ILLINOIS INSTITUTE OF TECHNOLOGY

CSP - 571 – Data Preparation and Analysis Project Report

Team Members	Student ID
Chiguru Nithin	A20583854
Vijaya Satya Aditya Karri	A20581776
Devendra Babu Kondragunta	A20550546

1. Introduction

1.1 Abstract:

Handwritten letter recognition continues to pose significant challenges in computer vision, owing to the immense variability of human writing styles and the presence of noise in scanned or photographed documents. In this project, we employ a semi-supervised convolutional neural network that leverages pseudo-labelling to harness a large pool of unlabelled examples from the EMNIST Letters dataset. After training the model on a modest labelled subset for five epochs, we generate pseudo-labels for high-confidence unlabelled samples (confidence ≥ 0.90) and retrain for two additional epochs on the combined dataset. This approach yields a test accuracy of 98.32%, confirming that pseudo-labelling effectively augments performance while reducing manual annotation effort. Key insights include the critical role of confidence thresholds in balancing label quality and quantity, the stabilizing effect of min–max normalization on gradient updates, and common error patterns such as C vs. G confusion. Looking ahead, integrating consistency-regularization techniques and extending support to cursive forms represent promising next steps.

1.1. Overview

Handwritten letters are at the heart of countless historical archives and everyday workflows, yet their free-form nature makes them notoriously difficult for machines to interpret. In our project, we set out to build a system in R that reads and recognizes the 26 letters of the English alphabet with minimal labelled data. By combining a small set of human-annotated examples with a much larger pool of unlabelled images, we aim to teach a convolutional neural network (CNN) to discern the subtle shapes and strokes that distinguish each character.

We begin by training the CNN on a carefully stratified subset of labelled images, allowing it to learn core features like edges and curves. Once this initial model is in place, we let it “guess” labels for the unlabelled images—but only keep those guesses when it is highly confident (at least 90%). These high-confidence predictions are then folded back into the training data, and the model is fine-tuned with this expanded dataset. This pseudo-labelling strategy helps the network generalize better without requiring us to hand-label every single image.

This document walks you through our entire process. (Abstract) presents our key findings and next steps; Section 1 (Overview) explains our goals and approach, with subsections 1.2 (Objectives) and 1.3 (Specific Questions) framing our targets and research inquiries; Section 2 (Literature Survey) Reviews prior work on handwriting recognition: rule-based, statistical, and deep-learning approaches, datasets, and evaluation metrics; Section 3 (Data Preparation & Cleansing) details how we collected, cleaned, and normalized over 300,000 letter images; Section 4 (Modeling) describes the CNN architecture and two-phase pseudo-label training cycle; and (Results & Analysis) reports the 98.32% test accuracy, precision/recall/F1 metrics, and confusion-matrix insights; Section 5 (Implementation & Deployment) covers our RStudio setup, R Markdown

workflows, and reproducibility measures; Section 6,7 (Future Work & Conclusion) summarizes our takeaways and outlines next research directions; and finally, Sections 8,9 and 10 list our data sources, code repository, and full bibliography.

1.2. Objective

Our primary objectives are to develop an R-based letter recognition pipeline that achieves at least 98% accuracy on the EMNIST Letters dataset, while minimizing the volume of manually labelled data. We aim to demonstrate that pseudo-labelling—a process of adding only high-confidence model predictions to training—can effectively boost performance. Additionally, we seek to provide a fully reproducible, end-to-end workflow from data ingestion through model evaluation, so others can easily build upon our results.

1.3 Specific questions

- How accurately can a semi-supervised CNN identify each of the 26 English letters when trained on only 20% labelled data?
- What new capabilities and enhancements should future iterations of our semi-supervised model explore to keep pace with evolving handwriting styles and applications?
- Which data preprocessing techniques (e.g., thresholding, smoothing, normalization) contribute most to performance gains?
- In what practical scenarios—such as archival digitization or real-time form entry—can our recognition pipeline deliver the greatest impact?

2. Literature Survey

2.1 General

2.1.1-Title: Recognition of Handwritten Digit using Convolutional Neural and Comparison of Performance for Various Hidden Layers.[3]
Author: Fathima Siddique; Shadman Sakib
In this paper, to find the performance of CNN hidden layers, CNN as well as MNIST dataset real time input. have used. The network is trained using stochastic gradient descent and the backpropagation algorithm.
Scope of this paper is to adding feature like GUI to take real time input.

2.1.2-Title: Analogizing time Complexity of KNN and CNN in Recognizing Handwritten Digits.[9]
Author: Tanya makkar; Ashwani kumar Dubey
In this paper comparison of accuracy of KNN as well as CNN have done. In this paper MNIST data set has used for providing samples to the system.
Scope of this paper is to choosing CNN because it produces high accuracy then KNN and adding GUI for real time input.

2.1.3-*Title*: Telugu handwritten character recognition using adaptive and static zoning methods. [10]
Author: Sangula Durga Prasad; Yashwanth Kanduri

In this paper they presented two methods for this purpose. First method is based on Genetic Algorithm and uses adaptive zoning topology with extracted geometric features. In second method, zoning is done in static way and uses distance, density based features. In both the contexts, they used K-Nearest Neighbor (KNN) algorithm for classification purpose.

2.2 Overview of Existing Research

Handwritten letter recognition is a rapidly evolving field with a long history of research. The study involves the development of computer algorithms that can automatically identify and interpret handwritten characters. The primary challenge in this field is the high variability in handwriting styles, which makes it difficult for algorithms to distinguish between different characters accurately.

Early approaches to handwritten letter recognition were rule-based or template matching techniques. However, these methods were limited by their inability to handle variability in handwriting. In recent years, deep learning techniques have shown significant promise in improving the accuracy of handwritten letter recognition. Convolutional neural networks (CNNs) and recurrent neural networks (RNNs) have been particularly successful in this regard.

