Computer Vision Final Project

Course: 22928 - Introduction to Computer Vision **presenter:** Ofir Paz

Date: 04.02.2024 **Id:** 329202717

Introduction

Font classification is a prevalent task in various applications today. It plays a crucial role in tasks such as optical character recognition (OCR), where script from images needs to be translated, and in converting text from real-world images into digital formats. In this project, we delve into the font classification problem, focusing on analyzing a specific dataset generated synthetically to train and evaluate models for this task. I will present a convolutional neural network (CNN) architecture that classifies letter images from seven possible fonts, and the algorithms that were implemented to further assist with the model's predictions. Throughout the whole project, I restrained myself from viewing the images in the test dataset since as a test dataset, it is not meant to be visible and to be a part of the decision-making process.

Data Exploration

As said before, the dataset consists of large images that contain synthetically generated words with given bounding boxes for the words and letters. Each word has a specific font out of seven possible fonts:

- Skylark
- Sweet Puppy
- Ubuntu Mono
- VertigoFLF
- Wanted M54
- always forever
- Flower Rose Brush

And the words are taken from some newspapers, as described in (Gupta, Vedaldi, & Zisserman, 2016).

Here are some examples of the first few images from the labeled dataset:

Images at indexes 0 and 10 of the dataset, on the left there are the word bounding boxes with the fonts of the words and on the right, there are the characters bounding boxes.

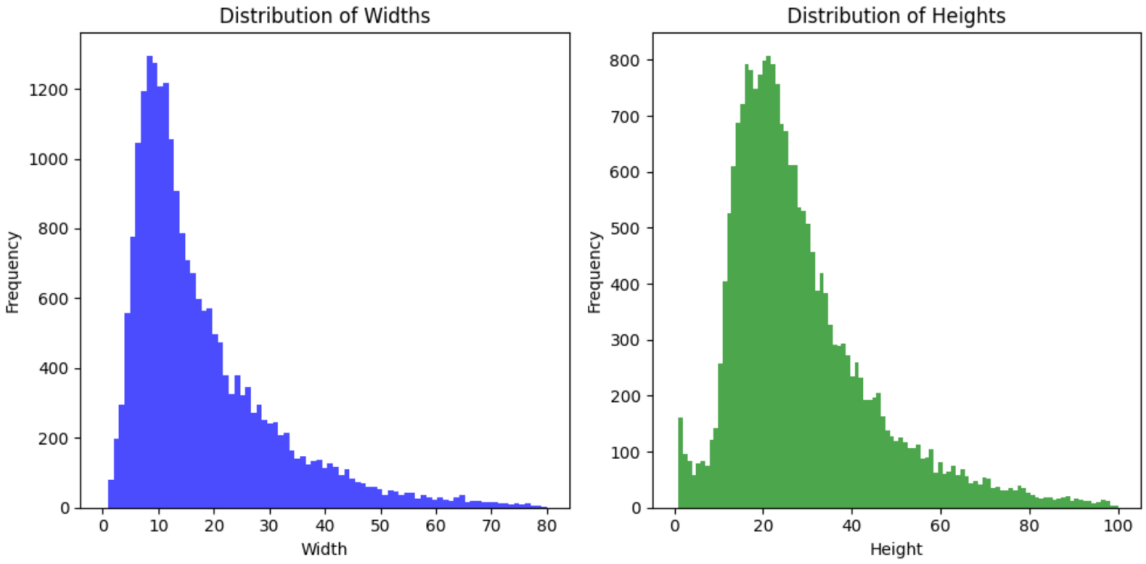
Notice how some of the characters are very hard to view and notice of. This will be one of the major issues in this dataset, that there are many words colored with the same color as their background (This is initially why I wanted to use color jitter augmentation, to make some variance between the background and the letters themselves).

After plotting some images, I created a function to crop out letter images for the larger images. Notice how the bounding boxes are not all parallel to the large image bounds, so I needed to implement some kind of perspective transformation. I used the public library OpenCV, and acquired the results:

The first image is a cropped and perspective transformed word, and the rest are each letter in the same order. Notice how each letter is a different image and has a different size. The word was taken from the large image at index 0 in the labeled dataset.

