Large Assignment of Machine Learning: Traditional Approachs for EMNIST Handwritten Character Recognition problem

Instructor: Nguyen Duc Dung, PhD[1,2]
Students: Ly Tran Tan (ID: 2353077)[1,2 *], Le Nguyen Nhat Minh (ID: 2352743)[1,2 *]

[1]Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology (HCMUT), 268 Ly Thuong Kiet, HCMC, Vietnam

[2]Vietnam National University Ho Chi Minh City (VNU-HCM), Linh Xuan Ward, HCMC, Vietnam

January 14, 2026

Abstract

This report investigates the problem of single handwritten character recognition using the EMNIST Balanced dataset. Today, we explore the transition from classical Machine Learning approaches to the basis of Deep Learning techniques. Three models were implemented and compared: Support Vector Machine, Random Forest classifier representing the classical approachs, and a Convolutional Neural Network representing the Deep Learning approach. In below sections, we will explain why the Deep Learning techniques outstand those of Machine Learning ones. Experiments demonstrate that while classical methods provide a strong baseline, Deep Learning significantly outperforms them (approximately 90% accuracy) by effectively capturing spatial hierarchies in image data. The study also highlights the specific challenges of the EMNIST dataset, such as class ambiguity (for instance, '0' vs 'O') and data orientation issues.

1 Problem Statement and Motivation

You can imagine that the archives of an old hospital, a bustling post office, or a national library. These institutions possess millions of physical documents and patient records, handwritten letters, and historical manuscripts. To make this information searchable and accessible to the modern world, we must digitize it. However, while scanning a document into an image is trivial, extracting the semantic meaning from that image is exponentially more difficult. A computer can easily read standardized fonts, just like the common one is Times New Roman, but it stumbles significantly when faced with the chaotic nature of human handwriting. This "Big Problem" of full handwriting recognition on whole context, or known as Optical Character Recognition problem, is actually a composite of thousands of smaller, fundamental failures. If an automated system misinterprets a doctor's handwritten in a dosage instruction, the entire digitization pipeline fails. Therefore, before we can hope to build systems that understand sentences or paragraphs, we must first master the atomic unit of this problem, which is the arise problem of this report, **achieving as highest accuracy as possible in Single Handwritten Character Recognition problem**.

*These authors with * contributed equally to this work.

The motivation for this study stems from the realization that the solved problems of the past are no longer sufficient benchmarks for this realworld complexity. For years, the computer vision community relies on the MNIST dataset, you can access at *[9]*, which only contains digits from 0 to 9, as the gold standard. However, distinguishing ten distinct digits is vastly simpler than the reality of alphanumeric writing, where structure, orientation, similarity create high ambiguity, which leads to the failure in recognition those characters. To address this, we turn to the **EMNIST Balanced** dataset, published by *[4]*. By expanding the classification task from 10 classes to **47 classes**, including digits, uppercase, and distinct lowercase letters. It allows us to determine whether modern Deep Learning techniques merely offer incremental improvements or if they fundamentally outstand Classical Machine Learning approaches in extracting features from this messy, high-dimensional problem.

2 Existing Solutions

Historically, **Handwritten Character Recognition** has been approached using a wide spectrum of techniques, ranging from simple machine learning algorithms to complex deep neural networks. Below paragraphs will analyze the most common and well-known solutions to this problem. One of the most intuitive methods is **k-Nearest Neighbors (k-NN)**, written in *[6]*, which classifies a new image by finding the k most similar images in the training set based on *Euclidean distance*. While k-NN is conceptually simple and requires no training phase, it seems to not be suitable for real-time applications on large datasets like EMNIST because its distance calculating to all 112,800 training samples during the inference phase is computationally prohibitive, and the method remains highly sensitive to pixel shift and noise.

Another classical machine learning approach is the **Support Vector Machine (SVM)**, demonstrated detailed at *[5]*, which attempts to find the optimal hyperplane separating classes with the maximum margin. For image data, non-linear kernels such as the *Radial Basis Function* are typically employed to capture complex patterns. Although SVMs are mathematically robust and effective in high-dimensional spaces, they scale poorly with dataset size. This is reason why we prefer *Linear Kernel Function* for huge size dataset in order to increase computation efficiency, since the training complexity often ranges from $O(N^2)$ to $O(N^3)$.

A more scalable classical approach is the **Random Forest** algorithm, published at *[1]*, an ensemble method that constructs a multitude of decision trees during training to output a consensus class prediction. Random Forest is highly efficient for this task as it can handle raw pixel data without heavy data-preprocessing phase and utilizes parallel processing (using all CPU cores) for rapid training (since we do not have the NVDIA GPU with us but an integrated on-board graphic card from the CPU). However, this way of approach will achieves lower like other classical methods, it lacks spatial awareness, for example, it treats pixel $(0, 0)$ and pixel $(0, 1)$ as independent features, which effectively ignores the structural relationships inherent in visual data (especially the image data).

In contrast to these classical techniques, **Convolutional Neural Networks (CNN)**, coming from *[10]*, represent the current state-of-the-art (SOTA) for computer vision. Unlike the previous methods, CNN preserve the spatial structure of the input image, leveraging translation invariance to recognize characters regardless of their position. By automatically learning hierarchical feature, which progresses from simple edges to abstract shapes so as to help CNN effectively handle the variability of handwriting. However, they requires significant computational power and large amounts of labeled data to prevent overfitting (since unlike many machine learning algorithms, deep learning can perform well on huge size dataset without prior bias knowledge).

Efficiency and Selection Strategy - what is the best approach?

When selecting a solution for EMNIST, the choice depends on the constraint. Firstly, we consider **maximum Accuracy**, which is our main desire, **CNN is the most appropriate solution** for high accuracy Handwritten Character Recognition tasks where computational resources allow. It is the only method that effectively handles the geometric distortions inherent in handwriting. Then secondly, for the **best classical machine learning approach in accuracy metric**, **SVM will be the best candidate between all classic ways**. Thirdly, **for low-resource environments (as in our situation now), Random Forest is the most efficient**. It provides trade-off accuracy with a fraction of the compute cost and faster training as well as inference time than both SVM and CNN. In a nutshell, we will implement all three approachs, which are CNN, Random Forest and SVM, in their primitive invention, however, in the section 5, we will demonstrate an improvement to those classic machine learning approach by combining *Feature Extraction* before feeding the preprocessed data into the model, which will help to reduce data complexity, improve model accuracy, and optimizes computational efficiency. You will question us the reason why we obmit the k-NN here, the answer are those k-NN is too slow at inference compared to Random Forest, with the similar accuracy, which means worse trade-off than Random Forest, and also its accuracy is outstanded by CNN and SVM.

3 Methodology

In this section, we detail the experimental framework designed to evaluate and compare the performance of Deep Learning versus Classical Machine Learning on the handwritten character recognition task. Our methodology proceeds in three distinct stages: firstly, the selection and preprocessing of the EMNIST Balanced dataset to ensure a fair and rigorous benchmark, secondly, the implementation of a custom Convolutional Neural Network (CNN) architecture optimized for spatial feature extraction, and thirdly, the development of an optimized Random Forest classifier to represent a robust classical baseline. Below sections will describe the dataset characteristics, models' architecture, and optimization strategies employed for each approach.

3.1 Dataset: EMNIST Balanced

The EMNIST (or Extended MNIST) dataset serves as a challenging successor to the classic MNIST digits dataset, expanding the scope to include handwritten letters while maintaining the same 28×28 pixel format, which is an ideal benchmark for evaluating model performance on more complex character recognition tasks. For this study, we specifically selected the **EMNIST Balanced** split. While the full dataset (the ByClass one) contains 62 classes, including 10 digits, 26 uppercase and 26 lowercase, the 'Balanced' split reduces this to **47 classes** by merging uppercase and lowercase letters that are structurally identical in handwriting, for instance, 'C' vs 'c', 'O' vs 'o', etc.

We prioritized the 'Balanced' split over other variants, like ByMerge or ByClass, because it provides an equal number of samples per class. This balance allows us to train using standard loss functions without needing to adjust class weights (for example, **Defered Re-Weighting** showed in *[2]*) or perform extensive resampling (for instance, **Label Powerset Random Oversampling** method proposed at *[3]*) to handle the severe class imbalance present in the larger splits.

The selected dataset comprises **112,800 training samples** and **18,800 testing samples** spanning 47 classes, including **digits (0-9), uppercase letters (A-Z), and the 11 distinct lowercase letters (a, b, d, e, f, g, h, n, q, r, t)**. Prior to training, the data underwent a series of data-preprocessing steps to ensure compatibility. Since raw EMNIST images in CSV format are inherently rotated 90 degrees and flipped (in MatLab format as the authors have

mentioned), a transpose operation was first applied to correct their orientation. Also, all pixel values were normalized from the range $[0, 255]$ to $[0, 1]$ to ensure efficient gradient descent and support model convergence.

