Hanoi University of Science and Technology
School of Informations and Comunication of Technology

Project I
Handwritten characters and digits classifiers

Student Trần Hữu Đạo
Student ID 20220061

Instructor: **Dr. Trần Vĩnh Đức**

December, 2024

Contents

Abstract

Recognizing handwritten characters and digits is a useful application of Artificial Intelligence technology. For the sake of understanding AI algorithms, I used K-Nearest Neighbor and several Artificial Neural Network models, to complete this task and evaluate their effectiveness in doing so. The findings are anticipated to provide insights that could improve the accuracy and efficiency of character recognition systems. This report not only details the experimental procedures and outcomes but also discusses the implications of these findings in real-world applications, thereby setting the stage for future advancements in this area.

1. Introduction

Character recognition has emerged as a crucial technology in the domain of computer vision and pattern recognition. This report delves into the intricacies of my project focused on character recognition using the EMNIST dataset. It aims to provide an exhaustive exploration of various machine learning algorithms and their efficacy in recognizing handwritten characters.

1.1 Motivation

The ability to recognize characters accurately is a fundamental requirement for numerous applications in today's digital world. Character recognition technology underpins a wide array of systems and services, such as:

- Optical Character Recognition (OCR): Enhancing the digitization of printed materials, facilitating easier storage, retrieval, and editing of textual content.
- Handwriting Recognition: Assisting in the digital conversion of handwritten documents, improving accessibility, and preserving historical manuscripts.
- Automated Data Entry: Reducing human error and increasing efficiency in data processing tasks, especially in banking, postal services, and administrative documentation.
- Assistive Technologies: Empowering individuals with disabilities by providing tools that convert handwritten or printed text into speech or braille.
- Security Systems: Enabling automated identity verification through the recognition of handwritten signatures or alphanumeric codes.

Recognizing the paramount importance of character recognition, this project aims to explore and enhance the effectiveness of various machine learning models in accurately identifying characters from the EMNIST dataset.

1.2 My Scope

This project aims to evaluate and compare the performance of several machine learning algorithms on the EMNIST dataset, which consists of handwritten characters. The algorithms under consideration include:

- K-Nearest Neighbors (K-NN): A non-parametric method that classifies characters based on the majority label among the nearest neighbors in the feature space.
- Feedforward Neural Networks (FFNNs): A type of artificial neural network where connections between the nodes do not form cycles. Information moves in one direction—from the input layer, through hidden layers, and to the output layer.
- Convolutional Neural Networks (CNNs): Deep learning models specifically designed to process grid-like data, such as images, by automatically learning spatial hierarchies of features.

My approach involves training each of these models on the EMNIST dataset and evaluating their performance based on metrics such as accuracy and confusion matrix. By comparing these models, I aim to identify the strengths and weaknesses of each algorithm in the context of character recognition and provide insights into their practical applications.

In summary, this report will detail the methodologies and findings of my project, offering a comprehensive analysis of the performance of various machine learning models in the task of character recognition. The ultimate goal is to advance the understanding of these models and their applicability in real-world scenarios where accurate character recognition is essential.

1.3 The Extended MNIST database

The EMNIST dataset is a set of handwritten character digits derived from the NIST Special Database 19 and converted to a 28x28 pixel image format and dataset structure that directly matches the MNIST dataset. There are six different splits provided in this data set, including:

- EMNIST ByClass: 814,255 characters. 62 unbalanced classes.
- EMNIST ByMerge: 814,255 characters. 47 unbalanced classes.
- EMNIST Balanced: 112,800 characters. 47 balanced classes.
- EMNIST Letters: 145,600 characters. 26 balanced classes.
- EMNIST Digits: 280,000 characters. 10 balanced classes.
- EMNIST MNIST: 70,000 characters. 10 balanced classes.

Each of them is provided in two formats including Binary and CSV. In csv file, each row is a separate image with 785 columns. The first column is the class label (see mappings.txt for class label definitions). Each column after represents one pixel value (784 total for a 28x28 image).

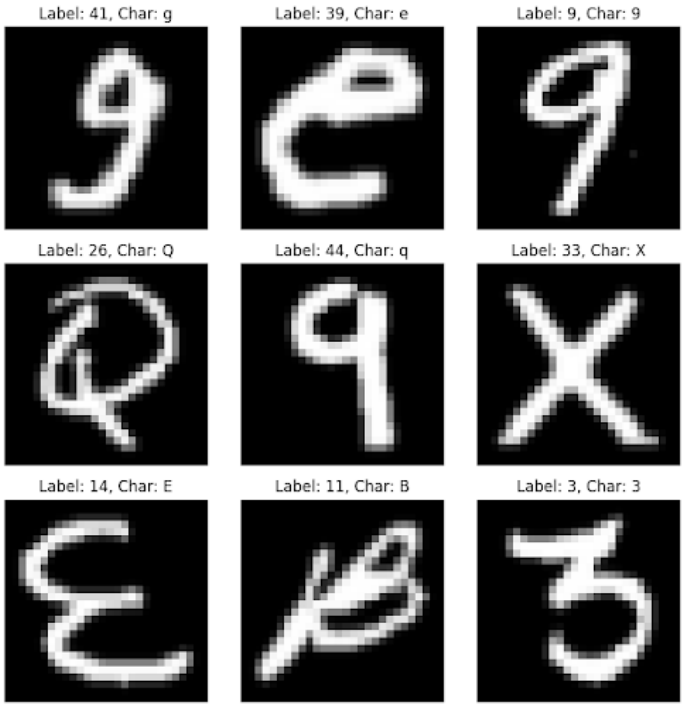
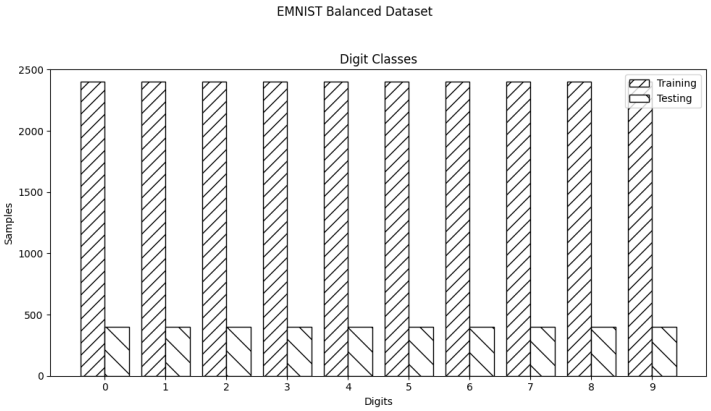
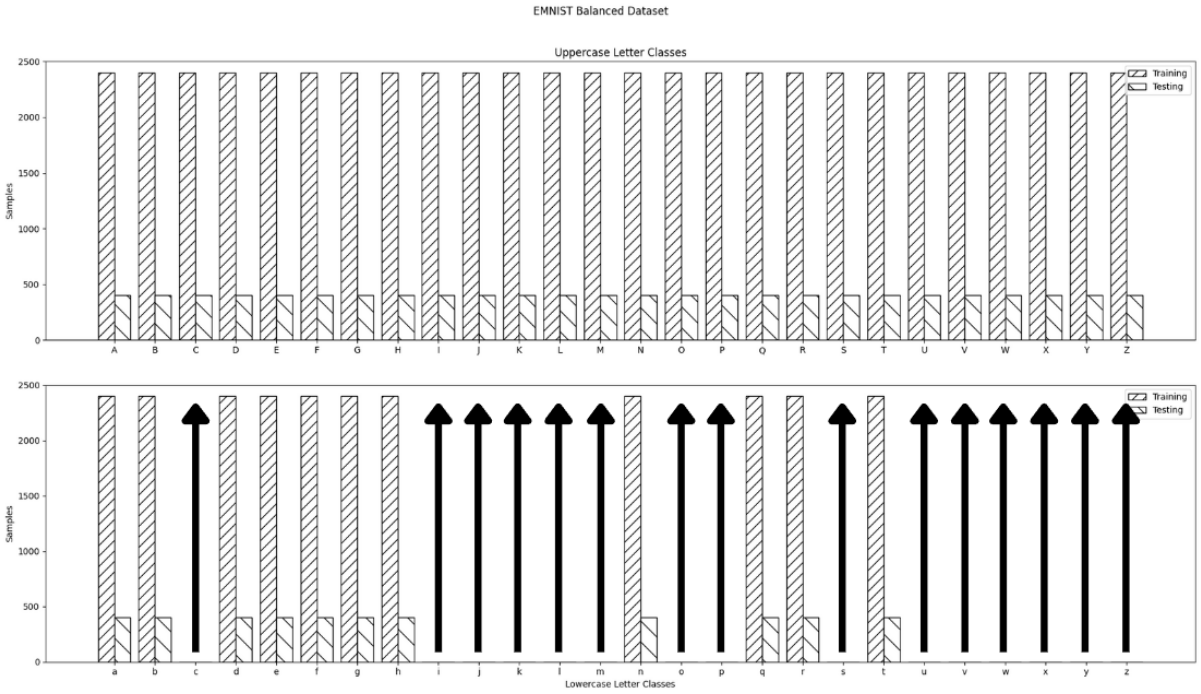


Figure 1: Examples of EMNIST pictures

In this project, I choose EMNIST Balanced dataset for several reasons. The EMNIST Balanced dataset proves to be a good choice since it has a balanced class distribution, ensuring equal representation of all classes, avoiding bias, and improving generalization while being significantly smaller in size and easier to handle computationally compared to the larger ByClass and ByMerge datasets.



(a) digits



(b) letters

Figure 2: EMNIST balanced dataset

2. Methodology

In this project, I will apply some typical machine learning methods to solve the "Handwritten characters and digits classifiers", including Supported Vector Machin (SVM), K-Nearest Neighbor (KNN), Random Forest (RF), Feed-forward Neural Network(FNN) and Convolutional Neural Network (CNN).

