# Handwritten Greek Characters Classification

# Abstract

Different models can be used to build a machine learning system capable of performing a character classification task. One of the classification tasks most often used as a benchmark for classification models is character recognition. This report describes a comparison among these different models, trained on a dataset of modern greek handwritten letters, with different hyperparameters and optimization features. It is well known that Convolutional Neural Networks (CNN) perform the best in image classification tasks, however, reaching high accuracy scores also requires hyperparameter tuning and strategies to solve high variance. Using these optimizations, the best model trained in the experiments reaches an accuracy score of up to 94%.

# Contents

**Introduction** 1

**Preliminaries** 3
**Metrics** 3
**Precision** 3
**Recall** 4
**Accuracy** 4
**Confusion matrix** 4
**Models** 5
**Artificial Neural Network (ANN)** 5
**Convolutional Neural Network** 5
**Optimization of a model** 6

**Dataset** 8
**Merging some classes** 10

**Method** 12
**Dataset preprocessing** 12
**Models architecture** 13
**Optimization strategies** 15
**Metrics** 17

**Implementation** 19
**Dataset preprocessing** 19
**Models implementation** 20
**Optimization implementation** 21
**Metrics** 21

**Experiments and results** 23

**Conclusions** 33

**References** 34

# 1 Introduction

Character classification tasks have been in the spotlight of ML for decades. They present a good way to benchmark models of image classification and perform studies on machine learning models and neural networks. Therefore, there are numerous datasets of glyphs, characters, and digits, among which the most common is the MNIST dataset, comprised of digit glyphs from 0 to 9. This project has a similar goal: building different models for character classification, comparing them, and looking at optimization strategies. The classification task of characters is going to be achieved for greek characters, using a dataset of greek handwritten characters.

Automatic recognition of handwritten characters can help in numerous real-world applications, from data entry to automatic mail sorting, logistics, and so on. This is especially true for the alphabets currently used. Other real-world applications concern the digitization and automatic analysis of historical documents. This is true for older alphabets or older forms of alphabets. One such example is the old polytonic Greek alphabet, which differs from the modern one in the use of special accents. A classification algorithm must then be able to also recognize the accents, increasing the number of classes.

The Greek alphabet, due to its historical significance and its relevance also outside of the Greek language (e.g.: greek characters are very often used in mathematics), is one of the most important and recognizable collections of glyphs (here glyph meaning character or generic symbol). Implementing a classification algorithm for greek handwritten characters is not a difficult task today, however, the resources (datasets, studies, etc.) about this topic are relatively limited when compared to other, more popular, sets of glyphs, such as Arabic digits and Latin characters. Being less explored, finding large datasets with hundreds of thousands of samples is also a problem, this needs to be considered when implementing the correct model for the dataset at hand.

The report describes the various models and techniques used to accomplish the task. As expected, CNNs perform better in the task than fully artificial neural networks (ANNs).

The first Preliminaries section will introduce the concepts used in the whole report, then the Dataset section describes accurately the dataset and its peculiarities. In the Method and Implementation sections, we describe the way in which the models have been built. Method delves into the theoretical details while Implementation lists the libraries and tools used for each task. The experiments section describes the training process listing the specific hyperparameters for each model, this section will also show the loss and accuracy change as the model is trained, with the relative graphs. The Results section presents the metrics (precision, accuracy, recall), identifies the best model, and shows the relative confusion matrices for each model.

Finally, in the Conclusions section, the final statements are written, as well as proposals to eventually improve even further the model.
Having a good model for the classification of greek handwritten characters can be a good step in the direction of increasing attention to the topic, thus hopefully also increasing the amount of collected data and furthering research in a direction that would benefit numerous real-world applications, from industry and services, to research in antiquities.

# 2 Preliminaries

To start our discussion some concepts that will be used throughout our report need to be introduced.
The classification model that will be built is going to take as input the pixels of an image (the image is a photo of a greek handwritten character) and output a 48-dimensional array, each element of the array represents a confidence value attributed by the model to one of the 48 classes (classes are: upper-case alpha, lower-case alpha, upper-case beta, etc. up to zeta). The class with the greatest confidence value is the predicted class for the image.
The first thing concept introduced is the way in which these models are evaluated: using metrics of interest such as precision, recall, accuracy, and confusion matrices. Then the specifics of various classification models are discussed.

## 2.1 Metrics

Metrics are an important part of the project. It's the tool that will be used to measure our models' efficacy. These metrics will determine what is the best model in a rigorous way. These metrics rest upon the definition of true-positive (TP), true-negative (TN), false-positive (FP), and false-negative (FN) classifications. In binary classification, with a true/false output label, a TP classification is a correct classification, that assigns the true label correctly to an input image (or whatever the input data is), a TN correctly assigns the false label to the input image, an FP assigns wrongly the true label to the input image, and finally a false-negative wrongly assigns the false label to the input image. These concepts can be extended for multi-class classification (such as this project's task) and are used in computing precision, recall, and accuracy, as well as the confusion matrix.
Aside from this introduction, the detailed formulas used for these experiments can be found in the Metrics section of the Method chapter.

### 2.1.1 Precision

Given a class, the Precision (also known as positive predictive value) is the ratio of TP over all inputs that have been classified as belonging to that class (i.e.: TP and FP): $P = \frac{TP}{TP+FP}$.
In a balanced dataset, we can then average over the number of classes (N) to get an average of the precision for all classes: $P = \frac{\frac{\sum_i TP_i}{\sum_i TP_i+FP_i}}{N}$. This formula is called macro-averaging precision and is described in [4] Table III. Precision can be seen as a measure of how the model is qualitatively good (the number of correct predictions

among all predictions made). A model with a precision of 0.8 will be correct 80% of the time when making a prediction.

### 2.1.2 Recall

Recall (also known as sensitivity) can instead be seen as a measure of how the model is quantitatively good. A model with a recall of 0.8 will only classify as true 80% of all true instances. Recall in fact is the ratio of TP over the sum of TP and FN (this is all the instances that were correctly classified as true and the instances that had to be classified true but were wrongly classified as false): $R = \frac{TP}{TP+FN}$. It's possible to take an average over all classes as it was done for Precision.

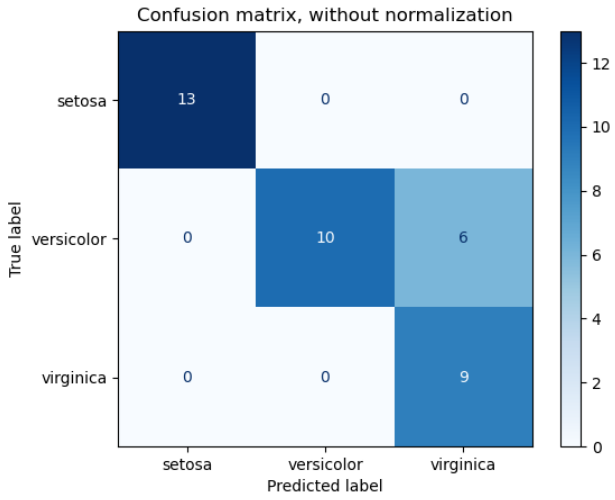
### 2.1.3 Accuracy

Precision and recall describe different things. Accuracy thus is used to compute a unified metric that can describe the overall performance of a classifier. It is the metric of choice for classification. It is the ratio between the number of correct predictions and all predictions. In the binary classification example the correct predictions are both TP and TN, therefore Accuracy is $A = \frac{TP+TN}{TP+TN+FP+FN}$.

### 2.1.4 Confusion matrix

A confusion matrix is an immediate and concise way to represent the results of a classifier, it can summarize in a table precision, recall, and accuracy. The figure shows an example of confusion matrix for multi-class classification, each cell contains the number of predictions, where the horizontal position determines the predicted class while the vertical position determines the true class.

An example of a confusion matrix, of the same kind and color scheme for this project:




## 2.2 Models

There are many models for classification, among them those of classical machine learning such as support vector machines (SVMs), and those of deep learning, like ANNs and CNNs. The ones that yield better results for generic tasks of image classification are CNNs. ANNs however were also used in the past for image classification and can serve as a baseline to compare other models.

