# Handwritten Letter Identification with CNNs

Scarlett Dong, Shaash Sivakumar, Sunil Trivedi, and Marc Walden

March 14, 2025

**Abstract**

In this report, we explore the effectiveness of a trained CNN model in identifying handwritten digits. The model was trained using the EMNIST Letters dataset. The model parameters were optimized using an extension of the stochastic gradient descent in the Adam optimizer. The model performed strongly on a test set, achieving 92.79% accuracy. The model also showed the ability to identify different letters with fairly equal proficiency. Moving forward, we believe that we could improve our model by implementing data augmentation and utilizing random search in place of grid search to optimize hyperparameters.

## 1 Introduction

Our project goal is to use the EMNIST Letters data set, which contains 103,600 grayscale, 28x28 pixel images of handwritten English letters, to train a CNN model to identify handwritten letters from images. This is quite practical, as letter recognition is very important in a variety of tasks including but not limited to analyzing handwritten documents (forms, applications, surveys), enabling handwritten input on tablets, and sorting mail in post offices by identifying handwritten addresses. CNNs are innately good at image classification due to local spatial invariance, parameter sharing, translation invariance, feature hierarchy, and robustness and generalization, which makes them perfect for this project. We will begin by pre-processing the EMNIST Letters data set to make it suitable for the CNN. Then, we will build our CNN architecture, including multiple convolutional layers, max-pooling layers, fully connected layers, and a flatten layer. We will train this model using a cross-entropy loss function, ReLU activation functions, and the Adam optimizer. Finally, we will evaluate the effectiveness of the model by calculating accuracy and other performance metrics.

## 2 Background

Our CNN model takes in a dataset where each input image consists of 28 rows and 28 columns of pixels, totaling 784 neurons. From there, the input layer passes through different types of layers—convolutional layers, pooling layers, and fully connected layers. The order of these layers may vary, and some layers may appear multiple times. We shall explore this aspect of our model in the model section. In this section, we will focus on explaining the core purpose of each layer.

#### 2.0.1 Convolutional Layer

In a convolutional layer, a filter (kernel) is applied to small regions (patches) of the input image. The filter slides over the image with a certain stride. At each position, the filter performs scalar multiplication with the corresponding patch of the image and adds all values together, resulting in a single scalar value. For example, assuming the input region has elements $1, 2, 3, 4, 5, 6, 7, 8, 9$ organized into a $3 \times 3$ matrix and the kernel has elements $0, 1, 0, 0, 1, 0, 0, 1, 0$ also organized into a $3 \times 3$ matrix, then the output would be:

$$10 + 21 + 30 + 40 + 51 + 60 + 70 + 81 + 90 = 15.$$

This operation is repeated over the entire image, producing an output matrix (called the feature map) that represents the detected features across the image.

Note the following hyperparameters:

- **Depth**: The depth of a convolutional layer represents the number of filters (kernel matrices) used for each patch of input data. Each filter detects a specific feature in the input image that is learned after. For example, if there are two filters, both are applied to every region of the image, producing two numerical values per region. Each value represents the strength or presence of a particular feature in that specific region.
- **Stride**: With stride $s$, the filter moves $s$ pixels at a time across the image.
- **Padding**: Ensures the filter can cover edge pixels without losing information.

After applying the filter across the entire image, you get a feature map that has reduced dimensions (due to the filter size) compared to the input image. For example, applying a $5 \times 5$ filter to a $32 \times 32$ image results in a $28 \times 28$ feature map (with stride = 1 and no padding). This layer essentially identifies local features like edges, textures, or simple patterns in the image.

### 2.0.2 Pooling Layer

In this layer, the idea is to pool small areas of the feature map with a larger stride (typically stride = 2). From there, we measure either the max or the mean of all the entries in the pooled area. This process is repeated for all pooled areas. This way, we lower the dimension/size of the feature map. The hyperparameter for the pooling layer is simply the size of it.

### 2.0.3 Fully-connected Layer

In a fully connected layer, each neuron is directly linked to all neurons in the previous and next layers, without skipping any connections. This structure is similar to how neurons are arranged in a traditional artificial neural network (ANN). The hyperparameters in the Fully-Connected layer are the number of deep layers and the number of neurons per layer.