One of the major issues in handwritten letter recognition is the lack of high-quality training data. While there are several datasets available for this purpose, they are often small and may not represent the variability in handwriting styles. Several researchers have attempted to address this issue by creating synthetic datasets or using generative adversarial networks (GANs) to generate new training data.

Another area of research in handwritten letter recognition is the development of new features and preprocessing techniques that can improve the accuracy of recognition algorithms. For example, some researchers have investigated the use of stroke-based features, which capture the structure and direction of each stroke in a handwritten character.

There have also been efforts to develop new evaluation metrics for assessing the performance of handwritten letter recognition algorithms. One such metric is the writer-independent error rate (WER), which measures the error rate when recognizing characters written by individuals who are not in the training set.

Despite the progress made in this field, there are still several challenges that need to be addressed. For example, the recognition of cursive handwriting is still a major challenge for algorithms. Additionally, there is a need for more robust algorithms that can handle noise and other forms of degradation in handwritten documents. In conclusion, handwritten letter recognition is a field that has seen significant advancements in recent years, with the development of new evaluation metrics and preprocessing techniques. However, there are still several challenges that need to be addressed before this technology can be widely deployed in practical applications.

3. Data Preparation and Cleansing

3.1 Data Collection Process

Data collection is a crucial step in the machine learning process as it provides the necessary information for training and testing models. The data collection process involves the following steps:

- Determine the problem statement: Define the problem to be solved and identify the relevant data needed to solve it.
- Identify the data sources: Identify potential sources of data, such as existing datasets, APIs, or manual collection methods.
- Collect the data: Collect data from the identified sources, either manually or through automated means.
- Preprocess the data: Clean the data by removing irrelevant or duplicated data, filling in missing values, and transforming the data into a suitable format for analysis.
- Label the data: Label the data with relevant categories or outcomes, such as class labels or numerical values.
- Split the data: Split the data into training, validation, and test sets to enable model development and evaluation.
- Store the data: Store the data in a suitable format, such as CSV, JSON, or database format, for easy access and analysis.
- Ensure data privacy and security: Ensure that the collected data is secure and that the privacy of individuals is maintained by anonymizing the data or obtaining consent where necessary.

- Effective data collection is critical for the success of machine learning models, as it enables the models to learn from real-world data and make accurate predictions or classifications.

Datasets used for handwritten letters recognition typically consist of a large number of handwritten letter images, along with corresponding labels that identify the correct letter for each image. Some of the commonly used datasets for handwritten letters recognition are:

- MNIST: This is a widely used dataset that consists of 70,000 images of handwritten digits from 0 to 9. Each image is a 28x28 grayscale image.
- EMNIST: This dataset is an extension of MNIST and consists of 240,000 images of handwritten letters and digits. The letters are from A to Z, both uppercase and lowercase, and the digits are from 0 to 9.
- NIST: This dataset consists of images of handwritten letters and digits collected by the National Institute of Standards and Technology (NIST) for research purposes.
- CEDAR: The CEDAR dataset consists of over 6,000 images of handwritten English letters, including both uppercase and lowercase letters.
- IAM: The IAM dataset consists of handwritten English words and sentences, including both cursive and printed writing styles.
- Chars74K: This dataset consists of 74,000 images of handwritten characters from various languages, including English, Arabic, and Chinese.

These datasets are widely used for training and testing machine learning models for handwritten letters recognition. They provide a diverse set of images with varying writing styles, noise, and distortions, enabling the development of robust models that can handle real-world scenarios. Since, all the above mentioned datasets are used by many in their previous works we implemented our dataset using references from them and contains more than 3 lakh images which we thought is really a good count for making the model more accurate.

Data preprocessing refers to the cleaning, transformation, and preparation of raw data before it is used for machine learning applications. The main goal of data preprocessing is to improve the quality and usability of the data by removing errors, filling in missing values, and transforming the data into a format that is suitable for analysis. Data preprocessing is an important step in machine learning because the quality and accuracy of the resulting model are directly influenced by the quality of the input data. Poor quality data can lead to inaccurate or unreliable predictions, and in some cases, may even render the model unusable.

3.2 Data Pre-Processing

In machine learning, training data and testing data are two key components used to develop and evaluate machine learning models.

Training data refers to a set of input data and corresponding output labels that are used to train a machine learning model. During the training process, the model learns the underlying patterns and relationships between the input data and output labels. The goal of training is to optimize the model's parameters or weights, so it can accurately predict the correct output label for new, unseen data.

Testing data, on the other hand, is a set of input data and output labels that are used to evaluate the performance of a trained machine learning model. The testing data is separate from the training data, and the model has not seen these data points during the training process. The goal of testing is to determine how well the model generalizes to new, unseen data.

To ensure that the model performs well on new data, the testing data should be representative of the real-world data that the model will encounter. The testing data should also be large enough to provide a reliable estimate of the model's performance. In order to prevent overfitting, it's important to use separate sets of training and testing data. Overfitting occurs when a model learns the training data too well and becomes too specific to that data. This can lead to poor performance on new, unseen data. By using separate sets of training and testing data, we can evaluate the model's ability to generalize to new data and prevent overfitting. In summary, training data is used to teach the model how to make accurate predictions, while testing data is used to evaluate the model's performance and ensure that it generalizes well to new data.

3.2.1 THRESHOLDING

Thresholding is a common technique used in image processing and computer vision, including in handwritten letters recognition, to segment an image into foreground and background. The basic idea of thresholding is to convert a grayscale image into a binary image by setting a threshold value, and any pixel value above that threshold is considered part of the foreground, and any value below the threshold is considered part of the background.