Continuing with my data exploration efforts, I addressed the issue where the letter images vary in width and height. Of course, for a CNN, we must resize all input images to a fixed

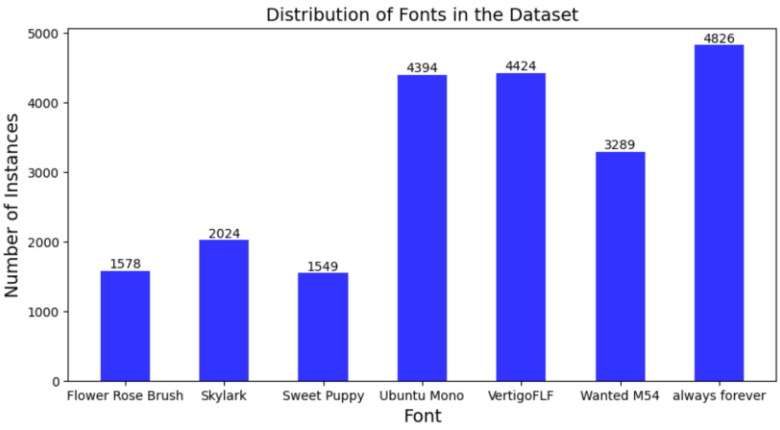
size (or do we? See the 'More Ideas' section). To address the size issue, I generated histograms to visualize the distribution of widths and heights for the letter images with different fonts. These histograms provide insights into the variability in image dimensions across the dataset. I also ran a script to calculate the mean of the widths and the heights of the bounding boxes, to choose a global size to resize all the cropped letter images for it. Here are the results:



$$\mu_{width} = 18.13, \qquad \mu_{heigt} = 29.13$$

I chose to exclude width and height values that exceed a certain threshold in the histograms for better visualization.

Additionally, I plotted the number of instances per class to understand the class distribution within the dataset. These visualizations offer valuable insights into the distribution of the fonts in the dataset, guiding subsequent steps in data preprocessing and model development.



We can see from the graph that the imbalance classes issue is present, but not on a great scale, which is a good thing for us overall. In many classification problems, classes are imbalanced to the point at which they need to be heavily treated. In this case, I decided not to see this as a problem since the largest difference between the number of occurrences of a class is only a factor of 2.5.

Data Preparation

In this section, we outline the steps taken to prepare the synthetic dataset for the font classification project.

The first stage of data preparation involved validating the integrity of the provided dataset. This included verifying the accuracy of the word bounding boxes and character bounding boxes to ensure the reliability of the annotations. In the validation, I found some images where the character bounding boxes are a single pixel! This is a major problem in the dataset, but I decided not to remove those samples since the test set had these (what I consider to be noise from the synthetic generation) bounding boxes. Another issue that came up, was that some of the word bounding boxes had point values that were outside of the larger images. Some of the x and y coordinates were even negative. I assume this again has to do with errors in the generation of the dataset. Anyways, this issue seems to not affect any aspect of this project since I used the word bounding boxes solely for plotting, and all the plotting function I used managed to handle this problem without even raising any warning.

Next, I considered applying data augmentation. Data augmentation refers to the process of artificially increasing the diversity of a dataset by applying various transformations to the existing samples. These transformations can include operations such as rotation, scaling, flipping, color jittering, and affine transformations. By augmenting the dataset in this manner, we introduce variations in font size, style, and orientation, thereby providing the model with a more comprehensive understanding of different font styles. I experimented with data augmentation and discovered the surprising result that in our case, it is less than helpful. Why is that? First, I only experimented with data augmentation that is relevant to our task. The bounding boxes indicate the shape of the letters, so random angle augmentation was irrelevant. To add to that the bounding boxes also make random crops and scaling irrelevant. Flipping is also irrelevant since letters are not prone to be flipped (Unlike cats for example that can be flipped horizontally but not vertically). So, the only common augmentation left is color jittering. In color jittering, you can randomly change the brightness, contrast, saturation and HUE. Experimenting with the first three, I was not surprised that they decreased performance. The images in the dataset are mostly bright,

have normal contrast and saturation. But I expected that adding random HUE jitter (random color change) will increase performance, but it turns out that it slightly worsened them. I expected performance to increase since the words are generated with pretty much random color, but my best assumption for these results is that the data is generated synthetically and therefore changes to it will make it difficult for the model to adapt to the nature of the colors that are generated. So, to sum up, in the final product augmentation was never used.

Following augmentation, I searched for values to resize all the images to. I decided to go with the values close the averages I calculated and presented in the last section, which where 20 and 30 width and height respectively. Those values were specifically chose because they are close enough to the average and I could make some clever kernel size choices later on.

