The German Traffic Sign Benchmark (GTSB) Traffic Sign Classification

Diogo Silva, Nº MEC.: 89348
DETI
Universidade de Aveiro
diogo04@ua.pt

Pedro Oliveira, Nº MEC.: 89156
DETI
Universidade de Aveiro
pedrooliveira99@ua.pt

***Abstract*—GTSB is a multi-class, single-image classification challenge held at IJCNN 2011. Automatic recognition of traffic signs is required in advanced driver assistance systems and constitutes a challenging real- world computer vision and pattern recognition problem. A comprehensive, lifelike data-set of more than 50,000 traffic sign images has been collected. It reflects the strong variations in visual appearance of signs due to distance, illumination, weather conditions, partial occlusions, and rotations. The data-set comprises 43 classes with unbalanced class frequencies.**

***Index Terms*—Supervised Learning, Classification, Deep Neural Networks**

I. Introduction

This report serves as a compendium and thorough explanation of all the techniques, methodologies and algorithms adopted for the elaboration of the first TAA - Tópicos de Aprendizagem Automática (Topics of Automated Learning) - project at Universidade de Aveiro. In sum, the task consisted in the application of Machine Learning techniques, either developed during class or self-taught, in the solving of one of the several problems, previously proposed by the course's head teacher, Pétia Georgieva.
More concretely, the issue we decided to tackle was entitled *The German Traffic Sign Benchmark (GTSB)* and consisted in the design and creation of an algorithm capable of accurately identifying traffic signs. Right from the start this topic peaked our curiosity due to its prevalence and importance in one of the most commonly known and talked about applications of Machine Learning nowadays - The creation of self-driving, AI-powered, autonomous cars. All the buzz we hear about on a daily basis served both as inspiration as well as a driving force that made was want to focus our utmost efforts into achieving the best possible solution to the problem.
In the following sections we will be describing our approaches to the problem at hand, starting with the analysis of the given data-set in section 2, we will then proceed to talk about the way we decided to view and tackle the issue, and the reasoning behind the adoption of a Neural Network architecture as a solution, as well as showing the initially obtained results in section 3. Section 4 focuses on the methodologies utilized to improve both the individual images and the data-set as a whole whilst section 5 is used to report adjustments made to the Neural Network's model and hyper parameters. Finally, section 6 presents both the final combination of algorithms chosen and a brief conclusion alongside a discussion of the results obtained.

II. Data-set Analysis

A. Data-set description

The data utilized for the conclusion of this project, made available in this link, consists in a total of 51883 images of varying dimensions, distributed over three different sets - Meta, Test and Train. Each image contains one of forty-three possible traffic signs and is included in the directory correspondent to the set it belongs in. In practice, what this means is that there are forty-three different classes that an image can be classified as.
The **Meta** set contains an example of each of the traffic signs present in the other sets' images and is used mostly by us as a way to know what the appearance of the signs we are attempting to identify. All images in this directory are named a number from 0 to 42, indicating what class they belong to.

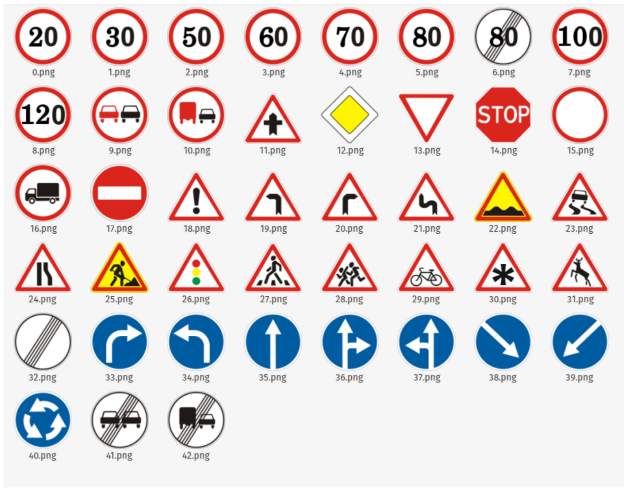

Fig. 1. All images contained in the Meta set

The **Test** set contains 12631 different images of distinct signs all shuffled together. These are the images that we will be using to test the accuracy of our model with, hence the quality - generally measured in terms of lighting conditions, blurriness, rotation and/or focus/scale of the sign itself - differs greatly from image to image in order to assure that the data-set is as diverse and lifelike as possible, parting from the idea that in real life visibility conditions are in constant change and so we should able to accurately classify signs, regardless of how easy it is for us to see them.

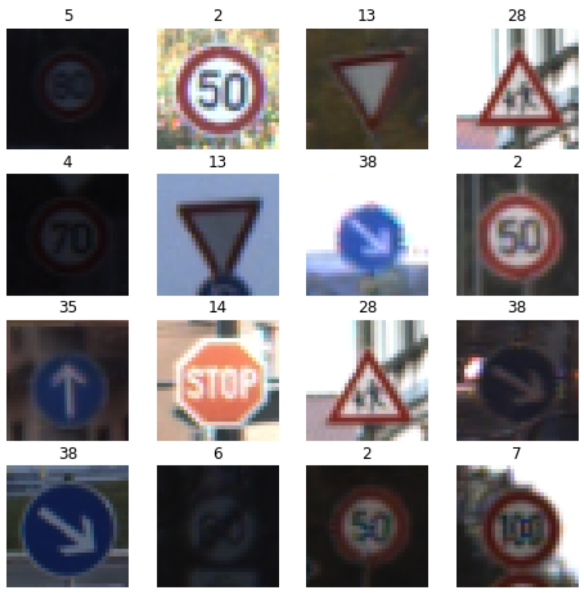

Fig. 2. A subset of images from the Test data-set. The class of each image can be seen above itself

Finally, the **Train** set contains 39209 images divided into 43 folders corresponding to what class the image belongs to. Just as the Test set, images differ greatly in quality to provide an ample range of examples, of varying difficulty, to train our model with.

Besides the three pre-organized sets, our data-set also provided us with three CSV files containing information on the images present in each set - indicating their width, height, relative path and most importantly, what class they belong to.

B. Statistical analysis

As aforementioned, our data set contains images that portrait one out of forty three possible traffic signs, however, it should be stated that we do not possess an equal number of representations of each traffic sign. This issue reaccures on both the Test and Train set, but it is particularly troublesome when it comes to the latter, since this is the data that will be fed to our model. Effectively, this means that we will be training using skewed data, which can lead to the underrepresentation of certain classes. Furthermore it could also lead to problems when creating our Cross Validation subset. Both these topics are expanded upon in section IV.

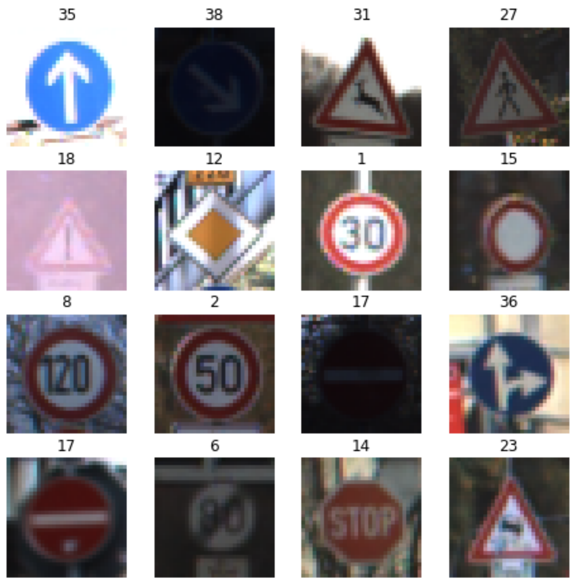

Fig. 3. A subset of images from the Train data-set. The class of each image can be seen above itself

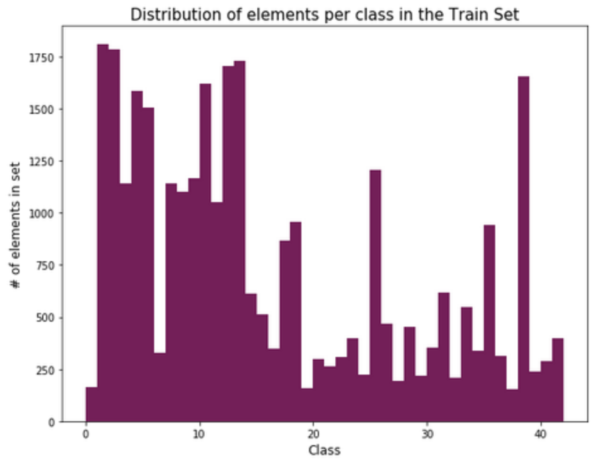

Fig. 4. Number of images per class in our training set

III. Deep Neural Network and initial configuration

A. The problem

As stated previously, the topic of traffic sign recognition boils down to a classification problem, where each image of the traffic sign corresponds to a single category, represented here as a number between 0 and 42.
Before trying to come up with our own solutions, and knowing how prevalent this challenge is, especially since it had been introduced during the IJCNN (International Joint Conference on Neural Networks), we researched the problem and read several articles detailing possible approaches. While they varied in the hyper-parameters, such as the used learning rate, they all shared one thing in common: the usage of a Deep Convolutional Neural Network, composed of convolutional

blocks and fully connected layers. The majority of results from these articles looked promising too: most of the articles we saw had a test accuracy greater than 93% [1] [3] [4], with one even managing to get a 98% accuracy [2].
So it was apparent that we had to implement a convolutional neural network to get the best result. A vast majority of the articles we found specified the usage of a LeNet5 network, a concept we'll explain in the following section, but that, in summation, boils down to a classic Convolutional Neural Network architecture, designed to recognize pixels from images with minimal preprocessing [11].
As for our optimizing function, we opted to go with Adam, basing ourselves in the reports we have looked at. It's an optimizing function that's said to have the advantages of an adaptive gradient algorithm, namely the way it maintains a per-parameter learning rate; and the root mean square propagation, namely how it calculates the learning rate based on an average of the magnitudes [12]. Finally, we have to talk about the categorical crossentropy loss function: it's based on the binary logistic regression loss function, adapted to include all the different categories.

$$-\sum_{j=0}^{M}\sum_{i=0}^{N} y_{ij} * log(\hat{y}_{ij})$$

B. LeNet5 Architecture

The LeNet5 network consists of two convolutional blocks, composed of a convolutional layer and an average pooling layer; a flattening layer, two fully connected layers, and one softmax layer. With the exception of the last layer, all use a ReLU activation function: a simple function that changes the negative numbers to equal 0, while the positive numbers remain unchanged. It's a widely used function for neural networks since there is no complicated computations, and, due to it being a linear function for positive numbers, there's no risk for plateaus or saturation as there would be with other activation functions.

