**Handwritten Character Recognition**

Project Submission Report

Antonio Chedid, Paul El Kouba, Elio Saade

**Abstract**—In the following report, we present a detailed documentation of our project. To begin with, we will introduce the importance of the following project and introduce the functionalities of our new note taking mobile application. To continue, we will elaborate on the process of building the machine learning classifier that detects the handwritten characters starting from the pre-processing and the features extraction, moving on to the different models implemented and the final model choice based on some metrics computed, and concluding with a brief discussion about the integration between the model and the User Interface (UI) developed using React-Native.

**Index Terms**— Handwritten, Non-parametric, Machine Learning, PEAS, React-Native

—————————— ◆ —————————

Contents

[1 Introduction 1](#_Toc197440791)

[2 PEAS 2](#_Toc1061107611)

[3Pre-processing 2](#_Toc1991857508)

[3.1 Bounding Box 2](#_Toc2048914642)

[3.2 Feature Extraction 2](#_Toc332185414)

[4Machine Learning Models 3](#_Toc1369603003)

[4.1 Random Forest 3](#_Toc1526244151)

[4.2 Neural Network: Developed by Us 3](#_Toc688761940)

[4.3 Neural Network 4](#_Toc483826043)

[4.4 CNN 4](#_Toc84969489)

[4.5 Transfer Learning 5](#_Toc1113971980)

[5 Deployment 6](#_Toc333080007)

[6 Conclusion 6](#_Toc1895295860)

[7 References 6](#_Toc198085765)

[8 Appendix 7](#_Toc151259489)

# Introduction

The rapid digitization of business processes led to a growing interest in Optical Character Recognition (OCR) technologies to reduce manual tasks and labor costs. A crucial component of OCR is Handwritten Character Recognition (HCR), which can be defined as the computer’s ability to take and interpret handwritten character images and to change them into a machine-readable format for further processing.

This paper presents the improved version of our note-taking mobile application, which allows the user to take notes by typing on the keyboard or by writing characters on a canvas. The application embeds a supervised, parametric, HCR agent, which reads the canvas and converts it to written characters in the notes. The paper also highlights the scientific process followed in order to build and compare different learning agents.

# 2 PEAS

We can define the PEAS for our handwritten character recognizer as follows:

* **Performance** measure: percentage of successfully labelled handwritten characters
* **Environment**: mobile application
* **Actuator**: display of the label of the character in the mobile application
* **Sensor**: color pixel matrix of the character

Also, the PEAS of our agent were characterized by: fully observable, fully known, static, stochastic, episodic, single agent, and discrete.

# 3 Pre-processing

## 3.1 Bounding Box

The first step in our pipeline is to pre-process the handwritten image in order to normalize its dimensions and extract the character by itself, thus filtering out any white noise. To do so, we use a library called OpenCV. We start off by implementing a threshold on the image to convert the RGB values into two values: either 0s or 255s. Then, we get the contours of the images, which are then used to get the bounding boxes.

However, two additional functions were implemented: *merge\_near*, and *remove\_contained.* As it is, the function that generates the bounding boxes in OpenCV might sometimes produce unwanted results: for instance, if a user draws the letter ‘E’, but the three horizontal lines are slightly disconnected from the main vertical stem, we will get 4 different boxes, one for each line. Another example is the letter ‘A’: if the user draws the middle stem in a way that is slightly disconnected from the main triangle, OpenCV would generate two different bounding boxes. The same concept applies to the letter ‘i’: we want the dot and the stem to be considered as one letter, and not two separate shapes. Thus, *merge\_near* and *remove\_contained* alter the original bounding boxes to accommodate for these cases. To merge the boxes, we used a certain *threshold* that defines a radius of proximity for both the horizontal and the vertical dimensions. We merge the boxes in 2 cases: if the boxes are fully contained within each other, or if they’re partially contained within each other such that the distance between their extremities fall within the aforementioned threshold. An example of the above can be found in the appendix, with more in the notebook accompanying this report.

After getting the bounding box of the image, the result is then converted into a binary grayscale image and resized to 28x28px. These dimensions were chosen as such to follow with some industry standards, specifically the image size used in the MNIST dataset.

## 3.2 Feature Extraction

In order to classify the handwritten characters with high accuracy, we had to calculate and extract certain features and characteristics from the raw images. We implemented 7 categories of features. We first started by calculating the invariant moments, which are pure statistical measures of the pixel distribution around the center of gravity. These 7 moments are invariant to the position, size and orientation of the character. The second set of features were histograms of the pixel distribution in the image, both vertically and horizontally. We divided the rows of the image into bins of 5 rows each and calculated the number of pixels in each bin; then, we repeated the same approach for the columns. Also, we created 6 virtual lines, 3 of which were vertical and 3 of which were horizontal and identified the number of intersections between each line and the character. Furthermore, we calculated the percentage of pixels in the top half of the image as well as the percentage of pixels in the left half of the image and stored them as features. In our fifth set of features, we implemented the principle of zoning, which consisted of dividing the image into 12 equal sub-images and computing the percentage of black pixels in each sub-image. Another feature vector consisted of getting the coordinates of the division points in the image. A division point is the center through which the image can be divided into 4 sub-images having the closest possible number of black pixels. After the coordinates of the division point are stored in the feature vector, the process is repeated for the 4 obtained sub-images for multiple levels. Our final group of features studied the contour of the character: we got the locations of the first black pixel found in all directions (left, right, top, bottom), then we calculated the derivative of the obtained values to get insights into the smoothness (slope) of the contour; finally, we calculated the width and height of the character at 6 locations by subtracting the right and left coordinates, as well as bottom and top coordinates.