For our convolutional neural network, we use the categorical cross-entropy loss function to measure how well the predicted probabilities match the true labels, which is defined as:

$$E = -\sum_{n=1}^{N} y_n \log(p_n)$$

Where:

- $y_n$ is the true label for sample $n$ (an integer indicating the probability predicted of the correct class)
- $p_n$ is the predicted probability for the correct class, which is typically output by the softmax activation function in the final layer and depends on the weights.
- $C = 10$ is the total number of categories (ten digits)

Our goal is to minimize this function via gradient descent. The idea behind this methodology is to move towards a direction in the space of $W$ such that the loss function decreases towards its minimum value. Therefore, from a general standpoint, we update $W$ at each step using the idea shown below:

$$W_{i,j}^{k+1} = W_{i,j}^{k} + \alpha \frac{\partial E}{\partial W_{i,j}^{k}}$$

Where:

- $W_{i,j}^{k}$ and $W_{i,j}^{k+1}$ represent the weights of the current iteration and next iteration respectively.
- $\alpha$ represents the learning rate, which is assigned by hyperparameter tuning.
- $\frac{\partial E}{\partial W_{i,j}^{k}}$ represents the gradient of the loss function with respect to the weights of the current iteration, which can be done via backpropagation (further explanation below).

However, more particularly, we also incorporate momentum and an adaptive learning rate through the Adam optimizer to address some of the common issues with gradient descent, such as getting stuck at less important local optimums. This also incorporates the update of a moving average of the gradients and squared gradients to adjust the learning rate dynamically. The full algorithm that also incorporates adaptive and momentum-based learning can be found on slide 34, Lecture 15, but we omit it for brevity purposes in this report.

By the way that the neural network is designed, note the following relations:

$$a_i^k = \sum_{j=1}^{n_{k-1}} W_{i,j}^{k-1} z_j^k + b_i^{k-1}, \quad z_i^k = h(a_i^k)$$

Where:

- $a_i^k$ represents the linear combination of the $i$th node in the $k$th layer of the weights and bias term.
- $h(x) = \max(0, x)$ for all layers except for the last layer. For the last layer, we use softmax: $h(x) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$, where $C$ is the number of classes.
- $W_{i,j}^{k-1}$ represents the weight parameter of the layer $k-1$ for each $i$ and $j$.
- $b_i^{k-1}$ represents the bias parameter of the layer $k-1$ for neuron $i$.
- $n_{k-1}$ represents the number of neurons in the layer $k-1$.

In order to compute the gradient, we use the following formula:

$$\frac{\partial E_n}{\partial W_{i,j}^k} = \sum_{l=1}^{n_{k-1}} \frac{\partial E_n}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial W_{i,j}^k} = \frac{\partial E_n}{\partial a_i^{k+1}} \frac{\partial a_i^{k+1}}{\partial W_{i,j}^k},$$

where

$$\frac{\partial a_i^{k+1}}{\partial W_{i,j}^k} = z_j^k.$$

We know that for the bias term,

$$\frac{\partial E_n}{\partial b_i^k} = 1$$

so all that remains is to calculate

$$\frac{\partial E_n}{\partial a_i^{k+1}}.$$

Let $L$ be the total number of layers. Then, we can obtain $\frac{\partial E_n}{\partial a_i^{L+1}}$ from the loss function directly and use that to calculate $\frac{\partial E_n}{\partial a_i^L}, \frac{\partial E_n}{\partial a_i^{L-1}}, ..., \frac{\partial E_n}{\partial a_i^1}$ by noting by chain rule that:

$$\frac{\partial E_n}{\partial a_j^{k-1}} = \sum_{i=1}^{n_{k-1}} \frac{\partial E_n}{\partial a_i^k} \frac{\partial a_i^k}{\partial a_i^{k+1}} = \sum_{i=1}^{n_{k-1}} \frac{\partial E_n}{\partial a_i^k} W_{i,j}^{k-1} h'(a_j^{k-1}).$$

Using these equations, we present a clear way to calculate the gradient $\frac{\partial E_n}{\partial W_{i,j}^k}$ and $\frac{\partial E_n}{\partial b_{i,j}^k}$ for all $i$ and $j$ for all layers $k$ in the network. For the purposes of this project, we will use packages from Keras that do all of these operations automatically. See our code for more detail on this.