1) Convolutional Block: A convolutional block consists in a convolutional layer, made to recognize image patterns, like lines and figures, and an average pooling layer, to reduce the dimensionality of the input.

2) Flattening Convolutional Layer: A flattening convolutional layer is a layer with the goal of flattening the input, which means turning the complex matrix given into a vector.

3) Fully Connected Layer: The purpose of this layer is to decrease the dimension of the flattened input.

4) Softmax Layer: The last fully connected layer has the particularity of using a SoftMax activation function instead of the regular ReLU function. This function will turn the input nodes into probabilities that sum to one, ensuring an output vector that represents the probabilities for the classification of the inputted image [9].

C. Base Implementation

So now that we have the basis of what a Convolutional Neural Network looks like, it was time to implement it into code. We used the library Tensorflow and its Keras modules, as it provided a very high level way of building the neural network. The initial structure of the Neural Network we built has been illustrated in Fig. 5.

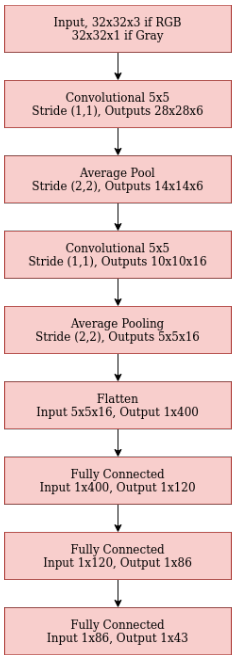

Fig. 5. Implementation of CNN

We then decided on the hyperparameters based on the articles we have seen, mainly a batch size of 200, a learning rate of 0.002, and 50 epochs.

D. Initial configurations and Graphs Used

It's also important to establish the metrics we used to check our progress. The most basic ones were the final accuracy and loss function values for each data-set (Train, CV and Test). Since these metrics are generic over all the data, e.g. the accuracy was the ratio of the number of correct guesses and the number of guesses, they quickly proved to be inadequate: since we had a great amount of classes, if the model fails miserably in one of the less represented classes (as would be expected), we could still be presented with a pretty good overall accuracy, hence not being able to detect that some classes may be suffering; with this in mind we added two other graphs that could better represent how our model was doing: a confusion matrix heat-map and a bar graph of accuracy per class.
The (relative) confusion matrix is a graphical representation of the relative frequency of what our model predicted the image's

class to be for each different class whilst the accuracy per class bar graph is a graph showcasing the overall accuracy for each individual class. These two representations will serve as a better basis to assert if our model is improving or not.
Finally, before moving on to our results, we should establish that before feeding the data to our model, we divided the training dataset, taking 20% of all the examples to create our Cross Validation Dataset. This division is completely random, not taking into account the number of imbalance in the dataset, as we'll analyse later. Finally, all datasets were shuffled before being fed to the model.

IV. Image Preprocessing and Data-set preparation

A. Description

Before we move on to a more in-depth analysis of how we played with and altered the base model talked about in the prior section, we will first be succinctly explaining all treatments applied to our base images as well as what modifications we made to our data-set. It should be mentioned that this data preprocessing was our first way of attempting to improve our model's results as, after the analysis of the images' overall quality, we felt that if we could somehow standardize and improve their visibility we would be facilitating the computer vision's identification of the images, hence leading us to a better outcome. The remainder of this section will tackle all of the techniques we tried, and their results.

B. Resizing

Firstly we started by trying to homogenize our images by standardizing their size. Initially the images could vary in both width and height, with high disparity between each other. We agreed upon a fixed size of 32x32 pixels as those were the most commonly used dimensions throughout the papers we researched. This step was an obligatory one as our Neural Network required be given an input of a standard size in order to function.
Following are the obtained initial results after resizing. Showcased in Table 1 are the overall results for accuracy and loss values over the utilized sets, whilst Fig. 6 and 7 report the relative confusion matrix heat-map and the accuracy per class, in that order.

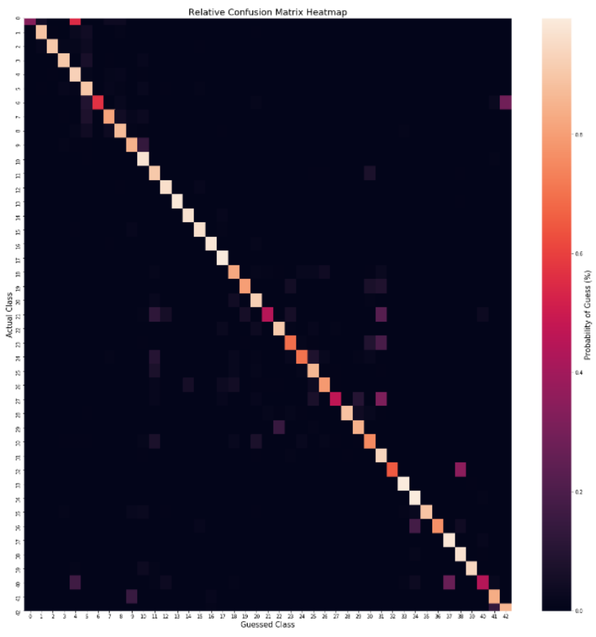

Fig. 6. Initial relative confusion matrix heatmap

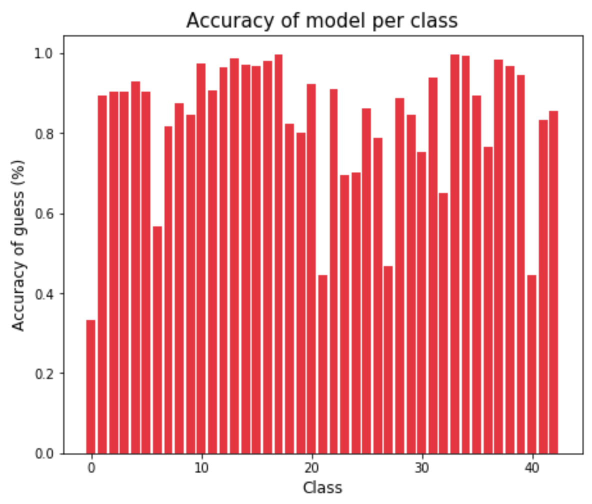

Fig. 7. Initial accuracy per class

TABLE I
Initial Accuracy and Cost results

Set	*Accuracy*	*Loss*
Train	0.9918069	0.031324406663282988
CV	0.96977425	0.2507336913898897
Test	0.89136976	1.538338404115782

C. K-Fold Cross Validation

It should be noted that, in the previous section, we presented results for a so called CV set. This is in fact our Cross Validation set, obtained after applying a 3-Fold on our dataset. We have mentioned that, initially, our dataset was already divided into 3 sets - Meta, Train and Test - but the Meta set was inconsequential and served no use in the overall training and testing of our model, so what we did was, as per the teacher's recommendation, take 20% of our images in the Train set, and use them to create a new Cross Validation (CV) set.
The reason for using a Cross Validation set is because it "helps us better use our data, and it gives us much more information about our algorithm performance" [8], allowing us to fine tune our model's parameters and choose the best one.
The way we did this at first was by randomly picking 20% of the images in the Train set, and transferring them into the new CV set. A problem with this approach, however, was that, due to the highly skewed data, it could happen that

the least represented classes in the Train set would be nigh nonexistent in the newly created set. This was obviously an issue, but for the results presented in the previous section, that was the way we formulated the CV set.

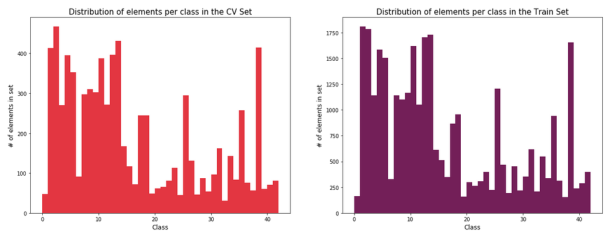

Fig. 8. CV (left) vs Train (right) class representation using random distribution

After some deliberation we changed our approach by creating an algorithm that would take 20% randomly chosen images of each class' subset in the Train set and transferring them into our CV set. Our aim was to create a CV set that would have a representation of each class similar to the one in our Train set, hoping that by assuring that each class is, at least minimally, represented in the cross validation set, our model's training would better fit the parameters and output a more accurate model. As shown below in Fig. 9, our CV set's silhouette using this technique is much more similar to the Training set's the overall and per class accuracy both improved compared to the previously obtained using the random distribution CV, as observed in Fig. 10 and 11.

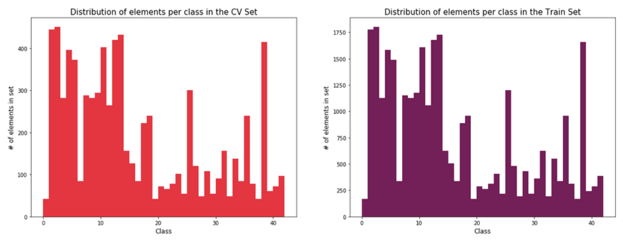

Fig. 9. CV with equal class distribution (left) vs Train (right) class representation

TABLE II
ACCURACY AND COST RESULTS USING CV WITH EQUAL CLASS DISTRIBUTION

Set	***Accuracy***	***Loss***
Train	0.9913606	0.06068361176813464
CV	0.97627854	0.26256666635979714
Test	0.9223278	1.8555431108631635

D. Conversion to Grayscale

Moving back to image preprocessing, we decided it would be a good idea to try to convert our images into grayscale. This was done both for two main reasons. The first was to attempt to decrease the total amount of data our model would have to process, since we would only have to work with images containing 32x32 pixels worth of data, as opposed to RGB images which each contain 3 channels of 32x32 pixels (one channel for each colour) [1], [2], [7]. The other reason was because, allegedly, "rejecting color information can even boost the final result" [1]. Effectively, what we verified in practice is that both premises held true, the processing speed for the 32x32 RGB images, averaging over 5 seconds per epoch, was reduced to about 3 seconds per epoch with grayscale images. As shown in Fig. 13 and 14 the results per class were similar to the ones before, despite the overall accuracy and loss, shown in Table III, having gone down a bit. We decided to keep using Grayscale for the remainder of this work.