2.1 K-Nearest Neighbor

2.1.1 Main ideas

The k-Nearest Neighbors (KNN) algorithm is a non-parametric, supervised learning classifier that is used for classification and regression. In classification, the output is a class membership. KNN operates on the principle that similar instances are near each other, using proximity to make classifications or predictions about the grouping of an individual data point. When a new input is given, KNN finds the k-nearest instances from the training data and assigns the class most common among them. It is one of the simplest and most popular classifiers used in machine learning today.

The aim for training this model is to calculate the distance between each pixel of the input image and the corresponding pixels of every character in the training dataset, and then return a prediction based on the K-nearest neighbors found.

Calculating Distance and Normalising

To measure the similarity between characters, KNN relies on a distance metric. Each character image is normalized by scaling the pixel values to a range of 0 to 1, ensuring that each feature contributes equally to the distance calculation. This normalization is crucial to prevent features with larger ranges from disproportionately influencing the distance metric.

Finding Nearest Neighbors

For each test instance, the distance to all training instances is calculated using different distance formulas, facilitated by the Scikit-learn library. The k instances with the smallest distances are selected as the nearest neighbors. This process is implemented in the *evaluate_knn* function, which computes the distances and identifies the closest training samples, storing them in a list for further processing.

Voting and Final Prediction

In the classification task, each of the k-nearest neighbors votes for their respective class. The class with the highest number of votes is assigned to the test instance.

2.2 Random Forest

2.2.1 Decision Tree

Decision Tree Classifier is a type of Binary Tree that generate classifications base on given data. Each Decision Tree is constructed with Decision Node which contains two parameters: feature and threshold. This Decision Node divides input list into two output lists base on whether the value of feature higher than the threshold or not. As a result, Decision Trees

divides a multiples dimension data set into hyperregtangles. A leaf node is a node that does not have children node. Instead, it labes the input list to the highest presented class.

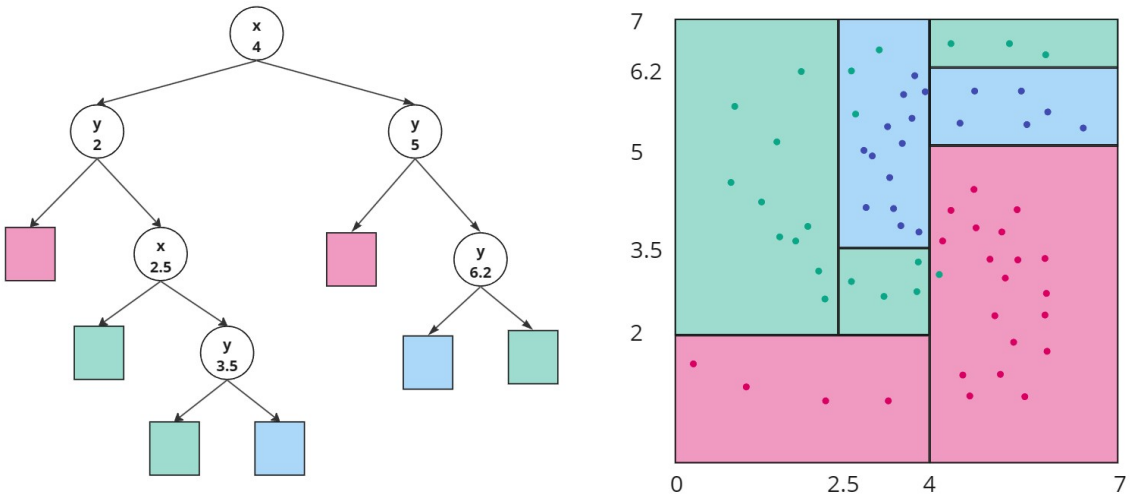


Figure 3: Example of a Decision Tree Classifier

Building a Decision Tree classifiers means finding the best pair of feature and threshold to split the dataset assuming that they are best at classifies the test set. There are several criteria to evaluate the classification ability of a split that are entropy and gini impurity.

Gini impurity is calculated using the following equations:

$$Gini(D) = 1 - \sum_{i=1}^{n} p_i^2 \tag{1}$$

, where D is the list of instances, n is the number of classes.

$$Impurity(f,t) = \frac{Gini(D\_left) \cdot |D\_left| + Gini(D\_right) \cdot |D\_right|}{|D\_left| + |D\_right|} \tag{2}$$

, where f, t are feature and threshold of the split; D_left and D_right are two output list of a node; $|D\_left|$ and $|D\_right|$ are number of instances in each list.

Entropy and information gain is calculated using the following equation:

$$Entropy(D) = -\sum_{i=1}^{n} p_i \cdot log_2(p_i) \tag{3}$$

, where D is the list of instances, n is the number of classes.

$$Gain(f,t) = Entropy(D) - \frac{Entropy(D\_left) \cdot |D\_left| + Entropy(D\_right) \cdot |D\_right|}{|D\_left| + |D\_right|} \tag{4}$$

, where f, t are feature and threshold of the split; D_left and D_right are two output list of a node; $|D\_left|$ and $|D\_right|$ are number of instances in each list. In general, gini imprity is more simple and runs faster while entropy is more sensitive.

2.2.2 Ensemble Learning

One problem of Decision Tree is that is get overfit easily when max_depth increases. By using multiple Decision Trees, each of them are built using different dataset using bootstrap sampling to randomise a dataset and conclude the output class as the most chosen output of trees, the problem of overfit reduces without having to reduce the max_depth.

$$forest\_prediction = mode(tree\_predictions\_list) \tag{5}$$

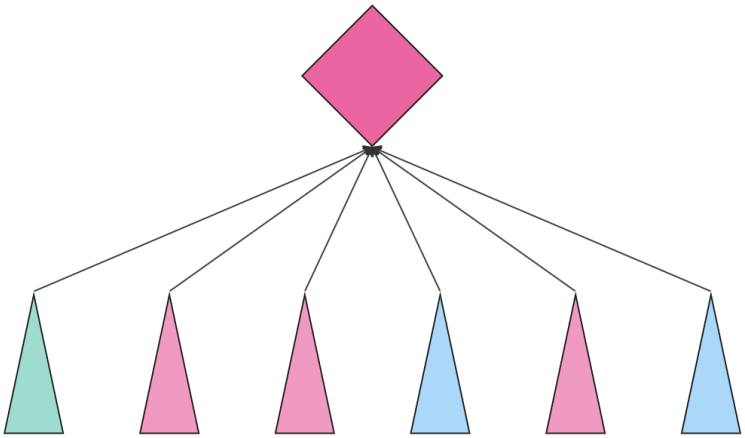


Figure 4: Example of a Random Forest Classifier

2.3 Feed-forward Neural Network

Feed-forward Neural Network (FNN) is an ANN in which neurons are divided into fully connected layers, the neurons of the previous layer only trigger neurons of the following layer.

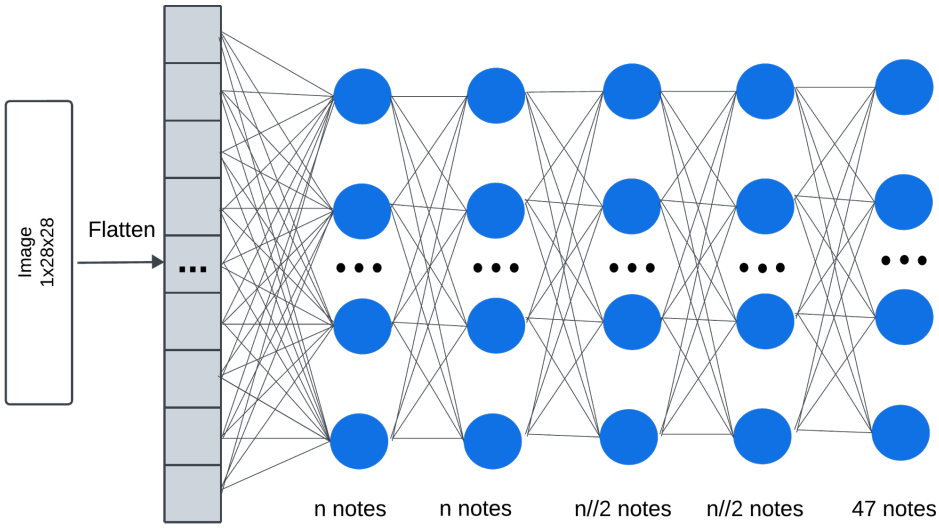


Figure 5: Architecture of Feed-forward Neural Network

The model starts with a Flatten layer, reshaping the input data from a 2D image format to

a 1D vector. Layer 1 consists of a fully connected layer with 784 input features and n output features, followed by a ReLU activation function and a dropout layer with a 0.5 probability to prevent overfitting. Layer 2 has another fully connected layer with n input and output features, also followed by a ReLU activation function and a dropout layer with a 0.5 probability. Layer 3 includes a fully connected layer with n input features and n//2 output features, a ReLU activation function, and a dropout layer with a 0.3 probability. Layer 4 has a fully connected layer with n//2 input and output features, a ReLU activation function, and a dropout layer with a 0.3 probability. The output layer is a fully connected layer with n//2 input features and 47 output features, corresponding to the number of classes.

The experiment was conducted with the following configurations:

- The number of hidden units n was set to 128, 256, and 512.
- The loss function used was Cross Entropy Loss.
- The optimizer employed was Stochastic Gradient Descent (SGD) with learning rates of 0.01, 0.1, and 0.5.
- The models were trained for 5, 10, and 20 epochs.