3.2 Approach 1: Deep Learning - Convolutional Neural Network

In this section, we detail the architecture and training strategy of our custom CNN model designed to capture the spatial dependencies inherent in the EMNIST handwriting samples.

3.2.1 Model Architecture

To address the complexity of the EMNIST Balanced dataset, we designed a custom Convolutional Neural Network constructed as a sequence of three convolutional blocks followed by a fully connected classification head. Each convolutional block is engineered to progressively extract features of increasing complexity, from simple edges in the initial layers to abstract shapes and character structures in the deeper layers.

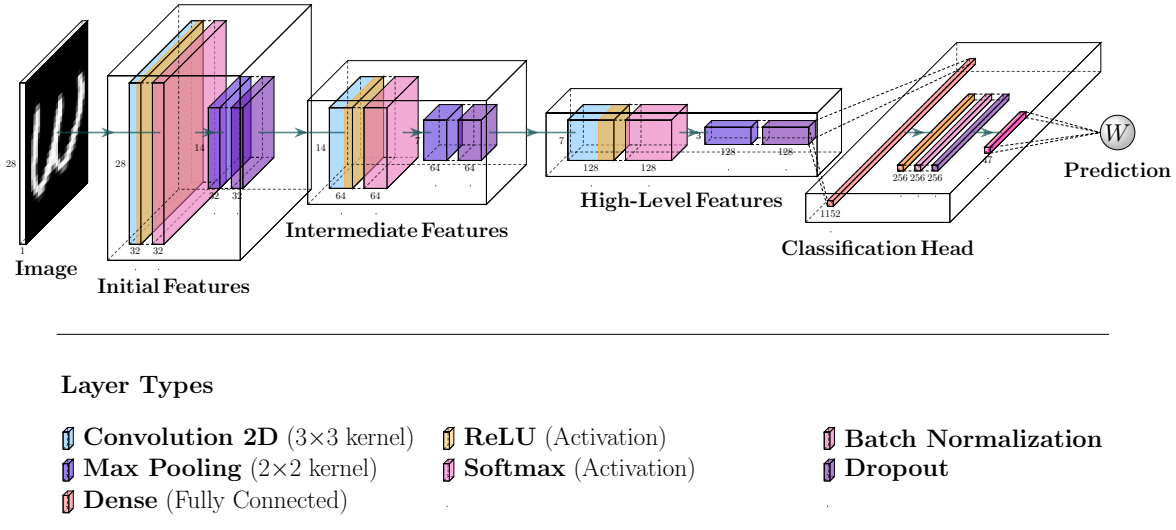


Figure 1: Overview of the Convolutional Neural Network architecture. The network takes a 28×28 input image and processes it through a sequence of three convolutional and max pooling blocks for hierarchical feature extraction, followed by drop out, finally, resulting feature maps are flattened and passed through two fully connected dense layers for final classification

From this point, we have the figure provided for you to obtain our CNN model's overview more clearly. The first block consists of a `Conv2D` layer with 32 filters (3×3 kernel), utilizing 'same' padding to preserve spatial dimensions and ReLU activation for non-linearity. This is immediately followed by `BatchNormalization` to stabilize the learning process (since it normalizes the data within each batch by calculating the mean and variance of data in a batch and then adjusting the values so that they have similar range, also included the scaling and shifting the values so that model learn effectively) and `MaxPooling2D` (2×2) to reduce spatial dimensionality. To prevent overfitting at this early stage, a `Dropout` with L2 regularization are applied. The second block mirrors this structure but increases the network depth to 64 filters. The third and final convolutional block expands to 128 filters, enabling the model to capture high-level semantic features.5.

The classification head flattens the 3D feature maps into a 1D vector, which is fed into a dense layer of 256 units. This fully connected layer also combines a L2 regularization, is followed

by a Batch Normalization, and a robust Dropout to enforce generalization. The final output layer consists of 47 neurons with Softmax activation, corresponding to the 47 distinct classes in the EMNIST Balanced dataset. The detailed architecture and parameter count are summarized in Table 1.

Table 1: Detailed Architecture of our Proposed CNN Model

Layer (type)	**Input Shape**	**Output Shape**	**Number of Params**
Block 1: Initial Feature Extraction			
Conv2D	(28, 28, 1)	(28, 28, 32)	320
Batch Normalization	(28, 28, 32)	(28, 28, 32)	128
Max Pooling 2D	(28, 28, 32)	(14, 14, 32)	0
Dropout	(14, 14, 32)	(14, 14, 32)	0
Block 2: Intermediate Features			
Conv2D	(14, 14, 32)	(14, 14, 64)	18,496
Batch Normalization	(14, 14, 64)	(14, 14, 64)	256
Max Pooling 2D	(14, 14, 64)	(7, 7, 64)	0
Dropout	(7, 7, 64)	(7, 7, 64)	0
Block 3: High-Level Features			
Conv2D	(7, 7, 64)	(7, 7, 128)	73,856
Batch Normalization	(7, 7, 128)	(7, 7, 128)	512
Max Pooling 2D	(7, 7, 128)	(3, 3, 128)	0
Dropout	(3, 3, 128)	(3, 3, 128)	0
Classification Head			
Flatten	(3, 3, 128)	(1152)	0
Dense (Fully Connected)	(1152)	(256)	295,168
Batch Normalization	(256)	(256)	1,024
Dropout	(256)	(256)	0
Dense (Output)	(256)	(47)	12,079
Total params			**401,839**
Trainable params			**400,879**
Non-trainable params			960

Table 1 outlines the quantitative complexity of our proposed architecture, which contains a total of **401,839 parameters**. The vast majority of these are **400,879 trainable parameters**, representing the weights and biases in the Convolutional and Dense layers that are updated during backpropagation. Also, Batch Normalization account for a small percent only, since 50% parameters of the Batch Normalization parameters are trainable, and the remaining 50% ones are non-trainable, which is the reason for the existence of **960 non-trainable parameters**. These specific non-trainable parameters differ from fixed weights and instead represent the moving mean and variance statistics tracked by the four Batch Normalization layers, which are updated during the forward pass rather than through gradient descent. Notably, the transition from the feature extraction blocks to the classification head (the first Dense layer) accounts for approximately 73% of the total model size (295,168 parameters), highlighting that while the convolutional base is computationally efficient, the dense classification head is responsible for the majority of the model's memory.

3.2.2 Training and Optimization

The training process was optimized to ensure convergence, early stoping for time efficiency and maximize generalization performance. The model was compiled using the **Adam optimizer**, initially set with a learning rate of 0.001. We employed **Categorical Cross-Entropy** as the loss function, which is standard and common for multi-class classification tasks.

To effectively manage the training dynamics, we implemented specific callbacks. We use scheduler to monitor the validation accuracy, which is `ReduceLROnPlateau`; if improvement stalled for 3 consecutive epochs, the learning rate was automatically reduced by a factor of 0.5. This dynamic adjustment allows the model to take large steps initially and finer steps as it approaches the optimal solution. Additionally, a `ModelCheckpoint` callback was employed to save only the best-performing model weights based on validation accuracy, ensuring that the final evaluated model was not an overfitted version from the last epoch but the most generalized version achieved during training. The network was trained for 50 epochs with a batch size of 64, using real-time data augmentation (rotation, zoom, and shift) to further improve robustness against handwriting variations.

3.3 Approach 2: Classical Machine Learning - Support Vector Machine

To provide a comprehensive comparison against the Deep Learning approach, we implemented a classical machine learning baseline using the Support Vector Machine (SVM). This method is mathematically rigorous and particularly effective for high dimensional data, though it presents unique computational challenges.

3.3.1 Preprocessing and Dimensionality Reduction

We knew that SVMs rely on distance calculations, specifically Euclidean distance in the RBF kernel. If features have different ranges, those with larger magnitudes will dominate the objective function, preventing the model from learning the correct decision boundary. To mitigate this, we do some **feature scaling**, or a `StandardScaler`, to normalize the pixel intensity values, centering them to have a mean of 0 and a standard deviation of 1.

Also, the EMNIST images consist of $28 \times 28 = 784$ features. Since the computational complexity of non-linear SVMs scales quadratically with the number of samples and linearly with the number of features, training on raw 784-dimensional vectors is computationally prohibitive. Honestly, we have tried the way of training on raw vector with 784 features, consequently, the time for training on train data and inference on valiadtion data would take approximately 30 minutes. However, we want to use grid search to find the best combination of parameter for our model, that would take us almost roughly a day time to train all combinations and inference in each iteration of **Grid Search Cross-Validation**, which will be defined later, to find the best model of all. To address this, we must **reduce feature dimension**, specifically integrated **Principal Component Analysis (PCA)** into the pipeline, which we adapted from *[12]*. PCA projects the high dimensional pixel data onto a lower dimensional subspace while preserving the maximum variance, since arcording to Shanon in information theory, the information of the data is the variance of the data. As our experiments in section 4, we treated the number of principal components as a hyperparameter to be optimized, balancing the trade-off between information variance and computational efficiency.