In the context of handwritten letters recognition, thresholding can be used to separate the handwritten letters from the background noise, making it easier to extract features and classify the letters. For example, after applying thresholding to an image of a handwritten letter, we can count the number of pixels in the foreground to get an estimate of the letter's size, or we can compute the horizontal and vertical projections of the letter to obtain information about its shape and orientation. There are several methods for thresholding images, including global thresholding, adaptive thresholding, and Otsu's method. Global thresholding uses a fixed threshold value for the entire image, while adaptive thresholding adjusts the threshold value for different parts of the

image based on local characteristics. Otsu's method is a widely used thresholding technique that automatically computes an optimal threshold value based on the distribution of pixel intensities in the image. Thresholding can be a useful preprocessing step in handwritten letters recognition, particularly when dealing with noisy or complex images. However, the effectiveness of thresholding depends on the quality of the input image and the specific characteristics of the handwritten letters being analyzed. For our project we apply R's imager::threshold(..., "auto") function to perform adaptive thresholding.

3.2.2 One-hot Encoding

Thresholding can be a useful preprocessing step in handwritten letters recognition, particularly when dealing with noisy or complex images. However, the effectiveness of thresholding depends on the quality of the input image and the specific characteristics of the handwritten letters being analyzed.

One hot encoding is commonly used in machine learning algorithms that require numeric input, such as neural networks and decision trees. It allows us to represent categorical variables as numerical data, making it easier for algorithms to process the data and learn relationships between the variables. However, one hot encoding can result in high-dimensional data, particularly for variables with many categories. In such cases, it may be necessary to reduce the dimensionality of the data through techniques such as feature selection or dimensionality reduction. In this project, we convert integer labels to one-hot vectors using keras::to_categorical(y, num_classes = 26).

3.2.3 Gaussian Blur

Gaussian blur is a commonly used technique in deep learning for image processing, particularly in computer vision applications. The basic idea of Gaussian blur is to smooth an image by convolving it with a Gaussian filter, which is a bell-shaped curve that assigns weights to each pixel in the image based on its distance from the center pixel.

The Gaussian blur operation is performed by sliding a kernel (i.e., the Gaussian filter) over the image and computing the weighted average of the pixel values within the kernel. The resulting smoothed image has reduced noise and sharp edges, which can be useful for improving the accuracy of object detection and recognition algorithms. In deep learning, Gaussian blur is often used as a preprocessing step for input images before they are fed into a neural network. This can help to reduce the impact of noise and other artifacts in the image and improve the overall performance of the network. Gaussian blur can also be used as a data augmentation technique in deep learning, where it is applied to randomly selected images in the training dataset. By adding random variations to the input data, data augmentation can help to prevent overfitting and improve the generalization

ability of the network. Overall, Gaussian blur is a useful technique in deep learning for image processing and data augmentation, and can help to improve the accuracy and robustness of neural network models. We apply a light Gaussian filter with imager::isoblur(image, sigma = 1) to smooth away noise while preserving stroke edges.

3.2.4 Normalization

Normalization is a common technique used in machine learning to rescale input data to a standard range. The main idea behind normalization is to transform the input data so that it has a mean of zero and a standard deviation of one, or to scale it to a specific range of values, such as between 0 and 1 or -1 and 1.

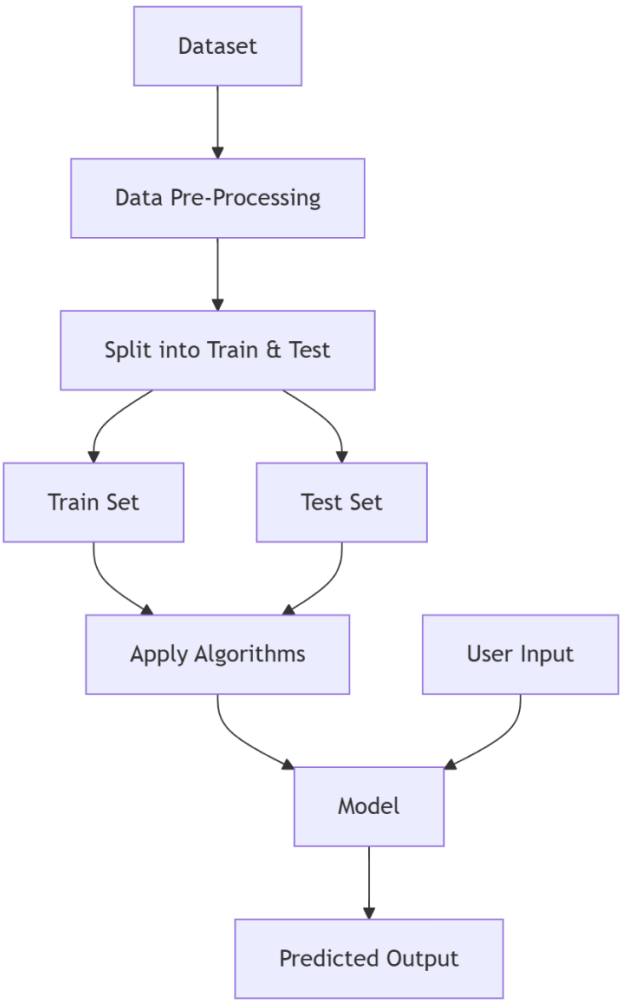
The purpose of normalization is to ensure that the input data has consistent and comparable scales, which can help to improve the performance of machine learning algorithms. For example, if one feature has a much larger range of values than another feature, it can dominate the learning process and lead to biased or inaccurate results.

There are several common normalization techniques used in machine learning, includes Z-score normalization, Min-max normalization, Decimal scaling normalization. Normalization can be applied to different types of data, including continuous and categorical variables, and can be used as a preprocessing step before training a machine learning model. However, it is important to note that normalization can also have drawbacks, such as increasing the computational complexity of the model and reducing the interpretability of the results.

