Introduction to Computer Vision 22928

Font Recognition in Images - Competition

Lior Michaeli*

The Open University of Israel
liorm0995@gmail.com

Abstract

Font recognition in images is a very important, challenging, and useful task today in Document Analysis, Pattern Recognition, and Computer Vision. This project presents a deep learning approach to recognizing the font of characters in images. In this paper, I aim to recognize the font of characters in images accurately and effectively by proposing several effective and smart methods to solve this task with smart algorithms that can be applied to a neural network and improve its performance drastically without changing the architecture at all.

Keywords

Font classification; Convolutional Neural Networks; Deep Learning; Network Architecture; Mini batch smart construction and selection; Computer Vision; Image processing

1 Introduction

Font recognition is a very important task, and it can help a lot to many people. For example, font is one of the most fundamental concepts in design, and therefore automatic font recognition from an image will help a lot to designers. As we know, Deep learning approaches are applied to many domains and tasks today, including Document Analysis tasks. In this competition, we needed to propose solutions for font recognition in images task on a special dataset, which we were provided with in the competition, and I will discuss it in Section 3. The duration of the competition was a month and a half.

2 Related Work and Novelty

There have been various approaches to recognizing fonts in images using deep learning approaches. Y. Wang et al. proposed a method based on deep learning and transfer learning [1]. They develop a fast and scalable system to synthesize huge numbers of natural images containing texts in various fonts and styles and use this synthesized data to train deep learning models that recognize fonts. Z. Wang et al. built up the first available large-scale VFR dataset, consisting of both labeled synthetic data and partially labeled real-world data [2]. Additionally, they developed Domain Adapted CNN that addresses the domain mismatch between real data and syntactic data. In addition, Tensmeyer at el. proposed a state-of-the-art method on a challenging dataset of 40 Arabic computer fonts, which classify small patches of text into predefined font classes [3]. They proposed a novel form of data augmentation that improves the robustness of the model to text darkness.

In this paper, I will mainly discuss some smart and effective methods to solve the font recognition task on the specific dataset of the competition. These methods mainly rely on meta-information and assumptions in our data and can improve the model performance drastically. One of these methods proposes special mini-batch selection and construction for the training procedure. I will discuss all these methods and some more methods in section 4. As far as I know and after having researched papers that deal with font recognition tasks, I did not see some of the methods I suggested, such as special mini-batch selection and construction based on meta-information in our dataset, being used. I do not know if the methods that I proposed are innovative, but I thought of them myself and I hope that these methods are an innovation.

3 Dataset and Preprocessing

My project uses a dataset that was given to us, specially for our competition. It was created by SynthText, a method for generating synthetic text images, provided by Gupta et al. [4]. In the competition, we get training and test datasets. The training dataset consists of 829 synthetic images, that contain 22,084 characters from seven fonts and the test dataset consists of 80 synthetic images that contain 2196 characters from seven fonts. In addition to the images, each image has word bounding boxes, character bounding boxes, and text as Meta information on the dataset. Of course, each image has a font label for each character, except for the test dataset that the labels of which we do not have, and only when we submit the predictions file for the test dataset on the competition page in Kaggle, we will get the accuracy of our model on the test dataset.

3.1 Data splitting

I decided to build datasets of characters instead of words, because in this way I will build datasets with more data (there are more characters than words), and I can apply smart algorithms (that I will discuss in section 4) thanks to the fact that I use characters. I split the training dataset that was provided to us, into training and validation datasets, when the training dataset consists of 745 synthetic images that contain 20,052 characters, and the validation dataset consists of 84 synthetic images that contain 2032 characters. The test dataset remains as we received it. I split the data into three datasets because this is important in ML to have training, validation, and test datasets and not only training and test datasets. In this way, we can evaluate the models and the algorithms that we proposed on the validation dataset and then we will test whether our algorithm or model is really as good as we thought on the test dataset. In addition, I put in the validation dataset approximately the same amount of data as in the test, to better evaluate the algorithms and the models I will propose before evaluating them on the test dataset (the same amount of data will probably be described in a similar way as it happens in the test dataset)

3.2 Data Preprocessing

Firstly, to create a dataset of characters for training, we need to extract the characters by their bounding boxes, which are provided to us as Meta information in our data. However, one issue that we need to solve is that the characters bounding boxes can be rotated, and must not be parallel to the axes. If we extract the chars in the classic and trivial algorithm of taking rectangle extraction that is parallel to the axes, by computing the four coordinates of the rectangle according to the highest and lowest x and y coordinates, we will have three problems:

- We will get an image of a char that contains extra information that was not in the original rotated bounding box of the image, and this can cause the model that we will develop, not to learn properly because we will disturb it with extra information.
- We will get many images with different degrees of rotation(some of which have very large differences in the degree of rotation), and thus our model will need to generalize on the factor of rotation of the chars in the images, and thus it will allocate a part of its learning resources in favor of this generalization process. This is not good because, as a result of this the model performance can decrease.
- We will get many images in different scales, and this is problematic because we need all the images to have the same resolution, to give them as input to our models (our models are not FCN and thus this is important for all the inputs of the model to be in the same resolution).