Another important step in data preprocessing is normalization. Normalization of the dataset is a crucial step in preparing the data for model training. After resizing the images to a specific size, I wrote a script to calculate the mean and standard deviation of the dataset, which are essential for normalization. These values,

$$\mu = (0.46775997, 0.48115298, 0.48016804)$$
$$\sigma = (0.25156727, 0.24406362, 0.27375552)$$

which I calculated over the three-color channels (RGB) are then used to scale the features to a standard range, ensuring uniformity and facilitating convergence during training. Normalization helps in improving convergence, stabilizing the training process, and enhancing model interpretability. It also ensures that features with different scales are treated equally during model training, ultimately leading to better model performance and generalization on unseen data. By meticulously preparing the given synthetic dataset and integrating it into the data pipeline, we have established a solid foundation for training robust font classification models capable of accurately categorizing font styles across various classes.

Lastly, after finishing preprocessing the data, I split the data into training and validation sets. I chose 90% of the large images to be the training set and the remaining 10% to be the validation set. I experimented with different values and found that those yield the best results (Note: the data is split randomly for the best generalization).

The Neural Network

In designing the font classification network, I chose residual blocks to address vanishing gradient issues in deep architectures. Skip connections facilitated gradient flow, enabling effective training of deeper models. To combat overfitting, I also implemented dropout regularization with a rate of 0.5 at the last fully connected layer and L2 regularization with a starting value of 0.005. I chose this value after careful consideration and extensive experimenting. In the train loop, after every specific number of epochs the weight decay parameter increases by 0.005 to mitigate overfitting at the end of the learning, when the learning rate is at its lowest. These methods were selected to balance model complexity and generalization. Additionally, learning rate decay and epoch tuning optimized training efficiency and convergence. Cost and accuracy graphs provided insights into training dynamics, guiding hyperparameter adjustments. This iterative process led to a robust architecture capable of accurate font classification. The architecture itself is built inspired by the famous Resnet (He, Zhang, Ren, & Sun, 2015), but with many changes. For starters, I found that using two convolutional layers at the start of the architecture instead of the usual one layer. I've also chosen specific kernel sizes, which are inspired by (Zhang, et al., 2022). Following the starting layers there are 4 residual blocks with increasing channels and an average pool in the middle of them. See the full network architecture in detail in the table below.

Layer	Kernel Size	Stride	Output Shape	Parameters #
Conv2D	(3, 3)	1	(30, 20, 32)	960
Conv2D	(5, 5)	1	(30, 20, 84)	67,452
Residual Block	(3, 3)	1	(30, 20, 84)	127,344
Residual Block	(3, 3)	1	(30, 20, 114)	213,408
Average Pooling	(2, 3)	(2, 3)	(10, 10, 114)	-
Residual Block	(3, 3)	1	(10, 10, 114)	234,384
Residual Block	(3, 3)	2	(5, 5, 134)	315,168
Fully Connected	-	-	512	1,715,712
Fully Connected	-	-	7	3,591

I excluded batch normalization and activation function layers to minimize the table's complexity. Of course, after every convolutional layer, there was a batch normalization

layer following the ReLU activation function. Reaching this specific network came after many days and even weeks of experimentation, running different architecture and work. I've tried for many hours to improve this network with no success, even changing the kernels' sizes and stride values jeopardized the results. Notice that the averaging pool has a filter that is not squared, I chose this because of the input image dimensions proportions. This directly impacts performance, as changing it to any other value decreases them. Overall, I think this network is built very thoughtfully with many considerations being made, and it performs very well on the validation set, reaching 92% accuracy on it!

But you might be asking, if the network got only 92% accuracy, how was I able to submit a 98.057% accuracy prediction on the test set? This will be discussed in the following section.

Further Improvements

Our dataset has a unique property. It is given that all the letters in a single word share the same font, as well as the lengths of the words plus the matching indexes. Using this property, we can enhance the model's predictions on a dataset. I've proposed and implemented four algorithms that each significantly improve the model's predictions. I will present all four of them since neither of them was ultimately superior to the others.