### 2.2.1 Artificial Neural Network (ANN)

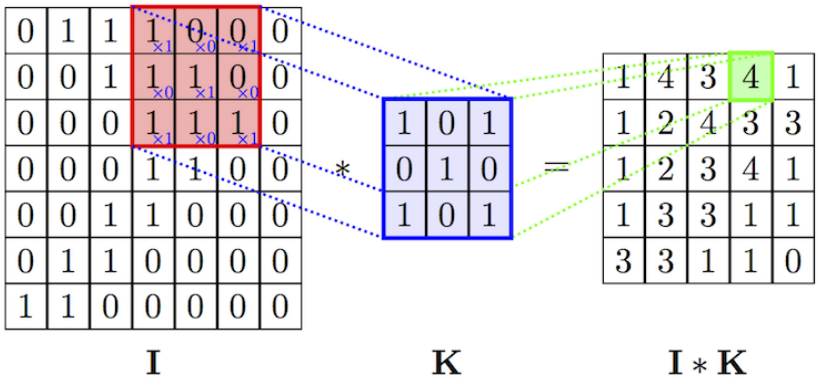
Fully connected artificial neural networks are the baseline and most elementary form of a neural network used in deep learning. Each node of a layer (except obviously the input layer) performs a weighted sum with a bias of all of a preceding layer's nodes, then applies an activation function to the result. Since the activation function is very often a non-linear function, these models are very large approximators of non-linear functions. Furthermore, differently from linear and logistic regressions, where input variables contribute independently to the output, in ANNs input variables interact with each other. Being complex non-linear models, they can be used for a wide range of tasks with a great degree of success. The weights and biases are initially set and then through a process of gradient descent, they are modified to minimize an error function of the predicted output and the real output.

### 2.2.2 Convolutional Neural Network

Images can be represented as simple vectors of pixels, they can therefore be given to ANNs as inputs. However, the ANN architecture isn't suited to capture image features. In fact, thinking of images as simple vectors of images, while true, isn't the

best approach: an important aspect of images is the spatial distribution of pixels: their proximity, distance, and disposition. This aspect is difficult to capture using ANNs. This is why convolutional neural networks (CNNs) come into play. Each node of a layer this time is not a weighted sum of all of a preceding layer's nodes, but just the weighted sum of a sliding "window" (called the kernel) of nodes in the preceding layer. This captures the adjacency of pixels, furthermore, the weights of the sliding window remain the same for all hidden units of a layer, which allows capturing the same feature by the sliding window. Thus, each kernel captures a feature, and the weights of the kernel, which determine the feature to detect, are learned by a process of gradient descent, minimizing an error function (which for CNNs and classification tasks is often the categorical cross-entropy). Another important aspect of CNNs is that they are good at ignoring features that are not discriminative and instead focus on discriminative features [3].

An example of a convolution, numerous kernels form a layer of a convolutional neural network:




### 2.2.3 Optimization of a model

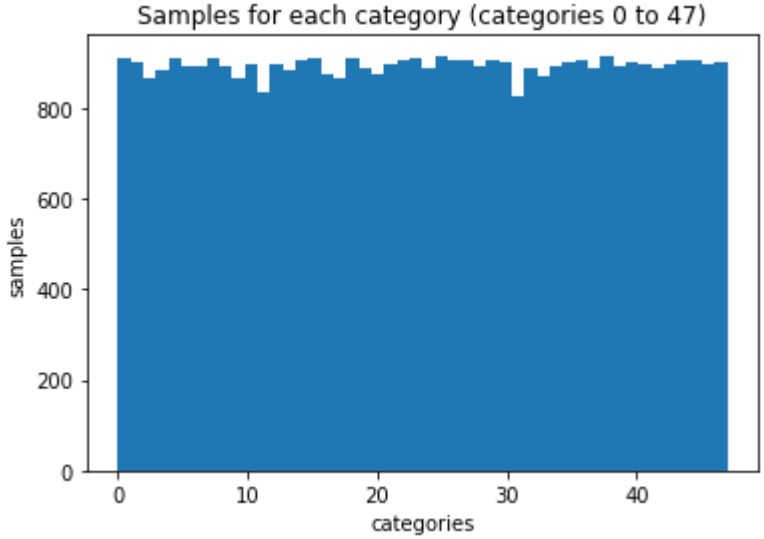
These models used need a lot of hyperparameters: for ANNs, the depth and width of a neural network, for CNNs, the number of kernels, their stride, and size, etc. It's also necessary to choose general network architecture, activation functions, loss functions, and other optimization features (regularization parameters, dropout probability, variable learning rate, use of batch normalization, use of data augmentation, etc.). To choose the correct combination it's possible to rely on experience, previous results with tested approaches, and searches for hyperparameter optimization. In this project, the model's hyperparameters and architecture have been chosen based on well-known previous results and approaches for optimization.

When training and testing a model two problems can arise: the model's high bias and high variance. A model has a high bias when it's oversimplified and has a high error on both train and test data (underfitting). A model with high variance instead performs well on train data but poorly on test data, this is due to the lack of generalization (overfitting). To resolve the high bias issue we can increase the complexity of the model so that it fits better with the training data, in a neural network this means increasing the number of layers and the width of layers. Increasing the depth of a neural network is generally preferred due to its generalizing power. To resolve the high variance issue instead other methods are used: such as L2 regularization, Dropout, and dataset augmentation during training. The bias and variance of a model exist in a relationship of trade-off, having a model that generalizes better (low variance) often implies simplifying the model thus increasing its bias while decreasing a model's bias often implies making it more complex and thus prone to overfitting (high variance).

# 3 Dataset

A dataset for old polytonic handwritten labeled characters has not been found. Finding a dataset for handwritten greek letters is more challenging than for other more widespread set of glyphs. However a dataset with a sufficient amount of samples has been found.

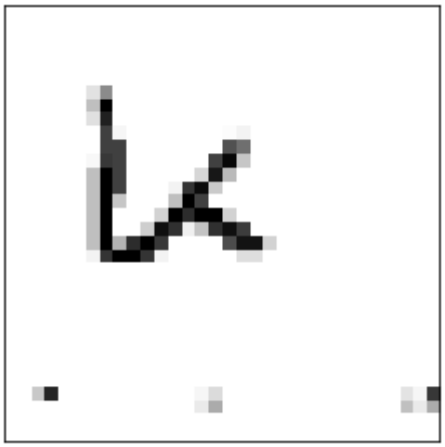
The dataset used comprises about 43,000 samples of handwritten modern greek characters, evenly divided into 48 classes, with 24 letters both upper case and lower case. The distribution of classes is uniform, with about something more than 800 samples for each class, as shown in the histogram below. The dataset was downloaded from Kaggle [1] and is originally from the GCDB database of greek handwritten letters [2]. This database is no longer available through its original website, yet the raw data can still be gathered from Kaggle. The dataset has an intuitive directory structure, however, for each character class (e.g.: capital alpha), there are two folders, one for the characters that were written individually, and one for characters that were extracted from words. They contain the same class of characters, they just differ in the way in which this data was gathered. To solve this problem and merge the folders containing the same characters, a Bash script was written (create_dataset.sh) to reorganize the data. This bash script created a dataset folder where there is a subfolder for each class, for a total of 48 folders.




The images in the dataset are of BMP3 format, an older BMP format. This is of no problem as the OpenCV library can read the image. The images only have one channel of grayscale and this is enough, since color is not a discriminative feature of handwritten characters. The sizes of different the images however are different, while all are about 45x45, some images have some more (or less) pixels in either width or height. This creates a problem since the Neural Network we are building

has a fixed input size. To solve this problem the images were resized to a 32x32 width and height format. The differences in pixel sizes of the original images were minor, 454 pixels, therefore resizing the images doesn't fundamentally change the aspect ratio of the image and its shape. Changing the aspect ratio of an image should be avoided if it changes some fundamental features of the image, preserving the aspect ratio of images is, therefore, an important aspect of image classification and image preprocessing, if the aspect ratio were to be changed by a significant amount, padding needed to be added to maintain the original aspect ratio, for this dataset the obvious choice is white padding since that is the background of images. Adding padding doesn't hinder learning because, as specified in the CNN section, CNNs are good at ignoring non-discriminative features, padding randomly added to all images being one.

Example of a sample from the dataset, a capital kapa letter:



The dataset also has some features, for instance, there are random straight lines below, above, and at the sides of letters, these come from the module that was used to gather the handwritten data, these features appear randomly in the images across all letters, so the CNN simply ignores them as they are not discriminative features between letters. The quality of the dataset can however be improved by removing these lines. Furthermore, the number of samples is not as high as for some other datasets of the same kind. The MNIST dataset, for a set of 10 glyphs, has 60,000 samples, while the dataset at hand has about 43,000 samples for 48 different glyphs. A higher number of samples could help in building more complex models, as a higher number of parameters in a model usually needs a higher number of training samples. More data could also improve the models implemented in this project.

As with all datasets in ML, this dataset has been split into training, validation, and testing sets. To increase the number of samples and diversity in the dataset, data augmentation has also been used for some of the models. This will be seen in more detail in the method section.