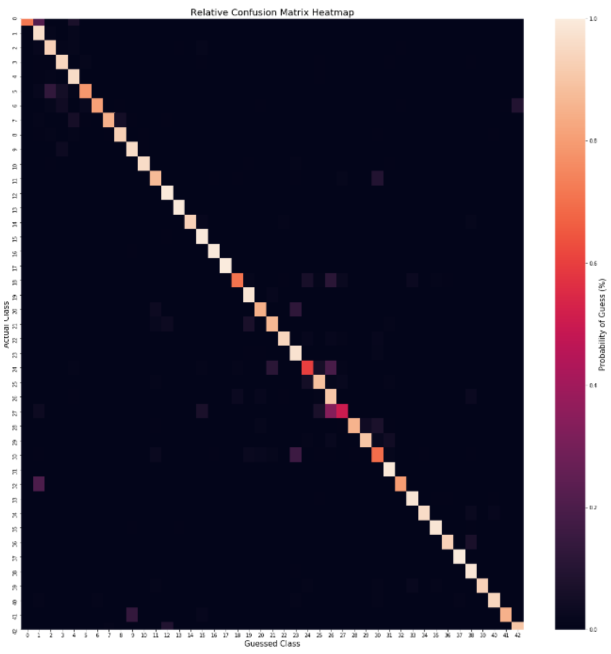

Fig. 10. Relative confusion matrix heatmap after changing the way we created the CV

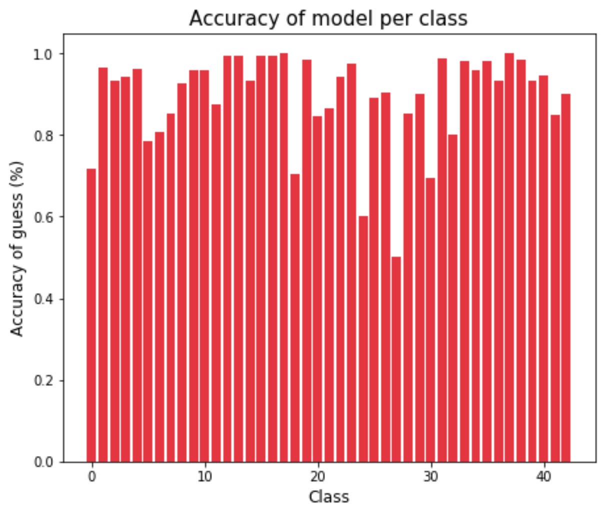

Fig. 11. Accuracy per class after changing the way we created the CV

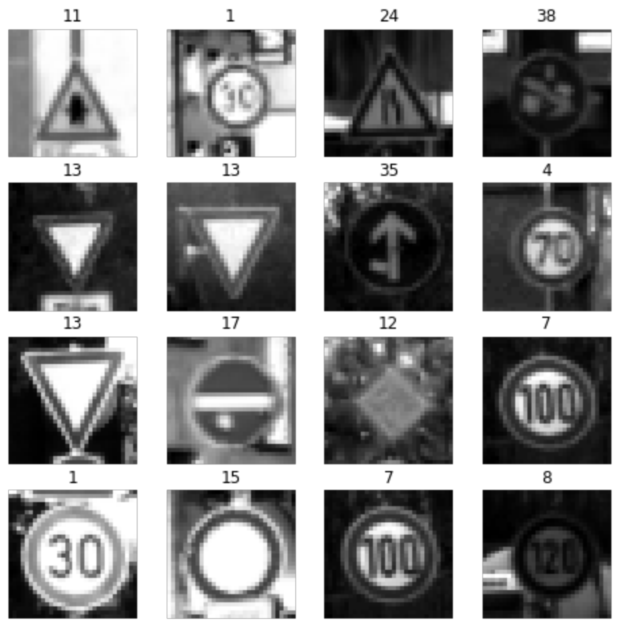

Fig. 12. A subset of images from the Test data-set after applying Grayscale conversion

TABLE III
GRAYSCALE ACCURACY AND COST RESULTS

Set	*Accuracy*	*Loss*
Train	0.99276334	0.02380184043415325
CV	0.9809973	0.1594154959352835
Test	0.9140143	0.899924260882272

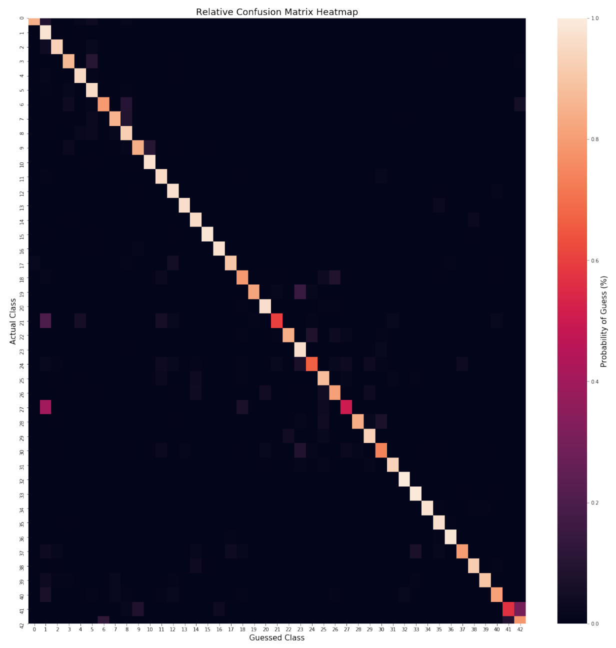

Fig. 13. Relative confusion matrix heatmap after applying Grayscale transformation

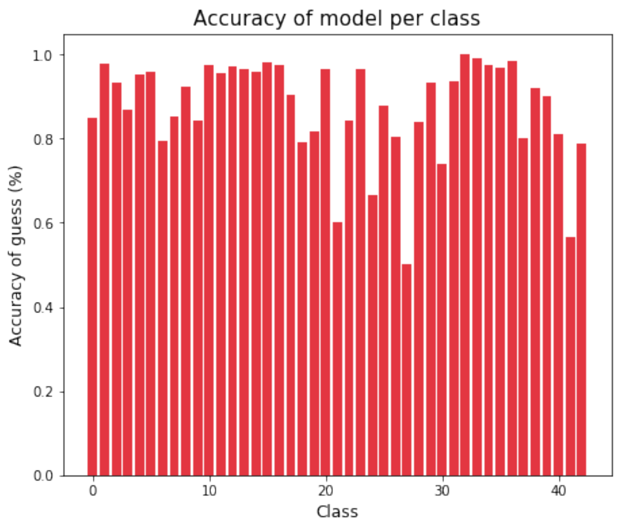

Fig. 14. Accuracy per class after applying Grayscale transformation

E. Normalization and Equalization

Overall, the results obtained after changing all images to grayscale were outstanding. Most classes were showing over 80% accuracy, with a lot of them even going over the 95%. There were, however, exceptions. Classes like class 27, had a lot of discrepancy comparatively to the rest. After analyzing some of the images in the Train set belonging to these classes we noted that they had particularly poor pictures in terms of visibility so we thought about trying to fix this issue. Subsequently, this lead us into doing some research about image treatment and quality improvement. Out of all techniques discussed in the internet, two in particular peaked our interest due to being mentioned on most reports we read about image classification using machine learning: Normalization and Equalization.

Both techniques produce similar results but they achieve them by applying different methods. Normalization simply consists in normalizing all pixel values in an image to make them fall into a given range (normally, between -1 and 1). On the other hand, equalization simply attempts to produce a histogram with equal amounts of pixels in each level of intensity. Up until now we were already applying a very simple form of normalization, that would, for each pixel of the image, subtract, and divide it by half the max intensity

$$\frac{PixelValue - 127.5}{127.5}$$

in order to make it fit into the [-1, 1] range, before feeding the images to our model, with the hindsight that working with numbers with a range as high as 255 would complicate the computer vision problem [1], [2]. However, we decided to also attempt other techniques of normalization and equalization in the hopes of producing images that would be more easily identifiable by our model following the same steps as some of our references, as well as trying other methodologies we found. We started by applying a simple **Histogram Equalization** on our images, which wielded the underwhelming results presented in Fig. 16, 17 and Table IV. Notice how we still have classes with a very decent accuracy rate, but some other classes actually presented worse results than before.

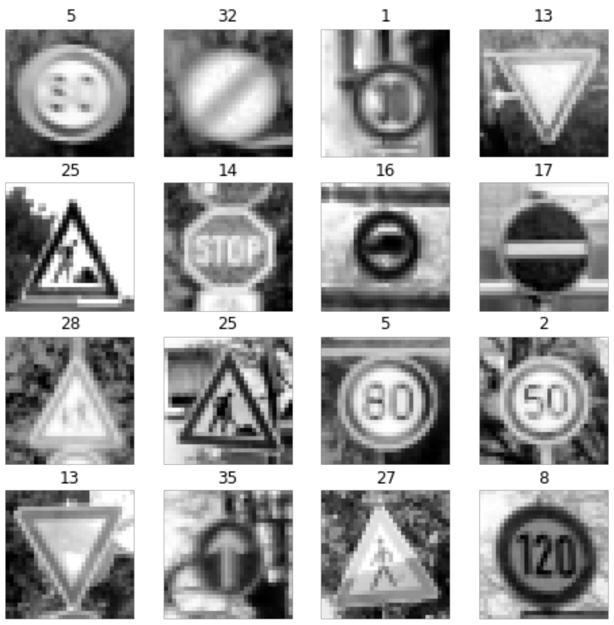

Fig. 15. A subset of images from the Test data-set after applying Histogram Equalization

TABLE IV
HISTOGRAM EQUALIZATION ACCURACY AND COST RESULTS

Set	*Accuracy*	*Loss*
Train	0.99553686	0.013727671238271915
CV	0.9767887	0.1320002153411293
Test	0.9055424	0.7252057633003602

As the last equalization technique ended up harming our model's performance, rather than helping it, we decided to try a different algorithm called **Contrast Limited Adaptative Histogram Equalization**. This method differs from the last because it's based around dividing each image into sections, and equalizing each of these individually. Supposedly, this should make it so our images have a better contrast to differentiate the sign from the background, but once again the results obtained were very disappointing (although better than after the last equalization), as once more the overall performance of our model was lowered, as shown in Table V and Fig 18 and 19.