## 3 Dataset

The dataset used in this project is the EMNIST (Extended MNIST) Letters dataset, which is one of the splits of the EMNIST dataset derived from the NIST Special Database 19 and structured to match the original MNIST dataset [1]. The EMNIST letters dataset contains a balanced set of handwritten English uppercase and lowercase letters and produces 26 classes. The dataset consists of 103,600 samples (88,800 training samples and 14,800 test samples). Each image is represented as a 28x28 grayscale pixel matrix, with each pixel value ranges from 0 to 255. The labels range from 1 to 26, corresponding to the letters 'A' to 'Z' [1].

Before training the CNN, preprocessing steps were taken to ensure model performance. After the dataset was loaded, labels and pixel values were stored as separate arrays, where labels were taken to be the first column of the dataset. Then, normalization was performed such that pixel values, originally ranging from 0 to 255, were scaled to the range [0, 1]. To ensure an upright position in visualization, the orientation of images were fixed by rotating 90 degrees counterclockwise and flipping horizontally. Next, the grayscale pixel matrices were reshaped into a 4D tensor by adding a channel dimension to match the CNN input format, so that each image has a format of 28x28x1. The labels, originally indexed from 1 to 26, were adjusted to 0 to 25 by subtracting 1 from each label. Train-validation split was also performed for hyperparameter tuning, and the training set was split into 80% training and 20% validation. Finally, visualization of sample images was included, and a selection of 5 EMNIST letters was displayed to verify alignment and orientation.

## 4 Model

Our model processes an input of a 28x28 grayscale image through 9 layers:

1. First Convolutional Layer: Depths of 64 and kernel size of (5×5) to extract initial features.
2. Max-pooling Layer: A 2x2 pooling to reduce the size of the retained feature map.
3. Second Convolutional Layer: We experimented with depths of 64 and 128 and kernel sizes of (3×3) and (4×4).
4. Max-Pooling Layer: A 2x2 pooling to reduce the size of the retained feature map.
5. Third Convolutional Layer: We experimented with depths of 128 and 256 and used kernel size of (3×3).
6. Flatten Layer: Convert the 2D feature maps into a 1D vector to pass into fully connected layers.
7. Fully Connected Layers:
   (a) A 128-unit dense layer with ReLU activation.
   (b) Dropout Layer: A 0.3 dropout rate is applied to reduce overfitting.
   (c) Output Layer: A softmax layer with 26 units, representing letters A-Z.

.

## 5 Methodology

We used the ReLU (Rectified Linear Unit) activation function in all convolutional layers. ReLU helps avoid the vanishing gradient problem and allows fast convergence. The model was trained using the Sparse Categorical Cross Entropy (where the labels are integer-encoded rather than one-hot encoded), the Adam optimizer, 10 epochs, and a Mini-Batch Stochastic Gradient Descent (SGD) with a batch size of 64. To ensure fair model evaluation and avoid data leakage, we strictly separated training, validation, and test sets and used validation accuracy to monitor overfitting.

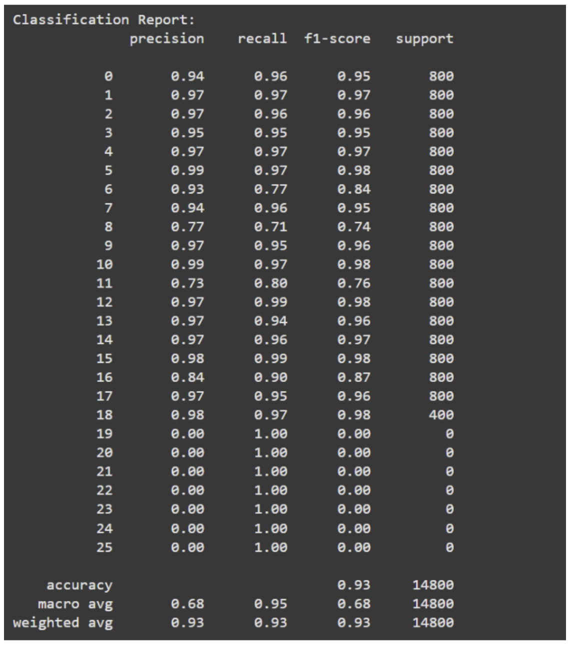
In the training process, we performed hyperparameter tuning process using grid search, iterating over different combinations of:

- Filter Depths: {64}, {64, 128}, {128, 256}
- Kernel Sizes: {5×5}, {3×3, 4×4}, {3×3, 4×4}

We found the hyperparameters that produced the best validation accuracy to be depths of 64 for the First Convolutional Layer, 128 for the Second Convolutional Layer, and 256 for the Third Convolutional Layer, with best Kernel Sizes of 5x5, 3x3, and 4x4, respectively. In total, this model had 2,209,178 trainable parameters

## 6 Results

We trained our own custom CNN evaluated on the EMNIST Letters dataset, which contains handwritten letters. Our goal was to accurately label a new, unseen letter as one of the 26 letters of the alphabet after training the model. The final evaluation of this model was performed on the unseen test set, and we achieved a high test accuracy of 92.79% (Training accuracy: 0.9602 - Training loss: 0.0992 - Validation accuracy: 0.9404 - Validation loss: 0.2094). This indicates our model's strong overall performance in recognizing and classifying handwritten letters. To evaluate the model's performance across all classes, precision, recall, and F1-scores were computed for each letter class (Figure 1). Confusion Matrix Heatmap was generated, where diagonal cells represent the number of correctly classified samples for every letter (Figure 2). The model showed balanced performance across most classes, indicating that it is capable of identifying different letters with relatively equal proficiency.



```
Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.96      0.95       800
           1       0.97      0.97      0.97       800
           2       0.97      0.96      0.96       800
           3       0.95      0.95      0.95       800
           4       0.97      0.97      0.97       800
           5       0.99      0.97      0.98       800
           6       0.93      0.77      0.84       800
           7       0.94      0.96      0.95       800
           8       0.77      0.71      0.74       800
           9       0.97      0.95      0.96       800
          10       0.99      0.97      0.98       800
          11       0.73      0.80      0.76       800
          12       0.97      0.99      0.98       800
          13       0.97      0.94      0.96       800
          14       0.97      0.96      0.97       800
          15       0.98      0.99      0.98       800
          16       0.84      0.90      0.87       800
          17       0.97      0.95      0.96       800
          18       0.98      0.97      0.98       400
          19       0.00      1.00      0.00         0
          20       0.00      1.00      0.00         0
          21       0.00      1.00      0.00         0
          22       0.00      1.00      0.00         0
          23       0.00      1.00      0.00         0
          24       0.00      1.00      0.00         0
          25       0.00      1.00      0.00         0

    accuracy                           0.93     14800
   macro avg       0.68      0.95      0.68     14800
weighted avg       0.93      0.93      0.93     14800
```

Figure 1: Classification report - precision, recall, F1-score per class.

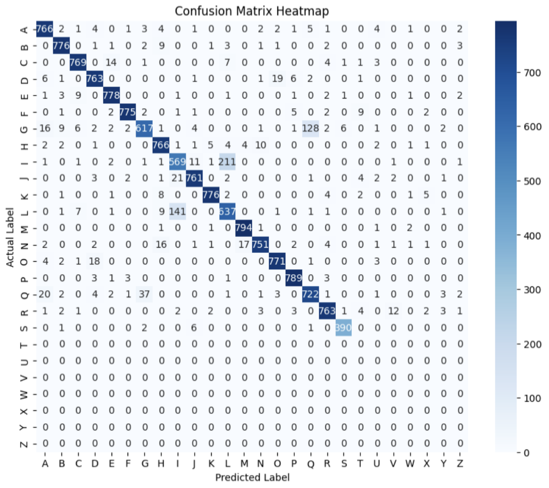


Figure 2: Confusion matrix heatmap.