Below an example of a dataset batch of 32:


## 3.1 Merging some classes

As it's possible to verify, some classes are extremely similar, the handwritten letters present the same features and it's difficult for a human to classify these letters correctly. For instance the upper case kappa letter and the lower case kappa letter. This is true for an ML classifier as well, which recognizes the same features and misclassifies some letters. The classifier, therefore, risks having consistently worse performance if such identical classes are not merged. This is a problem that is absent for digits (such as the MNIST) and depends on the fact that it is the project's aim to classify both upper cases and lower cases. This issue was already encountered by Vamvakas et al. [8] and they propose to merge some classes. In this project, we tackle this problem by training the model without merging classes, recording the results, and then merging some classes, therefore considering correct if the model classifies an upper case kappa as a lower case kappa and vice versa. Comparing the results for merged and not merged classes it's possible to notice greatly improved results. The table below shows the merged classes.

| | |
|---|---|
| Upper case theta | Lower case theta |
| Upper case pii | Lower case pii |
| Upper case xi | Lower case xi |
| Upper case omikron | Lower case omikron |
| Upper case iota | Lower case iota |
| Upper case psi | Lower case psi |
| Upper case tau | Lower case tau |
| Upper case kapa | Lower case kapa |
| Upper case fi | Lower case fi |
| Upper case ro | Lower case ro |
| Upper case beta | Lower case beta |

# 4 Method

This section describes the methods used to accomplish the task, with regard to dataset preprocessing it's image normalization, dataset augmentation, and dataset splitting into the train, dev (or validation), and test set. Next are described the model architectures used and their optimization strategies and finally details for metrics and model evaluation are presented.

## 4.1 Dataset preprocessing

The first task to accomplish in the realization of this project is data preprocessing. After resizing the images to a fixed size, the second step is image normalization. This step consists in modifying the pixel values for each image, instead of values from 0 to 255, the mean will be subtracted from each image's pixel and then it will be divided by the standard deviation. The new pixels will be floating-point values in an interval that is better for the model training. Images become clearer and normalization helps the models converge faster while training.
Following image normalization, the images have been subject to dataset augmentation. Models trained with dataset augmentation tend to be more robust. This method is used to increase the diversity and number of samples in the dataset, however, minor changes in the images have been applied to maintain the features of characters that the model needs to learn. The dataset has been augmented by applying three transformations: rotations, translations and zooming. Rotations have been of 0.052 , translations are of 10% of the image's height and width, while zooming is of 10% of the image's size, all transformations modify the images by relatively small amounts. Not all models will use dataset augmentation.
The dataset has then been split into three: a training set of 75% of all the training examples, a validation set of 15% of all the training examples and finally a testing set of 15% of all training examples. The dataset is split since the model is trained using the training set, the validation set is instead used to choose the hyperparameters and evaluate the model at each training step and finally the testing set is used to evaluate the model at the end of training. Splitting the dataset is very important since we are interested in the model's performance on the testing set (a set of samples that the model has not yet seen), not on the training set, which is comprised of all the examples that the model already has seen.

Example of an augmented dataset batch of 32:


## 4.2 Models architecture

After dataset preprocessing, the models should be built. The models used, as hinted before, are ANNs and CNNs. Activation functions are mostly ReLu since it is the better one and doesn't suffer from gradient vanishing as the sigmoid function does. In the output layer instead, the softmax activation function is used, this is to convert the output units into a probability distribution over the classes. In all cases, the categorical cross-entropy loss function is used. The number of epochs for training (i.e.: the number of total training steps), is determined with early stopping. The optimizer for the model is Adam [7] which is a better version of the simple stochastic gradient descent. Finally, the model is trained with batches of 32 images at a time. There will be a total of 4 trained models:

- ANN:

This model is a shallow neural network of just one hidden layer, it is used as our baseline model.

1. The input layer of 1024 units (32 x 32);

2. First hidden layer of 1440 units, ReLu activation function;
3. The output layer of 48 units, Softmax activation function.

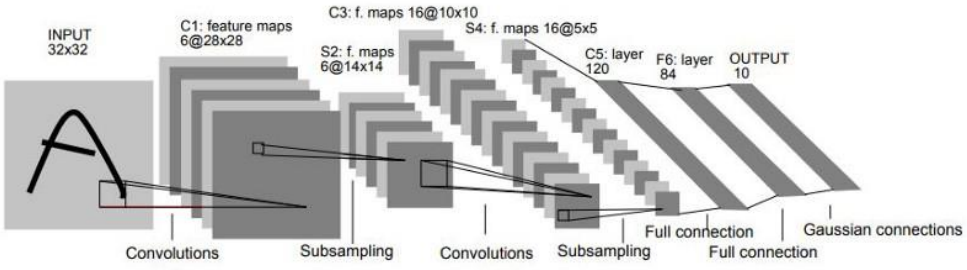
- LeNet5:

This model implements the LeNet convolutional neural network architecture as described by LeCun et al. in the 1998 article [5], it is a couple of convolutional layers followed by max-pooling and then a couple of fully connected layers preceding an output layer:

1. A convolutional layer of 6 kernels, with a kernel size of 5 and stride of 1;
2. A max-pooling layer of pooling size 2 and stride 2;
3. A convolutional layer of 16 kernels, with a kernel size of 5 and stride 1;
4. A max-pooling layer of pooling size 2 and stride 2;
6. A dense layer of 120 units;
7. A dense layer of 84 units;
8. An output layer of 48 units.

The input is a batch of images of size 32 x 32. This model will be trained with dataset augmentation.

The image below shows the LeNet architecture:




- BetterLeNet5:

This model is similar to LeNet5, it's comprised of some convolutional layers and dense layers. The difference between this model and the previous model is its depth and width, it is in fact deeper, with 2 convolutional layers before each max-pooling layer and then three dense layers before an output layer. This model also uses optimization strategies such as Dropout with a probability of 0.25, Batch normalization for each layer, early stopping, variable learning rate, and L2 regularization with parameter 0.0005. This model will be trained with dataset augmentation as well.

- LeNet5v2:

This model is also similar to LeNet5 and to our implemented model BetterLeNet5, it uses the same optimization strategies but has an even greater depth and width, both for the convolutional layers and the dense fully connected layers. This model won't be trained with dataset augmentation.

## 4.3 Optimization strategies

The choice of hyperparameters is one of the main tasks for algorithm optimization. In this project the hyperparameters for the BetterLeNet5 model are chosen using known values from previous results: the Adam optimizer with its default learning rate, the dropout with 0.25 probability, lambda parameter for L2 regularization of 0.0005, a variable learning rate of factor 0.2, batch size of 32 and 50-100 epochs. The optimization strategies described and mainly used in the BetterLeNet5 model are:

**Dropout**

Dropout is a strategy for improving the generalization of a model (and thus reducing variance) described in [6]. Using dropout, at each step of training, the backpropagation algorithm that trains the weights is applied to a different version of the network. In each step, there is a probability P that each neural unit is excluded from the training process and its output set to 0, on the other hand, with probability 1-P, each neural unit's output is multiplied by 1/P. The dropout probability P used here is 0.25 since probabilities of about 0.20 are proven to be the most effective. By introducing dropout a noise is introduced during the training steps and the model becomes more robust to noise. Furthermore, dropout forces the model not to focus on just one discriminative feature but to look at all the abstract features, this is because for some training steps the nodes associated with certain features may be zeroed, yet the model must still be able to make the correct choice. These factors contribute to making dropout a good strategy for reducing the model's variance since the network doesn't overly rely on certain nodes to make a decision. [3]

**Batch normalization**

Batch normalization is a technique used to improve the rate of convergence of stochastic gradient descent. It consists of normalizing the outputs of a node across all the values that a node has for a given batch of data. Given a node $z_i$, batch normalization follows the formula: $z_i = \gamma \frac{z_i - \mu}{\sqrt{\epsilon - \sigma^2}} + \beta$. This formula shows the process of normalization of $z_i$ where $\mu$ is the mean of all $z_i$ for a given batch of data and $\sigma$ is the standard deviation ($\epsilon$ is inserted to avoid dividing by zero). Gamma and beta are two learned parameters. Batch normalization reduces the "movement" between units. The reasons for batch normalization's effectiveness are not yet fully known,

however, the practical effects are significant, since this technique stabilizes the network and makes it converge faster [3].

**L2 regularization**

Regularization is a technique used to penalize complex hypotheses. As we have explained before, complex hypotheses can lead to high variance, which means that the model generalizes poorly and while having low error for the training set, it has high error for the validation and test set, this is called overfitting. This is accomplished by adding a component to the cost function: $Loss = Error + \lambda \cdot Complexity(h)$ where lambda is a hyperparameter and Complexity(h) is called regularization function and represents the complexity of the model. The way in which we choose the regularization function affects the variance of the model and depends mainly on the structure of the model. For instance, for polynomials a good regularization function is the sum of the squares of the polynomial's coefficients, this is called L2 regularization, and the regression process associated is called Ridge regression. For neural networks, we can use the same concept and sum the squares of the weights: $\sum_{i=0}^{N} \quad {w_i}^2$. Higher losses cause the weights to be relatively smaller and this "simplifies" the model. The chosen lambda hyperparameter that multiplies the regularization function is 0.0005.