# 4 Machine Learning Models

In order to know what model to use, we implemented several ones and trained and tested them. And to determine which one(s) to use, we checked their accuracy, and their time efficiency.

Our dataset contains 7988 images divided into 62 classes: [0-9], [a-z], [A-Z]. For the distribution of the dataset over the labels, refer to the figure in the appendix. To split this data, we used 2 methods based on the model to which the data was fed. For model that need features, we used the ‘sample’ and ‘drop’ functions from pandas to separate the original DataFrame into training and testing DataFrames. For the models that require the image itself as an input, we used the utility function ‘image\_dataset\_from\_directory’ from Keras. This function forms a labeled dataset from images that are categorically divided into folders and makes it easier to feed this data to Keras models.

## 4.1 Random Forest

To understand Random forest, we should first briefly go over normal Decision Trees.

A decision tree is a flowchart-tree structure that helps predicting and classifying objects. Each node in the tree is an attribute and the branches represent the outcomes of the node

Now going back to random forests, they are a collection of decision trees that operates as an ensemble where the output that is predicted by the most trees in this forest will be the output of the entire ensemble.

After studying multiple non-parametric models in Project 1, the random forest was the most accurate one with 77.9%. Thus, we tried replicating the results with the new dataset we added. We obtained a precision of 86.1%; a recall of 47.3%; an F-value of 57.7%; and an accuracy of 48.9% which is a lot lower than what we previously had.

This could be the result of the new data added where we have some noise in the images.

The Cross Validation plot for a k=10 and the Confusion matrix of the random forest model are attached in the appendix.

## 4.2 Neural Network: Developed by Us

In an initiative to fully understand the inner workings of the machine learning models used in well-known libraries such as Tensorflow and PyTorch, we set out to build our own neurak network API. We built 5 modules that follow a similar syntax to the Tensorflow API: a fully connected layer, which is the equivalent of the dense layer in Tensorflow, a layer for the ReLu activation function, a layer for the Softmax Activation, a layer for Cross Entropy Loss, and finally an API for the Sequential model, which allows the stacking of different layers.

We build each of the layers mentioned above as classes. Therefore, we first develop their initialization function, followed by a *forward* function which computes the forward pass across the layer, and a *backward* function that computes the backpropagation to update weights when necessary.

In the case of the Sequential model, it simply consists of stacking the layers together. We then create a *fit* function that trains the model by computing the forward pass across all its layers, then updating the weights of each through calls to the *backward*  function. Finally, we write a *predict* function that uses the current settings of the weights to generate a prediction on a given set of input.

After testing out the functionality of each module, we proceed to use the model for our application. We import the data, build a simple Sequential model consisting of 2 Fully Connected Layers, a Softmax layer, and cross entropy loss. However, we were unable to get testing accuracies for this neural network Neural Network.

## 4.3 Neural Network

We used TensorFlow and Keras to build a neural network. We began by splitting the dataset into training and testing with 80-20% ratio. Then we passed the images through our pre-processing function to generate cropped and normalized (28x28x1) images, which were used to generate normalized features (between 0 and 1). The features vector corresponding to each image had a size of 264. The next step was to encode the 62 labels of our dataset. We used the LabelEncoder class from Sci-kit Learn to perform one-hot encoding. Now that the data was ready, we built our Neural Network using Dense and Dropout layers. The Dense layers are the typical connected layers, while the Dropout layers randomly set input units to 0 in order to prevent overfitting. A summary of our model is found in Figure 4. The next step consisted of training the model with different values of epochs in order to determine the optimal number. We chose a range of 55 to 115 with steps of 10, and plotted the loss, accuracy, precision and recall versus number of epochs. The corresponding plots are found in the Appendix, Figure 5. The best performance was recorded at epochs = 105, having **loss = 0.765**, **accuracy = 0.7484**, **precision = 0.8037**, and **recall = 0.7021**. After tuning the number of epochs, we performed K-fold cross-validation to come up with the best model and ensure that the training data was not biased. We set the number of splits to 5 and created 5 models, such that each model would train with 4 sets and validate with 1. The accuracy scores of the 5 models are found in Figure 6. The best model had a training **accuracy = 0.7613**. Finally, we evaluated the best model using the test data and got **loss = 0.8771**, **accuracy = 0.7347**, **precision = 0.786**, **recall = 0.6871**.