Thus, we need to find another solution to this issue. I thought of using the following algorithm:

1. I extracted each char from its original image (an image that contains words) with homography transformation, and in order to make the homography transformation, I used the char bounding box that was given to us.
2. I computed the average resolution of the results after step 1 and got that the average resolution of the results after step 1 is of approximate shape of (30, 19, 3). I rounded this to the shape of (32, 16, 3) because I wanted to use width and height which is a power of two (this is more acceptable and more convenient). One little note is that I write the shapes in the format of (Height, Width, Amount of channels).

This algorithm solves the issue that the characters' bounding boxes can be rotated and solves the problems that are caused by the classic and the trivial algorithm.

In Fig. 1, there is an example of the result of the classic and the trivial algorithm versus the algorithm I proposed on a specific example from the training dataset:

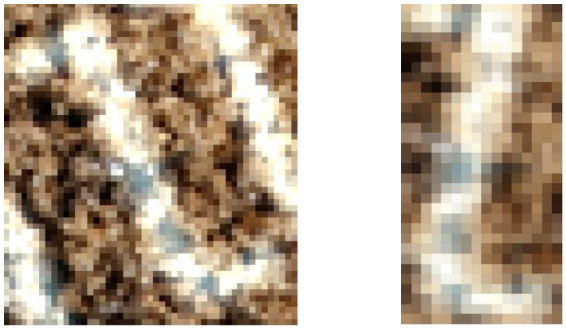

Figure 1: Results of the classic and the trivial algorithm (left) versus the algorithm I proposed (right) on specific example from the training dataset.

We can see that the algorithm I proposed gives good results, and give better results compared to the classic and the trivial algorithm.

In addition, I normalize all the data according to the mean and the std of the training dataset.

3.3 Data distribution

This is very important to know what our data distribution, because thanks to this we will know if we have a problem of imbalance in our data and if yes, we will know that we need to solve this issue when we build our models and algorithms.

In Fig. 2, there are the character distributions in the training, validation, and test datasets.