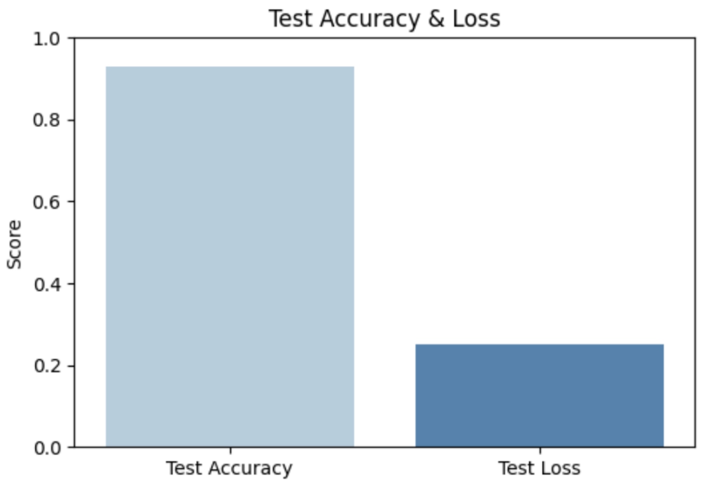
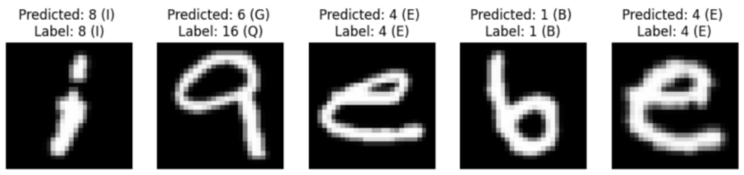


Figure 3: Accuracy and loss of model on testing data set.

The model's high accuracy was consistent across different folds in the cross-validation process, indicating that the model generalizes well to unseen data. Due to limited computational abilities, we weren't able to try out many different combinations of hyperparameters. However, we are satisfied with the results and proud of our model design. Its robustness suggests that the architecture, including the choice of convolutional layers and the fine-tuning of hyperparameters, is well-suited for the EMNIST Letters dataset. One important insight that we noticed when analyzing our results is that, in many cases, the incorrect predictions made by the algorithm were ambiguous for us to interpret. For example, this prediction of the letter ‘q’ seen below as the letter ‘g’ could potentially be a mistake made by ordinary humans. This suggests that our model produces predictions that are aligned with human intuition in many instances, and that it is able to generalize its decision-making process in a way that mirrors human reasoning.



## 7 Conclusion

The overall goal of our project was to train a CNN model to identify handwritten letters from images, a task that is applicable in many real-world scenarios. We formulated a custom model architecture that proved successful in achieving our goal. Our model was able to achieve a high accuracy rate when applied to a test set and demonstrated its robustness and ability to generalize to unseen data in its consistent accuracy across all classes. We believe that our model satisfies all project requirements, but we do acknowledge that some improvements could be made with more time and resources, such as improved model accuracy by performing data augmentation and improved model efficiency by utilizing random search in place of grid search to search for the best hyperparameters.

## 8 Author Contributions

1. Scarlett Dong: Proposal formulation, model architecture, dataset, data preprocessing, main model coder.
2. Shaash Sivakumar: Proposal formulation, model architecture, background, model/methodology, main model coder.
3. Sunil Trivedi: Proposal formulation, model architecture, abstract/introduction, conclusion, report compilation, model code help.
4. Marc Walden: Proposal formulation, model architecture, background, model/methodology, results, model code help.

## 9 Acknowledgements

We would like to acknowledge Professor Tingwei Meng, TA Xinzhe Zuo, and classmate Jason Liu for the advice and help they provided us during the completion of our project.

## 10 GitHub

`https://github.com/shaashwathsivakumar/Math156Final`

## References

[1] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre van Schaik. EMNIST: Extending MNIST to handwritten letters. Data retrieved from `https://www.kaggle.com/datasets/crawford/emnist/data?select=emnist-letters-test.csv`, 2017. arXiv preprint arXiv:1702.05373.

[2] R. Dixit, R. Kushwah, and S. Pashine. Handwritten digit recognition using machine and deep learning algorithms. *arXiv preprint arXiv:2106.12614*, 2021.

[3] K. O'Shea and R. Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[4] B. Siddhesh. Lenet-5 architecture explained. `https://medium.com/@siddheshb008/lenet-5-architecture-explained-3b559cb2d52b`, 2020. Medium.

[5] TensorFlow. Mnist: Handwritten digit classification. `https://www.tensorflow.org/datasets/catalog/mnist`, 2021. TensorFlow Datasets.