3.3.2 Model Architecture

We utilized the `SVC` implementation from the Scikit-Learn library. The core of our architecture relies on the *Radial Basis Function (RBF)* kernel. The reason for not using other kernel is that: while a Linear kernel is faster, handwriting data is inherently non-linear; the separation between

complex characters cannot be defined by a straight line. The RBF kernel maps the input space into an infinite dimensional feature space, which will guarantee the best boundary of all kernel but with higher computation.

To handle the multi-class nature of the dataset with $N = 47$ classes, we employed the *One-vs-One* strategy. This approach constructs a binary classifier for every pair of classes, resulting in $N(N-1)/2 = 1081$ classifiers. While this increases the number of models compared to the *One-vs-Rest*, each model is trained on a significantly smaller subset of data, which mean each classifier only trains on the samples belonging to the two specific classes.

3.3.3 Hyperparameter Optimization Strategy

Here we employed **Grid Search Cross-Validation** to identify the optimal model configuration. Rather than training a single model with default parameters, we defined a comprehensive hyperparameter search all possible spaces to mathematically explore optimal structural configurations. However, due to the high computational cost of the RBF kernel and my own limitation on the hardware, the search space was carefully constrained.

To ensure the model generalizes effectively to unseen data, we performed hyperparameter optimization using **5-Fold Cross-Validation**. For each iteration, the model was trained on four partitions (80% of the data) and validated on the remaining partition (20%) to provide the fairness to the above CNN approach. This process was repeated five times for every parameter combination, ensuring that the performance metrics were not biased towards any specific subset of the data. Ultimately, the final model selected was the estimator that achieved the highest mean accuracy across all five folds, which will provide a confident basis for generalization to unseen data. The search space focused on two critical parameters: the regularization parameter (C) and the dimensionality of the feature space. For C, we evaluated values in the set $\{1, 5, 10\}$, balancing the trade-off between maximizing the decision margin (low value of C) and strictly penalizing training misclassifications (high value of C). In parallel, we determined the optimal level of dimensionality reduction by testing the retention of $\{50, 100, 150\}$ principal components. This step was essential to identify the minimum number of features required to accurately distinguish characters while mitigating excessive noise and computational overhead.

3.4 Approach 3: Classical Machine Learning - Random Forest

We selected the Random Forest algorithm in order to establish a robust baseline for comparison against the Deep Learning approach. As an ensemble method based on decision trees, Random Forest is widely known for its versatility, resistance to overfitting, and ability to handle high-dimensional data effectively. This section describes the necessary preprocessing steps to adapt image data for classical algorithms and outlines the configuration of our ensemble model.

3.4.1 Preprocessing and Feature Engineering

Unlike Convolutional Neural Networks, which can process raw 2D image matrices directly, the Random Forest algorithm requires input data to be structured as flat feature vectors. Therefore, we applied a flattening transformation to the EMNIST dataset prior to training. Each 28×28 grayscale image was reshaped into a single 1-dimensional vector of size 784.

Consequently, the spatial relationships between pixels (for example, that pixel $(0, 0)$ is vertically adjacent to pixel $(1, 0)$) are discarded, and each pixel is treated as an independent feature. We also incorporated an optional standardization step using `StandardScaler` to normalize feature distributions. The labels were converted from One-Hot encoded vectors to integer class indices to be compatible with the Scikit-Learn functions and modules that we used.

3.4.2 Model Architecture

For this classical ensemble machine learning approach, it constructs a multitude of decision trees at training time and outputs the class that is of the classes (classification) of the individual trees. This architecture was chosen for its inherent robustness to noise and its ability to handle high-dimensional data without extensive feature engineering.

The internal structure of our Random Forest was configured to balance model complexity with generalization capabilities. Central to this architecture is the ensemble size (`n_estimators`), which aggregates predictions from multiple independent decision trees to reduce individual variance and ensure prediction stability. To prevent any single strong feature from dominating the learning process, we enforced a randomized split criterion (`max_features`), where each node considers only a subset of features, specifically $\sqrt{N_{features}}$ during the splitting phases. This mechanism effectively decorrelates the trees, allowing the ensemble to capture a more diverse range of structural patterns. Furthermore, while the EMNIST Balanced dataset is nominally uniform, we implemented automatic class weighting to dynamically adjust weights inversely proportional to class frequencies, thereby safeguarding against any minor sampling imbalances that may arise during bootstrap aggregation. From a computational perspective, the implementation was optimized for parallel execution, leveraging all N available CPU cores to significantly minimize training latency.

3.4.3 Hyperparameter Optimization Strategy

Similar to SVM, to maximize performance without relying on a static validation set, which inherently limits the data available for model training, we employed **Grid Search Cross-Validation** (`GridSearchCV`) strategy. This "small" optimization process involved systematically testing combinations of key parameters, including the number of estimators, tree depth, and leaf granularity. Specifically, we evaluated ranges of $[100, 200]$ for the number of estimators to identify the optimal trade-off between voting stability and computational inference cost. Furthermore, we explored maximum tree depths of $[10, 20, 30]$ to control model complexity. Constraining the depth is critical for 28×28 pixel inputs, as unbounded trees tend to memorize specific noise patterns (overfitting), whereas limited depth forces the trees to learn more generalizable geometric shapes. We also tested `min_samples_leaf` values of $[2, 4]$ to regulate leaf granularity. By requiring a minimum number of samples at each leaf node, we enforce smoothness in the decision boundaries and prevent the model from isolating single outlier images. Similarly to SVM, this search was conducted using **5-Fold Cross-Validation** the same as the two mentioned approach to ensure statistical robustness, and moreover, to provide the fairness to both the CNN and SVM approach.

4 Experiments and Demonstration

4.1 Experimental Setup

Hardware and Software Environment. All experiments were conducted on a local workstation running the **Ubuntu 24.04.03 LTS** operating system. Due to specific hardware constraints, namely the absence of dedicated GPU acceleration or CUDA support, all computational tasks, which includes model training and inference, were performed strictly on the CPU. The underlying hardware configuration consists of a CPU **12th Gen Intel Core i5-1235U** (10 cores, 12 threads, with a clock speed up to 4.40 GHz) paired with **16 GB of RAM** and integrated **Intel Iris Xe Graphics**. The software framework employed for this study relies on **Python 3.10**, utilizing **TensorFlow/Keras** for deep learning implementations, **Scikit-Learn** for classical machine learning models, and **Pandas** for data processing and analysis.

Training Data Partitioning. To ensure rigorous evaluation, the dataset was partitioned using stratified sampling to maintain the uniform class distribution of EMNIST Balanced. For the CNN model, the data was split into **80% training** and **20% validation** sets. To maintain parity, the Random Forest model was evaluated using **5-Fold Cross-Validation**, where each fold effectively utilizes an identical 80/20 training-validation ratio, ensuring a fair comparison between the two approaches.

Training setup for CNN. The Convolutional Neural Network was trained using the **Adam optimizer** with an initial learning rate of $\boldsymbol{\alpha = 0.001}$. To enforce sparsity and mitigate overfitting, we applied **L2 regularization** (Ridge Regression) to the kernel weights of all convolutional and dense layers, with a regularization strength of $\boldsymbol{\lambda = 0.001}$. This adds a penalty term $\lambda \sum ||w||^2$ to the loss function, discouraging extreme weight values. The training loop was configured for a maximum of **50 epochs** with a **batch size of 32**. To further optimize convergence on the CPU architecture, we implemented a dynamic callback strategy. A `ReduceLROnPlateau` scheduler monitored the validation accuracy. If performance stagnated for **3 consecutive epochs**, the learning rate was automatically **reduced by a factor of 0.5** (bounded by a lower limit of 10^{-6}). Simultaneously, an `EarlyStopping` mechanism was employed to terminate training if validation accuracy failed to improve for **7 consecutive epochs**, ensuring that the final model utilized the weights from the epoch with the highest generalization performance rather than the final training step.

Training setup for Support Vector Machine. For the SVM implementation, we utilized a deterministic **Grid Search Cross-Validation** (`GridSearchCV`) strategy to identify the optimal balance between model complexity and generalization. However, the search space for the SVM was tightly constrained due to the quadratic computational complexity of the **RBF kernel**. The optimization grid comprised **9 distinct combinations**, evaluating Regularization parameters $C \in \{\mathbf{1}, \mathbf{5}, \mathbf{10}\}$ and PCA dimensionality reduction targets `n_components` $\in \{\mathbf{50}, \mathbf{100}, \mathbf{150}\}$. Furthermore, to ensure convergence stability while preventing infinite loops on non-separable vectors, a hard limit of **2,000 iterations** (`max_iter=2000`) was imposed. Crucially, regarding hardware utilization, we were forced to deviate from the parallel processing used in Random Forest. The SVM optimization was executed parallel with `n_jobs=2`. This strict constraint was necessary to prevent memory overflow and system crashes caused by the massive size of the kernel matrix during concurrent fold evaluations on our local machine. The number of CPU jobs above is maximum concurrent threads for my local workstation since we had broken our system many times when trying to set for all available workers, or 12 threads here, and 4 cores, or 4 threads here, both make my laptop restart time to time, finally, we can conclude that 2 threads are appropriate for us. However, if anyone want to retrain our model, just replace the number of threads as the capability of your machine.