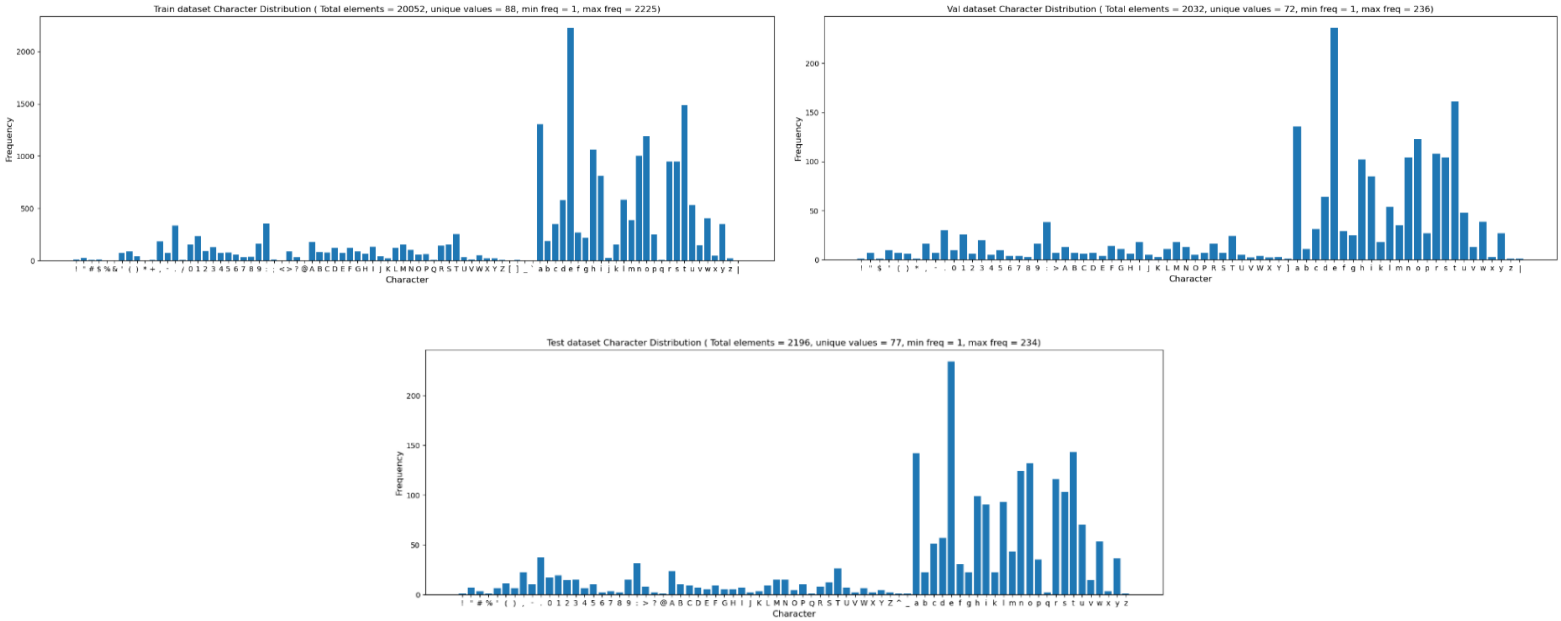

Figure 2: Characters distribution in the training, validation and test datasets

In addition, In Fig. 3 there are the fonts distribution in the training and validation datasets (As I mentioned, we not have font labels for the test dataset).

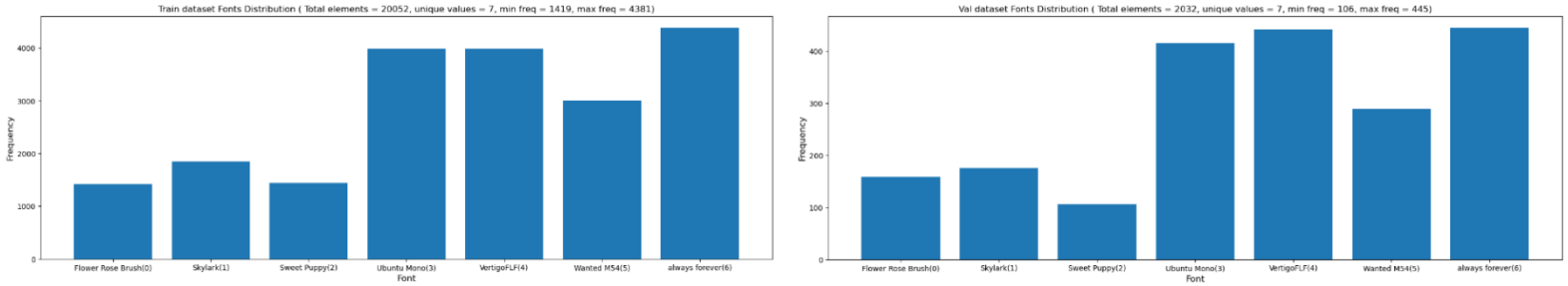

Figure 3: Fonts distribution in the training and validation

We can see that the font distribution in each dataset is imbalanced. This is an issue that we need to solve, because if we do not solve this issue, our model can tend in its predictions to specific fonts, probably the fonts that appear the most in the data. In addition, the distribution of the characters in each dataset is also imbalanced. This also may be an issue, because if we do not solve this issue, our model in low-quality images can assume more that these images describe specific chars (probably the characters that appear the most in the data), and this can lead it to wrong predictions.

Something important and interesting that we can see is that the fonts and characters distribution in each dataset are similar to each other in their pattern and how they are distributed. This makes sense, because our training, validation, and test datasets come from the same source, and therefore they have the same structure of data and the same characteristics. For example, if I have two fonts and one of them appears the largest number of times in the training dataset and the other appears the least number of times in the training dataset, so probably this ratio of appearances between these two fonts, will also happen in the validation and test datasets.

I will discuss solutions to the issue of the imbalanced distributions in section 4.

3.4 Data sample examples

Our datasets contain both low-quality and high-quality images.

In Fig. 4, Fig. 5, and Fig. 6 there are some examples of characters from the training dataset, validation dataset, and test dataset.

Figure 4: Examples of data samples from the training dataset (each data sample contains image, character txt and font label)

Figure 5: Examples of data samples from the validation dataset (each data sample contains image, character txt and font label)

Figure 6: Examples of data samples from the test dataset(each data sample contains image and character text). As I mentioned, we not have font labels for the test dataset

4 Methods

4.1 Baseline

The baseline method that I proposed is a simple architecture described in Fig. 7, and I am trying it to start solving our task with the baseline method and after I see the results of this network, I will start improving my baseline method.