**Variable learning rate**

The variable learning rate is a technique used to optimize models. Fine-tuning the learning rate and thus reducing the model's loss, getting closer to the global minimum, while also helping the model in converging faster. In this project's case, the learning rate will be reduced once the learning process hits a plateau. The learning rate will be reduced by multiplying by a factor of 0.2 each time a plateau is reached. More specifically the learning rate will be reduced once the validation loss, computed for each epoch, hits a plateau. When hitting a plateau, instead of keeping a high learning rate, a reduced learning rate helps in reducing further the loss value. This is because when the value of the loss is not changing and hovers around a value, we may have reached a point where smaller steps are needed to continue the descent and reach the global minimum. With an higher learning rate we are instead hovering around this global minimum, unable to descend further and decrease the loss.

**Batch size**

The batch size used for the experiments is 32, which is in line with the LeNet [5] architecture. Instead of only optimizing only one training sample at a time, the model optimizes over a batch of samples, in this case, 32 samples. Then the average

of the gradients of all the training examples is used to update the parameters of the model. An epoch is an iteration over all the batches until reaching the end of the dataset. The training is performed for a number of epochs, therefore iterating over the whole training dataset multiple times.

**Number of epochs and early stopping**

Models have been trained with maximum epochs being 50 and 100. It's maximum epochs because the model is not guaranteed to train for up to 50 or 100 epochs. The model will in fact stop training once a value stops improving. In this project, the value of interest is the validation accuracy and the patience is 10 epochs when the maximum amount of epochs is 100 and 5 epochs when the maximum amount of epochs is 50. So, while training for 100 epochs, if for 10 epochs the validation accuracy doesn't improve, then the model stops training and the saved weights of the model are the weight that gave the best validation accuracy. The same is true with 50 maximum epochs, but only if there are 5 epochs without improvement. This allows the model not to train excessively and only save the best weights.

## 4.4 Metrics

Finally, we consider the metrics used. These are the same as the ones described generally in the Preliminaries section, the difference is that the task at hand is not a binary classification task, instead, it is a multi-class classification, so we will use the appropriate formulas, in part already shown. In multi class classification we use this definition for true positives for each class i ($TP_i$) and false negatives for each class i ($FN_i$): $TP_i$ is the number of test examples of class i that have been correctly classified as class i. $FN_i$ is the number of test examples of class i that have been mistakenly classified as different from class i. $FP_i$ is the number of test examples of different from class i that have been mistakenly classified as class i. The formulas that will be described in the following section perform an average of all the relative precisions and recalls for each class (precision for each class i: $P = \frac{TP_i}{TP_i+FP_i}$; recall for each class i: $R = \frac{TP_i}{TP_i+FN_i}$). This is called macro-averaging and gives the same weight to different classes, this is good for datasets where the data is uniformly distributed among classes, like the one at hand.

For N classes (N=48 characters) the formula for precision that we will use is $P = \frac{\sum_i TP_i}{\frac{\sum_i TP_i+FP_i}{N}}$, while the formula for the recall is $R = \frac{\sum_i TP_i}{\frac{\sum_i TP_i+FN_i}{N}}$. These are macro averaging precision and recall as described in [4]. Accuracy is $A =$

$\frac{number\ of\ samples\ classified\ correctly}{number\ of\ all\ samples}$ and finally the confusion matrix has size NxN (48x48), where there are 48 rows, rows being associated to a ground-truth value, and 48 columns, columns being associated with a predicted value. Another result that is going to be shown is the loss function for the models, this is the categorical cross-entropy: $Loss = -\sum_{i=0}^{N} y_i log(h_i(x))$ where N is the number of classes (and thus the output size of the model, since the model outputs a probability distribution over all classes), $y_i$ is i-th component of the ground-truth value and $h_i(x)$ is the i-th component of the prediction computed from input sample x. This is the function that the optimizers try to minimize.

The metrics that are going to be presented can be divided in two: metrics gathered during the training process and metrics from the model evaluation with the test set. The metrics gathered during the training process are loss and accuracy, both computed for the training set and validation set. The loss is expected to lower over epochs while the accuracy is expected to increase over the epochs. Then. there are the metrics from the model evaluation with the test set: these metrics are precision, recall, accuracy and the confusion matrix. Precision, recall and accuracy are computed for the test set amount to scalar values, while the confusion matrix can also be plotted as a matrix with a color scheme.

The metrics will show what is the best model.

# 5 Implementation

## 5.1 Dataset preprocessing

The first task, which was the reorganization of the dataset into a correct directory structure, has already been described in the Dataset section. This task was accomplished by coding a simple bash script that accomplishes the task. The bash script is called create_dataset.sh and uses simple Linux commands such as mv, cp, for loops, and the zip utility to unpack archives. It also uses the rename tool (a useful tool to rename files and folders using Perl regular expressions), which therefore needs to be installed to run the script. Once the script is executed a "dataset" folder will contain the dataset organized in a way that is compatible with the Python code. The data was extracted from the zip archive downloaded from Kaggle.
The dataset is then loaded onto the Jupyter notebook, this is accomplished using the OpenCV library and the imread() function. This function reads images and converts them into matrices of integers in the interval from 0 to 255. Using the cv2.IMREAD_GRAYSCALE flag it is possible to import the image as a simple matrix with only one channel: no red, green, and blue. The BMP3 file format is impossible to read using the Tensorflow functions to decode BMP files, this is why OpenCV was used. Then, as described in the Dataset section, it's necessary to rescale the images to have them all of the same width and height. This is made possible using another OpenCV function: the cv2.imscale function. We then proceed to rescale the images to the desired 32x32 size. It is not needed to fix the aspect ratio with padding since the images were already somewhat squares, with only a few pixels of width and height difference. This is done for all images, furthermore, the labels of the images are taken directly from the folder structure, thus an image inside the CAP_ALPHA folder (upper-case alpha letter) will have the CAP_ALPHA label.
As described in the Method section, we then proceed to normalize the images, by subtracting the mean activity for all pixels of an image ad dividing by the standard deviation, this can be easily done since OpenCV images are NumPy arrays, so we can use the NumPy mean() and std() function to compute the mean and standard deviation. This is a code snippet that illustrates the normalization of images:

```
mean_px = img.mean().astype(np.float32)
std_px = img.std().astype(np.float32)
img = ( img - mean_px ) / std_px
```

The result of this dataset preprocessing are two Tensorflow tensors, one containing the images and one containing the label for each image. The image at position i in the images tensor, has the i-th label in the labels tensor. Finally, using the Keras function to_categorical the labels are converted from numeric values (numbers 1 to 48), to one-hot encoding values.
Once images and labels are loaded into the notebook, the next step is creating the TensorFlow Dataset (tf.data.Dataset), a TensorFlow class for representing the

dataset with a powerful API to apply further preprocessing and for iterating through the training, validation, and test examples.
To accomplish this we use the from_tensor_slices((images, labels)) method, which creates a dataset of (image, label) pairs. We then apply some preprocessing functions with the map() method, which applies the function for every data point. We use the map() method to reshape further the images in the (32,32,1) shape, this is because the 2D convolutional layers of Keras take as inputs in the form of (dimension1, dimension2, channels), since our images have only one channel the last element of the shape is 1. Then we split the data using the take() and skip() methods of the Dataset class, we obtain a training dataset a validation dataset, and a testing dataset. Finally, we apply the dataset augmentation strategies described in Method using the map() method. To realize dataset augmentation we use Tensorflow Keras layers RandomRotation, RandomZoom, RandomTranslation. We then use the shuffle() method to shuffle the examples in the training dataset and the repeat() method so that when the iterator reaches the end of the dataset it starts again, thus we can train for multiple epochs.
Finally, one batch of the training dataset, with the corresponding labels, has then been visualized.