2.4 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are a powerful class of deep learning algorithms primarily used for analyzing visual data. Designed to automatically and adaptively learn spatial hierarchies of features, CNNs consist of layers like convolutional layers, pooling layers, and fully connected layers. The convolutional layers apply filters to the input image to create feature maps, capturing essential patterns and spatial relationships. Pooling layers then reduce the dimensionality, preserving important features while minimizing computational complexity. This hierarchical approach allows CNNs to excel in tasks such as image recognition, object detection, and facial recognition, making them indispensable in computer vision applications.

Convolutional Neural Networks (CNNs) have transformed the field of image processing and computer vision due to their architectural efficiencies which make them particularly suited for high-dimensional input data, such as images and videos. Two fundamental concepts that contribute to their effectiveness are parameter sharing and sparsity of connections.

Parameter sharing refers to the use of the same weights by multiple units in a layer. This is a defining characteristic of CNNs, primarily in their convolutional layers. Instead of learning separate sets of weights for every location in the input data, CNNs learn a single set of weights (i.e., a filter or kernel) that is convolved across the input. This drastically reduces the number of parameters in the model, leading to fewer computations and less overfitting. This property makes CNNs particularly powerful for tasks that require the detection of features that are useful across the entire input space, such as edges in images.

In CNNs, each neuron in a convolutional layer is connected only to a local region of the input. This locality principle ensures that the learned filters produce the strongest response

to spatially local input patterns. The sparsity of connections leads to more efficient learning and less computational complexity since neurons do not need to be connected to every part of the input. This allows for deeper, more complex networks by reducing the computational cost associated with each layer.

2.4.1 Core Concepts of CNNs

Convolutional Neural Networks made up by 3 types of hidden layers: Convolutional Layers, Pooling Layers and Fully Connected Layer (Flattening). To understand a Convolutional Layer, we need to be familiar with three concepts Filter/Kernel, Padding and Stride.

Filter / Kernel

Filters (also called kernels) are small, learnable matrices used to detect features in an input image. These filters slide over the image, performing element-wise multiplications and summations to create feature maps. Each filter is trained to identify specific patterns, such as edges or textures. Multiple filters are used in each convolutional layer, allowing the network to learn various features. This convolution operation is key to the CNN's ability to understand and classify complex image data effectively. Given a filter (or kernel) of size $f \times f$, which is

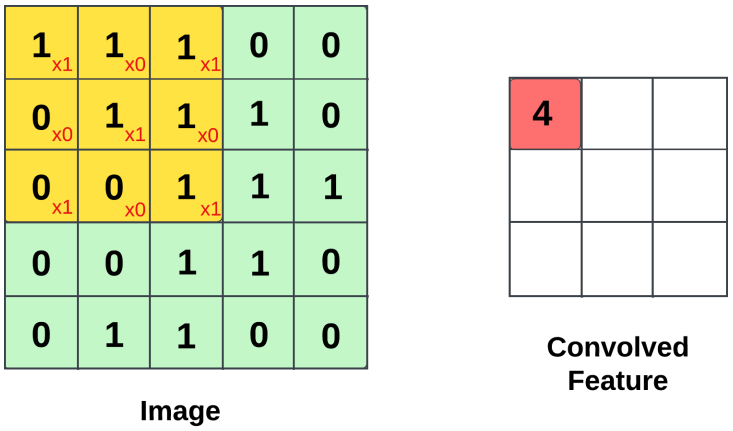


Figure 6: How filters work

the matrix of weights applied to the input image, an original $n \times n$ image will produce an output of dimensions $(n-f+1) \times (n-f+1)$ after the filter is applied.

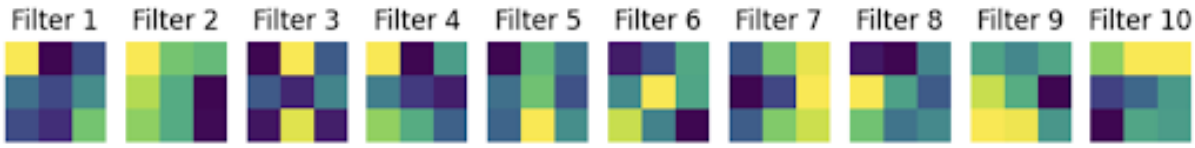


Figure 7: Filter visualization

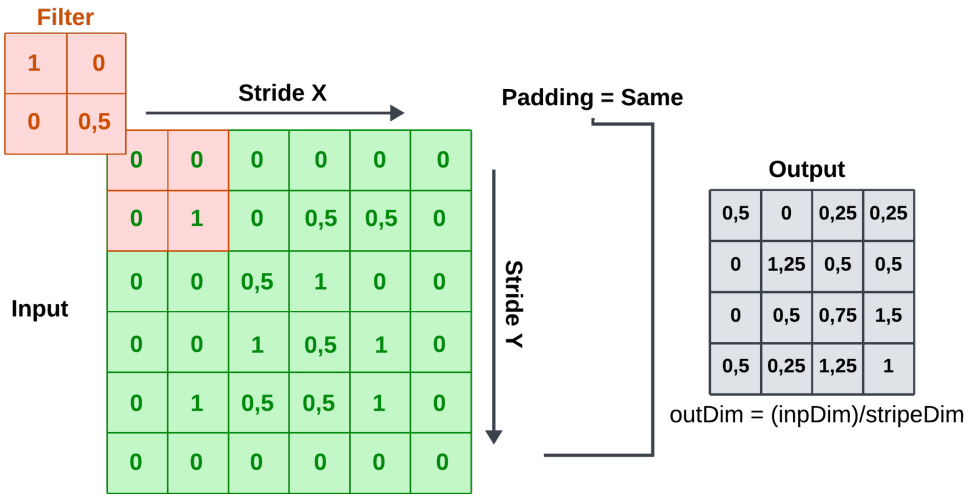


Figure 8: Padding

Padding

Padding involves adding layers of zeros around the input image. This technique allows the convolutional operation to be applied to the bordering elements of the input matrix, ensuring that the spatial dimensions are either preserved or controlled as desired through the network layers. Padding is crucial for controlling the size of the output volumes and to allow deep networks to maintain adequate feature resolution at higher layers. Given that p represents the number of layers of zeros added to the border of an image, an original $n \times n$ image will have dimensions $(n+2p) \times (n+2p)$ after padding. When a filter of size $f \times f$ is applied, the output dimensions can be calculated as $(n+2p-f+1) \times (n+2p-f+1)$.

Stride

Stride defines the step size with which the filter convolves around the input. A stride of one moves the filters one pixel at a time, while a larger stride results in wider spacing between applications of the filter. This affects the size of the output volume and the amount of overlap between regions processed by the filter applications. A larger stride results in smaller output dimensions, which can be beneficial for reducing the computational load and model overfitting. Given a stride s, which is the number of pixels the filter moves over the image, an original $n \times n$ image with padding p will have dimensions $(n+2p) \times (n+2p)$ after padding. When a filter of size $f \times f$ is applied with a stride s, the output dimensions are calculated as:

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \tag{6}$$

Pooling Layers

Purpose of pooling layers is to gradually shrink the representation's spatial size to reduce the number of parameters and computations in the network. The pooling layer treats each feature map separately. There are two methods for pooling that are max pooling and average pooling.

In the one hand, max pooling chooses the most significant element from the feature map. The feature map's significant features are stored in the resulting max-pooled layer. It is the most popular method since it produces the best outcomes. In the other hand, average pooling entails calculating the average for each region of the feature map. Pooling gradually reduces the spatial dimension of the representation to reduce the number of parameters and computations in the network, as well as to prevent over-fitting. If there is no pooling, the output has the same resolution as the input.

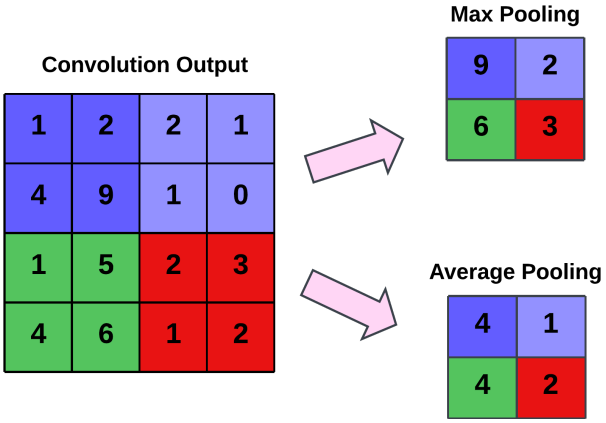


Figure 9: Pooling

Fully Connected Layers (Flattening)

After several convolutional and pooling layers, CNNs often include one or more fully connected layers. These layers connect every neuron in one layer to every neuron in the next layer. The input to these fully connected layers must be a flat vector, so the output from the previous layers is typically flattened into a single long feature vector. The fully connected layers integrate learned features from the convolutions to form the final outputs, such as the probabilities of different classes in a classification task.

