinates of regions of interest for each character, Next step is to modify the size for each picture of characters that is detected into 28x28 pictures for prediction. The image below shows a character after being detected

After making predictions, the output is an image with the bounding boxes surrounding characters.

III. Result

From the image above, it can be seen that most of the words are detected correctly except for the “!” and the icons which are not included in the dataset. The model detects them and compares them to the whole label and finds which is the most similar to them.

To gain more clarity about the behavior of the model, more image has been input for the model, for example, another result below.



From observations, 2 “L”s in the image have been merged into a single “U”. This does make sense because in real life that may be possible for some characters to be connected when writers write quickly. This leads to confusion for the model, it seems that the many characters as a single character, predict an incorrect result.

IV. Discussion

After taking some experiments, the model is still too simple, mapping real-life characters into incorrect predictions. So for this, all the labels that are predicted incorrectly can be re-labels and fed back to the model for the next training so that the model can have better performance in the future.

1. How People Around the World Solve the Problem

Handwritten text recognition is a well-studied problem in computer vision, and various approaches have been developed globally. A notable example is the Kaggle notebook by Eishkaran titled "Handwritten Text Recognition" (https://www.kaggle.com/code/eishkaran/handwritten-text-recognition). Key aspects of global approaches, including this example, include:

- Datasets: Commonly used datasets include MNIST (for digits), EMNIST (for digits and letters), and IAM (for handwritten words or sentences). These datasets provide labeled images for training and testing.

- Preprocessing: Images are preprocessed to standardize input. Common techniques include:

  - Converting images to grayscale.

  - Normalizing pixel values to a range (e.g., [0, 1]).

  - Resizing images to a fixed size (e.g., 28x28 pixels).

  - Applying data augmentation (e.g., rotation, flipping) to increase robustness.

- Models: Deep learning models, particularly Convolutional Neural Networks (CNNs), are widely used due to their ability to extract spatial features from images. Examples include:

  - Simple CNN architectures like LeNet-5 for smaller datasets.

  - More complex models like AlexNet, VGG, or ResNet for larger datasets or more challenging tasks.

  - Recurrent Neural Networks (RNNs) or Connectionist Temporal Classification (CTC) for sequence-based recognition (e.g., words or sentences).

- Training and Evaluation: Models are trained using labeled datasets, with techniques like train-test splits or k-fold cross-validation. Performance is evaluated using metrics such as accuracy, precision, recall, F1-score, and confusion matrices to identify misclassifications.

- Error Handling: Errors often arise from visually similar characters (e.g., "0" vs. "O", "1" vs. "I"). Techniques like data augmentation, ensemble models, or post-processing (e.g., spell-checking) are used to mitigate these issues.

- Example from Kaggle: Eishkaran's notebook likely uses a CNN-based approach (similar to LeNet-5 or a custom CNN) trained on a dataset like MNIST or EMNIST. It includes preprocessing steps (e.g., normalization, resizing), model training, and evaluation using accuracy and confusion matrices. The notebook may also explore techniques like data augmentation to handle variations in handwriting

2. How our group solves the problem

We outlines a systematic approach to handwritten text recognition using deep learning models, specifically AlexNet and LeNet-5, on the EMNIST dataset.

- Dataset:

- Description: The EMNIST dataset is used, which extends the MNIST dataset by including both handwritten digits (0–9) and letters (uppercase and lowercase). It contains grayscale images of size 28x28 pixels.

- Balance Check: The dataset is analyzed to ensure a balanced label distribution, preventing bias toward any class. The report confirms that the EMNIST dataset is perfectly balanced across labels.

- Preprocessing Phase:

- Alignment Correction: Some images in the EMNIST dataset are misaligned (e.g., the letter "W" misinterpreted as "3"). To address this, images are vertically rotated and flipped 90 degrees counter-clockwise.

- Normalization: Pixel values are normalized to the range [0, 1] to ensure consistent input to the models.

- Character Detection: For real-world application, images are converted to grayscale, noise is reduced using Gaussian blur, and contours are detected to identify regions of interest (ROIs). Each detected character is resized to 28x28 pixels for prediction.

- Method (Train-Test Split or Cross-Validation):

- Train-Test Split: The report implies a train-test split, as it mentions evaluating models on a validation EMNIST dataset. The validation data is used to assess model performance on unseen data and detect overfitting.