Training setup for Random Forest. For the classical approach, we also use **Grid Search Cross-Validation** (`GridSearchCV`) strategy similar to SVM. This method exhaustively evaluated every possible hyperparameter combination within our defined search space to guarantee the identification of the optimal configuration. The grid comprised **12 distinct combinations**, spanning `n_estimators` $\in \{\mathbf{100}, \mathbf{200}\}$, `max_depth` $\in \{\mathbf{10}, \mathbf{20}, \mathbf{30}\}$, and `min_samples_leaf` $\in \{\mathbf{2}, \mathbf{4}\}$. To ensure statistical robustness, each combination was validated using **5-Fold Cross-Validation**, which means that we have to build a total combination of **60 Random Forests**. Due to the hardware constraints of the local machine, the parallelization was strictly controlled by limiting the process to 2 CPU jobs, or `n_jobs=2`.

4.2 Training Process demonstration

In this section, we analyze the training dynamics, convergence behavior, and optimization results of the classical and deep learning approaches. All training metrics reported here reflect the performance on the validation subsets, which are Cross-Validation for Random Forest and SVM, and the fixed Validation Set for CNN, using the CPU-only hardware setup described in the Experimental Setup.

Traning and Optimization Phase of Random Forest. The `GridSearchCV` process exhaustively evaluated 12 unique hyperparameter combinations using 5-Fold Cross-Validation. The optimization process consumed a total of **29.47 minutes**. Table 2 presents the top performing configurations alongside the least effective one for comparison. A detailed examination reveals that **tree depth** was the primary determinant of model performance on the EMNIST dataset. Configurations restricted to a shallow maximum depth of 10 stagnated at an accuracy of approximately 72.5%. In contrast, relaxing this constraint to a depth of 20 or 30 resulted in a significant performance variability, boosting accuracy to over 80%. Notably, the performance differential between depth 20 and depth 30 was negligible (less than 0.01%), indicating that the decision trees successfully capture the majority of discriminative spatial features by depth 20, with further structural complexity yielding diminishing returns.

Trees	Max Depth	Min Samples Leaf	CV Accuracy	Train Time (s)
200	**20**	**2**	**80.40%**	**75.87**
200	**30**	**2**	**80.40%**	**77.81**
100	20	2	80.09%	39.28
100	30	2	80.08%	39.50
200	30	4	79.97%	73.61
200	20	4	79.95%	72.64
100	30	4	79.73%	37.64
100	20	4	79.71%	37.43
200	10	2	72.76%	50.72
200	10	4	72.71%	50.66
100	10	4	72.58%	26.68
100	10	2	72.58%	25.72

Table 2: Grid Search Results for Random Forest Optimization by Cross Validation

Regarding the **ensemble size**, or the number of trees, doubling the number of estimators from 100 to 200 provided a consistent, albeit marginal, accuracy improvement of approximately 0.3%. However, this slight performance gain incurred a linear computational cost, roughly doubling the training duration from 39 seconds to 77 seconds per fold. Weighing these trade-offs, our target model must get high accuracy after all, so the final model selected for the inference phase utilized **200 estimators, a maximum depth of 20 or 30, and a minimum samples leaf of 2**, achieving a peak cross-validation accuracy of **80.40%**.

Training and Optimization Phase of Support Vector Machine. For the SVM, we evaluated 9 hyperparameter combinations focusing on the interplay between dimensionality reduction (`n_components`) and regularization strength, representing by value of C. The results, summarized in Table 3, reveal a counter-intuitive finding regarding feature dimensionality. The configuration with the **lowest dimensionality of 50 components** consistently outperformed those with higher component counts from both 100 and 150 ones. Specifically, using 50 components achieved

a peak accuracy of **85.10%**, which is the same as we expected it to perform to overcome the another mentioned machine learning approach, Random Forest one with maximum of **80,4%**. However, increasing the components to 150 dropped the accuracy to approximately 82.00%. This means that the core variance of the EMNIST character set is effectively captured in the top 50 eigenvectors, and adding further dimensions likely introduces sparsity and noise that hinders the RBF kernel's ability to construct optimal hyperplane.

PCA Components	Regularization C	CV Accuracy	Train Time (s)
50	**5**	**85.10%**	**122.67**
50	10	84.70%	114.94
50	1	84.49%	153.82
100	5	83.68%	133.95
100	10	83.43%	126.64
100	1	83.09%	131.63
150	5	82.00%	878.73
150	10	81.81%	770.65
150	1	81.23%	978.79

Table 3: Grid Search Results for SVM Optimization by Cross Validation

Regarding the regularization parameter C, a moderate value of **5** proved optimal across all component settings from PCA, offering a balance between margin width and classification error. Furthermore, computational efficiency heavily favored lower dimensionality; training on 50 components required approximately **122 seconds** per fold, while training on 150 components surged to nearly **900 seconds** per fold due to the kernel matrix complexity scaling with feature size. Consequently, the final best SVM model selected utilizes **50 PCA components and Regularization term of C = 5**, offering the best trade-off between accuracy, of 85.10%, and computational feasibility.

Training and Optimization Phase of Convolution Neural Network. The Convolutional Neural Network was trained for 50 epochs, consuming a total of **34.64 minutes** on the CPU (averaging 41.57 seconds per epoch). The training log reveals a highly stable convergence trajectory, significantly aided by the learning rate scheduler. The learning trajectory of the Convolutional Neural Network is visualized in Figure 4.

As illustrated on the left of that figure, the training process exhibits three distinct phases. During the **Initial Phase (Epochs 1–11)**, the model demonstrated rapid convergence, with validation accuracy quickly stabilizing around roughly 86.5%. However, as shown in the loss landscape (the sub-figure on the right), the validation loss remained relatively high (approximately 0.77) during this period, indicating that the optimizer was oscillating around a local minimum. A critical inflection point is observable at **Epoch 12**, corresponding to the intervention of the `ReduceLROnPlateau` callback. This adjustment manifests as a sharp, precipitous drop in the loss curve (decreasing from 0.77 to 0.61) and a simultaneous discrete jump in accuracy to 87.8%, validating the effectiveness of the dynamic learning rate scheduler. In the final **Refining Phase (Epochs 21–50)**, subsequent learning rate reductions allowed the model to fine-tune its feature extraction weights. The loss continued to decrease steadily, reaching a global minimum of 0.353, while the validation accuracy peaked at **89.50%** at Epoch 48. Ultimately, unlike the classical Random Forest and SVM approach, which plateaued at about 80% and 85% respectively, the CNN's ability to extract deep hierarchical spatial features allowed it to break through that ceiling, outperforming the baseline by a significant margin of approximately **9.1%** and **4,4%** respectively.

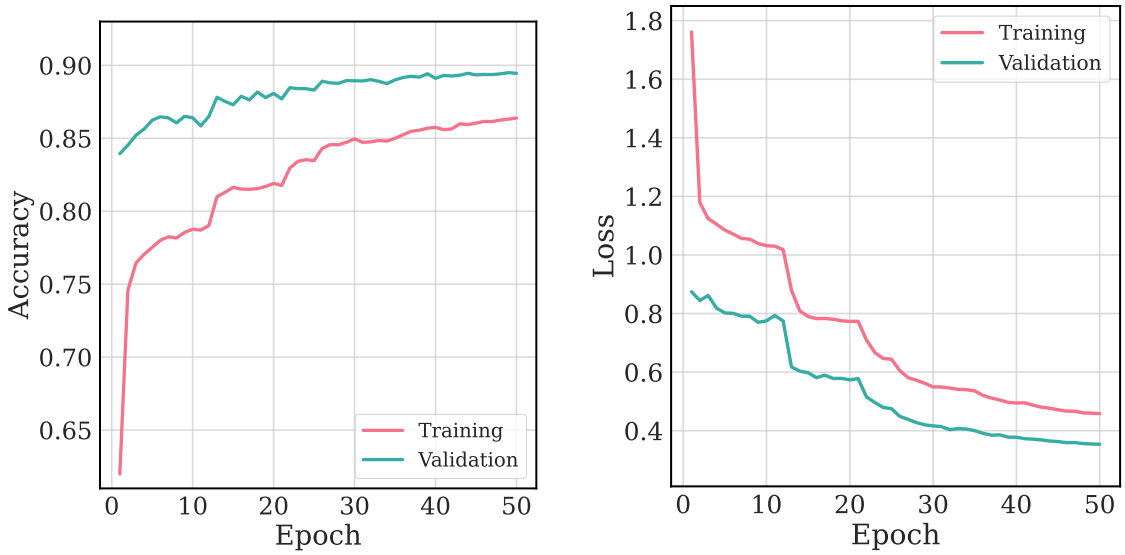


Figure 2: Accuracy and Cross-Entropy Loss respected to the number of Epochs from CNN

4.3 Performance Analysis

Table 4 presents a comprehensive quantitative comparison of the metrics for all three approaches. All experiments were conducted on the identical hardware configuration to ensure a fair assessment.