## 5.2 Models implementation

The models are defined inside Python functions that return the model, these functions take as input the shape of the input layer and the shape of the output layer (which is the number of classes). The models are implemented following the description of the Method section and can be consulted in the Jupyter notebook. The Keras API is used to implement the model in Tensorflow. The classes used are Dense, for dense fully connected layers, and Flatten as a layer that simply converts a 2D output of a previous layer to a 1D vector that can then be given as input to a Dense layer. Conv2D and MaxPooling are the classes used to implement convolutional layers and max-pooling layers, here it is possible to specify the number of kernels, kernel size, stride, etc. Activation functions are specified for each layer and all of them have ReLu, except the output layer which has a softmax activation function. The Model and Sequential classes contain a sequence of these layers. Then we compile the models using compile() method and specify the Adam optimizer, the loss function (categorical cross-entropy), and the metrics to record. Using the summary() method it's possible to visualize a table that summarizes the model briefly. Finally, the model is trained using the fit() method, which needs the Tensorflow tf.data.Dataset as a parameter as well as the number of steps per epoch, the number of epochs, the callbacks that will be used during the model's training (these are used to basically implement early stopping and the variable learning rate); then validation tf.data.Dataset needs to be specified as well as the number of validation steps, at the end of each epoch it's in fact possible to see the validation

accuracy and loss for this validation dataset. The validation dataset is also necessary for early stopping.

## 5.3 Optimization implementation

The general APIs to implement the model have been specified in the previous section, however we left out the implementation of the optimization strategies described in the Method section. The regularizer is implemented for each layer that requires it using the TensorFlow Keras function l2(lambda) with lambda = 0.0005. Dropout, BatchNormalization are implemented as layers in the Keras model using the tf.keras.layer.Dropout class (which also takes as attribute the probability of dropout) and the tf.kear.layer.BatchNormalization. After a BatchNormalization we use the Activation layer which applies the ReLu activation function. These layers are used in the BetterLeNet5 and LeNet5v2 models. Finally, EarlyStopping and ReduceLROnPlateau from the tf.keras.callbacks module are two functions used as callbacks that implement respectively early stopping and the variable learning rate for the model, that as we have explained, is reduced each time the algorithm encounters a plateau.

## 5.4 Metrics

Precision and Recall are implemented using tf.keras.metrics.Precision and tf.keras.metrics.Recall. These are used during training and evaluation. During training, using the fit() method, for each epoch the results of loss and accuracy for both training and dev set are recorded and stored in a data structure that will then be given as the return value of the fit() method. It's then possible to consult the values of loss and accuracy. When instead trying to get results for the final model, the method to use is the evaluate() method, which is run with the test set. This method returns the values of loss, accuracy, precision, and recall for the given model on the test set. Finally, as we have specified in the Dataset section, some classes will be merged and an evaluation on these new merged classes (but using the same model) will be performed. To implement this step more functions were required. First of all, it's necessary to compute the predictions for a given input, this is accomplished using the predict() method, then we change the prediction so that some predictions are merged (e.g.: upper case kappa is the same prediction as lower case kappa). This reduces the number of total categories to 37. The ground truth labels are also merged to 37 classes. This merging process is described in the Dataset section with a table. Then, after merging the predictions and labels to the new categories space, new functions for precision, recall, and accuracy have been realized. These implement the same formulas shown in the Preliminaries and Method section, which are also the formulas used by the Tensorflow classes. The implementation of the confusion matrix has instead been accomplished with the

confusion_matrix function imported from sci-kit-learn, while the visualization of the confusion matrix has been realized using Matplotlib and a function inspired from an example of a function shown in the sci-kit learn website. All plots have been realized with Matplotlib and they consist of histograms of data, plots of loss, and accuracy over epochs, confusion matrices, and a batch of data with the ground-truth label and the predicted label. From the confusion matrix it's also easily possible to compute the precision and recall using the same formula described in the method section.

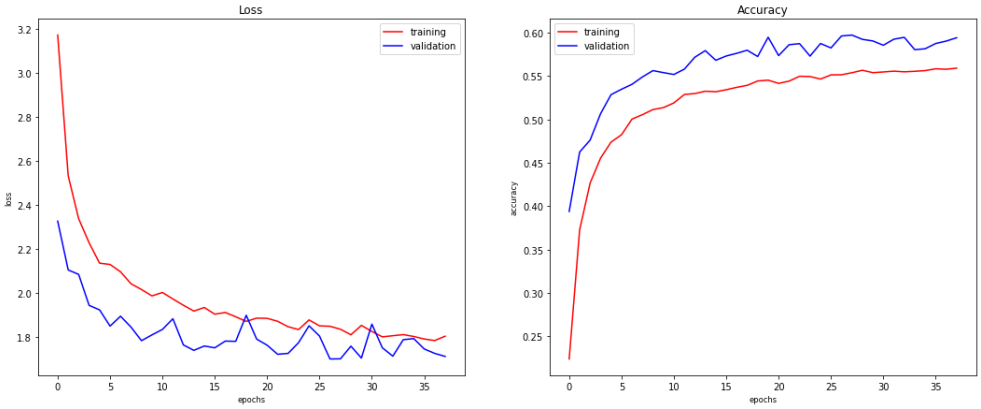
# 6 Experiments and results

These models have been run two times, once with 50 maximum epochs (and early stopping with patience 5) and once with 100 maximum epochs (and early stopping with patience 10). Of these two experiments runs only the best run for each model has been kept. These best models are those for which the results are shown. For the ANN, LeNet5, and LeNet5v2 models the weights trained with 100 epochs are the best ones, while for BetterLeNet5 the weights trained in 50 epochs achieved a better result. The training has been accomplished also using a GPU to speed up computation.

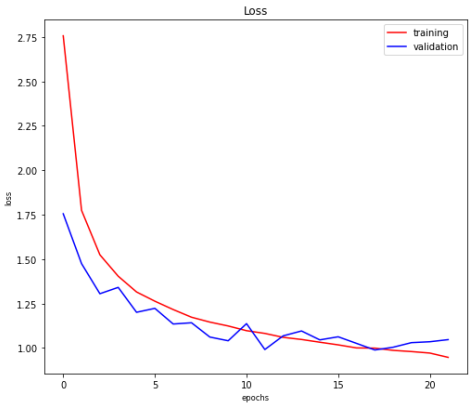
**Training metrics**

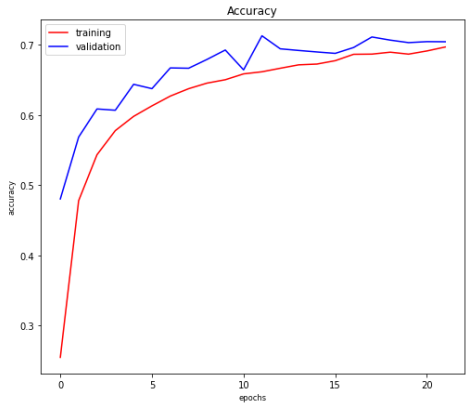
The first metrics are those of loss and accuracy during training. As expected the loss decreases as the training progresses, while the accuracy increases. It's also possible to notice that the validation accuracy and loss are lower than the training accuracy and loss, this can seem strange since the model should have better results on training, but considering dropout and data augmentation, which happen only during training, it's reasonable to think that the training process is made harder for the model, thus the training accuracy and loss are worse than its validation counterpart. During validation in fact all units are available to the network to compute the result. For each model:

- ANN

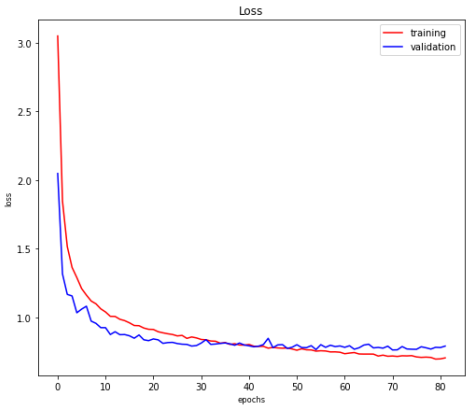


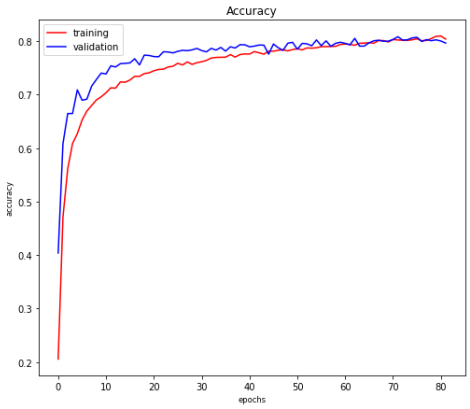

- LeNet5



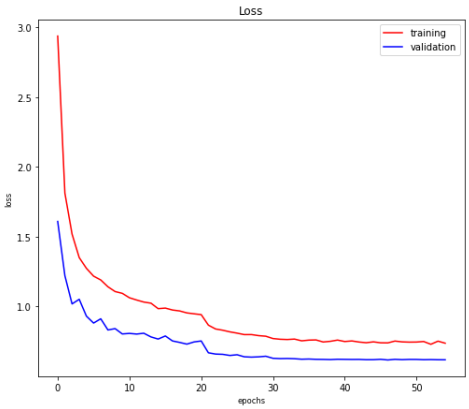



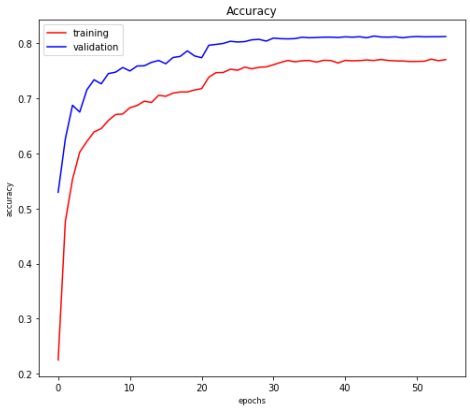

- LeNet5v2




- BetterLeNet5



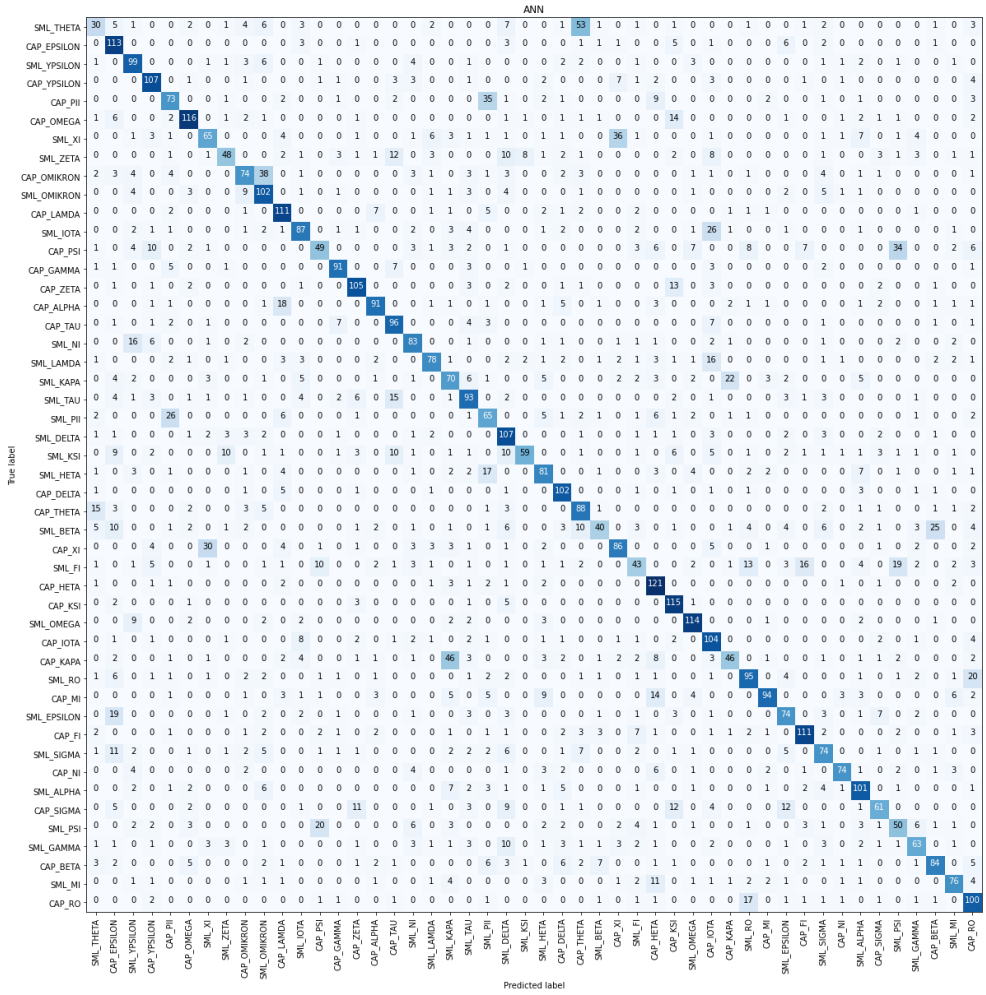

The best models are the convolutional networks, especially those which are both deeper and wider and which employ some regularization technique, these are LeNetv2 and BetterLeNet.

**Evaluation metrics**

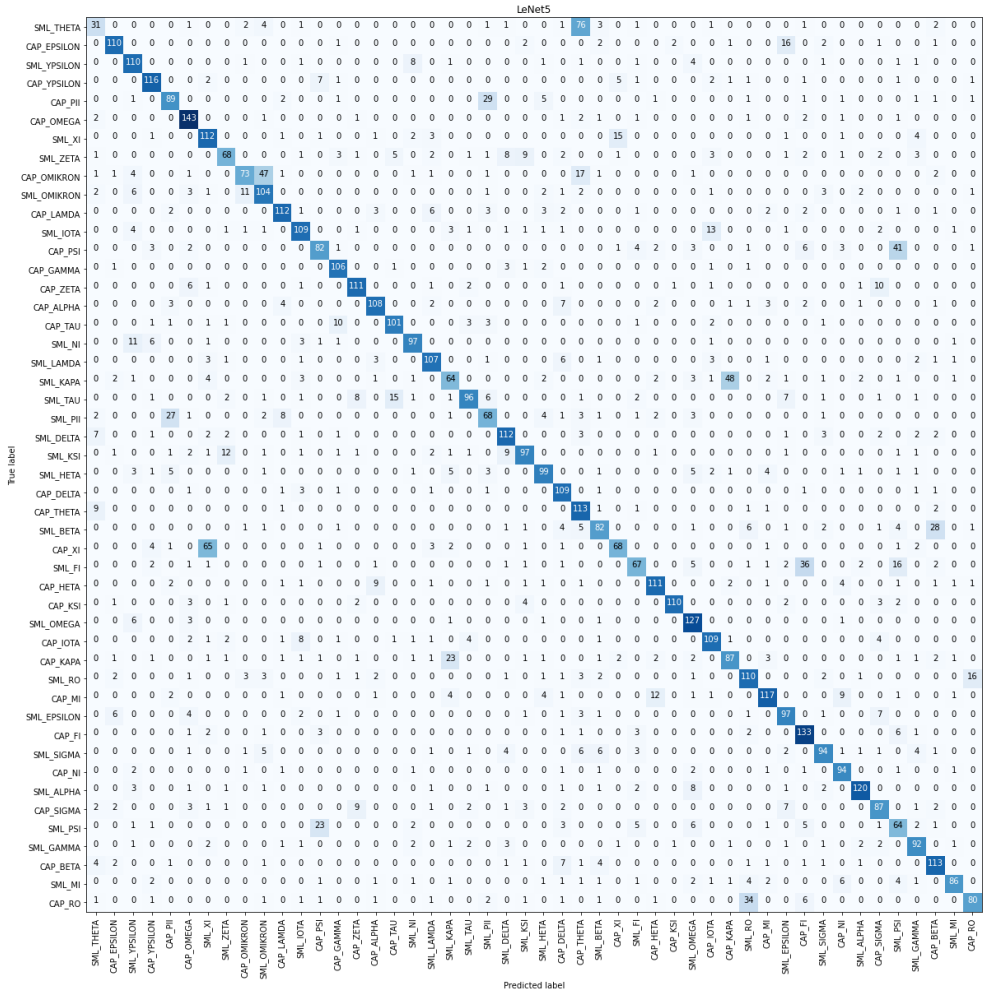
Then we have the tables of evaluation for each model and their confusion matrices:

| Model | Loss | Accuracy | Precision | Recall |
|---|---|---|---|---|
| ANN | 1.492 | 0.622 | 0.712 | 0.545 |
| LeNet5 | 0.901 | 0.726 | 0.784 | 0.689 |
| BetterLeNet | 0.400 | 0.880 | 0.903 | 0.861 |
| LeNetv2 | 0.737 | 0.805 | 0.842 | 0.782 |