## 4.4 CNN

Tensorflow presents a simple way to build complex CNN models. As such, we will be using it to build our CNN model. Note that a brief overview of the inner-workings of CNNs is present in the Jupyter Notebook that accompanies this report.

We choose the model architecture as follows: a convolutional layer with 32 filters and a kernel size of (3,3), a maxpool layer with stride 2, another Conv2D layer with 64 filters of size 3x3, a Maxpool layer, a third Conv2D Layer with 128 filters, and a final MaxPool layer. Then we use a Flatten layer to convert the output of the final MaxPool into an arary of values, which is then passed through 3 dense layers where the final layer consists of 62 neurons (to equate to the 62 classes that our model is to predict). A visualization of the architecture can be seen in Figure 8.

After training the model, we proceed to evaluation it. We plot the precision, ercall, accuracy, and loss of the function. Plots representing the mentioned metrics can be seen in figure 9. On the test set, we get an accuracy of 74%, a precision of 75%, and a recall of 73%. However, by inspecting the graphs in figure 9, we can easily detect that the model is overfitting: we have a really high training accuracy of 94%, while the validaiton and test accuracies lag behind.

Therefore, to decrease this overfitting, we proceed to add data augmentation layers to our model. We add a random rotation that rotates the images on every iteration; this would make the model more robust and less likely to overfit. The accuracy after augmentation increased to 78%, and indeed, overfitting was reduced. The corresponding graphs can be seen in figure 10. We also plot the confusion matrix (as seen in figure 11).

In an effort to understand the inner workings of the model, we plotted the outputs of the intermediary layers. They can be seen in figure 12.

However, these accuracies are still low in comparison to state-of-the-art CNN models, and we can see from the confusion matrix that the model is having issues with many letters. This is mainly due to the lack of extensive data; we used reduced datasets due to our inability (time-wise and computationally) to use the MNIST dataset, which consists of ~80K data points. Instead, we trained with only ~6.5K data points. That is why we proceed to use a pre-trained model through a transfer learning application.

## 4.5 Transfer Learning

Transfer learning is a machine learning technique where we take a pre-trained model and modify just enough so it would serve our use case.

In this project, we will be using the VGG-16 pre-trained model available on PyTorch. This model is a Convolutional Neural Network that is 16 layers deep. Its architecture can be seen in the image attached in the appendix.

Originally, the VGG-16 model is trained on Imagenet to correctly classify 1000 different object. It takes as input a tensor of size (244,244,3). Overall, it has a total of 138 million trainable parameters. Training them would take a lot of time. That is why we will only be training its last few layers that are Fully Connected Layers (or Dense Layers).

In order to do so, weights freezing must be applied to all the layers we do not wish to train.

Moreover, since the model can classify 1000 object, we need to change its last layer and set its output to 62 since we only have 62 classes (A->Z, a->z and 0->9).

After setting the learning rate to be 0.001, and the number of epochs to be 10, we got a low accuracy of around 34%. This could have been improved to be

# 5 Deployment

With React Native, we do not build a mobile web app, an HTML5 app, or a hybrid app; we build real mobile apps. React Native uses the same fundamental UI building blocks as regular iOS and Android apps: the developer just merges them using JavaScript and React. The use of react-native is advantageous since the code can run on different platforms and since the community around React and React-Native is large, and you can easily find the required documentation to build your application.

Using this technology, we were able to develop the note taking app allowing the user to choose between writing the note using a normal keyboard or by using our machine learning model. To interface between the app the backend application that will run the model, we use a Flask server to create a RESTful API that will accept a POST request from the application.

The pipeline for the functioning of the app is as follows:

1. User draws something on the Canvas in the app
2. Two seconds after the last stroke has been made, the app will automatically send a screenshot of the screen to the Flask server.
3. The Flask server will run a backend API that pre-processes the input, using the *get\_boudning\_box* function described in the pre-processing section.
4. According to the language that was chosen by the user (both English and Arabic are available), the Flask server will do a forward pass through one of the pre-trained models discussed in this notebook.
5. The server will return a string that represents the model’s prediction of the drawing. This string will then be processed by the app to be displayed on-screen.

# 6 Conclusion

To conclude, the aim of our project was to improve our note-taking mobile application which contains a Handwritten Character Recognition agent. We approached the design from the perspective of a customer who wishes to use the application, so we managed to create an easily understandable and manageable UI. We also followed a scientific procedure to design, train, tune, and test parametric machine learning agents. Some of the limitations faced are that our character recognition model cannot separate between a few similar-looking characters such as capital ‘C’, ‘V’ and ‘W’, and lower-case ‘c’, ‘v’ and ‘w’. Also, due to the normalized dimensions that we are using, the processed image of letter ‘O’ and number ‘0’ are alike; therefore, our model is not able to distinguish them. The main cause of these limitations is the lack of data; we used around 6500 samples for training while parametric learners require larger datasets. This was not feasible because of the lack of computational resources. The complete project can be found on this GitHub repository: [t0t0-01/Letter-Recognition-2 (github.com)](https://github.com/t0t0-01/Letter-Recognition-2)