Training details: I train my model with a cross-entropy loss function, learning rate of 5e-4, without weight decay, and batch size of 128. In addition, I applied a learning rate scheduler with a step rate of 20 and a decrease rate of 0.2, i.e. after every 20 epochs we will multiply the learning rate by 0.2. I trained the model for 30 epochs. I used Kaggle to train my model with the P100 GPU. I used the cross-entropy loss function because this is the most classic loss function to start with in classification tasks.

This baseline method gave me good starting results described in Section 5, and after this, I started to try other methods, because I understood that to improve the results in this task, I would need to use more complex methods.

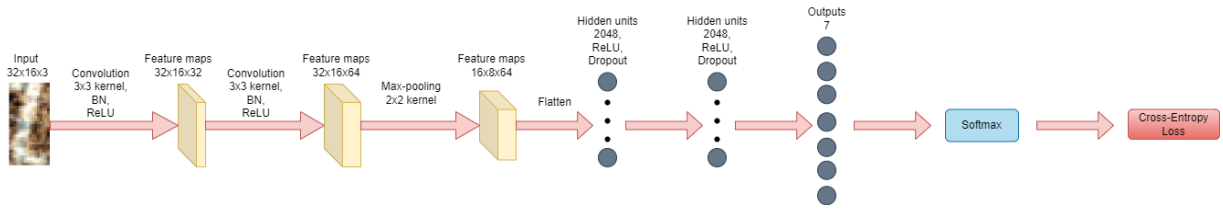

Figure 7: Architecture diagram of the Baseline method, that I created with the drawio site [5]

4.2 Custom ResNet

The second method that I proposed is to create a Custom ResNet model, i.e., create a model myself while taking inspiration from ResNet and the idea of Residual blocks. I chose to take inspiration from ResNet because ResNet is a very powerful and state-of-the-art architecture in DL, and it allows us to build deep models, while keep improving the model performance. So, I created my own custom ResNet described in Fig. 8, which is suitable to input with resolution like our image resolution. Our image resolution is small, and therefore I don't build a very deep and complex network.

Training details: I train my model with a cross-entropy loss function, learning rate of 1e-3, without weight decay, and batch size of 128. In addition, I applied a learning rate scheduler with a step rate of 15 and a decrease rate of 0.1. I trained the model for 30 epochs. I used Kaggle to train my model with the P100 GPU.

This method gave me better results compared to the Baseline method described in Section 5, but the results were still not good enough and I wanted to improve these results. This is what led me to go in the direction of thinking of smart algorithms that can be applied to neural network models.

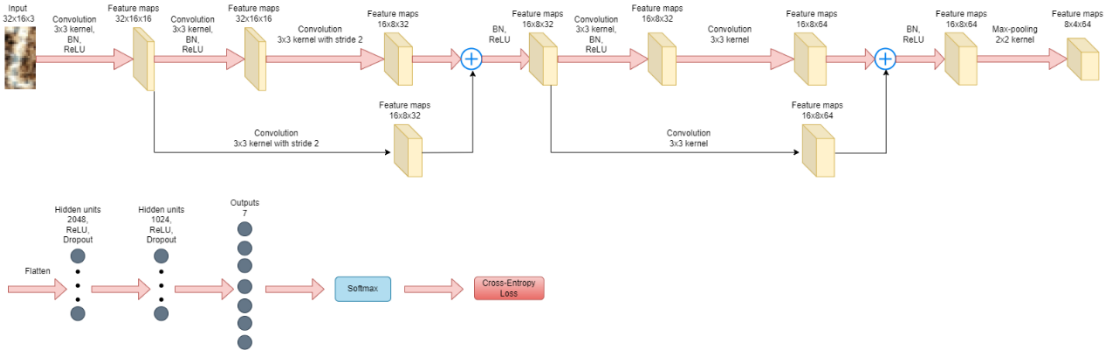

Figure 8: Architecture diagram of the Custom ResNet method, that I created with the drawio site [5]

4.3 First smart algorithm

After I tried the two previous methods, I understood that I needed to improve my model performance not by using a different architecture but by using a smart algorithm that will be applied to a model. The thought that got me thinking about smart algorithms is the thought that we have a lot of information and assumptions in our data that we can use to improve our model performance. The first smart algorithm assumes that in our data, in each word all the fonts of the characters are the same (this occurs almost always in reality too). Thus, after our model has predicted all the fonts of the characters in a specific word, we can take the most predicted font in this word as a final prediction for these characters. This is intuitive, because if the model has predicted that most characters are with font A, so probably all the characters in this word are with font A. In this way, we can avoid a lot of mistakes and improve the model's performance.

I applied this algorithm on my Custom ResNet, and the results improved drastically (the results described in Section 5).