1. Evaluate each letter's predictions probabilities, sum the probabilities that are of letters in the same word, take the argument of the maximum of the sum as the prediction for all the letters in the specific word. More formally:
   Let $l_1, \dots, l_N$ be letter images of some word. Denote our model as the function $h: \mathbb{S} \to \mathbb{R}^7$ ($\mathbb{S}$ is the letter images space), which returns a vector of size 7 where for each index $1 \leq i \leq 7$ in the vector, $h(l)_i$ is the probability that the letter image l_n has the ith font. Our algorithm assigns each letter $l_1, \dots, l_N$ the following font:
   $$\arg\max_i \sum_{n=1}^{N} \boldsymbol{h}(l_n)$$
2. Like the last algorithm, find the font that our model h predicted the most for the set of letters in some word, and if the number of occurrences is larger than 1, assign all the letters of that word the found font, otherwise stick with the original predictions.
3. This algorithm combines the last two. It first finds the most common font prediction of some word, and if the number of occurrences is larger than some threshold, set all the predictions of the letters in the word to the that font, otherwise use algorithm (1) to make the predictions.

4. This algorithm is like algorithm (1), but it incorporates a smarter scheme. Instead of just adding all the probabilities of the letters in the same words, it is multiplying each probability by the size of the original letter image size. Basically, we are using the original image size as a weight for the probability. Thus, we consider the probabilities of images with smaller original size to be less dominant, and the probabilities of images with larger original size to be more dominant. I've proposed this algorithm after noticing the diversity of the original images' sizes shown in the 'data exploration' section.

After implementing and testing all the presented algorithms, all of them showed very close results. A major improvement of the original 92% accuracy on the validation set to 98% was acquired, and 98.3% accuracy on the private Kaggle dataset. In the general case, I seemed to notice algorithm (4) and algorithm (3) with threshold set to 2 to be the best performing algorithms, while algorithm (2) to be the least performing of them all. In my final submission all the algorithms yielded the same results, which was a little bit surprising.

More Ideas

What I've discussed until now was the final approach I've used to tackle this classification problem. Other than the solutions presented, I've come up with a handful of other ideas.

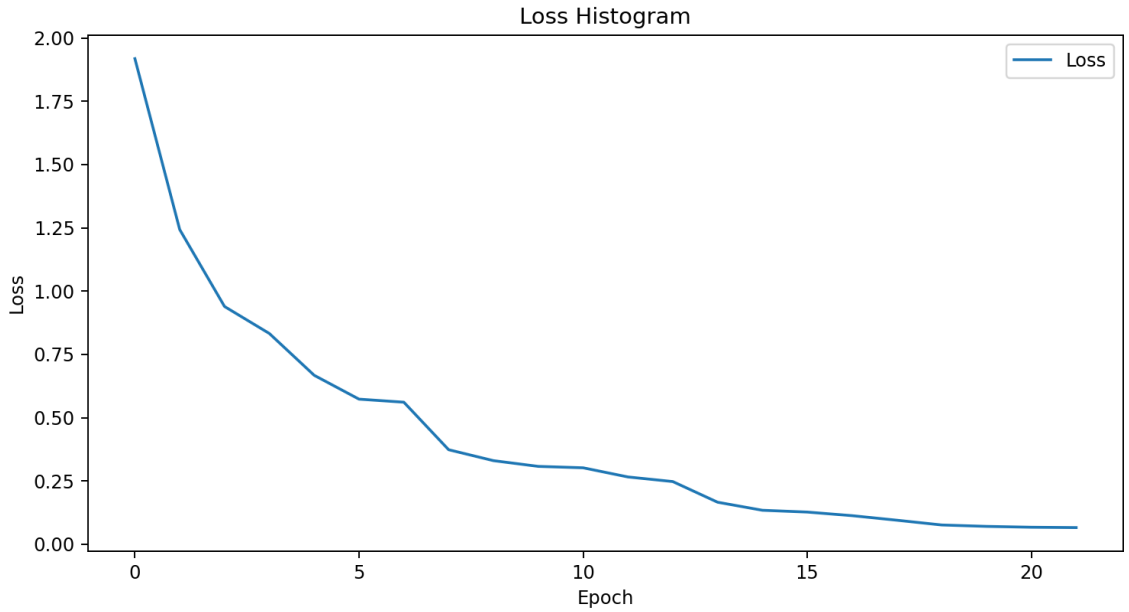
1. Due to the large diversity in images' sizes in the dataset, I experimented with an architecture that I've built that is not dependent on the input image size. Basically, the network is utilizing residual blocks and a dynamic averaging pool that is the size of the input image. To use this architecture, I've had to limit batch size to 1, which greatly impacted performance and training speeds. This led to ultimately making this technique not viable, so I've tried different approaches. The main reason for this idea to come up is that resizing images (with a function from the public library OpenCV) loses information.
2. Another idea was to utilize the architecture from the last idea, but to load in a mini-batch images with similar sizes and resize them all to their mean size. This idea I inferred to be the most promising of them all, but was ultimately left out due to complexity (implementing this would require me to implement pytorch's data loader)
3. The last idea was to create more data using the git-hub repository Egozi provided to us. I've tried to create more data for a few days, but since the repository was so outdated and designed for Linux (I use a Windows 11 machine), I've had to change too much of the source code which made this too much of a hassle. I proposed this

idea since I wanted to use a vision transformer model (ViT) (Bhojanapalli, et al., 2021) for the classification, which are known to perform very well on large datasets.