# 7 References

*[1] React Native Documentation* (n.d.). Retrieved from https://reactnative.dev/docs/getting-started

*[2] Expo Documentation* (n.d.). Retrieved from https://docs.expo.dev/

*[3] Ø. Due Trier, A. K. Jain, and T. Taxt, “Feature extraction methods for character recognition-A survey,” Pattern Recognition, vol. 29, no. 4, pp. 641–662, 1996.*

*[4] “Deep Network designer,” VGG-16 convolutional neural network - MATLAB. [Online]. Available: https://www.mathworks.com/help/deeplearning/ref/vgg16.html. [Accessed: 14-Dec-2022].*

*[5] Heutte, L., Paquet, T., Moreau, J. V., Lecourtier, Y., & Olivier, C. A structural/statistical feature-based vector for handwritten character recognition. Pattern Recognition Letters, 19(7), 629–641, 1998.*

*[6] G. Vamvakas, B. Gatos and S. J. Perantonis, "Handwritten character recognition through two-stage foreground sub-sampling," ELSEVIER, 2010.*

# 8 Appendix

Chart, bar chart, histogram

Description automatically generated

Figure 1 - Distribution of the dataset over the labels

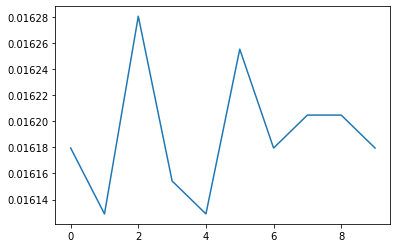


Figure 2 – Cross-validation RMSE for Random Forests model

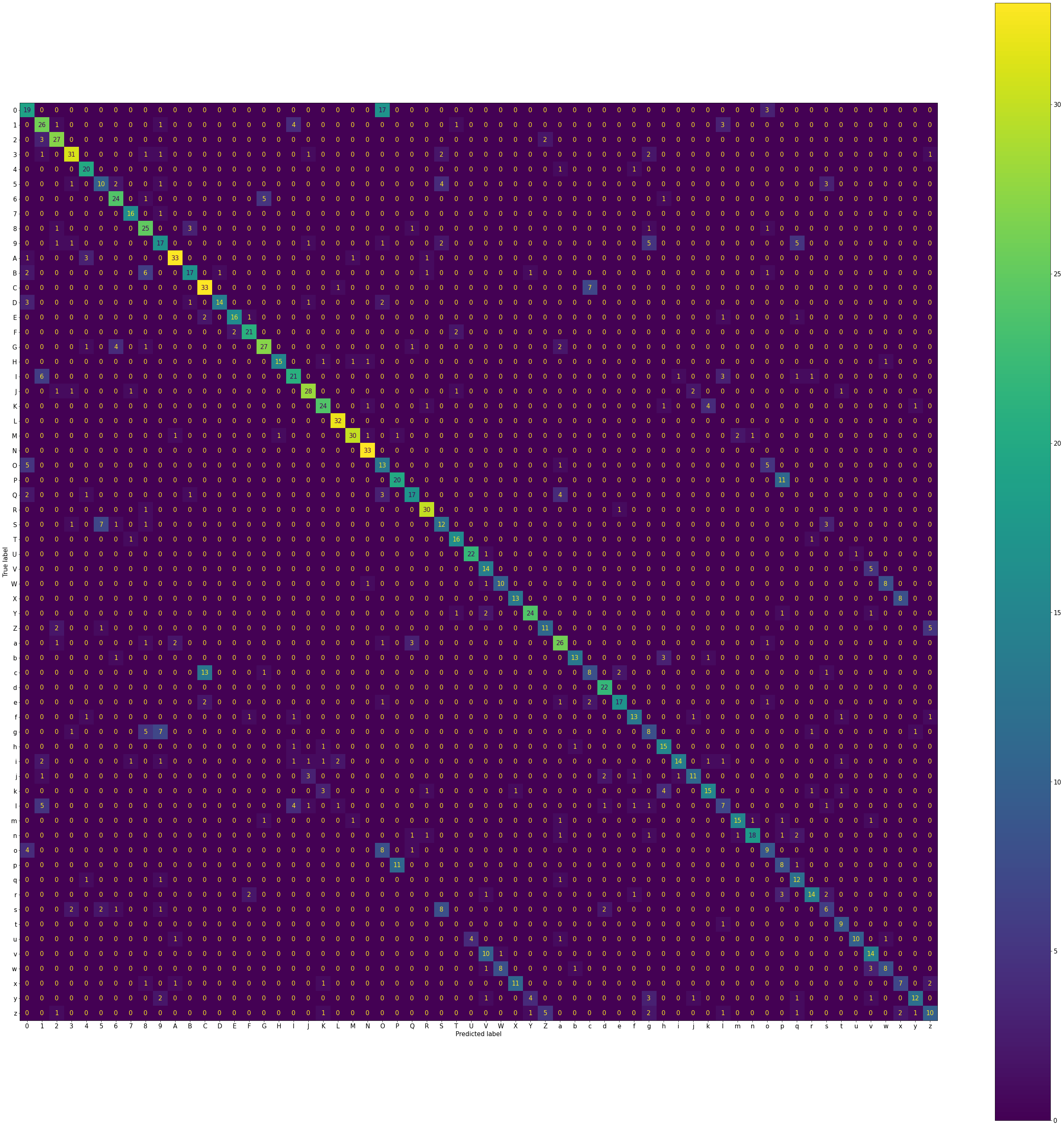