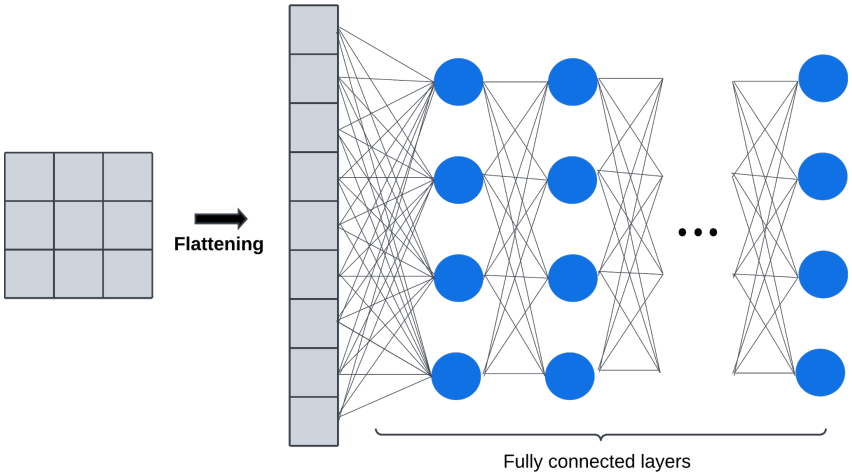


Figure 10: Fully-connected layers

2.4.2 What do filters of Convolution Neural Network learn?

I will introduce two methods for understanding what deep Convolutional Neural Networks (CNNs) are learning—Activation Maximization and Filter Visualization via Feature Inversion—offer distinct insights into the model's behavior. Activation Maximization involves selecting individual neurons within a CNN and identifying the input patterns that maximize their activation. This method is performed layer by layer, starting from the first convolutional layer and progressing deeper. By searching the dataset or using optimization techniques, one can find the image patches that lead to the highest activation values for each unit. This helps visualize the features each neuron is responsive to, with early layers detecting simple features like edges and textures, and deeper layers recognizing more complex patterns like object parts or entire objects. On the other hand, Filter Visualization via Feature Inversion employs techniques like feature inversion to reconstruct input images that produce specific activations in the network. This process uses optimization to reverse-engineer inputs that highlight what each filter in the CNN is looking for, often using bilinear interpolation to smooth out pixel values during reconstruction. This method provides general visualizations of the features that activate certain layers, illustrating the complexity and abstraction level of features learned at different depths. In comparison, Activation Maximization focuses on individual neurons and their maximal stimuli, offering a granular view at the neuron level. It directly shows what each unit prefers, while Filter Visualization offers a broader, more interpretative view of the network's feature detection capabilities by reconstructing inputs based on desired filter responses. Thus, Activation Maximization provides direct empirical insights, whereas Filter Visualization yields a composite view of the features that stimulate the network layers.

2.4.3 Types of CNNs

I utilize five distinct Convolutional Neural Network (CNN) architectures. The models implemented from scratch include Simple CNN, LeNet[3], and VGG-16[2]. Consistent with the Fully Connected Network (FNN) models, I employ Stochastic Gradient Descent (SGD) as the optimizer and Cross Entropy as the loss function. Each model is trained across multiple configurations, specifically for 5, 10, and 20 epochs, with learning rates set to 0.01, 0.1, and 0.5. Among these configurations, the VGG-16 model, trained for 20 epochs with a learning rate of 0.5, demonstrated superior performance, achieving a final test accuracy of 90%.

Simple CNN

This is the Architecture of Simple CNN used in this project:

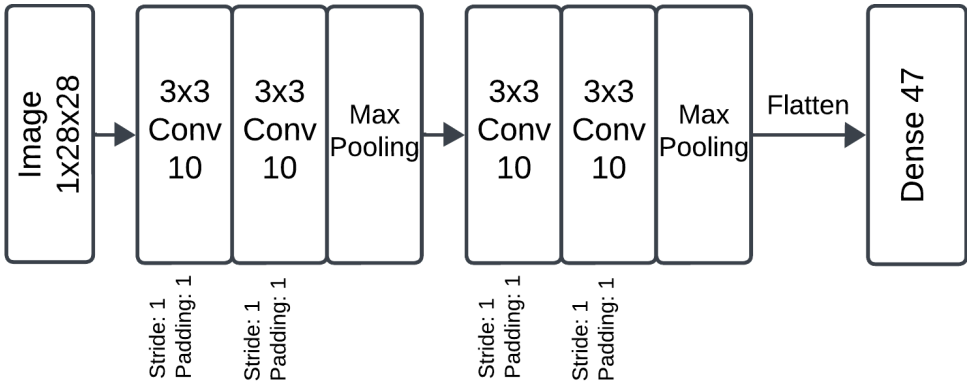


Figure 11: Architecture of Convolutional Neural Network

Number of parameters: 25,907

Convolution Block 1 consists of two convolutional layers, each with 10 filters, a 3x3 kernel size, stride of 1, and padding of 1, followed by ReLU activation functions. This block includes a max pooling layer with a 2x2 kernel size and stride of 2 to downsample the feature maps. Convolution Block 2 has two more convolutional layers with the same configuration and ReLU activations, followed by another max pooling layer with a 2x2 kernel size and stride of 2. The Classifier section flattens the feature maps into a 1D vector and includes a fully connected layer with 490 input features and 47 output features for classification.

VGG_16

This is the Architecture of VGG_16 model:

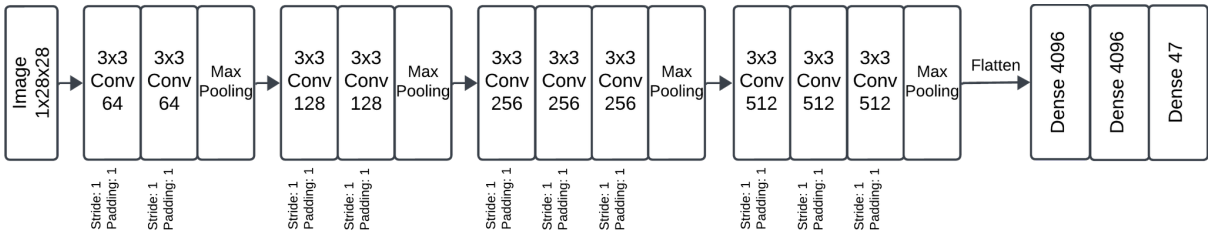


Figure 12: Architecture of VGG_16

Number of parameters: 26,714,607

The VGG-16 architecture consists of five convolutional blocks. Block 1 has two convolutional layers with 64 filters each, followed by batch normalization, ReLU activation, and max pooling. Block 2 includes two convolutional layers with 128 filters, again followed by batch normalization, ReLU activation, and max pooling. Block 3 contains three convolutional layers with 256 filters, followed by the same activations and pooling. Blocks 4 and 5 each have three convolutional layers with 512 filters. Throughout these blocks, the kernel size is 3x3, stride is 1, and padding is 1, maintaining a consistent feature extraction process and every max pooling layer with a 2x2 kernel size, stride of 2, reducing the spatial dimensions by half. The fully connected layers consist of two layers with 4096 units each, ReLU activation, and 0.5 dropout. The final output layer has 4096 input features and 47 output features for classification.

LeNet

This is the Architecture of LeNet model:

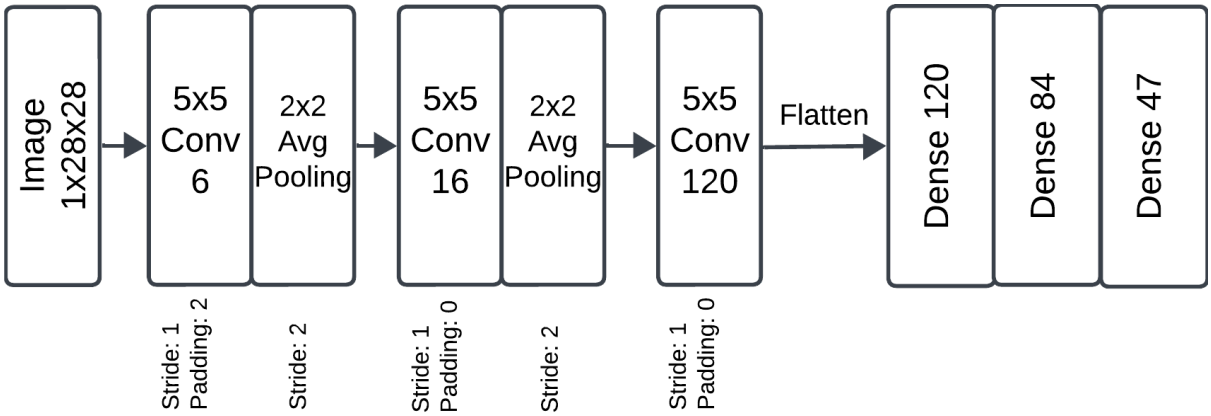


Figure 13: Architecture of LeNet

Number of parameters: 64,851 parameters

The LeNet architecture includes three convolutional blocks. The first block has a convolutional layer with 6 filters, a 5x5 kernel, stride of 1, and padding of 2, followed by ReLU activation and a 2x2 average pooling layer that halves the spatial dimensions. The second block features a convolutional layer with 16 filters, a 5x5 kernel, stride of 1, followed by ReLU activation and another 2x2 average pooling layer. The third block contains a convolutional layer with 120 filters, a 5x5 kernel, stride of 1, and ReLU activation. The classifier includes a fully connected layer with 120 input features and 84 output features, followed by ReLU activation, and an output layer with 84 input features and 47 output features for classification.

2.5 Principal Component Analysis

As stated before, the EMNIST Balanced dataset, while having a moderate size, still requires a lot of computational cost, having 785 values per character and 112,800 training as well as 18,800 testing characters. But for a few models, especially the Random Forest model, which requires training multiple Decision Trees, training and testing is incredibly time consuming. To combat that, I introduce Principal Component Analysis (PCA) to reduce the computational requirements and further improve my models in terms of time efficiency.

Principal Component Analysis (PCA) is a powerful technique in machine learning and statistics, used for dimensionality reduction, which is crucial in handling large datasets by transforming a large set of variables into a smaller one that still contains most of the information in the large set. Reducing the number of variables of a data set naturally comes at the expense of accuracy, but the reduction in dimensionality proves to be a beneficial trade-off. Because smaller data sets are easier to explore and visualize, and thus make analyzing data points much easier and faster for machine learning algorithms without extraneous variables to process.

2.5.1 Principle components

Principal components are new variables that are constructed as linear combinations of the initial variables. These combinations are done in such a way that the new variables (i.e., principal components) are uncorrelated and most of the information within the initial variables is squeezed or compressed into the first components.