After analyzing the resulting images of the last techniques we believe that the reason equalization might not have worked was due to the fact that, despite the algorithms really helping the darker images, making the traffic signs more noticeable, for blurrier and lighter images, the algorithm was actually

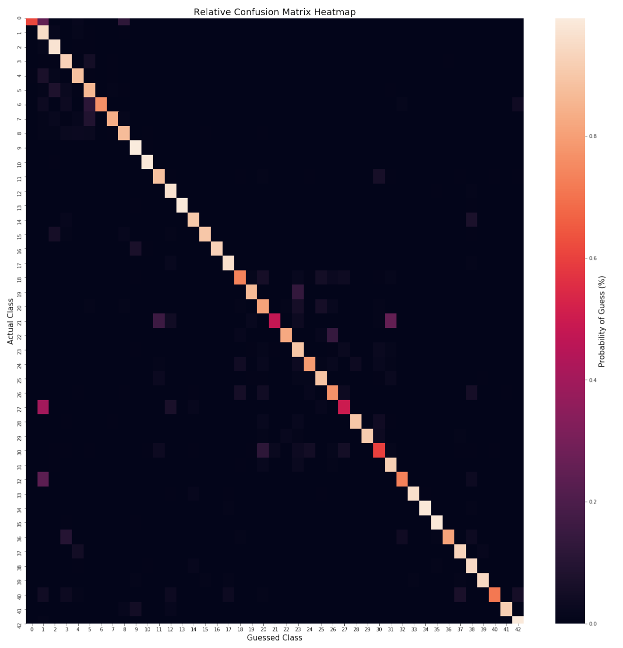

Fig. 16. Relative confusion matrix heatmap after applying Histogram Equalization to our images

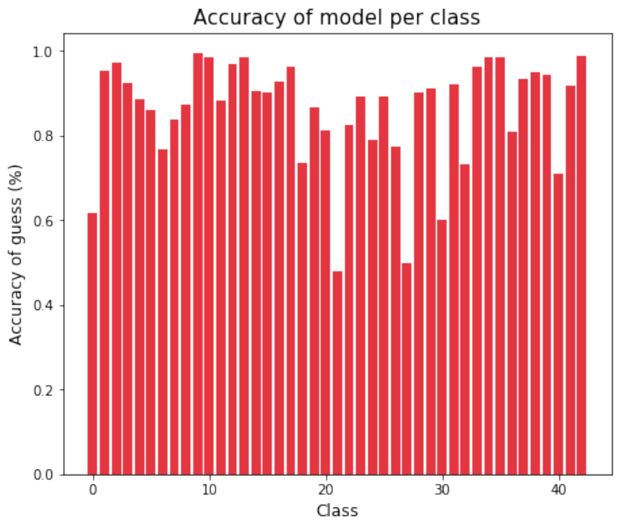

Fig. 17. Accuracy per class after applying Histogram Equalization to our images

TABLE V
CONTRAST LIMITED ADAPTATIVE HISTOGRAM EQUALIZATION ACCURACY AND COST RESULTS

Set	*Accuracy*	*Loss*
Train	0.9919982	0.028427384311084613
CV	0.9769162	0.1696492087726914
Test	0.9121932	1.0186666555292698

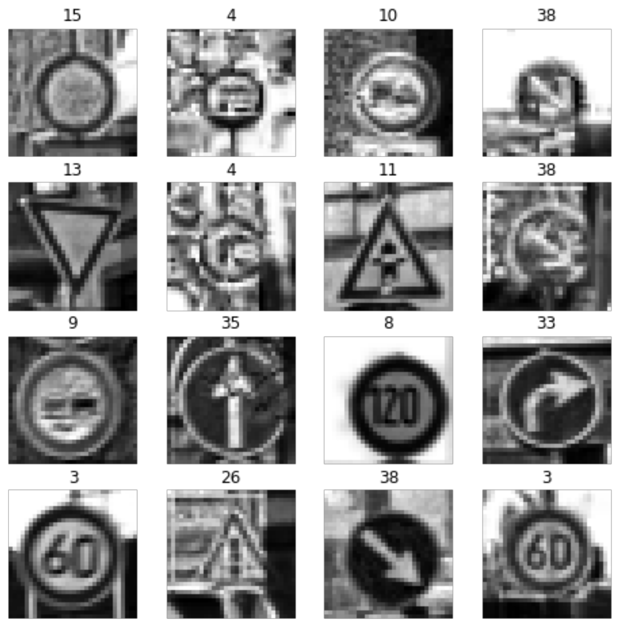

Fig. 18. A subset of images from the Test data-set after applying Contrast Limited Adaptative Histogram Equalization

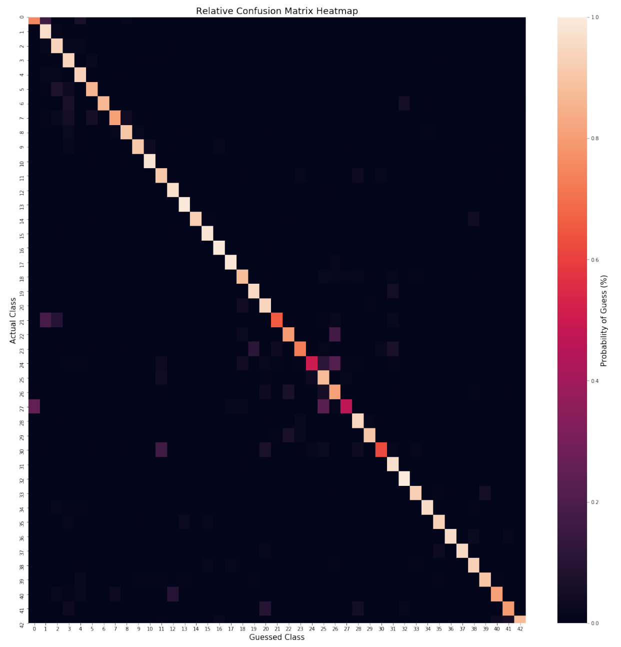

Fig. 19. Relative confusion matrix heatmap after applying Contrast Limited Adaptative Histogram Equalization to our images

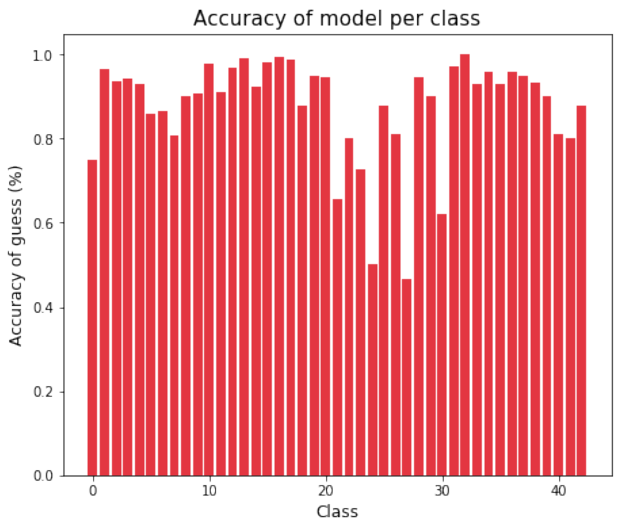

Fig. 20. Accuracy per class after applying Histogram Contrast Limited Adaptative Histogram to our images

hindering the overall visibility of the image. With this knowledge in hand we decided to try out a different technique of normalization. Rather than normalizing our images using the previously mentioned formula, we tried out OpenCV's algorithm which finds the lowest and highest intensity value pixel in the entire dataset, and determines a factor that will scale the lower pixel values closer to -1 and the higher ones to 1, supposedly creating a more uniform and appropriate to our dataset scaling of the values within that range. In practice, we did verify that the results obtained, observable in Fig. 21, 22 and Table VI, were better than the ones we had when using our original normalization, so for the following sections, we will be proceeding with this algorithm.

TABLE VI
OPENCV'S NORMALIZATION ACCURACY AND COST RESULTS

Set	***Accuracy***	***Loss***
Train	0.9967164	0.012512560527128116
CV	0.98622626	0.1027920527138769
Test	0.9360253	0.659059932324951

F. Contrast modification

With the surprisingly bad results of Contrast Limited Adaptative Histogram Equalization, we wondered if the reason as to why the algorithm failed was because it was inadvertently increasing the contrast of images that were already in high contrast, thus leading us to ruining perfectly acceptable images, in the hopes of fixing the bad ones. Figuring that what most images were suffering from was low exposure and focus on the traffic sign, we decided to try to, again, increase the contrast of the pictures, but this time apply that transformation only to images that were deemed to be in low contrast. This was accomplished through the usage of OpenCV's exposure detection method and contrast increase. However, it was to no

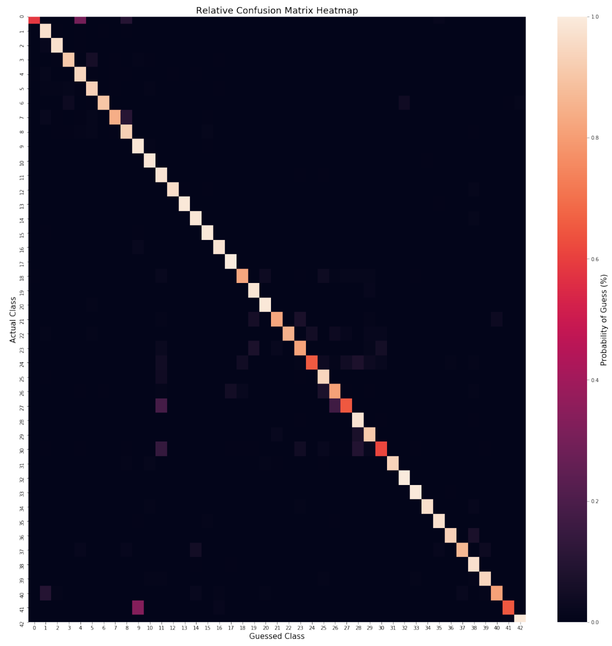

Fig. 21. Relative confusion matrix heatmap after applying OpenCV's Normalization to our images

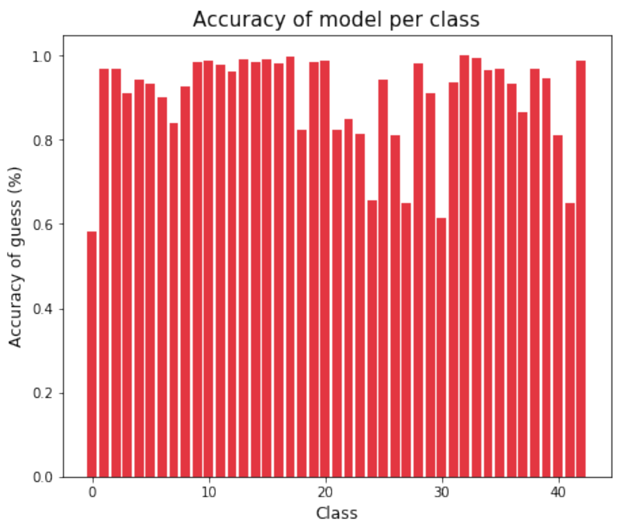

Fig. 22. Accuracy per class after applying OpenCV's Normalization to our images

avail as the results obtained were rather poor, as figures 23, 24 and Table VII illustrate.