Figure 3 - Confusion matrix displaying the results of the Random Forests model

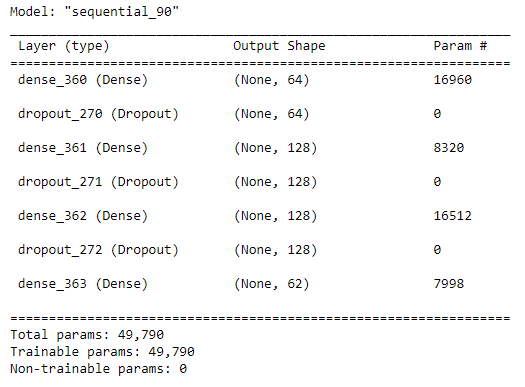


Figure 4 - Keras Neural Network model summary

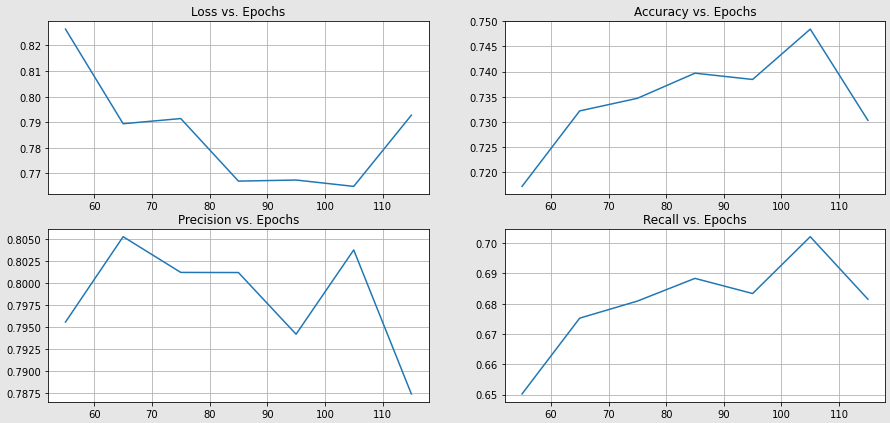


Figure 5 - Performance measures vs. number of epochs for the Keras NN model

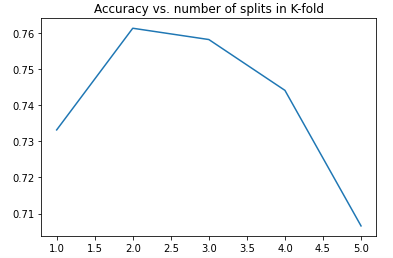


Figure 6 - Accuracy of the 5 NN models during K-fold cross-validation

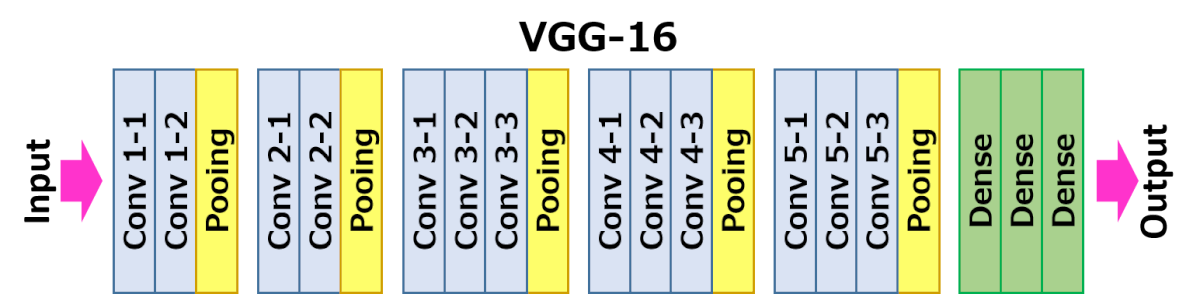


Figure 7 - VGG-16 model architecture

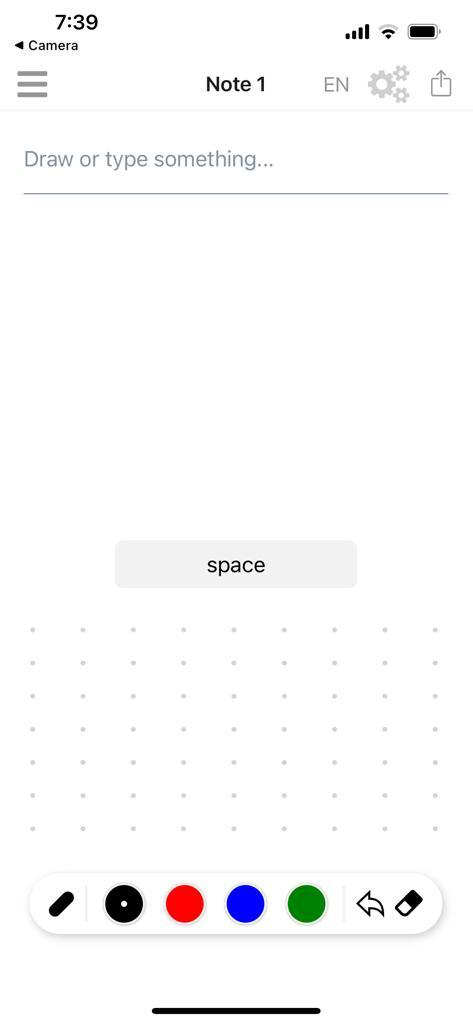
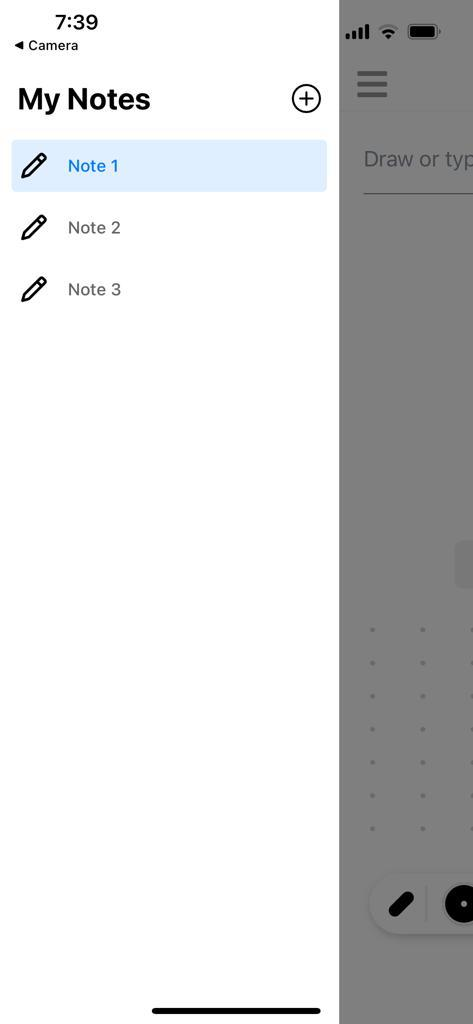
 