Constructing principle components follows 4 main steps:

Step 1: Centering

The aim of this step is to center the value of the continuous initial variables so that each one of them distribute around the initial point. This is to ensure the first principle components accurately reflect the direction of greatest variance. Without this step, the component may instead align more with the data's average.

$$z = x - \mu \tag{7}$$

Step 2: Covariance Matrix Computation

The aim of this step is to identify relationships between initial variables by calculating Covariance Matrix.

$$Cov(Z) = \frac{1}{n-1}\left(Z^T \cdot Z\right) = \begin{bmatrix} Var(z_1) & Cov(z_1,z_2) & \cdots & Cov(z_1,z_n) \\ Cov(z_2,z_1) & Var(z_2) & \cdots & Cov(z_2,z_n) \\ \vdots & \vdots & \ddots & \vdots \\ Cov(z_n,z_1) & Cov(z_n,z_2) & \cdots & Var(z_n) \end{bmatrix} \tag{8}$$

Step 3: Calculate and Ranking Eigenvectors

The aim of this step is to calculate eigenvalues and eigenvectors of the Covariance Matrix above. The eigenvectors are candidate feature vectors, which are then ranked by their eigenvalues in descending order. The highest ranked vectors are kept and combine into Feature Matrix, while lower ranked vectors are discarded. Thus, I have

- $v_1, v_2, ..., v_n$ are eigenvectors
- $\lambda_1 >= \lambda_2 >= ... >= \lambda_n$

$$F = [v_1, v_2, ..., v_p] \tag{9}$$

Step 4: Recast the Data Along the Principal Components Axes

In this step, the aim is to use the feature vectors formed above to reorient the data from the original axes to the ones represented by the principal components (hence the name Principal Components Analysis). This can be done by multiplying the transpose of the original data set

by the transpose of the feature vector.

$$X_{PCA} = Z \cdot F \tag{10}$$

In this project I intuitively choose 50 as the number of principal components. While some information is lost in the process, the new PCA-reduced dataset retains approximately 85.50% of the information from the original images. The advantage of reduced training time proves to be a very beneficial trade-off. The reconstructed image is visualized below.

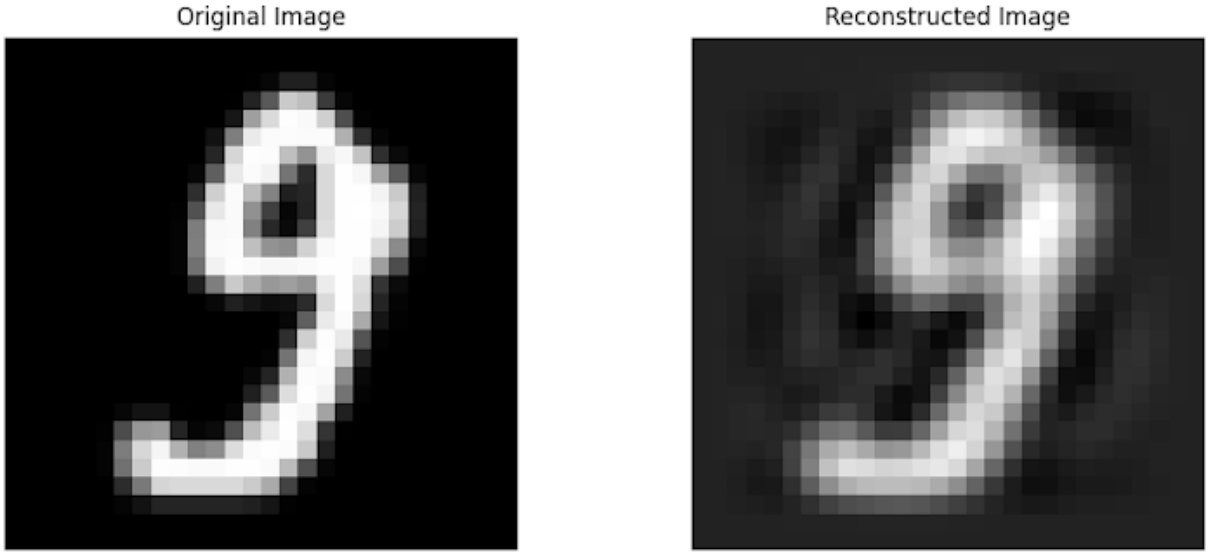


Figure 14: Example of reconstructed image from PCA-reduced data

2.5.2 Applying PCA for EMNIST

For the EMNIST character recognition problem, applying PCA offers several advantages:

Reduction of Computational Complexity:

By transforming the original 784-dimensional feature space into a lower-dimensional space (e.g., 50 dimensions), PCA significantly reduces the number of computations required for each split in the DecisionTree algorithm. This reduction in dimensionality leads to faster training times, making it feasible to train the model on large datasets.

Noise Reduction:

High-dimensional data often contains redundant and noisy features. PCA helps to filter out these irrelevant components by focusing on the principal components that capture the most variance. This noise reduction can improve the model's performance by preventing overfitting and enhancing generalization.

Improved Convergence:

With fewer dimensions, the optimization process for finding the best splits in the decision tree becomes more efficient. This efficiency leads to quicker convergence during training, making the overall process more time-effective.

Memory Efficiency:

Lower-dimensional data requires less memory for storage and processing. This reduction is particularly beneficial when dealing with large datasets, as it enables the use of more sophisticated models and algorithms within the available computational resources.

2.6 Performance Evaluation: Accuracy, Confusion Matrix

For EMNIST balanced data set, which is multi-classes and have instances equally distributed among classes, I consider 2 performance evaluation methods that are accuracy and confusion matrix

2.6.1 Accuracy

Accuracy is a metric used to evaluate the overall correctness of a classification model. It is defined as the ratio of correctly predicted instances to the total instances in the dataset. Mathematically, it is expressed as:

$$Accuracy = \frac{Numberof CorrectPredictions}{TotalNumberof Predictions} \tag{11}$$

2.6.2 Confusion Matrix

A confusion matrix is a table used to visualize the performance of a classification model by comparing the true and predicted labes. It provides detailed insights into the types of errors made by the model and helps in understanding the performance beyond simple accuracy. The confusion matrix is structured as follows:

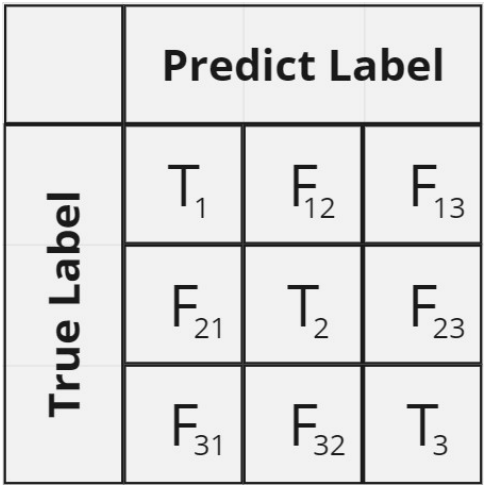


Figure 15: Confusion Matrix

3. Experimentation and Results

3.1 K-Nearest Neighbor

3.1.1 Hyper-parameter tuning

In the tuning phase to find the best model for the EMNIST data set, there are two things need to be considered: distance calculation method and parameter k (number of neighbours to consider). To do this, I used k-fold to split the training dataset into three parts that are equal in size, the we'll use two of them as the new training dataset and the other as the testing dataset, repeat it three times to find the best parameters.

Distance Calculation

In my tuning phase, I explore 3 distance calculation method: Mahattan, Euclidean and Chebyshev. While all three performed at a similar speed, there are a small difference in accuracy:

- Chebyshev distance clearly performed the worst, but only less than 3 percent worse than the other two distance calculations.
- Mahattan and Euclidean distance performed on a similar accuracy, but Euclidean distance ultimately came out on top, but only for about 1 percent.

In concludion, Euclidean distance was the most accurate distance measurement. This might be due to the fact that Euclidean distance measures the straight-line distance between two points in a multidimensional space, making it a natural choice for image data where pixel intensities represent spatial information.

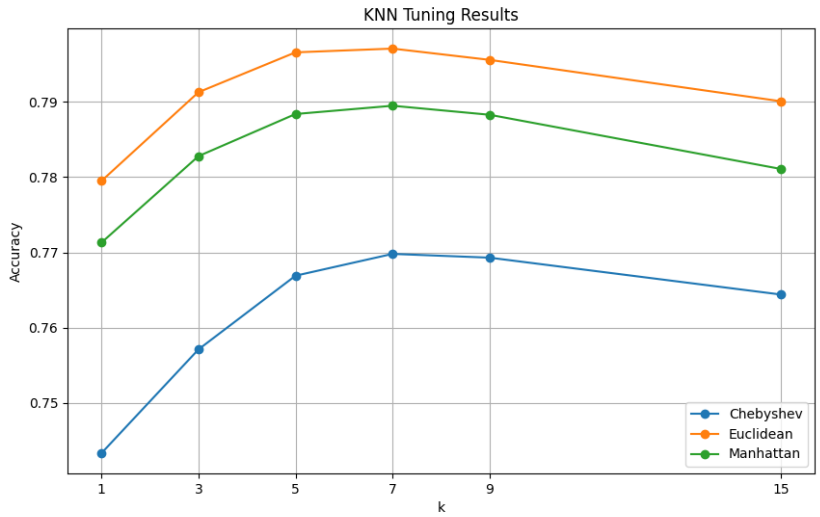


Figure 16: KNN Tuning Result

Choosing K

The parameter k, the number of neighbors to consider, is a critical hyper-parameter. In the one hand, small k value can lead to overfitting because the model becomes too sensitive to the specific instances in the training data, including any noise or outliers. In the other hand, A large k value can lead to underfitting because the model becomes too generalized, averaging over a large number of instances and potentially ignoring important local patterns. In my tuning phase, I chose six odd values (to avoid two label having the same amount of votes) of k for testing: k=1, k=3, k=5, k=7, k=9, k=15. The accuracy steadily increases with k, reaching its highest point at k=7, and then consistently decrease, making 7 the most optimal number of neighbors.