In terms of predictive capability, the hierarchy of model performance is distinct. The **Random Forest** baseline achieved the lowest test accuracy at **80.69%**, limited by its inability to capture non-linear spatial dependencies efficiently. The **Support Vector Machine**, utilizing the RBF kernel and PCA features, significantly improved upon this baseline, reaching **85.13%** accuracy on the unseen test set. This demonstrates the superiority of kernel methods in handling the complex topology of handwriting data compared to orthogonal decision trees. However, the **Convolutional Neural Network** remained the superior solution, achieving **89.39%** accuracy. The CNN's advantage of **4.26%** over the optimized SVM highlights the necessity of learnable spatial filters over static feature projection methods.

However, this accuracy hierarchy is inverted when considering deployment efficiency. The **Random Forest** model exhibited remarkable speed during the inference phase, processing the entire batch of 18,800 test images in just **0.84 seconds**, approximately 0.04 ms per sample. In contrast, the **SVM was computationally expensive at inference phase**, requiring **94.77 seconds** to process the same batch, or 5.04 ms per sample, which makes it over **100 times slower** than the Random Forest due to the cost of computing kernel distances for every support vector. The CNN struck a middle ground, requiring **2.20 seconds** for all test set, or 0.12 ms per sample, offering the best balance of high accuracy and reasonable real-time performance.

To provide a clear view of assessment to the CNN model's performance on unseen data, we randomly selected a batch of 25 samples from the test set for visualization, as presented in Figure 3. This image displays the raw handwritten inputs alongside their ground truth labels, the model's predicted class, and the associated confidence score from the CNN model. In that image, green border indicate correct classifications, while red border highlight error cases.

Model	Val Accuracy	Test Accuracy	Train Time	IT	TPS
Random Forest	80.40%	80.69%	29.47 min	**0.84 s**	**0.04 ms**
SVM	85.10%	85.13%	**10.2 min**	94.77 s	5.04 ms
CNN	**89.50%**	**89.39%**	34.64 min	2.20 s	0.12 ms

Table 4: Final Accuracy Metrics Comparison. We denoted that TPS is Time Per Sample in millisecond unit (ms), and IT is Inference Time on the Test set in second unit (s)

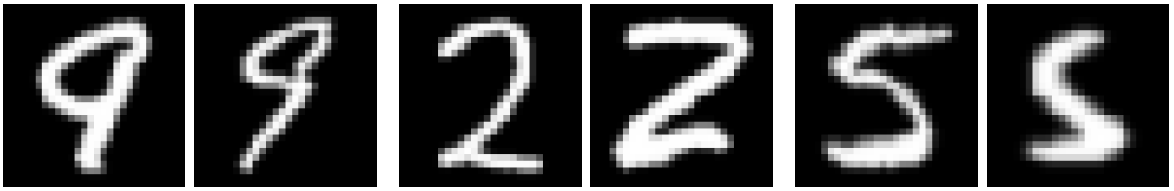
4.4 Error Analysis and Related Discussion

Despite the high performance of the CNN, neither model achieved perfect accuracy. Analysis of the misclassification logs revealed that the primary source of error for both models was **Class Ambiguity**, specifically arising from inter-class homology, which means that characters those are topologically distinct in theory but geometrically identical in many handwriting styles.

Figure 3: The predictions of CNN on 20 randomly picked Samples from Test set

Inter-class Ambiguity. While the EMNIST Balanced dataset explicitly merges structurally similar uppercase and lowercase pairs by the author (for example, 'C' and 'c', 'O' and 'o') to

mitigate case-based confusion, it retains distinct classes that share significant visual overlap. Our testing identified the confusion between **'q' (lowercase)**, in figure 4a, and **'9' (digit)**, in figure 4b, as the most frequent error source for the CNN. For example, you can see that the high variance in handwriting where the tail of a 'q' is often written straight, which means rendering it is indistinguishable from a '9' without broader context. Similarly, the models frequently struggled to differentiate **'2' (digit)**, in figure 4c, from **'Z' (letter)**, particularly when the 'Z' lacks a crossbar as in 4d, since both share a "zig-zag" topological structure. Another persistent source of ambiguity was the confusion between **'S'**, in figure 4f, and **'5'**, in figure 4e, which share a serpentine curve, and the vertical-line homoglyphs **'I', 'l', and '1'**, which often lack distinct serifs or terminators in rapid handwriting. For better visualization, we can look closer to the error cases in Figure 3. For instance, the first sample in the top-left corner (the 'V' sample) was misclassified as 'U' with 54% confidence. Similarly, the sample in the second row, specifically the 'Y' sample, was incorrectly predicted as '4' at 88% confidence, because the handwriting features a disjointed top stroke that mimics the crossbar structure of a digit '4'. These visual examples highlight that many reported "errors" are not failures of feature extraction, but rather inherent ambiguities where the isolated character lacks sufficient context for a definitive label.



(a) '9' (b) 'q' (c) '2' (d) 'Z' (e) '5' (f) 'S'

Figure 4: Inter-class ambiguity example visualization on 6 specific characters

Deep Learning versus Classical Machine Learning. The results highlight a clear hierarchy in handling the ambiguities of handwritten text. While the Convolutional Neural Network achieved the highest accuracy, of 89.39%, the gap between Deep Learning and Classical methods narrowed significantly with the introduction of the Support Vector Machine. The Random Forest, limited by its reliance on orthogonal splits of raw pixel intensities, struggled with slight rotations and shifts, achieving only 80.69%. The SVM, utilizing the RBF kernel and PCA features, successfully captured more complex non-linear topologies, boosting accuracy to 85.13%. However, a performance gap of approximately **4.26%** remains between the best classical model and the CNN. This deficit stems from the fundamental difference in feature extraction: while SVM and Random Forest rely on static feature projections or holistic distance metrics, the CNN leverages hierarchical, learnable filters. This allows the CNN to identify abstract geometric primitives, such as the loop of a '9' vs the stroke of a 'q', with a degree of spatial invariance that classical kernel methods cannot match.

Trade-offs in Classical Baselines. The comparison between the two classical approaches offers critical insights into the accuracy-efficiency trade-off. Although the Random Forest served as the weakest predictor, it proved to be the most computationally efficient, offering nearly instantaneous inference times, at approximately 0.04 ms per sample. In contrast, while the SVM provided a respectable boost in accuracy, it incurred a massive computational cost. Due to the complexity of the kernel matrix, the SVM was over **100 times slower** during inference, or 5.04 ms per sample, than the Random Forest and significantly slower than the one of CNN with 0.12 ms. This means that while SVMs can compete on accuracy for smaller datasets, they become computationally prohibitive for large-scale real-time Handwritten Character Recognition tasks

compared to the optimized forward flow of a Neural Network or the simple decision paths of a Random Forest.

5 Advance Enhancement on Machine Learning Approach

While the previous experiments established a baseline using raw pixel intensities, classical algorithms often struggle because raw pixels lack semantic information about shape and structure. To address this limitation and attempt to bridge the performance gap between Classical Machine Learning and Deep Learning, we adapted an advanced feature engineering step, the **Histogram of Oriented Gradients (HOG)** technique, which can be found at *[7]*.

5.1 Methodology of Histogram of Oriented Gradients

Motivation for HOG Implementation. Our decision to integrate Histogram of Oriented Gradients (HOG) stems from the inherent limitations of raw pixel representation in character recognition. When a classifier relies solely on raw pixel intensities, it is highly sensitive to **spatial translations** and **illumination variances**, for example, shifting a character by merely a few pixels changes the feature vector entirely, often leading to misclassification. HOG overcomes this by abstracting the image into a distribution of local intensity gradients, or edges, and orientations. This approach mimics human visual perception, which identifies characters by their stroke geometry, for instance, the curvature of a 'C' or the intersection of a 'T', rather than specific pixel values. By explicitly encoding the **direction** and **magnitude** of these strokes, HOG provides a robust feature set that is invariant to common handwriting deformations, such as slight rotations, varying stroke thickness, or changes in pen pressure. This transformation effectively simplifies the decision boundary for classical classifiers, allowing them to focus on the structural essence of the character rather than similar alignment along the images. Below here, we will configure the HOG descriptor with parameters specifically optimized for the 28×28 EMNIST images.