Figure 8 - User Interface

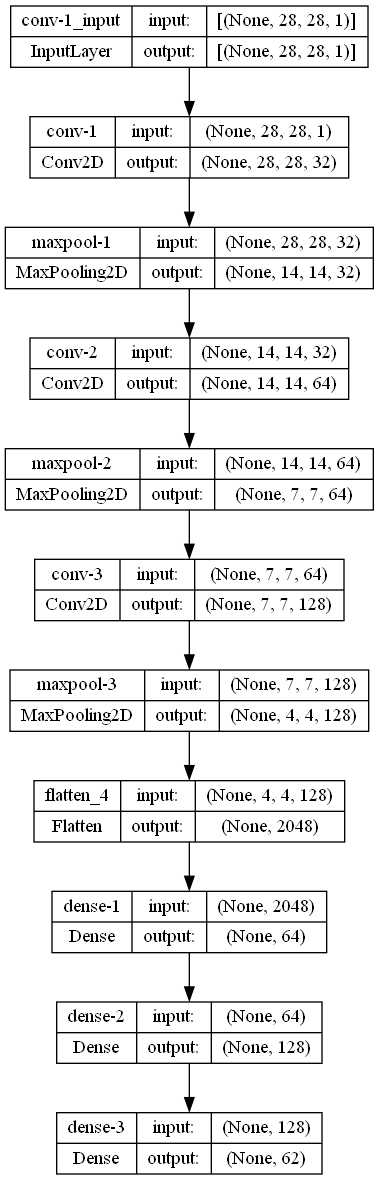


Figure 9 - CNN Architecture

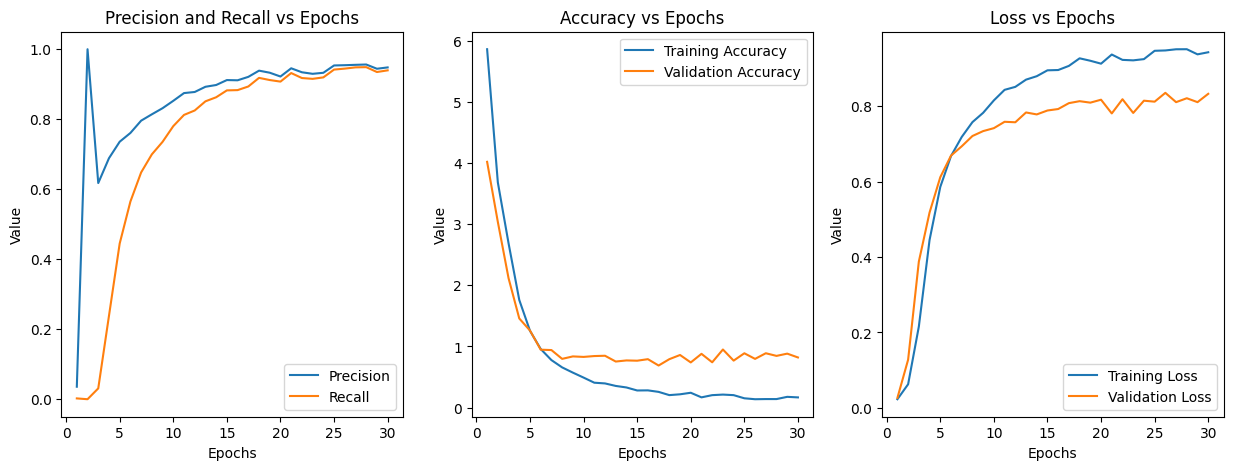


Figure 10 - Precision, Recall, Accuracy, and Loss graphs for 1st CNN

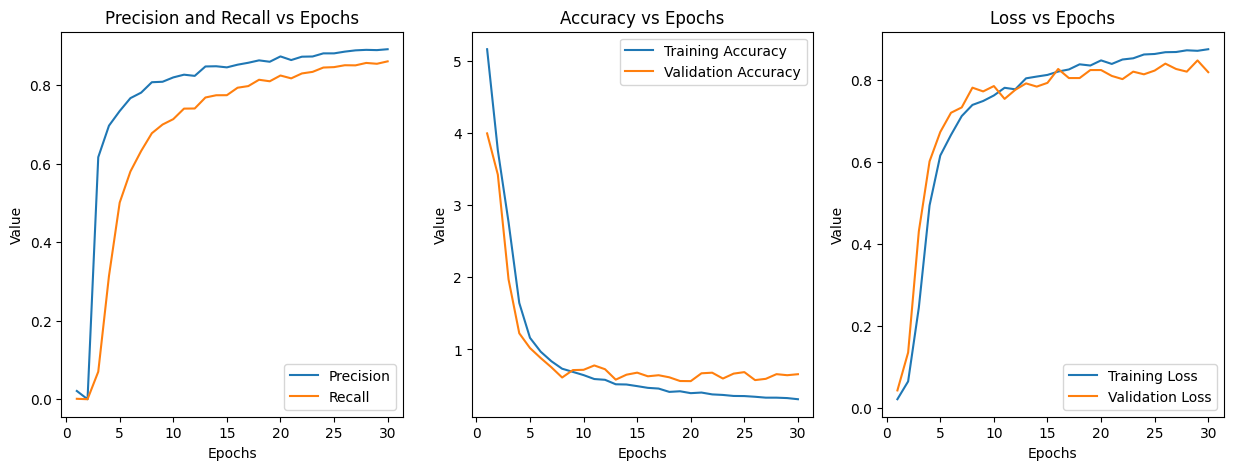


Figure 11 - Precision, Recall, Accuracy, and Loss graphs for Augmented CNN