Finally, I think these ideas are novel solutions for this classification problem, but due to limited time and resources I've sadly not had the opportunity to implement them properly and make experiments.

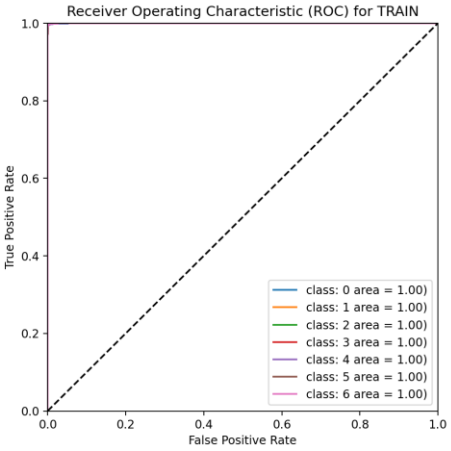
Results

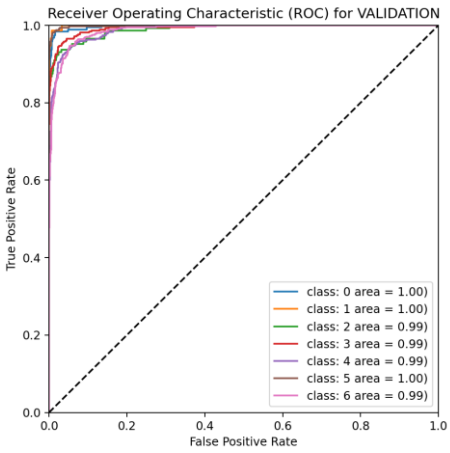
I am proud of my work and the results on the leaderboard. At the time of writing, I am in fifth place on the public leaderboards reaching 98.057% accuracy, and third place on the private leaderboards reaching 98.360% accuracy. The final submission was created with a carefully fine-tuned model, which was fine-tuned after 22 epochs where it was reaching 79.5% accuracy on the validation sets. Below is the cost graph for the first 22 epochs of training:



(Cost on the train dataset, calculating cost on the validation set impacted convergence of the learning, I suspect that must be some bug within the pytorch module)

I've also used the ROC graph and AUC for better understanding of the model's performance. It can be seen from these graphs that the model is performing very well on all 7 classes pretty evenly.





Conclusions

In conclusion, the font classification project successfully leveraged a combination of advanced techniques in network architecture design and regularization methods to achieve accurate classification results. By integrating residual blocks to address vanishing gradient issues and applying dropout and L2 regularization to mitigate overfitting, the model demonstrated strong generalization capabilities. Careful optimization of learning rate decay and epochs further enhanced training efficiency and convergence. Through rigorous experimentation and evaluation, the final architecture proved robust and effective in accurately classifying font styles.

References

Bhojanapalli, S., Chakrabarti, A., Glasner, D., Li, D., Unterthiner, T., & Veit, A. (2021). *Understanding Robustness of Transformers for Image Classification.* arXiv preprint arXiv: 2103.14586. Retrieved from https: //arxiv.org/abs/2103.14586

Gupta, A., Vedaldi, A., & Zisserman, A. (2016). *Synthetic Data for Text Localisation in Natural Images.* arXiv preprint arXiv: 1604.06646. Retrieved from https: //arxiv.org/abs/1604.06646

He, K., Zhang, X., Ren, S., & Sun, J. (2015). *Deep Residual Learning for Image Recognition.* arXiv preprint arXiv: 1512.03385. Retrieved from https: //arxiv.org/abs/1512.03385

Zhang, L., Shen, H., Luo, Y., Cao, X., Pan, L., Wang, T., & Feng, Q. (2022). *Efficient CNN Architecture Design Guided by Visualization.* arXiv preprint arXiv: 2207.10318. Retrieved from https: //arxiv.org/abs/2207.10318