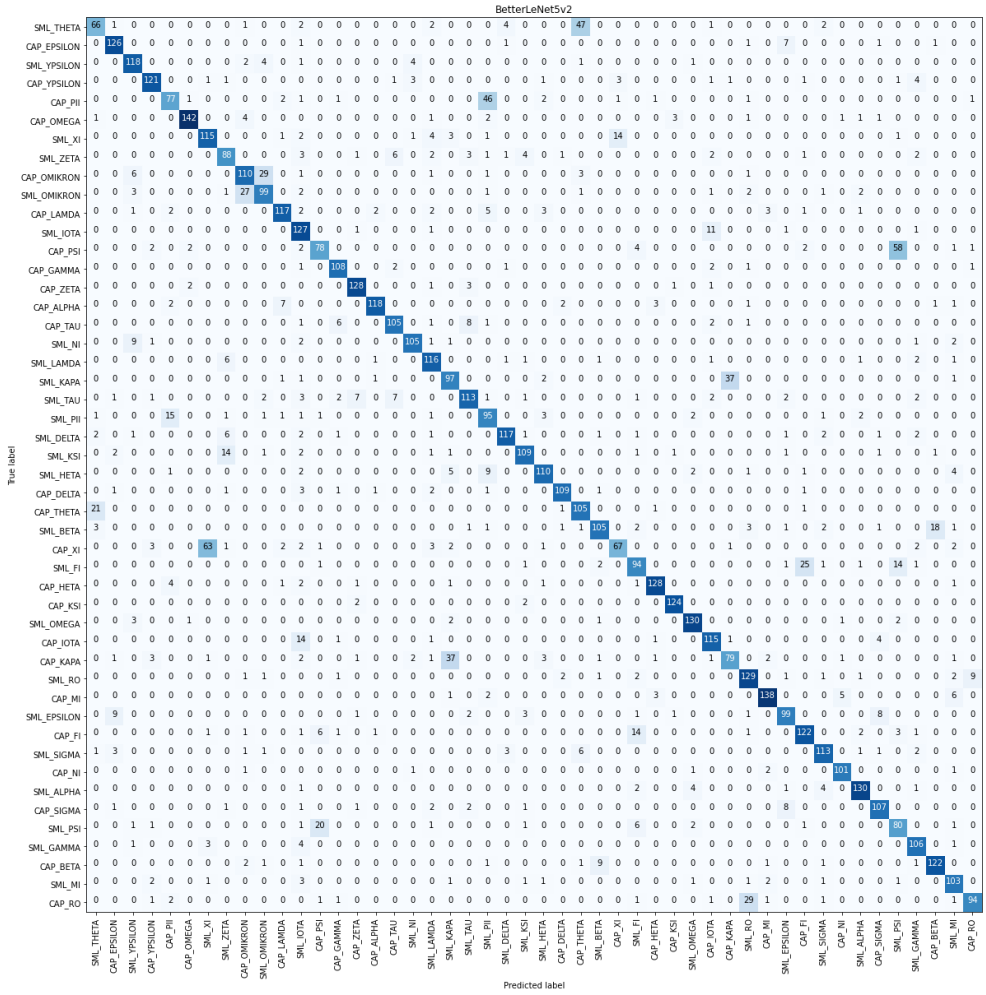
- ANN



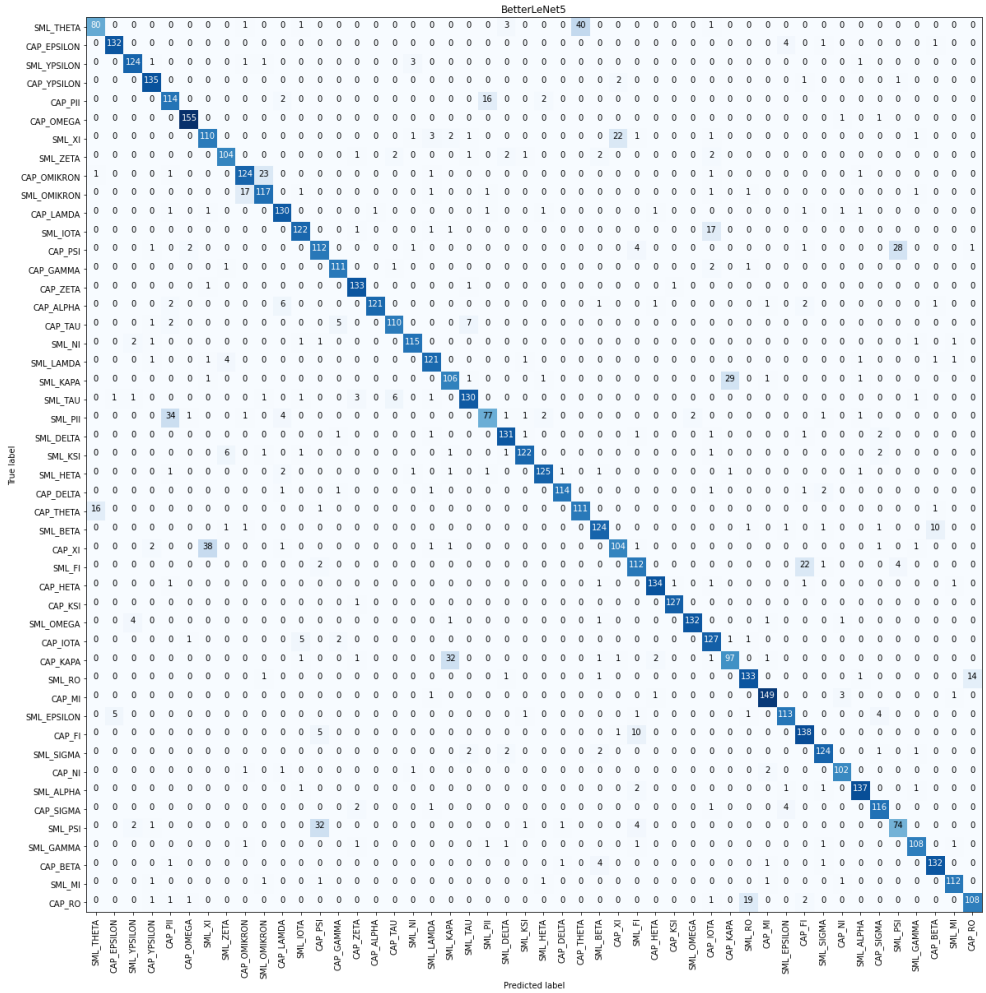

- LeNet5


LeNet5
True label
Predicted label
SML_THETA
CAP_EPSILON
SML_YPSILON
CAP_YPSILON
CAP_PII
CAP_OMEGA
SML_XI
SML_ZETA
CAP_OMIKRON
SML_OMIKRON
CAP_LAMDA
SML_IOTA
CAP_PSI
CAP_GAMMA
CAP_ZETA
CAP_ALPHA
CAP_TAU
SML_NI
SML_LAMDA
SML_KAPA
SML_TAU
SML_PII
SML_DELTA
SML_KSI
SML_HETA
CAP_DELTA
CAP_THETA
SML_BETA
CAP_XI
SML_FI
CAP_HETA
CAP_KSI
SML_OMEGA
CAP_IOTA
CAP_KAPA
SML_RO
CAP_MI
SML_EPSILON
CAP_FI
SML_SIGMA
CAP_NI
SML_ALPHA
CAP_SIGMA
SML_PSI
SML_GAMMA
CAP_BETA
SML_MI
CAP_RO

- LeNet5v2




- BetterLeNet5

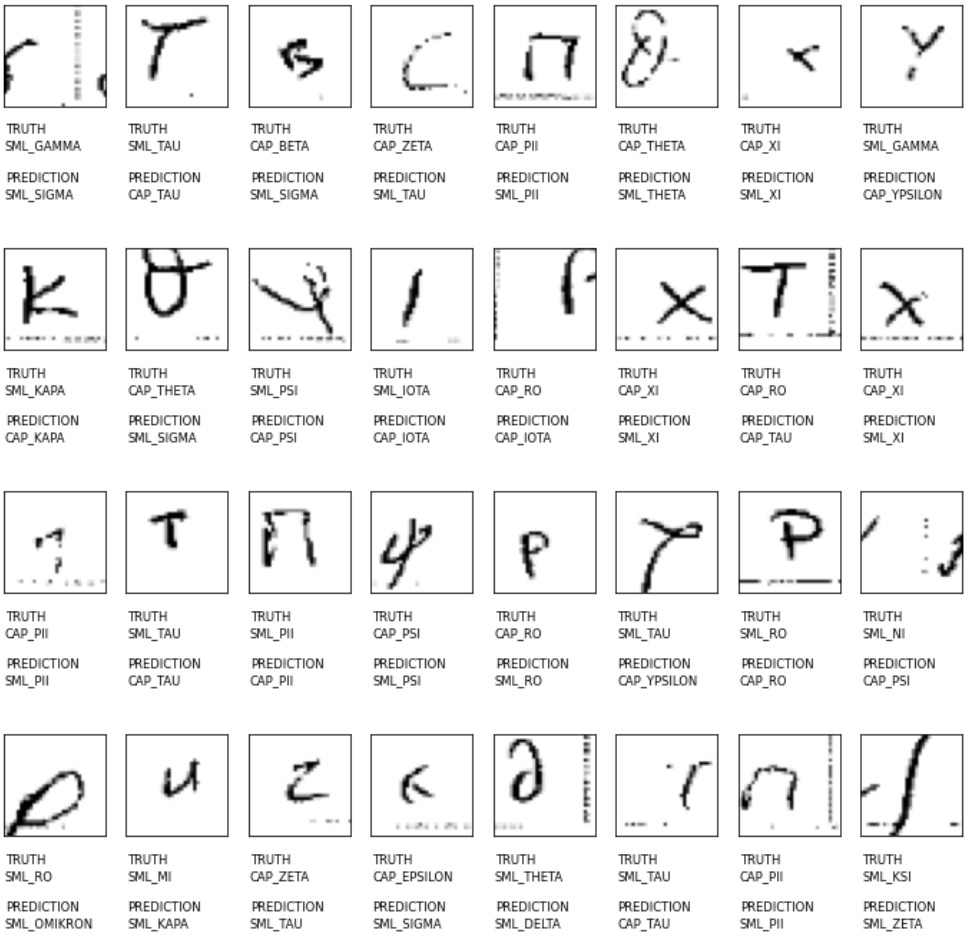



It's clear that the confusion matrices have a pattern of misclassifying the similar classes (e.g.: lower case kappa and upper case kappa), this is why some classes have been merged, as it is done also in [8].