- Callback Mechanism: A callback with `save_best_only=True` is used to save model weights with the highest validation accuracy or lowest validation loss, ensuring optimal performance.

- Models:

- AlexNet:

- Architecture: Consists of five convolutional layers (96, 256, 384, 384, and 256 filters) with ReLU activation, max-pooling layers, and two fully connected layers (4096 units each) with dropout (0.5). The output layer uses softmax activation for n classes (n = number of output classes, likely 47 for EMNIST digits and letters).

- Parameters: 21,599,306 parameters (82.39 MB).

- Training Time: 1 hour, 57 minutes, 29 seconds.

- LeNet-5:

- Architecture: Consists of two convolutional layers (32 and 48 filters) with ReLU activation, max-pooling layers, and two fully connected layers (256 and 84 units). The output layer uses softmax activation.

- Parameters: 369,174 parameters (1.41 MB).

- Training Time: 19 minutes, 3 seconds.

- Implementation: Models are implemented using Python and TensorFlow, trained on the preprocessed EMNIST dataset.

- Evaluation Performances:

- Metrics: Accuracy, precision, recall, and F1-score are computed using scikit-learn functions, with true labels (`y_test`) and predicted labels (`y_pred`) as inputs.

- Results:

- AlexNet: Accuracy = 0.98680, Precision = 0.98687, Recall = 0.98680, F1-score = 0.98680.

- LeNet-5: Accuracy = 0.99150, Precision = 0.99153, Recall = 0.99150, F1-score = 0.99149.

- Observation: LeNet-5 outperforms AlexNet across all metrics, with higher accuracy and fewer misclassifications.

- Comparison:

- LeNet-5 vs. AlexNet:

  - LeNet-5 achieves higher performance (accuracy of 0.99150 vs. 0.98680 for AlexNet).

  - LeNet-5 has a simpler architecture with fewer parameters (369,174 vs. 21,599,306), making it less prone to overfitting and faster to train.

  - Confusion matrices show that LeNet-5 has fewer incorrect predictions off the diagonal, indicating better class discrimination.

- Explanation: The report suggests that LeNet-5's simpler architecture allows better generalization to the EMNIST dataset, while AlexNet's complexity may lead to overfitting on the training data.

- Error Analysis:

- Common Errors: The models struggle with visually similar characters, such as:

  - Capital "O" vs. number "0".

  - Capital "I" vs. number "1" or lowercase "i".

  - Connected characters (e.g., two "L"s merged into a "U" due to fast handwriting).

- Character Detection Issues: The model misclassifies non-dataset symbols (e.g., "!") or icons, as they are not part of the EMNIST labels. It attempts to find the closest match, leading to errors.

- Improvement: Incorrectly predicted labels can be re-labeled and used for further training to improve model performance. Additionally, addressing word segmentation for word-by-word recognition (instead of character-by-character) is suggested for future work.

V. Conclusion

In this project, we explored the use of deep learning models for handwritten recognition, to develop a model that could accurately recognize a diverse range of handwritten text. Through experimentation and analysis, we found that using the AlexNet and LeNet-5 led to high accuracy in handwritten recognition, with the ability to recognize a variety of writing styles and variations. We also found that data augmentation techniques such as rotation, scaling, and flipping helped to improve the robustness of the models to different writing styles and variations.

Overall, our results demonstrate the potential of deep learning models for solving the challenging problem of handwritten recognition. And with further research and development, these models could be applied to a range of real-world applications, from document digitization to handwritten-based authentication.

On the other hand, there are still some limitations and tasks that need to be solved, such as word segmentation for word-by-word recognition instead of character-by-character.

In future works, we will test out more models to evaluate whether their accuracy is superior to our top training model, Lenet5. In addition, we are willing to detect whole phrases rather than just characters.

References

- *EMNIST Dataset: https://www.kaggle.com/datasets/oderationale/mnist-in-csv*

- *Kaggle Notebook:*

https://www.kaggle.com/code/viratkothari/image-classification-of-mnist-using-vgg16/notebook]

- *OCR Tutorial:*

https://www.pyimagesearch.com/2020/08/24/ocr-handwriting-recognition-with-opencv-keras-and-tensorflow

- *Reference comparison:*

https://www.pyimagesearch.com/2020/08/24/ocr-handwriting-recognition-with-opencv-keras-and-tensorflow