Gradient Computation. First of all, for every pixel, the magnitude and orientation of the gradient are calculated. This step is fundamental as it isolates the presence of edges, or strokes, and their specific directions, specifically vertical, horizontal, or diagonal, disregarding flat regions of the image. Since the input image is 28×28, the gradient calculation produces two matrices, specifically magnitude and orientation, of the same size of $28 \times 28 = 784$ pixel gradients

Cell Division of 4×4 pixels. The image is spatially partitioned into small connected regions called cells. We selected a fine-grained cell size of 4×4 pixels. This high resolution is crucial for character recognition, since standard configurations like 8×8 pixels would be too coarse, which will blur out minute structural details such as the loop in 'e' or the tail of 'a'. By that, we divide the 28×28 image by the 4×4 cell size, which results in a grid of 49 cells, since each side is 7 cells come from the side 28 pixels divided to 4 pixel in each cell's side.

Orientation Binning of 9 bins. Within each cell, the algorithm compiles a histogram of gradient directions distributed into 9 bins, spanning from 0 to 180 degrees. This mechanism effectively allows pixels to find their dominant structural element in that local region, creating a robust signature of the stroke orientation. Therefore, each of the 49 cells now contains a 9-value histogram.

Block Normalization. Finally, the cells are aggregated into larger 2×2 blocks to perform local contrast normalization. These blocks traverse the grid with a stride of 1 cell, employing the old day technique of sliding window mechanism to do the convolution operations similar to the modern day Deep Learning. This specific design choice is historically significant, since before the advent of end-to-end learning in CNNs, this sliding block technique was the gold standard in computer vision, famously introduced by Dalal & Triggs for pedestrian detection in [**?**], for achieving spatial invariance. By normalizing intensity within overlapping local neighborhoods, this step renders the extracted features robust to variations in illumination and stroke thickness, which is critical for distinguishing between bold and faint handwriting.

To quantify the resulting feature space, we observe that sliding a 2×2 window over a 7×7 cell grid without padding will yield 6 blocks per row and column, totaling 36 blocks per image. Each block encapsulates 4 cells, with each cell contributing a 9-bin histogram, resulting in 36 values per block. Consequently, the final feature vector expands to 36 blocks $\times$ 36 values $= \mathbf{1,296}$ features. This unfolding of the data from 784 raw pixels into a higher-dimensional representation provide it a complex spatial relationships, thereby enabling the classifiers to construct more accurate decision boundaries. However, for the SVM model, this **increased dimensionality poses a huge computational challenge**. Therefore, we apply PCA just like our baseline above to project these 1,296 HOG features down to 50 principal components, preserving the most discriminative variances while ensuring efficient kernel computation.

5.2 Difference between HOG and Convolutional Layers of CNN

There is a similarity between our applied HOG feature extraction and the initial layers of the Convolutional Neural Network, as both mechanisms share the fundamental goal of identifying local spatial structures through sliding windows. However, a critical divergence exists in the origin and adaptability of their feature detectors. In our classical approach, the HOG descriptor functions as a static, or **hand-crafted** feature extractor where the "filters", specifically the gradient magnitude calculations and 9-bin orientation histograms, are mathematically predetermined by human design. By enforcing this rigid framework, we explicitly assume that edge orientations are the definitive features for recognizing handwritten characters, effectively hard-coding the model's focus onto stroke directionality. In sharp contrast, the Convolutional layers in our Deep Learning model are initialized with **random weights** and possess the plasticity to **learn optimal feature** representations directly from the data during backpropagation. While the CNN often autonomously discovers edge detectors similar to HOG in its shallow layers, it retains the flexibility to evolve more complex, data-driven filters, such as those sensitive to loop closures, curvature continuity, or specific stroke intersections that a fixed algorithm might fail to capture. Consequently, while HOG successfully injects necessary spatial awareness into the SVM and Random Forest models, it remains bounded by its static definition, whereas the CNN's adaptive feature extraction allows it to uncover the latent, non-linear hierarchies inherent in the EMNIST dataset.

5.3 Experiments with HOG Technique

We applied this enhanced pipeline to our best-performing classical configurations, which are the **Random Forest (200 trees)** and the **SVM (RBF Kernel + PCA)**. However, we want to emphasize that **if the model *X* perform better than model *Y*, it completely does not mean that *X with HOG*, or $\mathbf{X_{HOG}}$, will perform better than *Y + HOG*, or $\mathbf{Y_{HOG}}$, currently, at here we just apply the HOG to prove that we have a way of improvement so that the model *X with HOG* can learn features from the raw data better than *X* only itself**. Maybe in the future, we will continue developing the project by apply HOG with **Grid Search Cross-Validation** to find the best combination of parameters.

The results, summarized in Table 5, demonstrate a clear improvement in predictive capability. As you can see, both classical models benefited from the spatial structural information provided by HOG feature extraction. The Random Forest saw a significant boost of **+2.08%**, rising to 82.77% on test set. This confirms that the orthogonality problem of decision trees on raw flatten pixels was partially mitigated by providing explicit edge features. The SVM also improved by **+1.67%**, reaching 86.80%, narrowing the gap with the CNN to just 2.59%. In a nut shell, while HOG successfully pushes the limits of classical machine learning, demonstrating that feature engineering can rival Deep Learning accuracy in specific contexts, the CNN remains the superior solution in accuracy (89.39%), as it integrates feature extraction and classification into a single and end-to-end optimized pipeline.

Model	**Input Type**	**Test Accuracy**	**Improvement**
Random Forest (200 trees)	Flatten Pixel	80.69%	–
	HOG Features	**82.77%**	**+2.08%**
SVM (RBF kernel)	Flatten Pixel + PCA	85.13%	–
	HOG Features + PCA	**86.80%**	**+1.67%**
CNN	Non-flatten Pixel	**89.39%**	–

Table 5: Impact of HOG Feature Extraction on Classical Models compared to itself and CNN also. As we have emphasized, the performance observed on raw data does not necessarily persist after feature engineering. The fact that Model X outperforms Model Y on raw pixels does not imply that X_{HOG} will strictly outperform Y_{HOG}. We implement HOG not to re-evaluate inter-model rankings but intra-model improvement, which demonstrates that gradient-based feature extraction enhances the representational capacity of a specific model relative to its own baseline.

6 Conclusion and Future Work

This comparative study comprehensively evaluated the performance of Classical Machine Learning versus Deep Learning for handwritten character recognition using the EMNIST Balanced dataset. Our experiments have revealed a distinct hierarchy in the trade-off between predictive accuracy and computational efficiency. Firstly, the **Random Forest** classifier served as an ultra-low latency baseline, with 0.04 ms inference time, but was limited by its reliance on raw pixel intensities, achieving a baseline accuracy of 80.69%. About the **Support Vector Machine**, utilizing an RBF kernel and PCA dimensionality reduction, successfully captured non-linear spatial topologies, improving accuracy to 85.13%, albeit at a significant computational cost and slower inference. Furthermore, our enhancement via **Histogram of Oriented Gradients** demonstrated that classical models are not obsolete. By explicitly engineering gradient-based features, we boosted the SVM's performance to 86.80%, effectively narrowing the gap with Deep Learning. However, our simple custom **Convolutional Neural Network** ultimately remained the superior solution, achieving **89.39%** accuracy with a highly efficient inference time (0.12 ms), proving that end-to-end learning of hierarchical spatial filters is more effective than static feature projection.

Future development of our work will diverge into the advancements in classical approaches and in deep learning integration to the system. Firstly, in terms of advancements in classical approaches, to address the computational bottlenecks encountered with the non-linear SVM, with the future work, we will explore **Kernel Approximation methods**, for example, Nystrom method or Random Fourier Features to make an approximation of the RBF kernel's high-dimensional mapping using linear solvers, potentially reducing the training and inference com-

plexity. We also plan to evaluate alternative feature descriptors beyond HOG, such as **Scale-Invariant Feature Transform**, as known as SIFT, which can be read at *[11]*, to better capture stroke curvature variations. Secondly, in terms of advancements in deep learning and system integration, we aim to resolve persisting class ambiguities, especially, 'q' vs '9', '1' vs 'l', by moving beyond isolated character recognition. We plan to integrate **context-aware mechanisms**, such as **n-gram language models** or **Recurrent Neural Networks**, as known as RNNs accessed at *[8]*, which can disambiguate characters based on word-level probability, which require us a transition to word-based datasets.