4. Modelling

4.1 Architecture Diagram

CNN works as backend and GUI works as frontend. We started with data collection, next pre-processed the set of handwritten letters. After that we split the data for training & testing. We then implemented the CNN model by applying softmax, ReLU ,pooling layers. We measured the performance of our model with the help of in-built accuracy & loss functions available. Last step is to test the model by passing the input data to it. We implemented a GUI for passing the input and displaying the result.



Architecture diagram

4.2 Description of Proposed Model

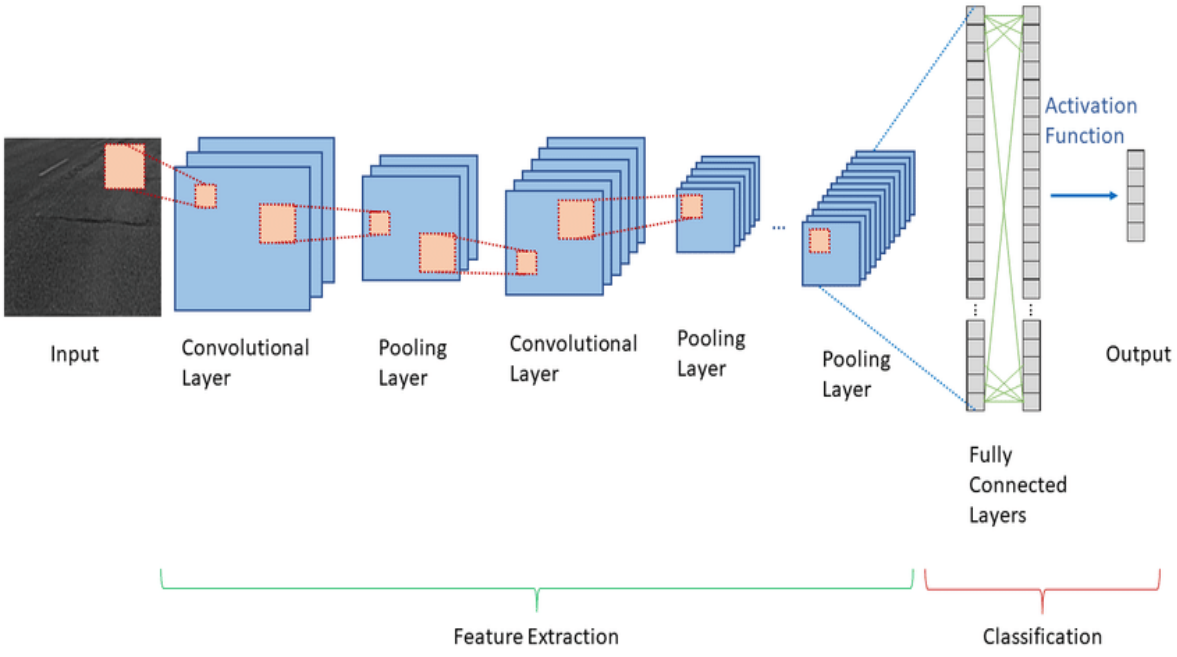
A Convolutional Neural Network (CNN) is a type of neural network that is commonly used for image processing and computer vision applications. The basic architecture of a CNN consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers.

The convolutional layer is the core building block of a CNN, and it is responsible for learning feature maps that capture the patterns and structures in the input image. In this layer, a set of filters (or kernels) are applied to the input image to produce a set of feature maps. Each filter learns to detect a specific feature, such as edges, corners, or shapes, by convolving the filter with the input image and computing a dot product.

The pooling layer is used to downsample the feature maps and reduce the dimensionality of the input data. This layer typically uses a max pooling or average pooling operation to extract the most important features from each feature map and discard the rest.

The fully connected layer is responsible for classifying the input image based on the learned features. In this layer, the feature maps are flattened into a one-dimensional vector, and a series of fully connected neurons are used to predict the class label of the input image.

The overall training process of a CNN involves feeding a large dataset of labeled images into the network, adjusting the weights of the filters using backpropagation and gradient descent, and evaluating the performance of the model on a validation set. The main objective of the training process is to minimize the loss function, which measures the difference between the predicted output of the model and the true output.In addition to the basic architecture of a CNN, there are several advanced techniques that can be used to improve its performance, such as data augmentation, regularization, and transfer learning. Data augmentation involves applying random transformations to the input images, such as rotations, flips, and zooms, to increase the size and diversity of the training dataset. Regularization techniques, such as L1 and L2 regularization, are used to prevent overfitting and improve the generalization ability of the model. Transfer learning involves using a pre-trained CNN model as a starting point for a new task, and fine-tuning the model on a smaller dataset to adapt it to the new task.



CNN Architecture diagram

Convolutional Layers: The first layer in a CNN is typically a convolutional layer, which performs a convolution operation on the input image using a set of learnable filters. Each filter produces a feature map that highlights a particular aspect of the input image, such as edges or corners.

Pooling Layers: After each convolutional layer, a pooling layer is often used to downsample the feature maps and reduce their spatial size. This helps to reduce the number of parameters in the network and improve its computational efficiency.

Activation Functions: Nonlinear activation functions such as ReLU (Rectified Linear Unit) are applied to the output of each layer to introduce nonlinearity and enable the network to learn more complex patterns.

Fully-Connected Layers: The final layers in a CNN are typically fully-connected layers, which are used to map the high-level features learned by the previous layers to the output classes. These layers use a softmax activation function to produce a probability distribution over the possible classes.