**Misclassification examples**

As said above, misclassification happens for similar classes, like indistinguishable upper case and lower case letters. This is the bulk of misclassification, the other cases are minor compared to this kind of misclassification. Still there is an issue of some similar characters being misclassified (characters which share some significant features, e.g.: lower case zeta and ksi, upper case psi and fi). The final and least common misclassification issue is those cases where characters are different but there may be some issue or extreme case with the data (the image is cut, or even a wrong ground truth label). The cases where the prediction is clearly different from the image of the digit are negligible.

Here is an example of 32 misclassified instances:



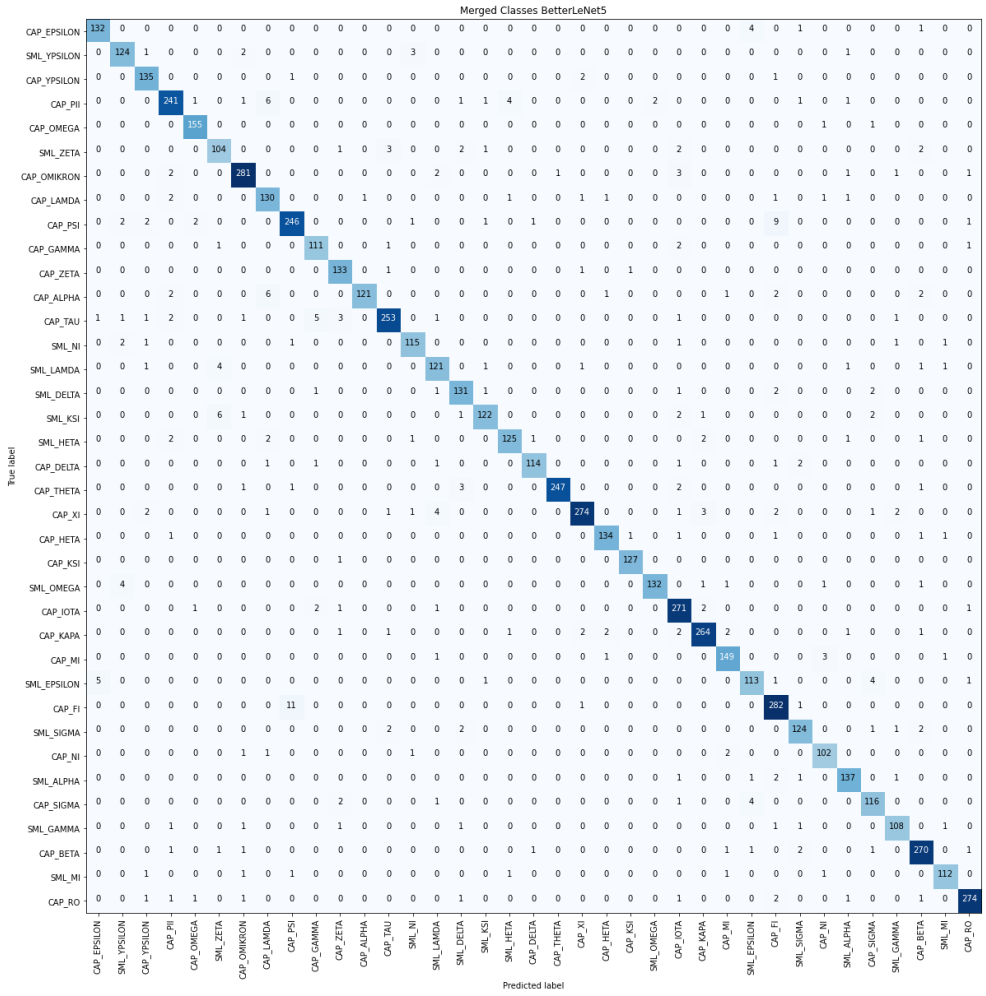

**Merging classes for the best model**

Merging the classes here are shown two results: the table of evaluation for all models and the confusion matrix for our best model, BetterLeNet5. The table shows an overall better performance, as predicted, but it also shows the best result so far achieved, while the confusion matrix show significant betterment of classification (the merged classes will of course appear darker as double the samples have been classified as that class):

| Model | Accuracy | Precision | Recall |
|---|---|---|---|
| ANN | 0.699 | 0.700 | 0.694 |
| LeNet5 | 0.815 | 0.810 | 0.810 |
| BetterLeNet | 0.948 | 0.946 | 0.946 |

| | | | |
|---|---|---|---|
| LeNetv2 | 0.892 | 0.888 | 0.888 |

This table shows the most impressive result yet, 94.8% accuracy for the BetterLeNet5 classifier.




What it's possible to notice here is that the bulk of misclassification doesn't happen anymore, however, there are still some instances of misclassification, a minor number of which depend on wrong ground-truth labels, while the majority of these errors come from the fact that the characters share some relevant features, such as upper case psi and upper case fi (they clearly share features such as the straight vertical line and some curved almost "o" shaped lines), lower case zeta and lower case ksi (it's a wiggly vertical line), lower case ni and lower case ypsilon, lower case heta and lower case pii, lower case gamma and capital xi. Sometimes these characters, just as the upper and lower case variants of the same character, are

difficult to recognize, since the handwritten script has a level of ambiguity. This increases the complexity of the task when trying to reach accuracy scores of 99% and over. A solution can be increasing the dataset size, collecting more data of handwritten greek characters, this will positively influence the model's capacity to recognize characters, even when similar.

**Example of classification of a batch of data**

Finally here is an example of classification using the best model (without merging classes):


As we can see the model correctly recognizes almost all characters. When merging similar classes the accuracy is even higher. The only error in this batch is slightly different from the errors of the sort described in the section above, here a capital alpha is misclassified as a capital fi, the alpha is not clearly recognizable (it's in the upper left corner) and there is also a vertical line in the middle of the image (a line that comes from the paper module used to collect data), a vertical line in the middle is an important feature of capital fi. This could be the reason for misclassification.

All other examples are correctly classified, except for lower case fi, beta, and pii and upper case fi, beta, and pii which would be considered correct if classes were merged.

# 7 Conclusions

The project achieves great accuracy for character recognition of handwritten letters, up to 94% with merged classes, and using some optimization methods. This is a great result that shows how this ML task is viable and can be applied productively in a number of real-world fields. To further confirm the results, multiple experiments can be performed and the variance of the best algorithm presented can be computed. However, these models can be improved further. Improving the quality (clearer pictures of digits, removing unnecessary features, etc.) but most importantly the quantity of data in the dataset, will guarantee greater accuracy, this is especially true for bigger models which due to their greater parameter space require more data to train. An even greater search for the best hyperparameters and model architecture using methods such as grid search, random search, etc. can greatly improve results. Finally, trying even more techniques to reduce variance and bias.

# 8 References

1] Used dataset https://www.kaggle.com/datasets/vrushalipatel/handwritten-greek-characters-from-gcdb
2] Margaronis, John, et al. "GCDB: a character database system." Proceedings of the International Workshop on Multilingual OCR. 2009.
3] Russell, Stuart J. Artificial intelligence a modern approach. Pearson Education, Inc., 2010.
4] Sokolova, Marina, and Guy Lapalme. "A systematic analysis of performance measures for classification tasks." Information processing & management 45.4 (2009): 427-437.
5] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.
6] Hinton, Geoffrey E., et al. "Improving neural networks by preventing co-adaptation of feature detectors." arXiv preprint arXiv:1207.0580 (2012).
7] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).
8] Vamvakas, G., et al. "Greek Handwritten character recognition." Proceedings of the 11th Panhellenic Conference in Informatics. 2007.