G. Data Augmentation

After settling upon a choice of image preprocessing algorithms - Grayscale with OpenCV's Normalization - it was time for us to finally address the issue of data disparity between classes. Most all of our references mentioned artificially increasing the number of data for the underrepresented classes in order to achieve a Train dataset that would better represent all traffic signs. We attempted

TABLE VII
CONTRAST INCREASE ACCURACY AND COST RESULTS

Set	***Accuracy***	***Loss***
Train	0.99795973	0.006693505189067708
CV	0.9799771	0.11181414452724814
Test	0.88994455	0.7147907885689634

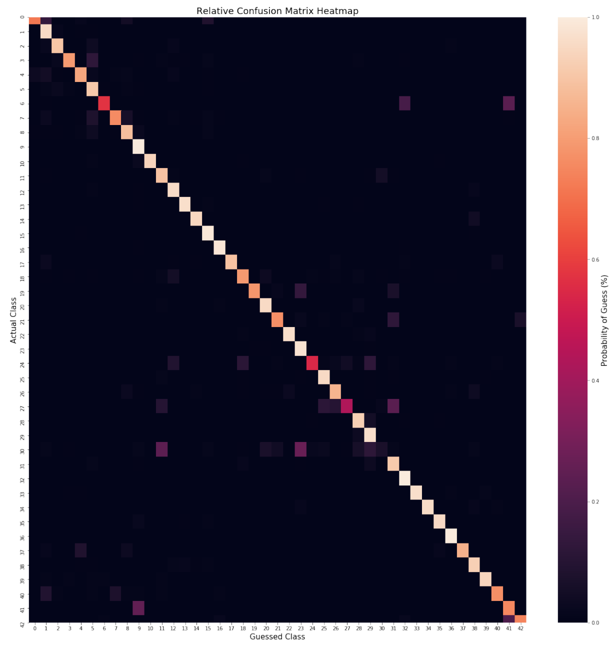

Fig. 23. Relative confusion matrix heatmap after applying Contrast modification

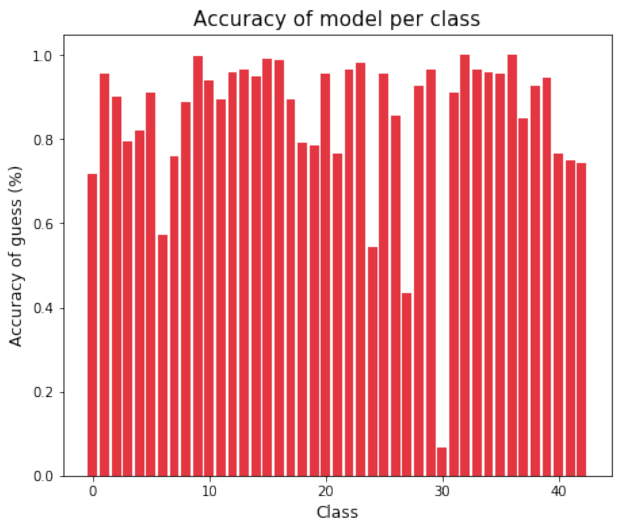

Fig. 24. Accuracy per class after applying Contrast modification

two forms of data augmentation. First we verified that the most represented signs rounded at about 1500 images, whilst the lowest were about a measly 200, so what we did was inject artificially created images into all subsets that had less than 1200 images until they reached that threshold. The way images were created was by randomly picking an image of the given subset and applying a random rotation and gamma adjustment. The obtained results weren't necessarily bad, but they were still worse than the ones obtained prior, as shown in Fig. 25, 26 and Table VIII.

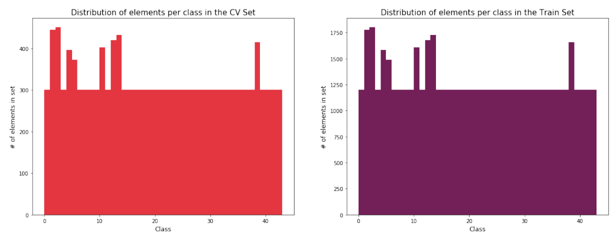

Fig. 25. Number of images per class on the augmented Train set and the CV set built from it

TABLE VIII
FIRST DATASET AUGMENTATION TECHNIQUE ACCURACY AND COST RESULTS

Set	*Accuracy*	*Loss*
Train	0.97684383	0.09540962100309129
CV	0.9633406	0.34100636054404126
Test	0.92494065	2.889168781619064

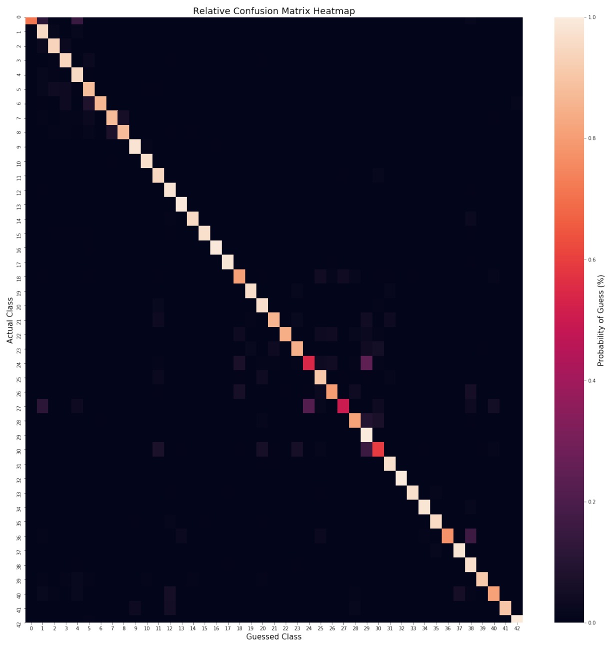

Fig. 26. Relative confusion matrix heatmap after applying the first Dataset augmentation technique

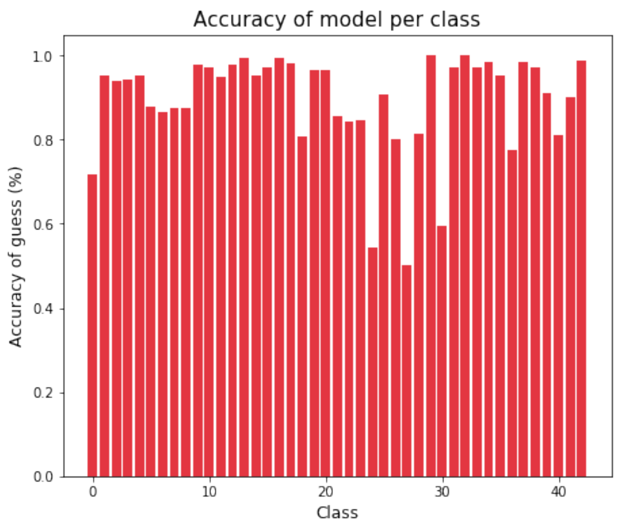

Fig. 27. Accuracy per class after applying the first Dataset augmentation technique

We figured that the poor results might have been caused due to the fact that, using the previous algorithm, the least represented classes in the original dataset would mostly constitute of artificial images in the augmented one. It should be said that, while the transformations used to create new images effectively create pictures distinct to the original one, they will still share a lot of similarities, hence they're not as effective at training the algorithm since there won't be enough variation between each other. With this in mind we gave data augmentation another shot, this time by creating artificial images so that all subsets would have a total of 5000 images and then picking, at random, 1200 images of each subset. Unfortunately, the results managed to be even worse than last time as can be seen in Fig. 29, 30 and Table IX.

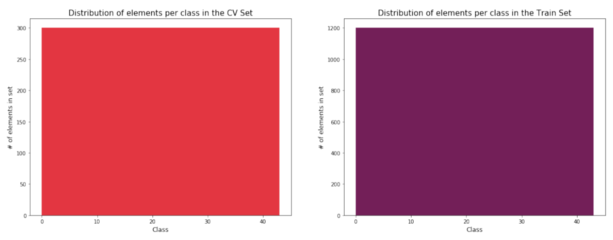

Fig. 28. Number of images per class on the augmented Train set and the CV set built from it using the second augmentation algorithm

TABLE IX
SECOND DATASET AUGMENTATION TECHNIQUE ACCURACY AND COST RESULTS

Set	*Accuracy*	*Loss*
Train	0.9837597	0.062307794080224146
CV	0.93418604	0.7422836606585702
Test	0.88654	12.478988825017291

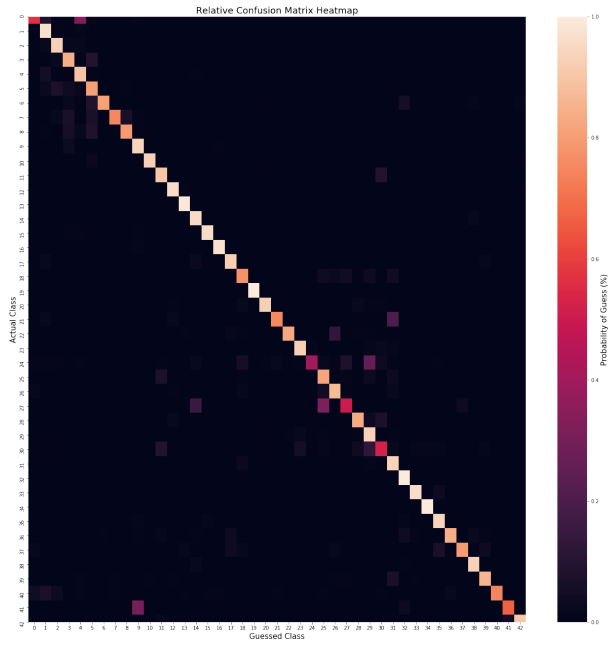

Fig. 29. Relative confusion matrix heatmap after applying the second Dataset augmentation technique

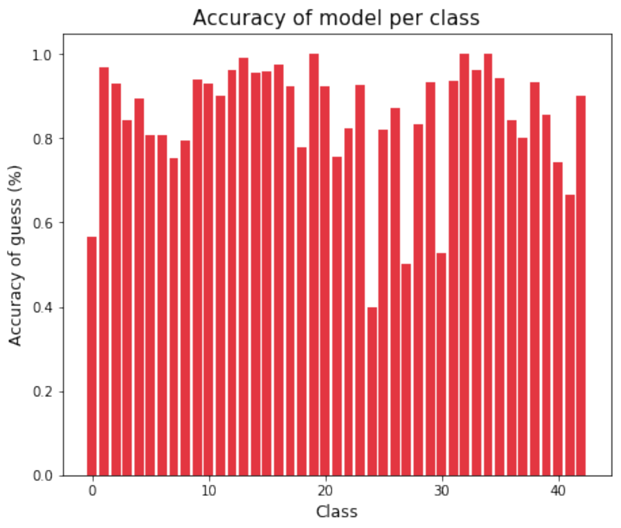

Fig. 30. Accuracy per class after applying the second Dataset augmentation technique

H. Conclusion

Overall we were rather disappointed with the low improvement of results between each of our attempts. Still, we managed to increase our overall accuracy and accuracy per class little by little. The reason as to why our preprocessing techinques, like equalization and contrast correction, didn't wield the results we expected might have been to the overall disparity in image variation, as applying an algorithm to certain images might improve them, but gravely worsen the visibility of others. Another possible reason might have been that we were focusing on the wrong preprocesses or that we should have added more such as blur correction.