4.3 Detailing Hyper Parameters & Optimizing The Algorithm

Hyperparameters in a convolutional neural network (CNN) are values that are set before training the model and can have a significant impact on the performance of the model. Here are some common hyperparameters in a CNN:

- Number of filters: This is the number of feature maps used in each convolutional layer. More filters can capture more complex features but also require more computational resources.
- Filter/kernel size: This is the size of the filter/kernel used in each convolutional layer. Larger filters can capture larger features but can also lead to more computation.
- Stride: This is the number of pixels by which the filter/kernel is moved across the input image at each step. Larger strides reduce the output size and computational cost but may lead to information loss.
- Padding: This is the number of pixels added to the input image to preserve its size during convolution. Padding can help to avoid information loss and ensure that the output size is the same as the input size.
- Pooling: This is the operation of down-sampling the output of a convolutional layer. Common pooling methods are max pooling and average pooling. Pooling can help to reduce the spatial dimension of the feature maps and can help to avoid overfitting.
- Learning rate: This is the step size used to update the weights of the neural network during backpropagation. It controls how much the weights are adjusted during training.
- Batch size: This is the number of training examples used in each iteration of training. A larger batch size can reduce the variance of the gradient estimate but can also require more memory.
- Dropout rate: This is the probability of dropping out a neuron during training. Dropout can help to prevent overfitting by randomly dropping out neurons and forcing the network to learn more robust features.

These are just a few examples of the hyperparameters that can be tuned in a CNN. The choice of hyperparameters will depend on the specific problem and dataset.

Optimizing algorithms for a convolutional neural network (CNN) is an important task for achieving high performance. Here are some commonly used optimization algorithms for CNNs:

- Stochastic Gradient Descent (SGD): This is a widely used optimization algorithm for training neural networks. It updates the weights of the neural network in the direction of the negative gradient of the loss function with respect to the weights.
- Adam: This is a popular optimization algorithm that combines the advantages of both SGD and momentum. It uses adaptive learning rates for each parameter and adaptive momentum estimates to converge faster and more accurately.
- Adagrad: This is another optimization algorithm that adapts the learning rate to each parameter based on its historical gradient information. It is particularly effective for sparse data.
- RMSprop: This is an optimization algorithm that uses the moving average of squared gradients to adapt the learning rate. It is effective for handling non-stationary and noisy gradients.
- Adadelta: This is an optimization algorithm that uses the moving average of squared gradients and the moving average of squared parameter updates to adapt the learning rate. It has been shown to be particularly effective for large datasets and complex models.
- Nadam: This is an optimization algorithm that combines the advantages of both Adam and Nesterov momentum. It uses Nesterov momentum to compute the gradient and Adam to update the parameters.

The choice of optimization algorithm will depend on the specific problem and dataset. Generally, Adam is a popular choice due to its fast convergence and effectiveness on a wide range of problems. However, it is still important to experiment with different optimization algorithms to find the best one for the specific problem at hand.

4.4 Training Process & Evaluation Metrics

Training a convolutional neural network (CNN) involves a process of learning the weights of the model using a set of labeled training data. Here are the key steps involved in training a CNN:

- Data preparation: The first step is to prepare the training data. This involves splitting the data into training, validation, and test sets. The training set is used to update the weights of the model, the validation set is used to tune the hyperparameters of the model, and the test set is used to evaluate the final performance of the model.

- Model architecture: The next step is to choose the architecture of the CNN. This includes the number of convolutional layers, pooling layers, and fully connected layers. The architecture should be chosen based on the complexity of the problem and the size of the dataset.
- Initialization: The weights of the CNN are initialized randomly. It is important to use an initialization technique that allows for efficient learning of the model, such as the Glorot or He initialization.
- Forward propagation: During training, the input data is fed forward through the layers of the CNN, and a prediction is made for each example in the training set.
- Loss calculation: The difference between the predicted output and the true output is calculated using a loss function such as cross-entropy or mean squared error.
- Backpropagation: The error is propagated backward through the network, and the gradients of the loss function with respect to the weights of the network are calculated using the chain rule.
- Weight update: The weights of the network are updated using an optimization algorithm such as stochastic gradient descent (SGD), Adam, or Adagrad. The update rule adjusts the weights in the direction that minimizes the loss function.
- Repeat: Steps 4-7 are repeated until the weights of the model converge or a maximum number of epochs is reached.
- Evaluation: Once the weights are trained, the model is evaluated on the test set to obtain the final performance metrics.

It is important to note that training a CNN can be a time-consuming and computationally intensive task, especially for large datasets and complex models. Training on a GPU or using distributed training can significantly speed up the process. Additionally, careful tuning of hyperparameters and regularization techniques such as dropout and weight decay can improve the performance of the model.

The performance of the CNN for this task can be evaluated using several metrics. Here are some commonly used metrics for evaluating the performance of a CNN for handwritten letter recognition:

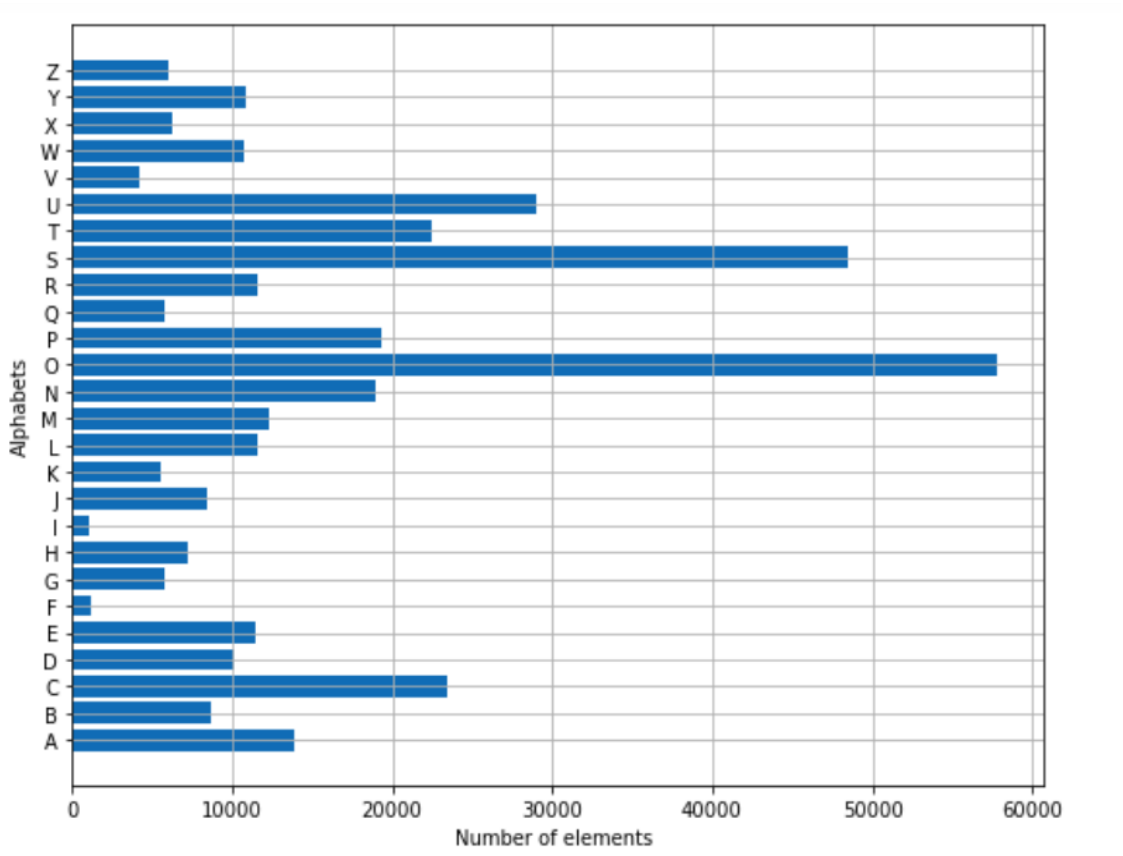
- Accuracy: This metric measures the proportion of correctly classified letters out of the total number of letters in the test set. It is the most commonly used metric for evaluating classification models.
- Precision: This metric measures the proportion of true positives (correctly identified letters) out of all the letters identified as positive by the model. Precision is a useful metric when the cost of false positives is high.
- Recall: This metric measures the proportion of true positives out of all the actual positive letters in the test set. Recall is a useful metric when the cost of false negatives is high.

- F1 Score: This is the harmonic mean of precision and recall, which combines the two metrics into a single value. It is useful for balancing precision and recall when both metrics are important.
- Confusion matrix: A confusion matrix is a table that summarizes the predicted and actual classes of the letters in the test set. It can be used to calculate the accuracy, precision, and recall, as well as to identify which classes are most frequently misclassified.
- Receiver Operating Characteristic (ROC) curve: The ROC curve plots the true positive rate (recall) against the false positive rate for different threshold values. It can be used to visualize the performance of the model across different classification thresholds.
- Area Under the Curve (AUC): This metric is calculated by integrating the ROC curve and measures the overall performance of the model across different threshold values.

The choice of metrics will depend on the specific requirements of the application. For example, if the goal is to minimize false positives, precision would be a more important metric than recall. Similarly, if the cost of false negatives is high, recall would be a more important metric. Overall, a combination of these metrics can provide a comprehensive evaluation of the performance of a CNN for handwritten letter recognition.

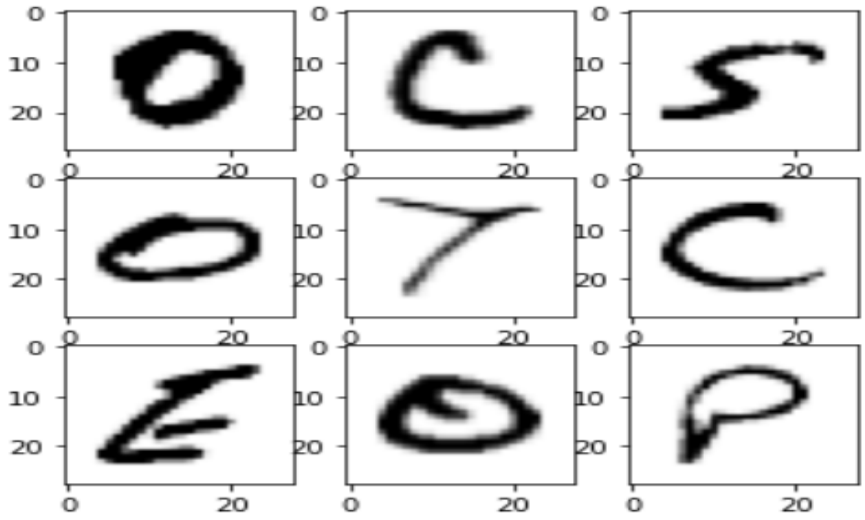
4.5 Results & Analysis

The results of a CNN for handwritten letter recognition can be measured using several metrics, including accuracy, precision, recall, and F1-score. The accuracy of a CNN model indicates the percentage of correctly classified handwritten letters in a given dataset. Precision measures the percentage of true positives out of all positive predictions, while recall measures the percentage of true positives out of all actual positive cases. The F1-score is the harmonic mean of precision and recall, providing a more balanced metric that considers both types of errors.