4.4 Ideas that did not improve the model performance

There are some ideas that I tried, but they did not improve the model performance. In this section, I will describe them shortly. Firstly, one idea that came to my mind to address overfitting in my model is to apply data augmentation like color jitter, random contrast, random brightness, and more. I tried this and it did not improve my model performance, and even gave me a slightly worse performance. First of all, this is not clear why this happened, because this makes sense that data augmentation should always help and be good, but this is not correct in all the cases. Our data is synthetic and not real

word data images, and thus there are many factors that we need to generalize on them in real data, that we do not need to generalize on them in synthetic data. Thus, if we use data augmentation in our task, this will not help to increase the performance and even can hurt the model performance, because the model allocates some of its learning resources for the generalization of some factors that it will never encounter. In addition, I tried to use transfer learning with VGG16 and ResNet50 pre-trained models on the ImageNet dataset, and this did not help to improve the performance. In addition, I thought to try to convert my RBG images to another format like gray-scale images or edge detection images, because I thought that the model must not get the RGB images to classify the font of the char. This idea failed to improve the model performance, and I assume this is because the model in the first convolutional layers probably gets the edge detection of the image input, and thus this is not does not matter so much if we give the model the edge detection of the image as input or give the model the original RGB image as input.

4.5 Second smart algorithm

After I tried the ideas that did not improve the results, I decided to see what the type of mistakes appear in predictions that my model made. I started checking and noticed that in many words there are more than one most predicted font, and my first algorithm chose without a smart thought one of these most predicted fonts, and therefore there are examples that one of the most predicted fonts is the correct font, but it not chosen by the first smart algorithm because there are some most predicted fonts. This brings me to understand that if I have some most predicted fonts, I need to choose the font that the model is most confident about in the characters that predict this font. We can see an example of a mistake of this type in Fig. 9.

Figure 9: Example of mistake that the first algorithm made. There are two most predictions(always forever and VertigoFLF), but the first smart algorithm takes the always forever, while the model has more confidence in the VertigoFLF font.

I applied this algorithm to my Custom ResNet, and the results improved, and it brought me to very good results that are described in Section 5. At this point, I was in a great position and all I had to do was to look for small things that could improve my model even more.

There is a pseudo code for the second smart algorithm.

Second smart Algorithm

```
initialization: trained_model = model
initialization: total_second_smart_predictions = null
words_lens = dataset.get_words_lens()
total_predictions_probs = trained_model.get_predictions_probs(dataset)
curr_idx = 0
for word_len in words_lens:
    word_probs = total_predictions_probs[curr_idx : curr_idx + word_len]
    word_predictions = get_predictions_according_probs(word_probs)
    most_predictions_in_word = get_most_predictions(word_predictions)
    final_prediction = get_prediction_with_most_prob(most_predictions_in_word, word_probs)
    for i in word_len:
        total_second_smart_predictions.append(final_prediction)
    curr_idx += word_len
end for
return total_second_smart_predictions
```

4.6 Weighted Cross entropy loss function and Mini-batch smart construction and selection

As we saw, the characters and fonts distributions in our data are imbalanced. Therefore, I thought of ideas that can help address this issue, and this led me to think about the idea of Weighted Cross entropy loss function and Mini-batch smart construction and selection. Due to the lack of time and as the competition was nearing its end, I didn't have time to try these ideas, but I suggest them anyway because I really think these ideas can improve the performance of the model,

especially the idea of Mini batch smart construction and selection, that can not only improve the performance of the model but also make the training process much faster and much more stable. Firstly, the idea of the Weighted Cross entropy loss function says that instead of using the classic Cross entropy function, we will also give weight to each class. For fonts that appear few in the data, we will give a higher weight, and for fonts that appear a lot in the data, we will give a lower weight. This idea will lead the model to be good not only on the fonts that appear a lot but also on fonts that appear little because now we will punish our model more if it makes mistakes on the minority fonts.

As for the second idea, I have a lot to say, but as this is not the main issue of this project, I will discuss this shortly. The idea of Mini-batch smart construction and selection says that instead of the use of the standard mini-batch gradient descent algorithm, that selects data samples randomly, we need to build algorithms that are based on meta-information of our data, and with this algorithm, we will increase our model performance, stability, and our model become faster. In our case, our model needs to handle variations of chars in images because each image represented another char, and this is also important that the model can understand the type of the font of the image. Thus, we can choose the char of each image as the meta-information in our data. We need to build an algorithm that will lead us to a situation where each mini-batch will contain the presence of each char, and in this way, our model will see all the chars in each mini-batch, which can help a lot to address the problem that the distribution of the characters in our data is imbalanced. Our ambition is always to reach a uniform distribution of all chars in each mini-batch. This algorithm will give us better results, and faster and more stable training. This algorithm is discussed in more detail by Dokuz & Tufekci [6]. They applied this to a speech recognition task. We can apply this algorithm in a very similar way to our task. In their task, this algorithm improves the results, and thus I believe that in our task will also improve the results. In addition, there are more strategies for mini-batch smart construction and selection that can be applied to our task, and also improve performance, like building and selecting each mini-batch according to the loss value in each data instance or data instances that are related to a specific font or char. This idea was presented by Loshchilov & Hutter [7].