3.1.2 Model evaluation

The advantages of KNN comes in its simplicity and flexibility: it is straightforward to implement and understand and easy to tune since it does not have a training phase. Yet despite that, with proper tuning, as demonstrated with k=7 and Euclidean distance, KNN can achieve reasonable accuracy.

The KNN model does have it fair share of limitations: it requires a significant amount of computation and memory, especially with a large and high-dimensional dataset such as EMNIST. It is also sensitive to noise and imbalance in the dataset itself. Fortunately, the EMNIST Balanced dataset, as the name suggests, is already balanced and deliberately reduced in size. With the help of PCA, the problems mentioned above are significantly minimized.

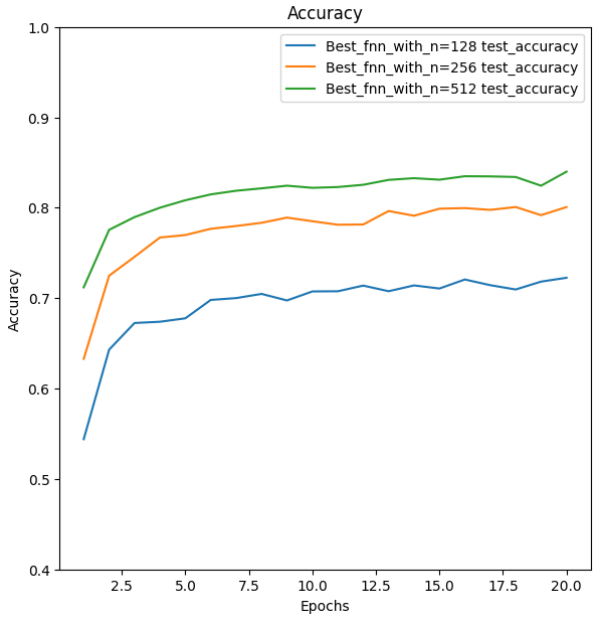


Figure 17: FNN Training Result

3.2 Feedforward Neural Network

3.2.1 Hyper-parameter tuning

In my hyper-parameter tuning process, I explored various learning rates and hidden unit configurations to optimize the performance of my Feed-forward Neural Network (FNN) for the EMNIST balanced dataset. Specifically, I experimented with learning rates (lr) of 0.01, 0.1, and 0.5, and hidden-unit parameters of 128, 256, and 512. The choices for these hyper-parameters were guided by the need to balance model complexity and training efficiency.

- Learning Rate 0.01: This learning rate provided a balanced and stable learning curve across different hidden unit configurations. The model showed consistent improvement in both training and test accuracy over epochs, with 512 hidden units achieving the highest test accuracy of 78.5% after 20 epochs.
- Learning Rate 0.1: This learning rate significantly accelerated the learning process. Models with 512 hidden units achieved a test accuracy of 83.% after 20 epochs. The rapid convergence is beneficial for quicker training times, though the performance gains plateaued sooner compared to a learning rate of 0.01.
- Learning Rate 0.5: This aggressive learning rate led to instability in training, with models frequently encountering 'nan' values in loss and accuracy metrics, especially with higher hidden unit configurations. The rapid parameter updates likely caused the models to overshoot optimal values, resulting in poor convergence.
- Hidden Units n = 128: Models with 128 hidden units showed steady improvement but achieved lower overall accuracy compared to larger configurations. This is likely due to the limited capacity to capture complex patterns in the data.

- Hidden Units n = 256: Increasing to 256 hidden units improved model performance significantly, providing a good balance between capacity and computational efficiency. The test accuracy reached up to 80% with a learning rate of 0.1 after 20 epochs.
- Hidden Units n = 512: The highest configuration in terms of hidden units consistently provided the best accuracy across different learning rates, demonstrating the benefit of increased model capacity. However, this also came with increased computational cost and potential overfitting risks.

Among these, the model that achieved the highest test accuracy in the final epoch had the following parameters:

- Number of hidden units n = 512
- Learning rate = 0.1
- Number of epochs = 20

This model attained an accuracy of 84% on the test dataset.

3.2.2 Model evaluation

Feedforward Neural Networks (FNNs) exhibit limitations when applied to image classification tasks due to their inability to effectively capture spatial hierarchies and local patterns inherent in image data. While they offer simplicity and faster training times, FNNs generally achieve lower accuracy compared to specialized architectures like Convolutional Neural Networks (CNNs). This is because each pixel in an FNN is treated independently, leading to a loss of critical spatial relationships. As a result, FNNs are less suitable for image classification, particularly for complex datasets, where CNNs or other deep learning models are more effective in extracting and leveraging detailed features.

3.3 Convolutional Neural Network

3.3.1 Hyper-parameter tuning

Simple CNN

- Learning Rate 0.01: Steady improvement, with training accuracy reaching 86.6% and test accuracy around 86.5% after 20 epochs.
- Learning Rate 0.1: Faster convergence, achieving training accuracy of 87.9% and test accuracy of 85.9% after 20 epochs.
- Learning Rate 0.5: Training was unstable, resulting in poor performance with consistent nan values in loss and accuracy metrics.

Among these, the model that achieved the highest test accuracy in the final epoch had the following parameters:

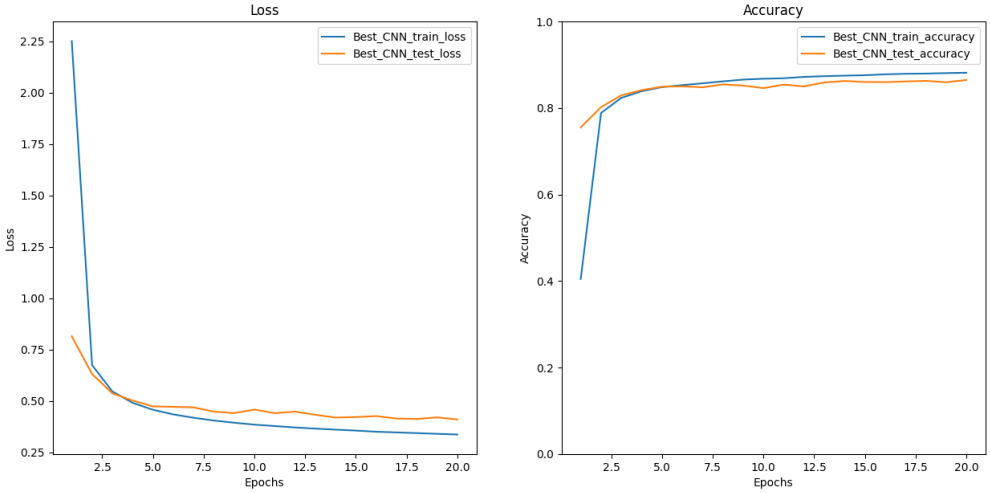


Figure 18: Best Simple CNN Model

- Learning rate = 0.01
- Number of epochs = 20

This Simple CNN model attained a test accuracy of 86.47% in the final epoch.

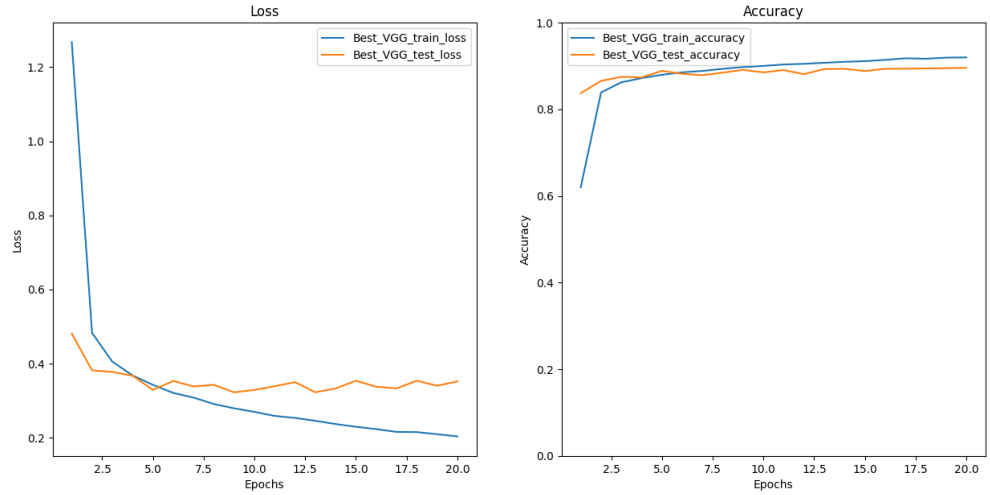


Figure 19: Best VGG_16 Model

VGG_16

- Learning Rate 0.01: High accuracy with slower convergence, achieving 96.3% training accuracy and 89.0% test accuracy after 20 epochs.
- Learning Rate 0.1: Balanced speed and stability, with 94.2% training accuracy and 89.4% test accuracy after 20 epochs.

- Learning Rate 0.5: Managed to converge but exhibited less stability compared to lower learning rates, reaching 91.8% training accuracy and 89.6% test accuracy after 20 epochs.

Among these, the model that achieved the highest test accuracy in the final epoch had the following parameters:

- Learning rate = 0.5
- Number of epochs = 20

This VGG model attained a test accuracy of 89.6% in the final epoch.