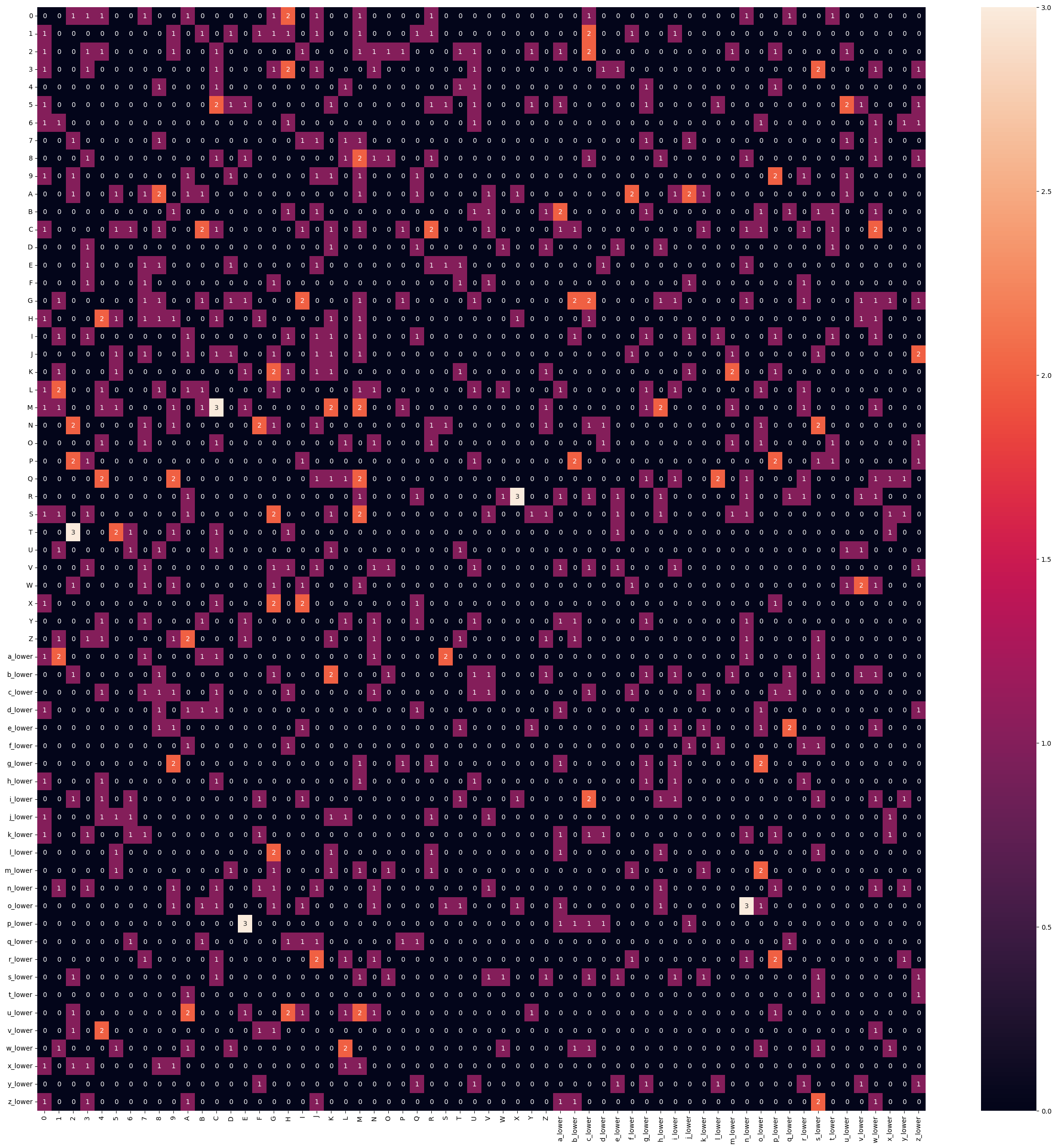


Figure 12 - Confusion Matrix of Augmented CNN

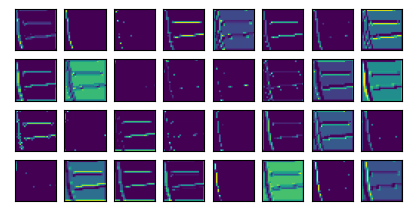


Figure 13 - Visualization of intermediary outputs

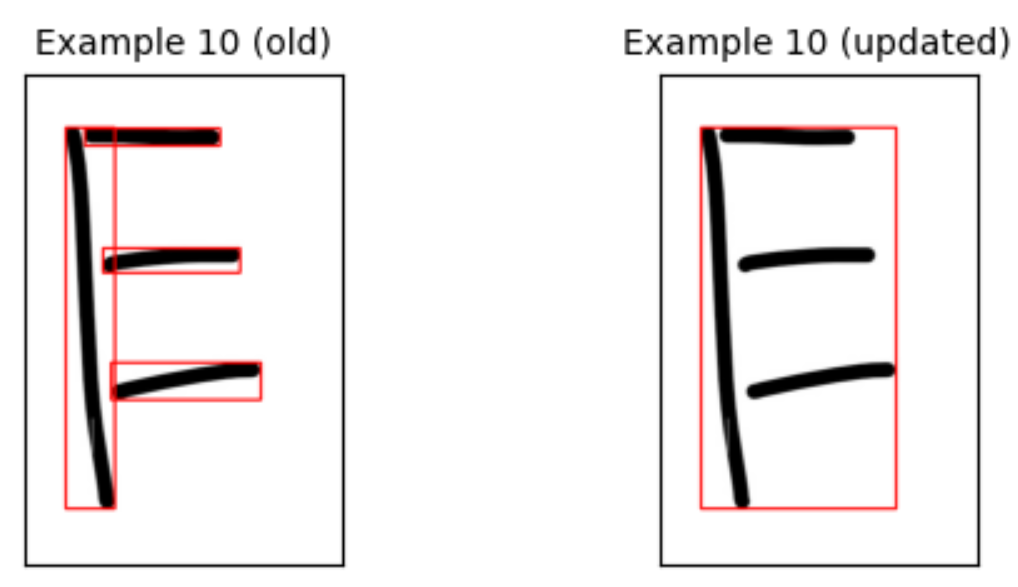


Figure 14 – Example of functionality of merge\_contained