In conclusion, these ideas are very good and creative, and I believe that they can improve the performance of our model in our task.

4.7 Multi neural networks

As we discussed before, in our data each image of char has meta-information that says what the char that the image represented is. In addition, we can intuitively be convinced that if our model task was to classify the type of the font of only images that represented only one specific character, our model performance would increase. This is because our model can assume that all the images that it classifies are for example represented with the char a, and thus it can focus more on determining the type of the font of char a. This assumption helps our model a lot, because types of fonts on specific char looked different compared to these types of fonts on another char, and thus when our model knows the char that it classifies, it will be much easier for it to predict the correct font for this char. So, I thought to take the second smart algorithm, and for each char compute the model accuracy on all the images that related to this char, and if the accuracy is smaller than some threshold I will determine (for example 97% accuracy) and the amount of the images that related to this char in the training and validation datasets bigger than some threshold1 and threshold2, I will create a new model that will specialize only in images related to this char. Finally in the prediction time, for each image, if the char that it represented has a model for this char, we will go to this model, and else we will go to the second smart algorithm. I thought about this idea a little bit after I thought about the first smart algorithm and a little before I thought about the second smart algorithm, and thus I implemented this method on the first smart algorithm. I tried to run and experiment with this method, but it took a lot of time to train and run it in my hardware, and I only did a small initial experiment of this method with the first smart algorithm that gave me a result of around 98% accuracy on the validation dataset but gave me a result of 95.9% accuracy on the test dataset. I think that if I had run this more times and of course run this on the second algorithm, this would have given me great results, and even improved my best results, but the problem was that the process with this method was very slow because of my hardware, and thus I gave up trying this method, but I think that this is a beautiful and good method that can help.

4.8 Optimized Custom ResNet - Hyperparameter Tuning with Optuna

Optuna is a very famous framework for hyperparameter tuning presented by Akiba et al. [8]. This framework makes all the hyperparameter tuning processes more automatic, effective, and easy. Therefore, we can get better results with less investment time than when we do manual hyperparameter tuning or use classic algorithms for hyperparameter tuning.

Hence, I use Optuna to optimize the second smart algorithm. I did hyperparameter tuning on these hyperparameters: learning rate, weight decay, number of Residual blocks, starting amount of channels, Number of Residual blocks, Number of fc layers(hidden fc layers, not including output fc layer), and number of out features in each hidden fc layer. I chose to tune these hyperparameters because these hyperparameters influence a lot the training process (and such as learning rate, weight decay), and I wanted to find the best architecture for our task, and as a result, get a better model performance.

This hyperparameter tuning with Optuna improved my model performance compared to little manual hyperparameter tuning that I did myself that led me to 97.024% accuracy on the test dataset. The results described in Section 5.

4.9 Improve the final method performance for production.

After I optimized the second smart algorithm, I trained the model that I had (Custom ResNet with optimization and second smart algorithm) with its weights on the training and validation datasets, i.e., I combined the training and validation datasets to one dataset and trained my pre-trained final model on this dataset. This is a common technique in DL, and this can improve our final model for production in many cases. Here also, I use Optuna to tune the following hyperparameters to train more effectively and easily the final model on all the training and validation datasets: learning rate, weight decay, and number of epochs.

This method improved my results, and this was my final method and idea. The results of this method are described in Section 5.

5 **Results**

5.1 Results Summary

Model	Train Accuracy	Validation Accuracy	Test Accuracy	Amount of params	Running time of the method - making predictions on the test dataset
Baseline	97.182325	87.696850	87.128	21009543	0.271759 seconds
Custom ResNet without smart algorithms	98.698384	88.139763	89.86	6360503	0.283349 seconds
Custom ResNet with the first smart algorithm	99.910233	96.456692	96.114	6360503	0.334891 seconds
Custom ResNet with the second smart algorithm	99.950129	97.440944	96.903	6360503	0.38439 seconds
Optimized Custom ResNet - Hyperparameter Tuning	99.132256	97.539370	97.692	13774527	0.3851 seconds
The final method for production	99.637746	X	97.996	13774527	0.3908 seconds

Table 1: Methods Train/Validation/Test Accuracy and running time

5.2 Hyperparameter Tuning on the Final Model Results

The parameters on these tables are not exact and are approx. To see the exact parameters, you can go to the project implementation .