7 Code Availability

In this report, we will only display the main source code for the Model Structure of CNN, SVM and Random Forest, which have been represented at section 7. The complete source code for this project, including data preprocessing, feature extraction, dimension reduction, model definitions, training scripts and plug-and-play inference scripts, is available at:
`https://github.com/tanlytran-hcmut-comsci/LA-MachineLearning-CO3117`

References

[1] Breiman, L.: Random forests. Machine learning (2001)

[2] Cao, K., Wei, C., Gaidon, A., Arechiga, N., Ma, T.: Learning imbalanced datasets with label-distribution-aware margin loss. In: Advances in Neural Information Processing Systems (NeurIPS) (2019)

[3] Charte, F., Rivera, A.J., del Jesus, M.J., Herrera, F.: Addressing imbalance in multilabel classification: Measures and random resampling algorithms. Neurocomputing (2015)

[4] Cohen, G., Afshar, S., Tapson, J., van Schaik, A.: Emnist: Extending mnist to handwritten letters. 2017 International Joint Conference on Neural Networks (IJCNN) (2017), `https://arxiv.org/abs/1702.05373`

[5] Cortes, C., Vapnik, V.: Support-vector networks. Machine learning (1995)

[6] Cover, T., Hart, P.: Nearest neighbor pattern classification. IEEE transactions on information theory (1967)

[7] Dalal, N., Triggs, B.: Histograms of oriented gradients for human detection. In: 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05). IEEE (2005)

[8] Elman, J.L.: Finding structure in time. Cognitive Science (1990)

[9] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)

[10] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE (1998)

[11] Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision (2004)

[12] Pearson, K.: On lines and planes of closest fit to systems of points in space. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science (1901)

Sourcecode for CNN model and Random Forest model

This appendix presents the core implementation details for both the Convolutional Neural Network and Random Forest classifiers.

Note: *To ensure the presentation remains concise and maintains a high degree of readability, the source code provided below has been streamlined. We have omitted auxiliary helper functions, redundant boilerplate, and extensive explanatory comments that do not directly pertain to the model architectures or the primary training logic. This format, as in in Listings 1, 2 and 3, allows the reader to focus exclusively on the critical structural definitions and algorithmic components that underpin the experimental results discussed in this study. Specifically, long lines have been wrapped to adhere to the page margins, preventing typographical overflow that would disrupt visual flow. Furthermore, many of my comments explaining standard deep learning concepts (for example, the definition of Batch Normalization or Regularization) were removed, assuming the reader's familiarity with fundamental neural network operations. Instead, we introduced structural annotations (as you can see below, as "# Block 1", "# Block 2") to explicitly delineate the architectural hierarchy, allowing for a clearer, high-level understanding of the model's design without the distraction of verbose implementation details.*

Source Code of Random Forest Model Implementation

```
import numpy as np
import joblib
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.base import BaseEstimator, ClassifierMixin
from skimage.feature import hog
import os
import time

class RandomForestOCR(BaseEstimator, ClassifierMixin):
    ### Compatible with sklearn's GridSearchCV for hyperparameter optimization.

    def __init__(self, n_estimators=200, max_depth=30, min_samples_split=10,
                 min_samples_leaf=2, max_features='sqrt', class_weight=None,
                 n_jobs=-1, use_scaler=False, use_hog=True, n_components=None):
        """
        Initialize Random Forest with optimized hyperparameters for EMNIST
        """
        # Store all parameters as instance attributes (required by sklearn's
    BaseEstimator)
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf
        self.max_features = max_features
        self.class_weight = class_weight
        self.n_jobs = n_jobs
        self.use_scaler = use_scaler
        self.use_hog = use_hog
        self.n_components = n_components

        # HOG parameters for 28x28 images
        self.hog_orientations = 9
        self.hog_pixels_per_cell = (4, 4)
        self.hog_cells_per_block = (2, 2)

        # Don't create model here - create it in fit() so GridSearchCV can update params
        self.model = None
        self.scaler = StandardScaler() if self.use_scaler else None
```

```
        self.pca = None  # Initialize PCA in fit() to ensure GridSearchCV parameter changes work
        self.is_trained = False
        self.training_stats = {}
        self.feature_importance = None

    def _extract_hog_features(self, X):
        """Extract HOG features from images"""
        # Ensure X is in correct shape (N, 28, 28) or (N, 28, 28, 1)
        if len(X.shape) == 4:
            X = X.squeeze(-1)  # Remove channel dimension
        elif len(X.shape) == 2:
            # Already flattened, reshape to images
            X = X.reshape(-1, 28, 28)

        hog_features = []
        for img in X:
            # Extract HOG features
            features = hog(
                img,
                orientations=self.hog_orientations,
                pixels_per_cell=self.hog_pixels_per_cell,
                cells_per_block=self.hog_cells_per_block,
                block_norm='L2-Hys',
                visualize=False,
                feature_vector=True
            )
            hog_features.append(features)

        return np.array(hog_features)

    def _flatten(self, X):
        # Converts (N, 28, 28, 1) to (N, 784) or extracts HOG features
        if self.use_hog:
            return self._extract_hog_features(X)
        else:
            # Use raw pixels
            if len(X.shape) == 2:
                return X
            return X.reshape(X.shape[0], -1)

    def _prepare_labels(self, y):
        # Converts One-Hot (N, 47) -> Integers (N,) since sklearn prefers
        # class indices but not one-hot vectors.
        if len(y.shape) > 1 and y.shape[1] > 1:
            return np.argmax(y, axis=1)
        return y

    def fit(self, X, y):
        ### Fit method compatible with sklearn's API (used by GridSearchCV)
        # Create model with current parameters (allows GridSearchCV to update params)
        self.model = RandomForestClassifier(
            n_estimators=self.n_estimators,
            max_depth=self.max_depth,
            min_samples_split=self.min_samples_split,
            min_samples_leaf=self.min_samples_leaf,
            max_features=self.max_features,
            n_jobs=self.n_jobs,
            random_state=42,
            verbose=1,
            class_weight=self.class_weight
        )

        X_flat = self._flatten(X)
        y_int = self._prepare_labels(y)

        # Apply PCA if enabled
        if self.n_components is not None:
            self.pca = PCA(n_components=self.n_components)
            X_flat = self.pca.fit_transform(X_flat)
        else:
            self.pca = None
```

```
112
113         if self.scaler:
114             X_flat = self.scaler.fit_transform(X_flat)
115
116         self.model.fit(X_flat, y_int)
117         self.is_trained = True
118         self.feature_importance = self.model.feature_importances_
119
120         return self  # sklearn convention: fit() returns self
121
122     def train(self, X_train, y_train, X_val=None, y_val=None):
123         # Create model with current parameters
124         self.model = RandomForestClassifier(
125             n_estimators=self.n_estimators,
126             max_depth=self.max_depth,
127             min_samples_split=self.min_samples_split,
128             min_samples_leaf=self.min_samples_leaf,
129             max_features=self.max_features,
130             n_jobs=self.n_jobs,
131             random_state=42,
132             verbose=1,
133             class_weight=self.class_weight
134         )
135
136         feature_type = "HOG features" if self.use_hog else "raw pixels"
137         print(f"Preprocessing data: (Extracting {feature_type})")
138         X_flat = self._flatten(X_train)
139         y_int = self._prepare_labels(y_train)
140
141         # Apply PCA if enabled
142         if self.n_components is not None:
143             self.pca = PCA(n_components=self.n_components)
144             X_flat = self.pca.fit_transform(X_flat)
145         else:
146             self.pca = None
147
148         # Apply scaling if enabled
149         if self.scaler:
150             X_flat = self.scaler.fit_transform(X_flat)
151
152         # Train model
153         print(f"Training {self.model.n_estimators} decision trees.")
154         start_time = time.time()
155         self.model.fit(X_flat, y_int)
156         train_time = time.time() - start_time
157         self.is_trained = True
158
159         # Calculate feature importance
160         self.feature_importance = self.model.feature_importances_
161
162         # Evaluate on training set
163         train_pred = self.model.predict(X_flat)
164         train_acc = accuracy_score(y_int, train_pred)
165
166         # Store training statistics
167         self.training_stats = {
168             'train_accuracy': train_acc,
169             'train_time': train_time,
170             'n_estimators': self.model.n_estimators,
171             'max_depth': self.model.max_depth,
172             'n_samples': X_flat.shape[0],
173             'n_features': X_flat.shape[1]
174         }
175
176         # Validation evaluation
177         if X_val is not None and y_val is not None:
178             val_acc = self.evaluate(X_val, y_val)
179             self.training_stats['val_accuracy'] = val_acc
180             print(f"Validation Accuracy: {val_acc*100:.2f}%")
181
182     def evaluate(self, X_test, y_test):
183         """Evaluate model accuracy on test set"""
```

```
        X_flat = self._flatten(X_test)
        y_int = self._prepare_labels(y_test)

        if self.pca is not None:
            X_flat = self.pca.transform(X_flat)
        if self.scaler:
            X_flat = self.scaler.transform(X_flat)

        predictions = self.model.predict(X_flat)
        return accuracy_score(y_int, predictions)

    def predict(self, X):
        ### Returns class predictions (sklearn convention)
        if not self.is_trained: raise ValueError("Model is not trained =)))")
        X_flat = self._flatten(X)
        if self.pca is not None: X_flat = self.pca.transform(X_flat)
        if self.scaler: X_flat = self.scaler.transform(X_flat)
        return self.model.predict(X_flat)

    def save_model(self, filepath):
        ### Save model, scaler, PCA, and HOG parameters to disk
        model_data = {
            'model': self.model,
            'scaler': self.scaler,
            'pca': self.pca,
            'n_components': self.n_components,
            'feature_importance': self.feature_importance,
            'use_hog': self.use_hog,
            'hog_orientations': self.hog_orientations,
            'hog_pixels_per_cell': self.hog_pixels_per_cell,
            'hog_cells_per_block': self.hog_cells_per_block
        }
        joblib.dump(model_data, filepath)
```