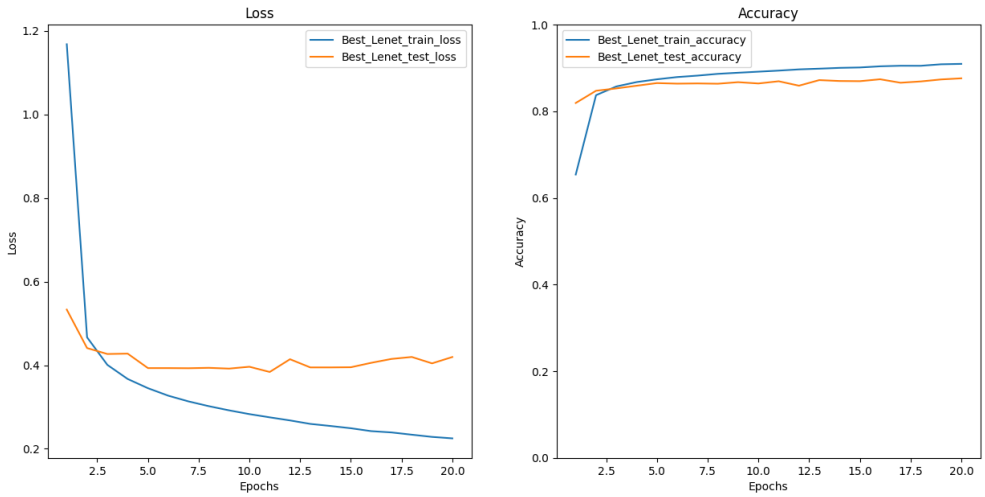


Figure 20: Best LeNet Model

LeNet

- Learning Rate 0.01: Achieved 88.2% training accuracy and 86.6% test accuracy after 20 epochs, showing consistent improvement across epochs.
- Learning Rate 0.1: Faster convergence with final training accuracy of 90.8% and test accuracy of 86.8% after 20 epochs.
- Learning Rate 0.5: Similar to the simple CNN, training was unstable with inconsistent performance metrics.

Among these, the model that achieved the highest test accuracy in the final epoch had the following parameters:

- Learning rate = 0.1
- Number of epochs = 20

This LeNet model attained a test accuracy of 87.6% in the final epoch.

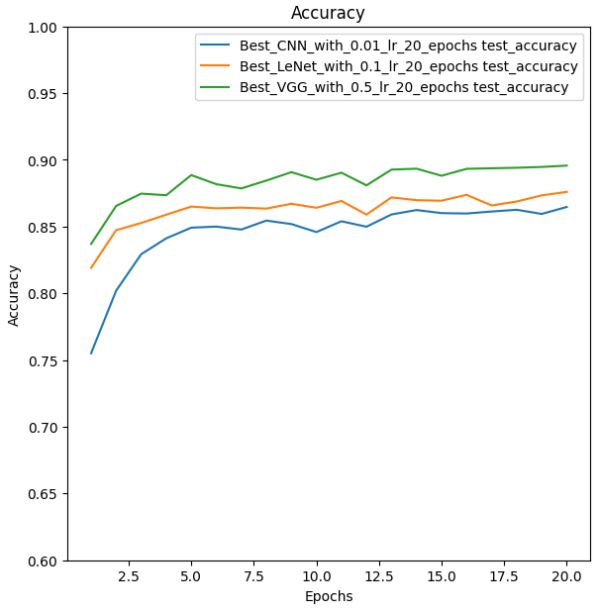


Figure 21: CNN Models Training Result

3.3.2 Model Evaluation

The VGG_16 model's higher performance can be attributed to its deeper and more complex architecture, which consists of 16 layers that enable it to learn a rich hierarchy of features. The multiple convolutional layers allow the model to extract and refine features progressively, leading to a more nuanced understanding of the image data. This results in higher accuracy, particularly for datasets with subtle differences and detailed patterns, like the EMNIST balanced dataset.

In contrast, the Simple CNN, with fewer layers and a simpler structure, still managed to achieve competitive results. Its ability to generalize well without overfitting highlights the effectiveness of basic convolutional operations in image classification tasks. The faster training times and lower computational requirements make it an attractive option for situations where resources are constrained.

LeNet, with its intermediate complexity, bridges the gap between Simple CNN and VGG_16. Its architecture, which includes convolutional layers followed by fully connected layers, provides a good balance between feature extraction and classification. The consistent improvement in accuracy across epochs indicates that LeNet can effectively learn from the data, though it does not capture as many intricate details as VGG_16.

3.4 Decision Tree & Random Forest

Decision Tree and Random Forest is predicted not to be able to classify well on a data set with a high amount of attributes, all of which are numeral attribute. To reduce the training time, we used Cython but PCA is still a must to reduce the number of attributes.

3.4.1 Hyper-parameter Tuning

There are two parameter to be tuned in this models those are max_depth and n_trees (number of tree in random forest).

Max depth

Max_depth is a primary parameter in tuning Decision Trees.

- A large max_depth in the other hand can lead to over fitting because the model becomes too strict on the classification and heavily depend on the dataset. For a dataset with a large number of attributes
- A small max_depth can lead to under fitting because number of attributes considered in classifying instances is not enough.

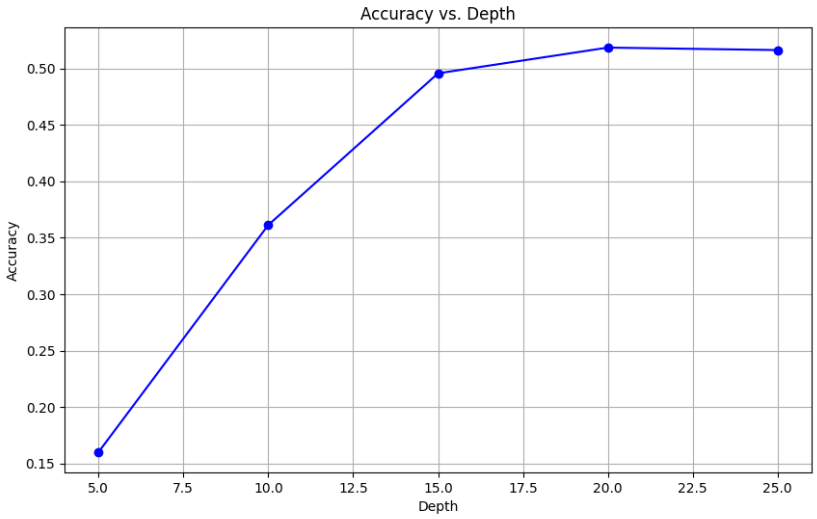


Figure 22: Decision Tree Tuning Result

The diagram above initialise that at $max\_depth = 20$, the model started to over fit. This is also the parameter we used to add to Random Forest and tune the n_trees parameter.

Number of trees

To tackle overfit create by high max_depth, we used multiple trees those are trained using different dataset (created by Boostrap Sampling). This is the performance of Random Forest with different n_trees:

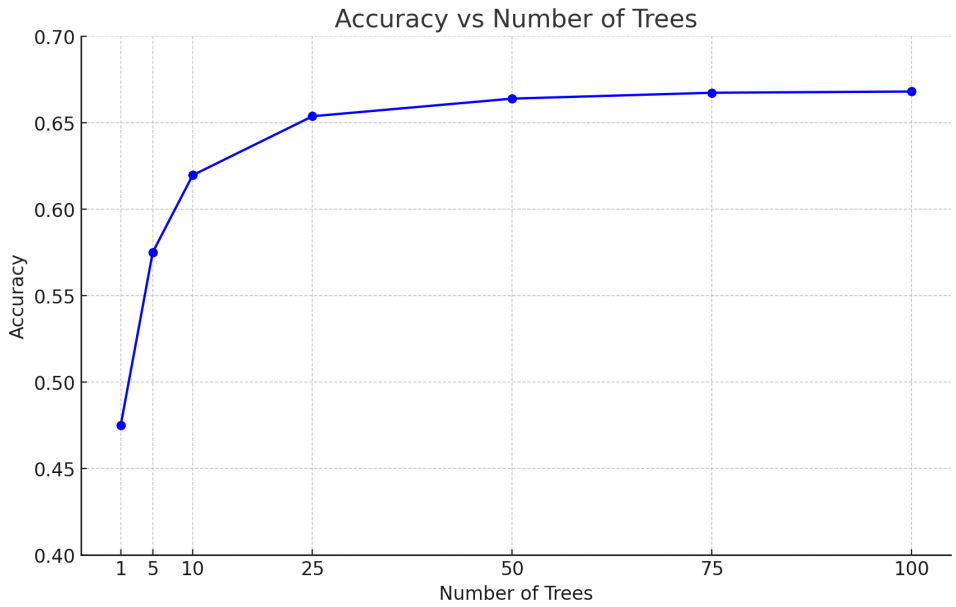


Figure 23: Random Forest Tuning Result

The diagram indicate that the accuracy of the model increases significantly from 51.63% to 66.81% when the results from multiple trees are combined.

3.4.2 Model evaluation

Decision Tree and its ensemble model Random Forest have several advantages compared to other approach. Firstly, they can capture non-linear relationships between features, which is advantageous for complex data like images. Secondly, Decision Trees do not require any assumptions about the distribution of the data.

However, Decision Tree have certain disadvantages including overfitting, instability and computational cost. Decisicion trees are prone to overfitting, especially with deep trees, unless pruning or other techniques are used. Also, small changes in the data can lead to completely different trees being generated. Random Forest somewhat tackle these problem while enhance another problem that is computational cost. Training deep trees can be computationally intensive and time-consuming. Random Forest contains numerous amount of Decision Tree multiply that computational cost.

Overall, after many trials to reduce training and testing time, this model is still time consuming while the accuracy is not as high as other models using different approaches.

3.5 Comparisons

3.5.1 Accuracy

This diagram compares accuracy of different models used in this project.