lr	Weight decay	Number of Residual blocks	Starting amount of channels	Number of fc layers	fc1 out features	fc2 out features	Train Accuracy	Val Accuracy	Test Accuracy
0.001443	0.000125	3	16	1	3297	X	99.13225	97.53937	97.692
0.002049	0.000652	3	32	2	6335	2478	99.98	98.27755	95.992
0.000907	6.98e-05	3	32	1	7727	X	98.78815	97.88385	96.721

Table 2: Hyperparameter Tuning on the final model.

lr	Weight decay	Num of epochs	Train Accuracy	Test Accuracy
0.0004886	7.239364e-05	4	99.637746	97.996
0.0001888	5.502165e-06	9	99.986415	97.571
2.9882548	3.899318e-05	7	99.411338	97.935

Table 3: Hyperparameter Tuning on the tuned final model for production.

5.3 Training and Validation Loss Curves

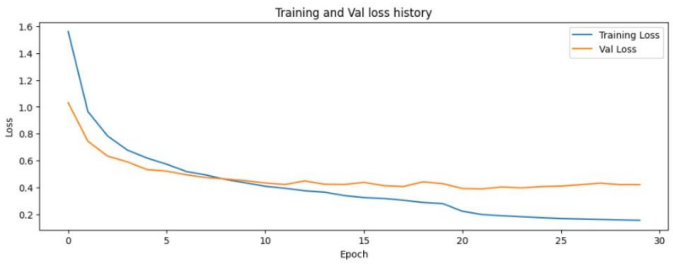

Figure 10: Training and Validation Loss curve in the Baseline model

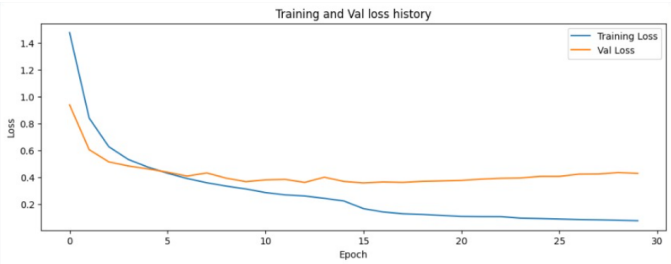

Figure 11: Training and Validation Loss curve in the Custom ResNet model

5.4 Training and Validation Accuracy Curves

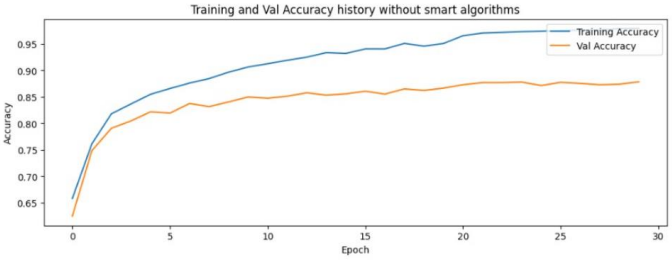

Figure 12: Training and Validation Accuracy curve in the Baseline model

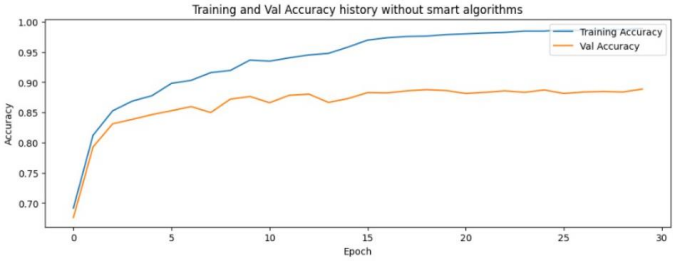

Figure 13: Training and Validation Accuracy curve in the Custom ResNet model

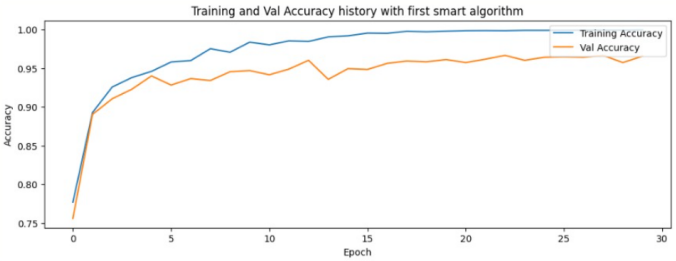

Figure 14: Training and Validation Accuracy curve in the First smart algorithm

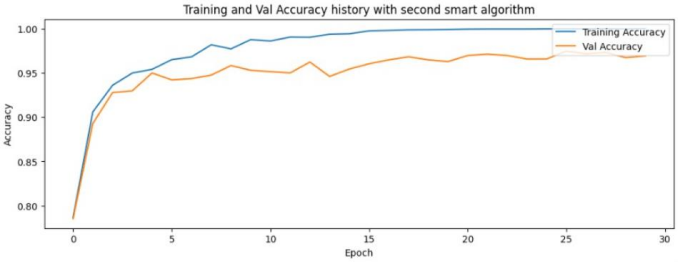

Figure 15: Training and Validation Accuracy curve in the Second smart algorithm

5.5 Evaluation graphs of the final method for production

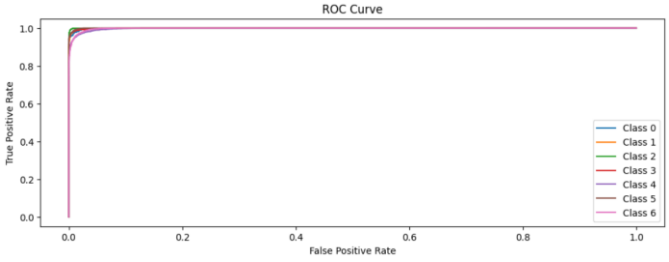

Figure 16: ROC graph

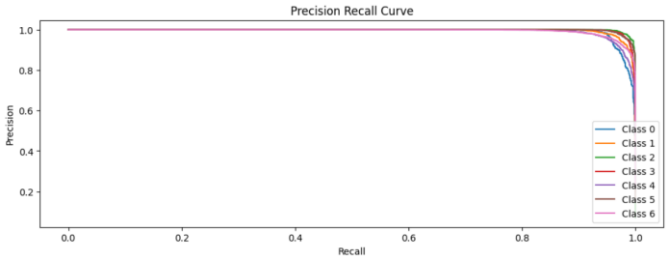

Figure 17: Precision-Recall graph

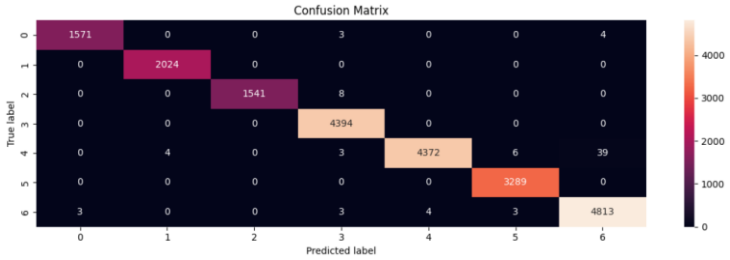

Figure 18: Confusion matrix graph

6 Conclusion and Future Work