When it comes to data augmentation, we believe the issue lied upon the fact that the least represented classes would constitute mostly of artificial images upon data augmentation, even after applying the second technique. Another possible problem might have the way we generated new images as the only transformations we applied were a rotation and gamma (brightness) change. This theory is further supported by the fact that the Train and CV set actually presented really good (and better than the previously achieved) accuracy, but when it came to the Test set the performance dropped considerably due to the fact that there were no artificial images in the Test set and our model had been trained using (artificial) images that were similar to each other and didn't present as much variance as needed to properly classify the pictures. Had we been given more time for this project we would perform more and more varied transformations when generating an artificial image to try to distinguish it as much as possible from the original one (by changing contrast, blurriness, saturation, etc.) so that, even for the least represented classes, our model would be trained with a properly varied dataset.

V. Model and Hyper-parameter alterations

A. Epochs

In the context of Machine Learning, an Epoch is defined as when the entire training database is passes forward and backward in the neural network. The model will then change its values after every epoch to get a better accuracy and lower loss values.

However, as it's to be expected, one epoch can take a very long time to be processed and may be costly for the computer, and after a certain number of epochs, the values will end up converging to a certain value.

So, we decided to study the evolution of the accuracy and loss values in relation to how many epochs have passed, resulting in the graphs contained in Fig. 31 and 32.

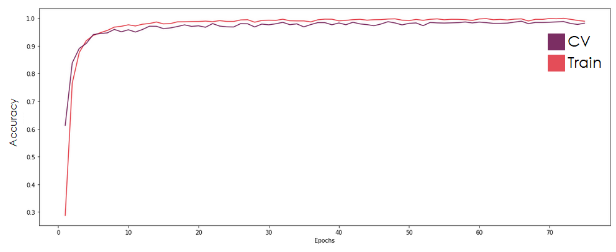

Fig. 31. Accuracy per Epoch

This shows there's not real reason to train the model for more than 25 Epochs, as the model has stabilized by then.

B. Changes in the Neural Network Structure

To follow the change in the hyper parameters, we decided to change our neural network's architecture and compare the obtained results. Firstly we started by entirely removing one of the convolutional blocks in our LeNet5, resulting in the results shown on Table X and Fig 33.

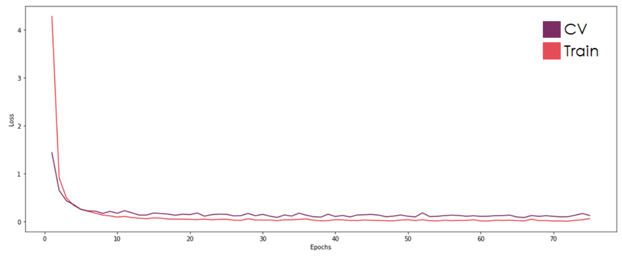

Fig. 32. Loss per Epoch

TABLE X
ACCURACY AND COST RESULTS WITH 1 CONVOLUTIONAL LAYER

Set	*Accuracy*	*Loss*
Train	0.9861643	0.046899364307072436
CV	0.9567657	0.2835945429848387
Test	0.8736342	1.5865751462990767

Quite clearly these results were showcasing overfitting, that is the parameters are adapting too much to the train set, but can't predict well enough new examples.
After this, we tried using only one fully connected layer, giving us the results shown on Table XI and Fig. 34.

TABLE XI
ACCURACY AND COST RESULTS WITH 1 FULLY CONNECTED LAYER

Set	*Accuracy*	*Loss*
Train	0.9966208	0.01381127847828818
CV	0.981635	0.10577674771742457
Test	0.91900235	1.6950519664691995

While showing better results than the previous network, it still doesn't justify the change in the network, as we're getting worse results than with our original LeNet 5 model.

C. Learning Rate

The next thing we have to study is the change of learning rate and how it affects the final results, as well as the overall

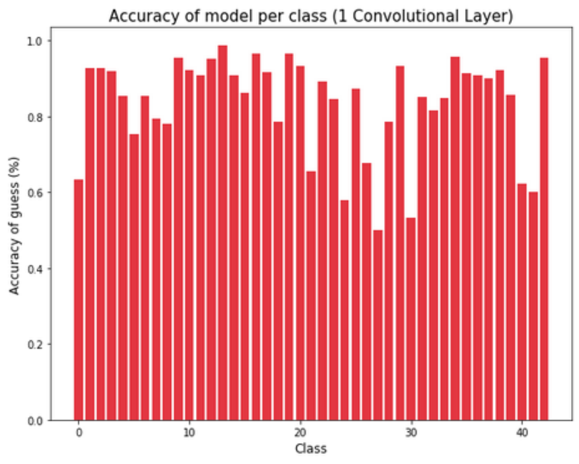

Fig. 33. Accuracy per Class with 1 Convolutional Layer

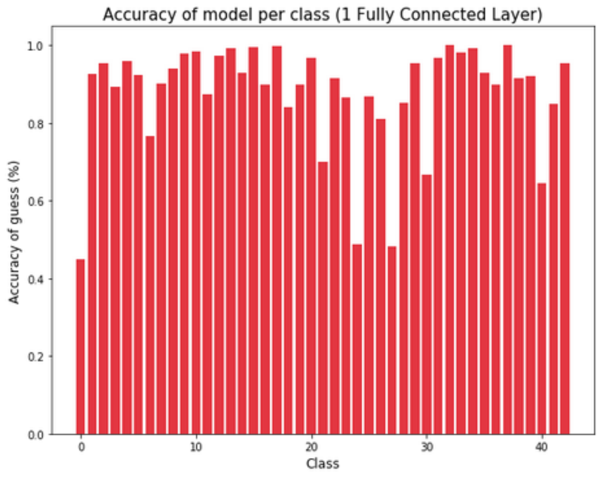

Fig. 34. Accuracy per Class with 1 Fully Connected

efficiency. Learning rate in neural networks can be summarised as how much should we change the node value based on its last cost. A learning rate too small and it can take too long for the model to reach a good final result; a learning rate too large and the values may never converge to reduce the cost function.

Up until now we have based our model on a learning rate of 0.002, based on what we have seen from the other reports and neural network models. We'll now change this learning rate, starting with 0.00025 and doubling it until it reaches approximately 0.1; and we'll compare the overall accuracy and loss function resulted from each different rate. The results are shown in the following graph (Fig. 35).

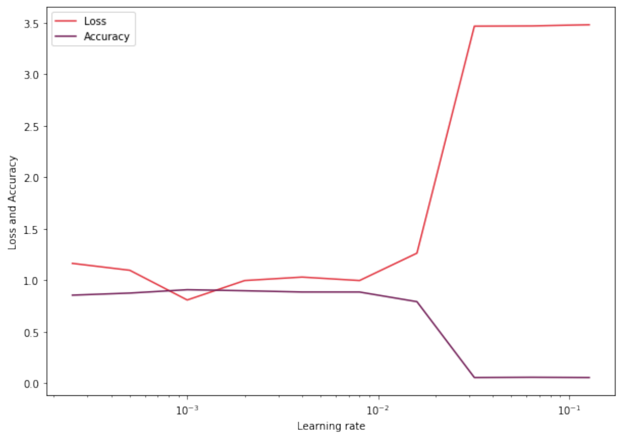

Fig. 35. Accuracy and Loss per Learning Rate

As expected, the greater the values the worse the final values of loss function and accuracy. It must be said, however, that we were not expecting such a quick decline in accuracy and increase in loss, as the values greater than 0.02 show abysmal results. As aforementioned, however, the learning rate also affects the time it takes to train our model, so we also reported on the time it took (in seconds) for each of the tested learning rate values. These results can be seen in Fig. 36 and, as expected, the smaller the learning rate value, the longer it takes for our model to train.

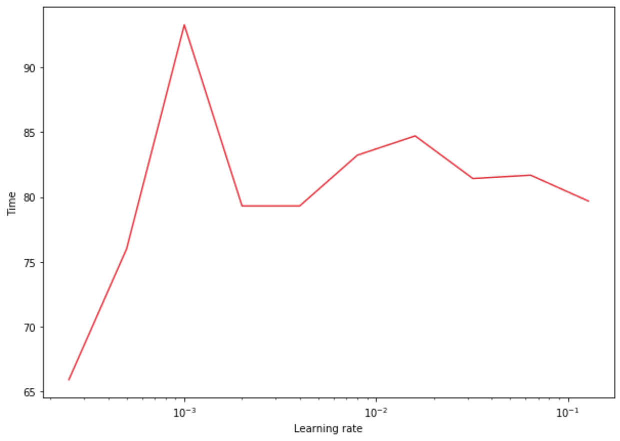

Fig. 36. Time taken to train per Learning Rate

While this proves that greater learning rates may impede the learning process of the model, we must also see the consequences of having a learning rate that's too low; the time taken to train must fall within an arbitrary, acceptable range of time without risking depleting the developer's patience.