Listing 1: Implementation of the Random Forest Class with HOG and PCA support

Source Code of Support Vector Machine Model Implementation

```
import numpy as np
import joblib
from sklearn.svm import SVC, LinearSVC
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.metrics import accuracy_score
from skimage.feature import hog
import time
import os

class SVMOCR(BaseEstimator, ClassifierMixin):
    def __init__(self, C=1.0, kernel='rbf', gamma='scale', use_hog=True, n_components
    =100):
        """
        SVM Wrapper for OCR with HOG feature extraction
        """
        self.C = C
        self.kernel = kernel
        self.gamma = gamma
        self.use_hog = use_hog
        self.n_components = n_components

        # HOG parameters
        self.hog_orientations = 9
        self.hog_pixels_per_cell = (4, 4)
        self.hog_cells_per_block = (2, 2)

        # PCA will be initialized in fit() to ensure GridSearchCV parameter changes work
        self.pca = None

        # SVM is highly sensitive to scale, so we always include a scaler
```

```
        self.scaler = StandardScaler()
        self.model = None
        self.is_trained = False
        self.training_stats = {}

    def _extract_hog_features(self, X):
        if len(X.shape) == 4:
            X = X.reshape(X.shape[0], 28, 28)
        elif len(X.shape) == 2:
            n_samples = X.shape[0]
            X = X.reshape(n_samples, 28, 28)

        hog_features = []
        for img in X:
            features = hog(
                img,
                orientations=self.hog_orientations,
                pixels_per_cell=self.hog_pixels_per_cell,
                cells_per_block=self.hog_cells_per_block,
                block_norm='L2-Hys',
                transform_sqrt=True,
                feature_vector=True
            )
            hog_features.append(features)
        return np.array(hog_features)

    def _flatten(self, X):
        """Convert images to feature vectors (HOG or raw pixels)"""
        if self.use_hog:
            return self._extract_hog_features(X)
        else:
            # Convert (N, 28, 28, 1) -> (N, 784)
            if len(X.shape) > 2:
                return X.reshape(X.shape[0], -1)
            return X

    def _prepare_labels(self, y):
        """Convert One-Hot (N, 47) -> Integers (N,)"""
        if len(y.shape) > 1 and y.shape[1] > 1:
            return np.argmax(y, axis=1)
        return y

    def fit(self, X, y):
        """Fit method compatible with sklearn's API"""
        print("Initializing SVM model...")
        start_fit_time = time.time()

        # Create SVM model - use LinearSVC for linear kernel (much faster)
        if self.kernel == 'linear':
            print("Using LinearSVC (optimized for linear kernel)...")
            self.model = LinearSVC(
                C=self.C,
                max_iter=2000,
                random_state=42,
                verbose=1,
                dual='gamma'
            )
        else:
            print(f"Using SVC with {self.kernel} kernel...")
            self.model = SVC(
                C=self.C,
                kernel=self.kernel,
                max_iter=2000,
                gamma=self.gamma,
                cache_size=2048,
                decision_function_shape='ovo',
                random_state=42,
                verbose=True
            )

        # Extract features and prepare labels
        X_features = self._flatten(X)
```

```
        y_int = self._prepare_labels(y)

        # Initialize PCA here to ensure GridSearchCV parameter changes work
        if self.n_components is not None:
            self.pca = PCA(n_components=self.n_components)
            X_features = self.pca.fit_transform(X_features)
        else:
            self.pca = None

        # Fit scaler and transform
        X_scaled = self.scaler.fit_transform(X_features)

        # Train SVM
        self.model.fit(X_scaled, y_int)
        self.is_trained = True
        return self

    def train(self, X_train, y_train, X_val=None, y_val=None):
        """Train SVM with optional validation"""
        # Train
        start_time = time.time()
        self.fit(X_train, y_train)
        training_time = time.time() - start_time

        # Training accuracy
        train_preds = self.predict(X_train)
        y_train_int = self._prepare_labels(y_train)
        train_acc = accuracy_score(y_train_int, train_preds) * 100

        # Validation accuracy if provided
        if X_val is not None and y_val is not None:
            val_preds = self.predict(X_val)
            y_val_int = self._prepare_labels(y_val)
            val_acc = accuracy_score(y_val_int, val_preds) * 100

            self.training_stats = {
                'training_time': training_time,
                'train_accuracy': train_acc,
                'val_accuracy': val_acc
            }
        else:
            self.training_stats = {
                'training_time': training_time,
                'train_accuracy': train_acc
            }

        return self.training_stats

    def predict(self, X):
        """Predict class labels"""
        if not self.is_trained: raise ValueError("Model is not trained yet.")
        X_features = self._flatten(X)
        if self.pca is not None:
            X_features = self.pca.transform(X_features)
        X_scaled = self.scaler.transform(X_features)
        return self.model.predict(X_scaled)

    def save_model(self, filepath):
        """Save model with HOG configuration"""
        if not self.is_trained: raise ValueError("Cannot save untrained model.")
        joblib.dump({
            'model': self.model,
            'scaler': self.scaler,
            'pca': self.pca,
            'n_components': self.n_components,
            'use_hog': self.use_hog,
            'hog_orientations': self.hog_orientations,
            'hog_pixels_per_cell': self.hog_pixels_per_cell,
            'hog_cells_per_block': self.hog_cells_per_block,
            'training_stats': self.training_stats
        }, filepath)
```

Listing 2: Implementation of the SVM Class with HOG and PCA

Source Code for Convolutional Neural Network Model Implementation

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
import numpy as np

class OCRModel:
    def __init__(self, num_classes, input_shape=(28, 28, 1), learning_rate=0.001):
        self.num_classes = num_classes
        self.input_shape = input_shape
        self.learning_rate = learning_rate
        self.model = None
        self.history = None

    def build_model(self):
        model = Sequential([
            # conv layer
            Conv2D(
                32,                             # number of filters
                (3, 3),                         # kernel_size
                activation='relu',              # activation function.
                kernel_regularizer=l2(0.001),   # lamda: regularization.
                input_shape=self.input_shape,   # input shape
                padding='same'                  # keep the input size
            ),
            BatchNormalization(),
            MaxPooling2D((2, 2)),
            Dropout(0.3),

            # conv layer
            Conv2D(64, (3, 3),
                activation='relu',
                kernel_regularizer=l2(0.001),
                padding='same'
            ),
            BatchNormalization(),
            MaxPooling2D((2, 2)),
            Dropout(0.3),

            # conv: captures high-level features
            Conv2D(128, (3, 3),
                activation='relu',
                padding='same',
                kernel_regularizer=l2(0.001)
            ),
            BatchNormalization(),
            MaxPooling2D((2, 2)),
            Dropout(0.25),

            # fully connected layer: classification
            Flatten(),
            Dense(256,
                activation='relu',
                kernel_regularizer=l2(0.001)
            ),
            BatchNormalization(), # faster convergence
            Dropout(0.5),
            Dense(self.num_classes, activation='softmax')  # Output layer
        ])

```

```
        self.model = model
        return model

    def compile_model(self):
        if self.model is None:
            self.build_model()

        self.model.compile(
            optimizer=Adam(learning_rate=self.learning_rate),
            loss='categorical_crossentropy',
            metrics=['accuracy']
        )

    def summary(self):
        # display model architecture
        if self.model is None:
            self.build_model()
        self.model.summary()

    def train(self, data_generation, X_train, y_train,
                  X_val, y_val, epochs=20, batch_size=32, callbacks=None):
        if self.model is None:
            self.compile_model()

        data_generation.fit(X_train)

        self.history = self.model.fit(
            data_generation.flow(X_train, y_train, batch_size=batch_size),
            epochs=epochs,
            validation_data=(X_val, y_val),
            callbacks=callbacks if callbacks else []
        )
        return self.history

    def save_model(self, filepath):
        if self.model is None:
            raise ValueError("We do not have model to save =)))")
        self.model.save(filepath)
        print(f"Model saved to {filepath}")

    def load_model(self, filepath):
        self.model = keras.models.load_model(filepath)
        print(f"Model loaded from {filepath}")

    def predict(self, X):
        if self.model is None:
            raise ValueError("We do not have model to predict =)))")
        return self.model.predict(X)
```

Listing 3: Implementation of the CNN Architecture using TensorFlow/Keras