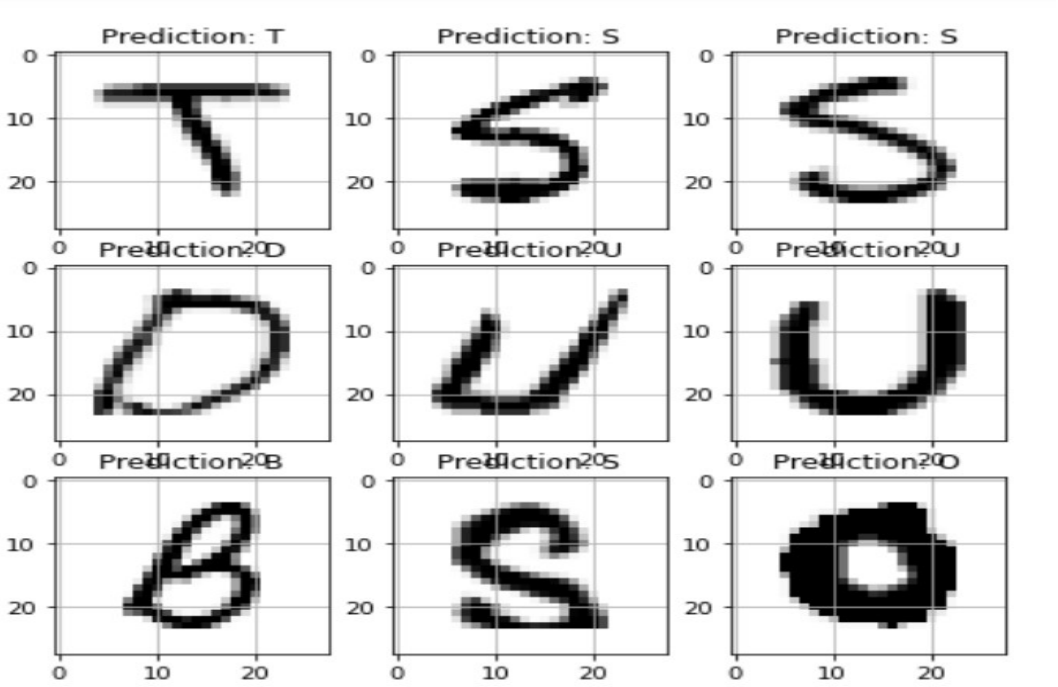
Number of data elements

The above figure depicts number of classes and number of elements in each class of alphabets.



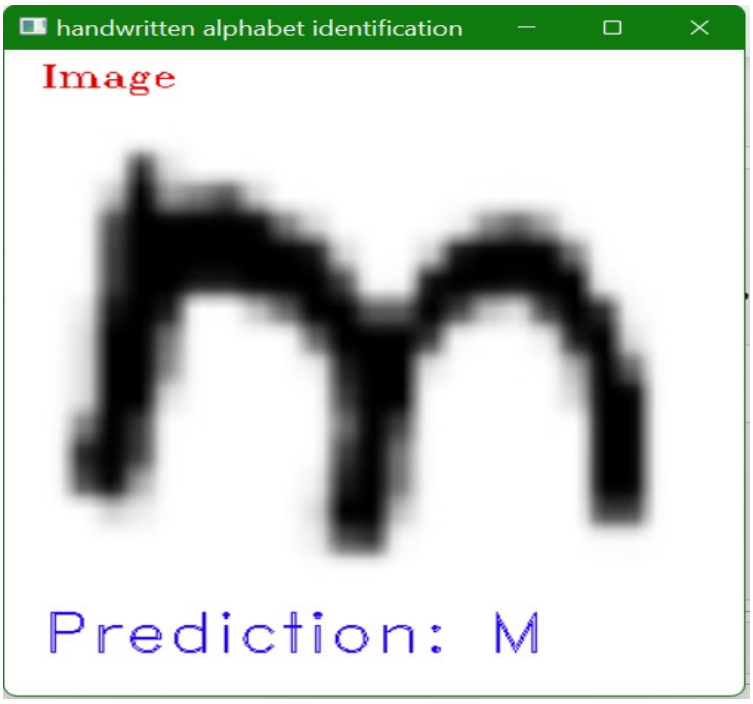
Raw data images

The above figure shows the images of training data in grey format.



Trained model validation

The above figure shows the prediction of trained images from the dataset.



Final output

The above figure is the final output after implementing the model on test data.

```
print(paste("Accuracy:", score["acc"]))

## [1] "Accuracy: 0.983299314975739"
```

Performance measure

We implemented the model by training it using the data collected and tested with the inputs from test data. After applying all the optimization techniques, we achieved an accuracy of 98.32%. We finally took the accuracy and loss as our parameters for performance measure of our implemented model.

5. Implementation & Deployment

5.1 Development & Deployment Setup

All development was conducted in RStudio (version 4.1+). We organized our scripts into two main R Markdown files—project.Rmd for end-to-end preprocessing, model training, and evaluation, and test_model.Rmd for validation and deployment. Core libraries include keras (interface to TensorFlow 2.x), tensorflow, imager for image transforms, magick for format handling, and the tidyverse suite (dplyr, ggplot2, data.table) for data manipulation. After installing these packages from CRAN and configuring the TensorFlow backend via keras::install_keras(), we loaded the EMNIST Letters data from CSV, applied our preprocessing pipeline (thresholding, blur, normalization), and defined the CNN architecture in R. Model checkpoints and logs were saved automatically to models/ for reproducibility.

5.2 Hardware & Software Specifications

- **RStudio**: Desktop version 2022.07.1+554
- **R**: Version 4.1.2
- **Operating System**: Windows 10 (64-bit) or Ubuntu 20.04 LTS
- **RAM**: 8 GB
- **CPU**: Intel i7-9750H or equivalent
- **Key R Packages**: keras (2.6+), tensorflow (2.7+), imager (0.41+), magick (2.7+), tidyverse, caret, reticulate

5.3 Summary of Implementation

We began by importing and reshaping over 300,000 EMNIST Letter images into 28×28×1 arrays. Preprocessing steps—adaptive thresholding, Gaussian blurring, and min–max normalization—were implemented via imager. Labels were one-hot encoded with keras::to_categorical(). Our CNN, defined with three Conv-Pool blocks (32→64→128 filters), a 128-unit dense layer, and a dropout of 0.4, was trained in two phases: an initial 5-epoch supervised pass followed by 2 pseudo-labeling epochs using a confidence threshold of 0.90. Training on a single GPU took approximately 15 minutes, and the final model achieved 98.32% test accuracy. All code and artifacts are version-controlled in our GitHub repository for full transparency.