D. Dropout

As we've seen throughout this report, the model has a great tendency to overfit to the training data. This, of course, has to be mitigated for better predictions for the test data. In this context, we were introduced to the concept of dropout: a regularization method that forces some of the output nodes in a layer to be randomly ignored.
The goal of dropping out nodes is to force other input nodes to have a greater responsibility, mitigating the noise caused during the training.
Of course, we didn't immediately know what values of dropout to pick, and on which layer should we put them. From other papers, we saw that the best results were when there was a dropout value in between the two convolutional blocks, and between the two fully connected layers. So we decided to study the values of accuracy and loss based on the value and its location on the network.
We started by choosing the best value (out of the ones we tried out) for dropout between the two convolutional blocks. The overall accuracy and loss values for the test set for each of the tested dropouts can be found on Table XII.

These results show us that the best value for a dropout value would be 0.1; so we trained the whole model again with this value, this time to see how it compares with the original network, taking note of all the measures: final accuracy for test set, data set and CV set (Table XIII), confusion matrix (Fig. 37) and accuracy per class (Fig. 38).

Here we see that some underrepresented classes are hurt badly by this method, not justifying its use despite the good result in the global accuracy.

TABLE XII
Accuracy and Cost Results with Dropout at Convolutional Blocks

Dropout	*Accuracy*	*Loss*
0.0	0.9108	0.7842
0.1	0.9283	0.4634
0.2	0.9072	0.6165
0.3	0.9098	0.4911
0.4	0.8990	0.4458

TABLE XIII
Accuracy and Cost Results with a Dropout p=0.1 at Convolutional Block

Set	*Accuracy*	*Loss*
Train	0.97988397	0.06783701262577657
CV	0.9792118	0.10608937150322803
Test	0.92114013	0.5530987583991562

After this, we applied the same method of study on a dropout at the fully connected layer. Table XIV shows all of the tested values.

TABLE XIV
Accuracy and Cost Results with Dropout at Fully Connected Layers

Dropout	*Accuracy*	*Loss*
0.0	0.8882	1.0522
0.1	0.9131	0.7051
0.2	0.9082	0.7238
0.3	0.9216	0.5324
0.4	0.9382	0.4076
0.5	0.9440	0.4010
0.6	0.9369	0.4398
0.7	0.9310	0.4905
0.8	0.9295	0.3830
0.9	0.0594	3.4628

In this table (XIV), we see that a dropout value above 0 does seem to improve the global accuracy, reaching a really good result when it's 0.5; just like how we did before, we'll use the global accuracy and loss function for all data-sets (Table XV), the confusion matrix (Fig. 39) and accuracy per class (Fig. 40) to better study how the model behaves.

TABLE XV
Accuracy and Cost Results with a Dropout p=0.5 at Fully Connected Layer

Set	*Accuracy*	*Loss*
Train	0.9803622	0.065988897759250766
CV	0.98597115	0.08362035378063079
Test	0.93705463	0.5070506157554827

Showing promising results, we came to the conclusion that a dropout value of 0.5 after the fully connected layer improved the overall performance of the model. We decided it'd be worth looking at how both values we studied more in-depth would work together: the following are the results with a neural network with a Dropout p=0.1 at the convolutional block; and

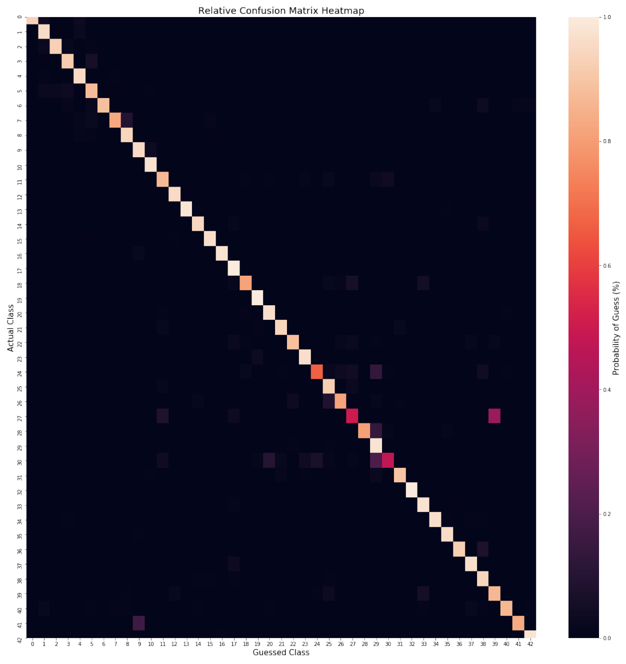
Fig. 37. Confusion Matrix with a Dropout p=0.1 at Convolutional Block

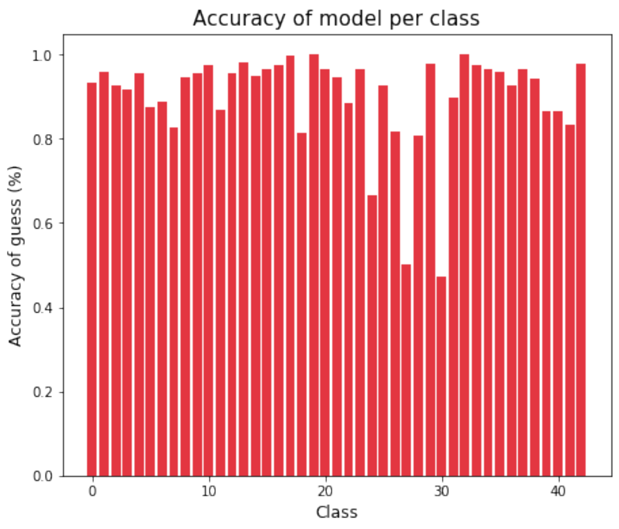
Fig. 38. Accuracy per Class with a Dropout p=0.1 at Convolutional Block

a dropout p=0.5 at the fully connected layer. These results are observable on Table XVI, Fig. 41 and Fig. 42.

Seeing how this showed a worse result in the training dataset, and did not seem to cause a major improvement on the neural network, we decided to keep the dropout value of the convolutional blocks at 0.0.

E. MaxPooling

We have been studying the way the model behaves when using an Average Pooling algorithm at the convolutional block, since it was what most studies we found on a subject had on their neural networks. However, we recently came

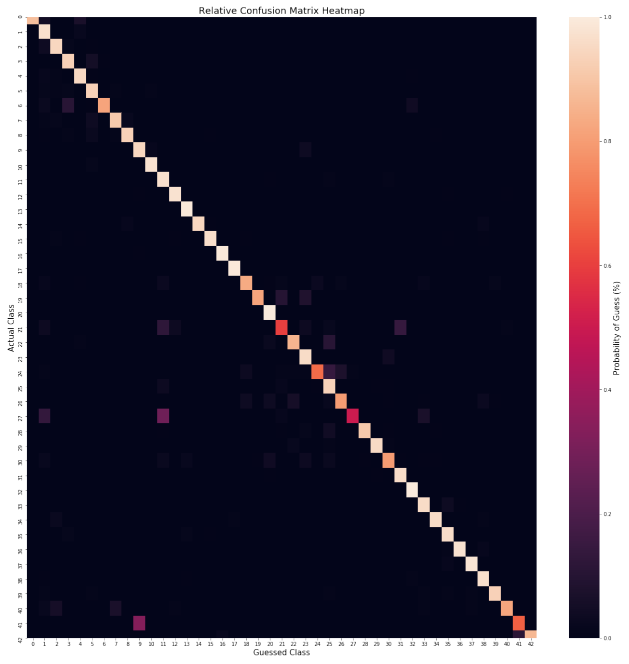
Fig. 39. Confusion Matrix with a Dropout p=0.5 at Fully Connected Layer

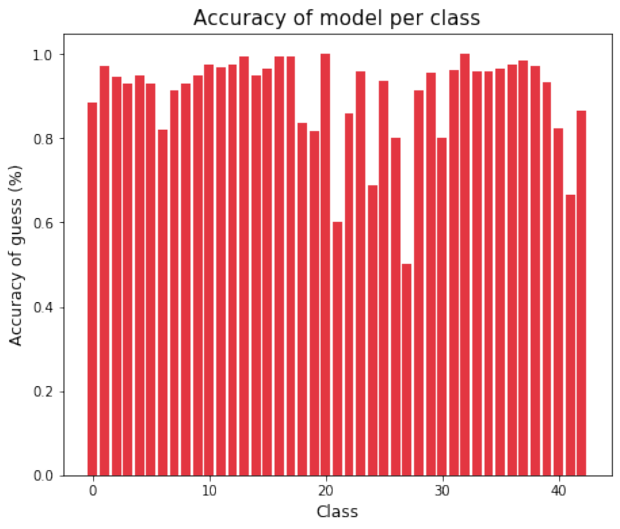
Fig. 40. Accuracy per Class with a Dropout p=0.5 at Fully Connected Layer

TABLE XVI
ACCURACY AND COST RESULTS WITH A DROPOUT ON BOTH LAYERS (P=0.1 FOR CONVOLUTIONAL AND 0.5 FOR FULLY CONNECTED)

Set	*Accuracy*	*Loss*
Train	0.95409334	0.15275168954068166
CV	0.9807423	0.08008680828443034
Test	0.935867	0.3276625476774604

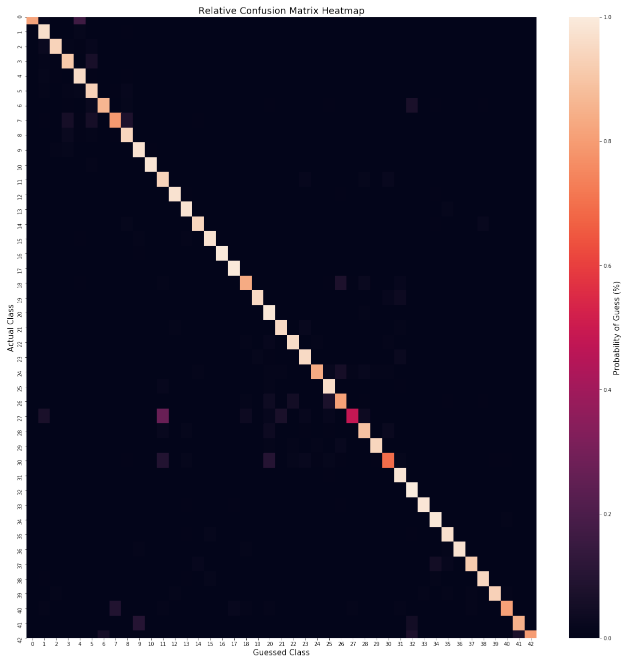

Fig. 41. Confusion Matrix with a Dropout on both layers

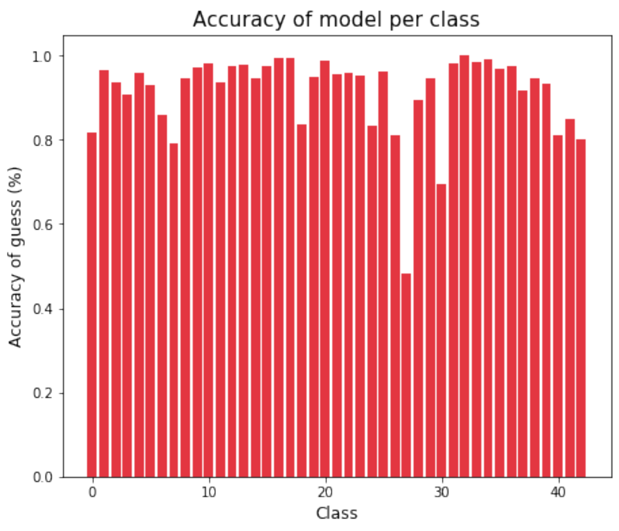

Fig. 42. Accuracy per Class with a Dropout on both layers

across a study talking about the max pooling algorithm. The pooling algorithms differ in how they choose the new value to replace a 2x2 matrix of values: the average pooling would average all values, while the max pooling will return the greatest value.
We decided that it would be interesting to use this algorithm to see how the behaviour of the network would change. These results are shown on Table XVII, Fig. 43 (the confusion matrix) and Fig. 44 (for the accuracy per class graph).