Overall, RF is the worst model with accuracy under 70% while other models have above 80% accuracy. SVM model, unexpectedly, has higher accuracy than the best FNN model, at

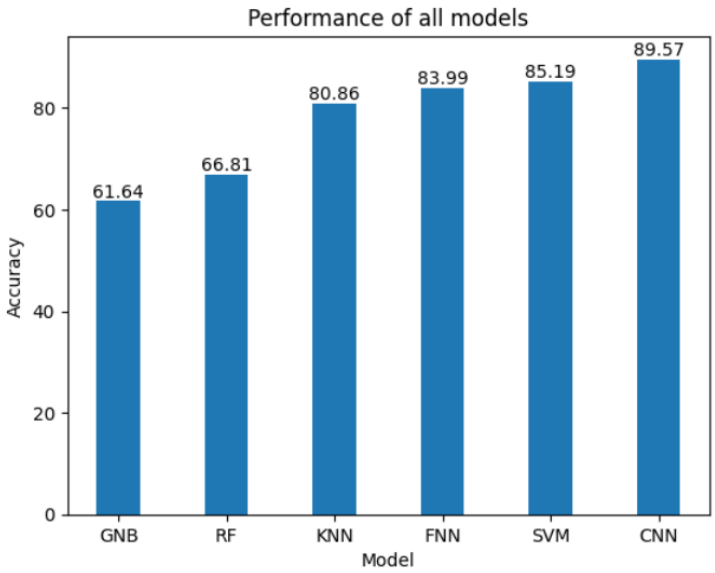
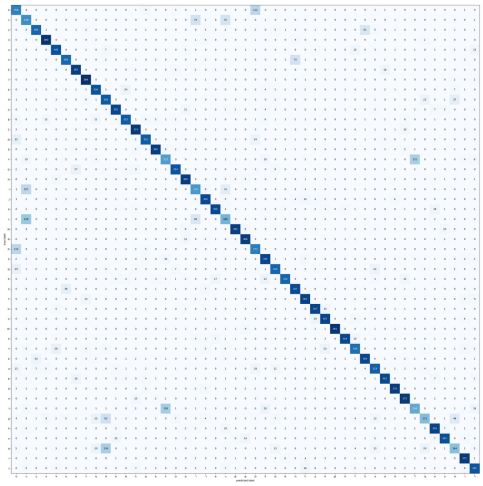


Figure 24: All models comparison

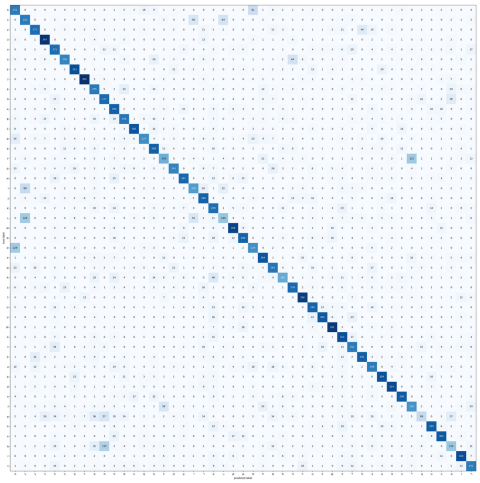
85%. CNN model, specifically VGG model, perform the best, with nearly 90% accuracy score.

3.5.2 Confusion Matrix

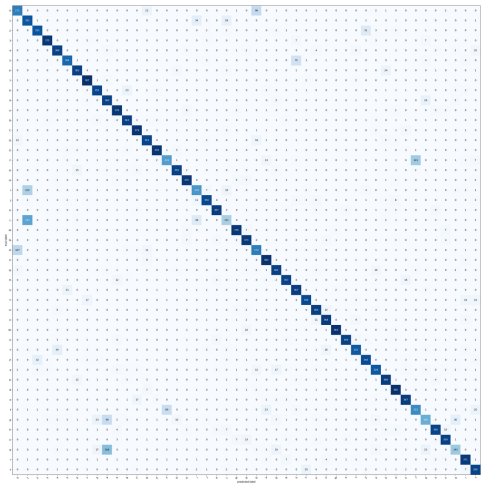
These are Confusion Matrix of my approaches to classify hand written digits and characters:



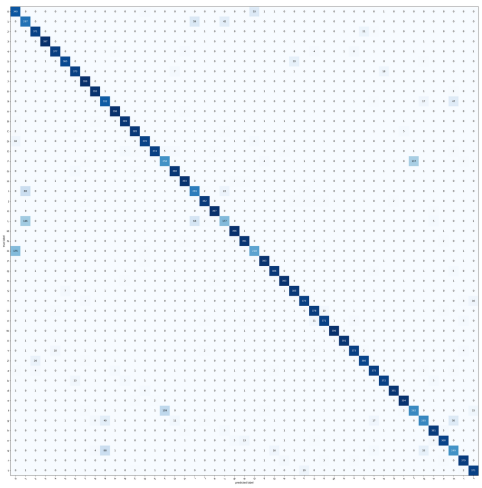
(a) KNN



(b) RF



(c) FNN



(d) CNN

Figure 25: Confusion Matrix

Overall, they provide insights into specific areas where the model performs well and where it needs improvement. Specifically, there are combinations that are easily confused:

- 0(digit), O(letter)
- 1(digit), I(letter), L(letter)
- f(letter), F(letter)
- 9(digit), q(letter), g(letter)

This is expected since all of these characters look similar to other characters in the dataset, sometimes even indistinguishable by human eyes. Also, this explain the limit of all after all tuning and adjustment, the prediction cannot exceed 90% accuracy in test set.

3.6 Model Deployment

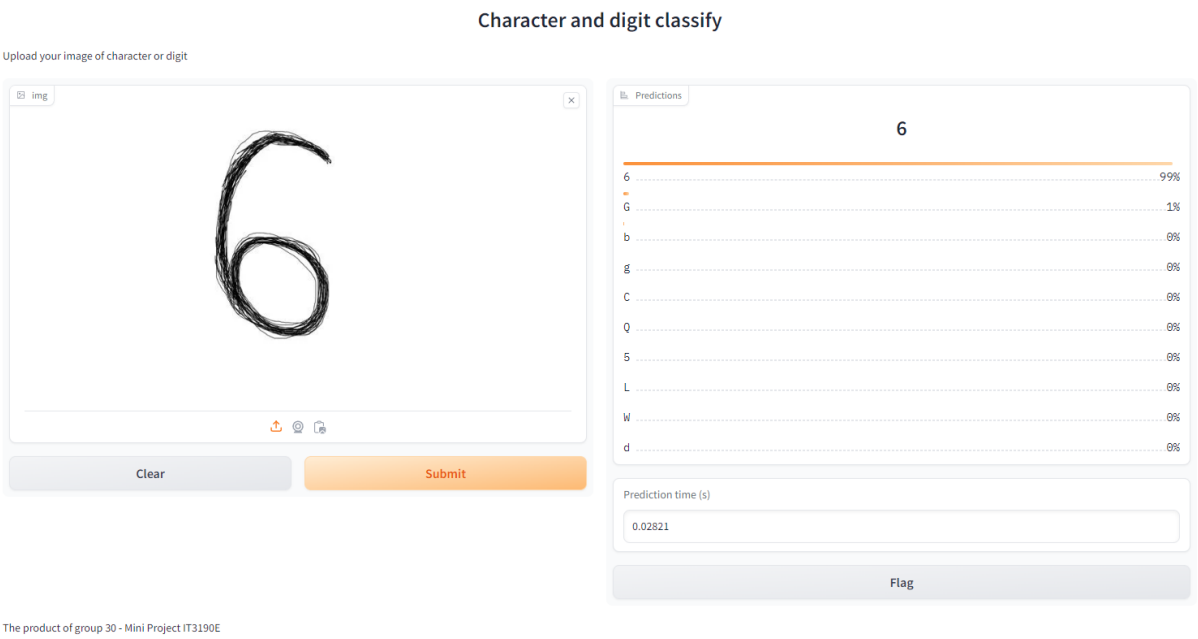


Figure 26: Character and Digit Classifier

After training a variety of models, I decided to deploy the model with the highest accuracy, VGG-16, which achieved a final accuracy of 90% after 10 epochs. This model was chosen due to its superior performance and reasonable training time of approximately 13 minutes. To facilitate deployment, I implemented a preprocessing step to transform input images to match the EMNIST dataset format, ensuring compatibility with the VGG-16 model.

The deployment process involves using Gradio, a Python library that simplifies the creation of web-based interfaces for machine learning models. The preprocessing step converts images to grayscale, applies Gaussian blur to reduce noise, uses adaptive thresholding for better character visibility, and centers the characters within a 28x28 pixel frame. This transformation ensures that the images are appropriately formatted for input into the VGG-16 model.

The Gradio interface allows users to upload images and receive predictions from the VGG-16 model. The interface displays the original and transformed images and outputs the prediction probabilities for each class along with the time taken for prediction. This setup provides an accessible and user-friendly way to utilize the VGG-16 model for image classification tasks, leveraging its high accuracy and robust performance on the EMNIST balanced dataset.

4. Conclusion

4.1 About used method

In this project, I used 4 different approach to solve the handwriten characters and digits classifier. I also used PCA as preprocessing method and have several adaptation for different approach. Overall, the result ranging from 60% to 90% accuracy with FNN and CNN are two

most accurate model. The Confusion Matrix shows the common mistake of all model, which entailed the limitation of all models lay on the characteristic of the dataset and the natural of the problem.

References

[1] The EMNIST Dataset - National Institute of Standards and Technology

[2] StatQuest with Josh Starmer (2021): Gaussian Naive Bayes, Clearly Explained!!!

[3] StatQuest with Josh Starmer (2018): StatQuest: Random Forests Part 1 - Building, Using and Evaluating

[4] Zakaria Jaadi (2024) - A Step-by-Step Explanation of Principal Component Analysis (PCA)

[5] Gradient-based learning applied to document recognition Very Deep Convolutional Networks for Large-Scale Image Recognition A review: Data pre-processing and data augmentation techniques MNIST Data Augmentation with Elastic Distortion