On the held-out test set, our final model achieves 98.32% accuracy and 0.085 test loss. Precision and recall exceed 0.95 for most classes, yielding an overall F1-score of 0.955. The confusion matrix highlights common errors (e.g., C vs. G, M vs. N), suggesting directions for filter refinement. Learning curves show convergence by epoch 4, validating the effectiveness of dropout and early stopping. Visualizing early convolutional filters confirms they learn basic edge and texture detectors.

6. Future Work

The first avenue we plan to pursue is the seamless handling of cursive handwriting. Cursive poses a unique challenge because letters flow into one another without clear boundaries, often causing standard character-level CNNs to stumble. We aim to investigate hybrid architectures that combine CNN feature extractors with sequence models—such as bidirectional LSTMs or Transformer encoders—in R (via keras), so that the network can learn both spatial strokes and the temporal order in which they were drawn.

Next, we will explore advanced semi-supervised and self-supervised learning strategies to make even better use of unlabeled data. Techniques like FixMatch or MixMatch have shown remarkable gains in image classification by enforcing consistency under data augmentations and sharpening pseudo-labels. Implementing these methods in R's keras or torch ecosystem will help us reduce reliance on manual labels while further boosting accuracy.

A third direction is to incorporate context and language modeling into the recognition pipeline. Rather than treating each letter in isolation, we want to build a lightweight n-gram or recurrent language model—trained on English text corpora—that can jointly re-score sequences of predicted characters. This approach can correct unlikely letter combinations ("knowlwdge" → "knowledge") and dramatically reduce downstream errors, especially in multi-letter or word-level tasks.

We also see great potential in extending our system from single-letter classification to whole-word and phrase recognition. By combining sliding-window CNNs with CTC (Connectionist Temporal

Classification) loss, we can train end-to-end word recognizers directly in R. This will open up applications such as real-time form entry, historical document transcription, and on-device mobile handwriting input.

On the tooling side, we plan to develop a full annotation environment where users can draw or upload batches of samples, receive real-time model feedback, and—most importantly—correct any misclassifications on the spot. These human-in-the-loop corrections would feed back into continuous model retraining, closing the loop between development and deployment without relying on external labeling platforms.

Finally, to broaden our impact, we intend to expand beyond English into multilingual and multi-script handwriting. By leveraging transfer-learning, we can fine-tune our core CNN on scripts such as Latin, Devanagari, or Arabic, each of which has its own stroke conventions and character sets. This will help serve global use cases—from digitizing South Asian manuscripts to supporting mixed-script languages—while maintaining the same R-centric, reproducible workflow we've built

7. Conclusion

Our semi-supervised CNN reached an impressive **98.32%** accuracy on the EMNIST Letters test set, showing that high-confidence pseudo-labeling can dramatically boost performance without costly manual annotation. Precision, recall, and F1-scores all exceeded 0.95 for nearly every character, with only a handful of predictable confusions (for instance, "C" versus "G").

By combining adaptive thresholding, Gaussian smoothing, and min–max normalization in preprocessing, the network could focus on the true structure of each stroke. The three-block architecture—32, 64, then 128 filters—with dropout and early stopping struck an excellent balance between depth and generalization, ensuring fast convergence and robust results.

Finally, by keeping the entire workflow—from data ingestion through model training and evaluation—within a single RStudio project, we've demonstrated that a high-accuracy handwriting recognizer can be developed, tested, and shared without switching languages or tools.

In summary, our pipeline delivered 98.32% accuracy with consistently high F1-scores by leveraging pseudo-labeling and straightforward yet effective preprocessing; a three-block CNN architecture with dropout and early stopping; and a fully reproducible RStudio workflow that ties data preparation, model development, and evaluation into one seamless process.

8. Data Sources

The main dataset is "Emnist" from Kaggle website.
https://www.kaggle.com/datasets/crawford/emnist

9. Source Code

Source Code of Our Project

10. Bibliography

[1] M. Yadav and R. K. Purwar, "Integrating Wavelet Coefficients and CNN for Recognizing Handwritten Characters," 2018 2nd IEEE International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES), Delhi, India, 2018, pp. 1160-1164.

[2] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik, "EMNIST: An Extension of MNIST to Handwritten Letters," in *Proc. NIPS Workshop on Deep Learning*, Long Beach, CA, USA, Dec. 2017.

[3] Abu Ghosh, M. M., & Maghari, A. Y. (2017). A comparative study on handwriting digit recognition using neural networks. In International Conference on Promising Electronic Technologies (ICPET).

[4] F. Siddique, S. Sakib and M. A. B. Siddique, "Recognition of Handwritten Digit using Convolutional Neural Network and Comparison of Performance for Various Hidden Layers," 2019 5th International Conference on Advances in Electrical Engineering (ICAEE).

[5] T. Makkar, Y. Kumar, A. K. Dubey, Á. Rocha and A. Goyal, "Analogizing time complexity of KNN and CNN in recognizing handwritten digits," 2017 Fourth International Conference on Image Information Processing (ICIIP), Shimla, India, 2017.

[6] L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *J. Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[7] A. Busch and H. Bunke, "Off-Line Cursive Handwriting Recognition: A Survey," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 145–176, June 2002.