We see here the accuracy on the more underrepresented classes were greatly improved, and we have reached a new

TABLE XVII
ACCURACY AND COST RESULTS WITH MAXPOOLING

Set	***Accuracy***	***Loss***
Train	0.98355013	0.05397802045190214
CV	0.98622626	0.07530226532331012
Test	0.9476643	0.35066120557064673

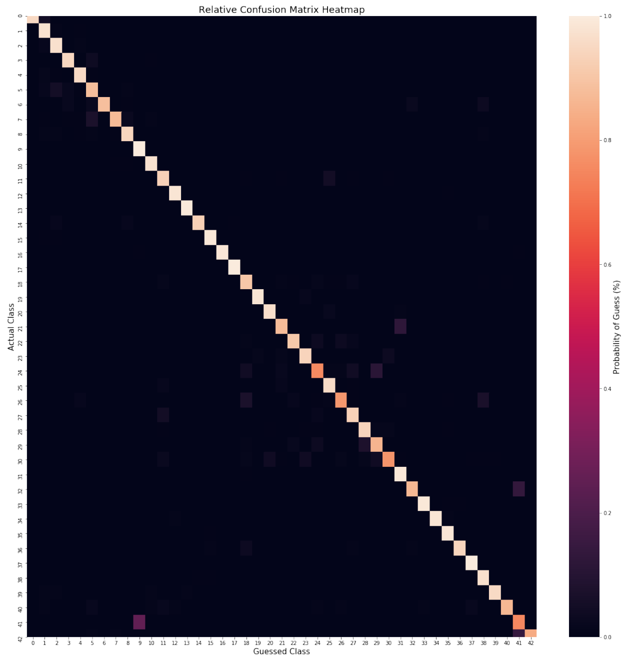

Fig. 43. Confusion Matrix with MaxPooling

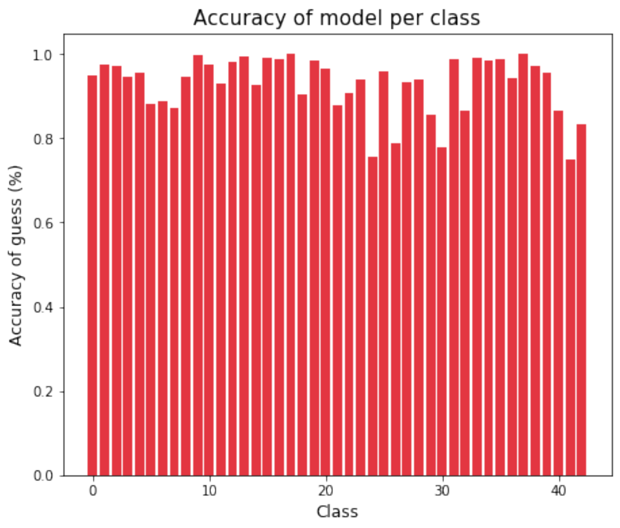

Fig. 44. Accuracy per Class with MaxPooling

final accuracy well above any of our previous tests.

F. Regularization

Lastly, we decided that it'd be interesting to see if adding a regularizer to our neural network would improve our overall results. It's a parameter introduced to constraint the coefficient estimates towards zero, in an effort to discourage it from learning very complex models, which would lead to the overfitting issue we've been seeing. The results obtained were put into Table XVIII, Fig. 45 and 46.

TABLE XVIII
ACCURACY AND COST RESULTS WITH REGULARIZER EQUAL TO 0.01

Set	*Accuracy*	*Loss*
Train	0.96219075	0.41597348621869934
CV	0.96964675	0.4006624079308572
Test	0.92644495	0.5791669460014983

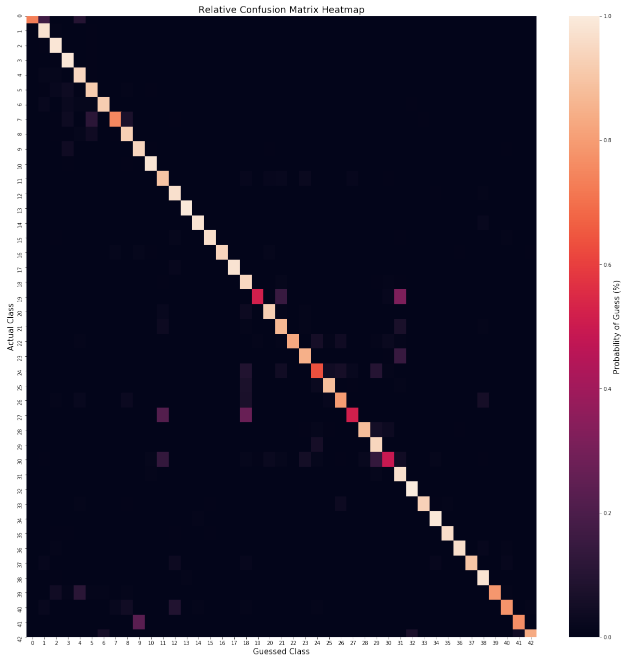

Fig. 45. Confusion Matrix with Non-zero Regularization

Unfortunately, as we can see, this did not improve our accuracy in any way. It's worth noting however that the accuracy for the training and CV dataset did decrease, showing that the model had a smaller tendency towards overfitting, even if it's still apparent.

VI. Final Conclusions and How to Improve the Model

All in all, we're pretty satisfied with our model. At the best configurations, we managed to reach a final accuracy of 98.3% on the train dataset, 98.6% on the Cross Validation dataset and 94.7% on the Test set. However there is still some room for improvement.

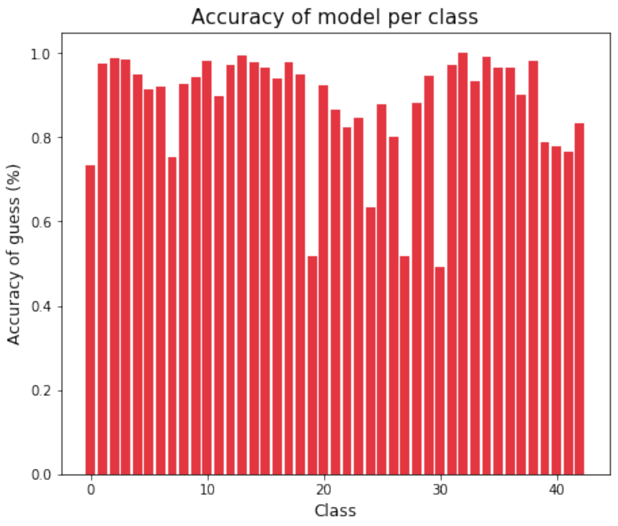

Fig. 46. Accuracy per Class with Non-Zero Regularization

Ater analysing other articles, about models with better final results, we saw that there are some important configurations that could result in a better final accuracy.
For instance, a different type of model could potentially go a long way; while we have tested small changes in our network, we kept the same overall model, a group of convolutional blocks, a flattening layer, and a group of fully connected layers, but we have looked at authors who structured their network differently, by doing three convolutional layers before applying a MaxPool [15].
Another change could be just changing the preprocessing steps: having better algorithms for the way data augmentation and the way the images are being normalized [14] [2] may have increased our overall accuracy. While we have tested for these variable, we used very primitive algorithms that may not have been suited for the job.

References

[1] Tomasz Kacmajor, "Traffic Sign Recognition using Convolutional Neural Network", 2017
[2] Eddie Forson, "Recognising Traffic Signs With 98% Accuracy Using Deep Learning", 2017
[3] Moataz Elmasry, "Deep Learning for Traffic Signs Recognition", 2018
[4] Harshit Sharma, "Identifying Traffic Signs with Deep Learning", 2017
[5] Param Aggarwa, "Intricacies of Traffic Sign Classification with TensorFlow", 2017
[6] LeNet-5 – A Classic CNN Architecture
[7] Pierre Sermanet and Yann LeCun, "Traffic Sign Recognition with Multi-Scale Convolutional Networks", Courant Institute of Mathematical Sciences, New York University 2011
[8] Dima Shulga, "5 Reasons why you should use Cross-Validation in your Data Science Projects", 2018
[9] Uniqtech, Understand the Softmax Function in Minutes, 2018
[10] Danqing Liu, A Practical Guide to ReLU, 2017
[11] LeNet-5, convolutional neural networks
[12] Jason Brownlee, Gentle Introduction to the Adam Optimization Algorithm for Deep Learning, 2017
[13] Categorical crossentropy
[14] Barney Kim, Traffic Sign Recognition, 2019
[15] Shekhar Bavanari, Traffic Sign Detection using Deep Learning, 2019

Addendum

A. Division of labor

For this work we met online via tools such as Jitsy and Zoom. Both students collaborated an equal amount of work hours developing all algorithms and analyzing every result in tandem.
Diogo Silva - 50%
Pedro Oliveira - 50%

B. Relative Confusion Matrix label mistake

Due to a nomenclature mistake that we only took notice whilst elaborating this report (at which point it was too late to change), the sidebar's (which shows the correspondence between color-value) label was dubbed ”Percentage of guess (%)”, when it should've in fact have been called ”Frequency of guess”.