In this work, I propose methods for solving the font recognition in images task. I have learned a lot about how to create an end-to-end ML project myself and to write on it a research paper and of course, I have had fun. I reached very good results in the competition with 97.996 % accuracy on the test dataset. In addition, I proposed creativity methods like mini-batch smart construction and selection that as I know and as I have researched papers that deal with font recognition, I did not see this method being used.

In the future, I would like to work on researching more about the Weighted Cross entropy loss function and Mini-batch smart construction and selection methods that I described for our task and implement these methods in our task. Especially the method of Mini-batch smart construction and selection is very interesting to me, and I very want to see if this method will improve the results of my final model as I have expected.

References

[1] Y. Wang, Z. Lian, Y. Tang and J. Xiao (2018). "Font Recognition in Natural Images via Transfer Learning"

[2] Z. Wang, J. Yang, H. Jin, E. Shechtman, A. Agarwala, J. Brandt and T. S. Huang (2015). "DeepFont: Identify Your Font from An Image". arXiv:1507.03196 [cs.CV].

[3] C. Tensmeyer, D. Saunders and T. Martinez (2017). "Convolutional Neural Networks for Font Classification". arXiv:1708.03669 [cs.CV].

[4] A. Gupta, A. Vedaldi, and A. Zisserman (2016). "Synthetic Data for Text Localisation in Natural Images". arXiv:1604.06646 [cs.CV]. GitHub link: https://github.com/ankush-me/SynthText

[5] Draw.io site, online diagram software for making own diagrams. https://app.diagrams.net/

[6] Y. Dokuz & Z. Tufekci (2021). "Mini-batch sample selection strategies for deep learning based speech recognition". Applied Acoustics.

[7] I. Loshchilov & F. Hutter (2016). "ONLINE BATCH SELECTION FOR FASTER TRAINING OF NEURAL NETWORKS". arXiv:1511.06343 [cs.LG].

[8] T. Akiba, S. Sano, T. Yanase, T. Ohta and M. Koyama (2019). "Optuna: A Next-generation Hyperparameter Optimization Framework". arXiv:1907.10902 [cs.LG].