Font Classification Project – Computer Vision

Ori Dabush

Table of contents

- The task and the dataset
- My working process
    - Research on pre-processing
    - Creating different datasets
    - Designing the model
    - Finding hyper parameters
    - Dealing with overfitting
    - Trying to create more data
    - Using the assumption that the whole word is in the same font
    - Creating the predictions .csv file
- Results
- How to run the code
- View my model's performance
- Reflection
- Appendices

The task and the dataset

The task in this project was to classify the font of a text in different images. The dataset contains images with synthesized text in them. The images were created using this code.

The dataset has 998 Images, 30520 Characters from 5 fonts. In addition to the images, each image has:

- Word bounding boxes
- Character bounding boxes
- Text
- Font label for each character

The possible fonts are:

- *Alex Brush Regular*
- Open Sans Regular
- Sansation
- Ubuntu Mono
- Titillium Web

An example of an image from the dataset and its bounding boxes:

My working process

Research on pre-processing

The first thing I did when I started the project was looking at the dataset and at the different images. I created a python notebook that views the images and the characters. I took each character and created its own 100x200 image using a projective transformation (an homography).

I noticed that some of the images are unclear and can be confusing, and in addition to the previous knowledge that I had about the need to perform data preprocessing, it motivated me to start researching about preprocessing methods.

I started reading articles about data preprocessing in deep learning, and specifically about images preprocessing. The links to some of these articles will be attached in the end.

After some research, I started listing the possible preprocessing methods and tried to implement them:

- Converting the image to grayscale
- Transforming each character to its own image using a homography (a projective transformation)
- Normalizing values from [0, 255] to [0, 1]
- Transforms:
    - Sharpening filter
    - Edge detection filter
    - Gaussian blur
- Normalization methods:
    - min-max normalization
    - z-score normalization

I played with these transformations and looked at the data after many combinations of these transformations, and decided to use the following preprocessing method:

- Converting the image to grayscale
- Normalizing values from [0, 255] to [0, 1]

- Applying gaussian blur to the full image
- Transforming each character to its own 100x100 image using a homography (a projective transformation)
- Applying a sharpening filter to each character
- Normalize each character using z-score normalization (the mean and std were calculated from the characters in the train dataset after all the previous transformations, and are saved in the 'data' folder as char_mean_100x100.txt and char_std_100x100.txt).

This is an example of the preprocessing sequence on a specific image:

I chose these specific transformations to be in the preprocessing because these are the transformations that made the characters the clearest and the least confusing. In addition, as I'll describe soon, this transformation sequence gave good results comparing to other transformations using a simple CNN model I started with.

The final transform functions can be seen here and also in the datasets that will be discussed next.

Creating different datasets

In this project I decided to use PyTorch. I decided to use it because I'm already familiar with it and its different functionalities.

In PyTorch, in order to train your model, you'll have to use a data loader. The data loader requires an object of a Dataset type. I implemented multiple datasets in order to use multiple functionalities.

In the beginning I thought that loading all the characters to the memory will be very heavy and won't be efficient, so I implemented a dataset that loads the characters only when the character is accesses. This is the SynthTextCharactersDataset class.

After using it, I noticed that although it loads the data very fast, it will waste time in the next levels because each character will be accesses multiple times, and it won't be efficient to load it every time it is accessed.

Because of the reason I mentioned, I decided to implement another dataset that loads all the characters and performs the transformations once in the constructor. This is the SynthTextCharactersDatasetRAM class. This dataset takes more time to load the data than the previous one, but it is more efficient in the next steps.

There was a memory vs. time tradeoff that I had to decide on, and I decided to sacrifice some memory and load all the characters once in order to save time when training the model and accessing these characters.

There is another dataset that I implemented which the reason that it is needed will be explained later. This is the SynthTextCharactersDatasetTest class. This dataset loads all the characters to the RAM just like the previous dataset, but its structure is different – accessing an element in this dataset will return multiple characters which assembles the word in the index that was accessed. This dataset will be used later for predictions only.

Data augmentation

In order to allow generalization and also in order to create more data I used data augmentation. Specifically, I randomly applied the following transformations to the given input as part of the models:

- Gaussian blur
- Horizontal flip
- Vertical flip
- Color invert
- Brightness and hue modification

These transformations allowed the model to train on more data, and I saw the improvement of the models with these layers.

Designing the model

When designing the model, I researched multiple architectures of image classifiers, and specifically of text and font classifiers. The main models I found are ResNet models (specifically ResNet 32), vgg models, and some transfer learning models I found. I decided to focus on the transfer learning and the ResNet 32 models.

I implemented both of these models as PyTorch requires the models to be implemented. These implementations can be seen here.

Both of the models' architecture can be seen in the appendices in the end of the report. The transfer learning architecture is simpler (it is less complicated and also faster) and also gave better results, and that's why I chose it to be the final model.

Finding hyper parameters

In order to find the best hyper parameters and to check the model performance with them, I created a script that runs over multiple combinations of parameters, trains a model, and logs its performance on the validation set.

The different hyper parameters I used:

- Learning rate
- Number of epochs
- Loss function
- Optimizer
- Batch size
- Train-validation percentage
- Model
- Weight decay (only with Adam optimizer)

For each combination of hyper parameters, I plotted an accuracy per epochs and average loss per epochs on both the train and validation sets, and also saved the model weights.

Dealing with overfitting

After some runs of the different models, I noticed that my model tends to overfit the training data. I noticed it in the performances – the accuracy on the validation

set was lower than the accuracy on the train set. In addition, it can be seen in the average loss per epoch graph, where we can see the train loss keep lowering and the validation loss start rising in some point.

I tried multiple methods to prevent this overfitting from happening, or at least lower its effect:

- Dropout layers – layers that zero some of the values of their input with a given probability.
- Batch normalization – normalizing the data in the current batch.
- Early stopping – stop the training after less epochs.
- Data augmentation – I already mentioned this method, but it also helped deal with overfitting as it created more data to train on.

I also tried to play with these layer's parameters, for example with the number of batch normalization layers and the probability of the dropout.

In the end, I decided not to use batch normalization because it lowered my model's accuracy and didn't help very much. I decided to use dropout layers in my model, and I also used early stopping by saving the model weights that lowered the average loss on the validation set. In addition, as I already mentioned, I used data augmentation.

The script that was used to find the different combinations and run the model can be found here (of course I changed the values of the parameters options manually and it is not updated).

Trying to create more data

Between trying different models and parameters and dealing with overfitting, I also thought of creating more data to train of.

I first started by looking at the SynthText repo that was given to us. This repo was a little bit broken and had some bugs in it, but I used this repo, which offers a nice UI to create data using the SynthText repo. I managed to generate more data, but it has a flaw in it that I didn't manage to fix – some of the text didn't fully appeared in the image.

After failing with the SynthText repo, I tried to search for other libraries that offer the same (or at least similar) service. I found a python library named synthtiger

which also generates text on images. I managed to modify its code and config files to fit in my needs, but the data that was generated didn't fit my needs – it wasn't similar to the dataset that was given to us, and it only made the performance worse.

So, unfortunately, I didn't use any additional data in the end.

Using the assumption that the whole word is in the same font

My next step in my way to improve my model was to use an assumption that we had on the data – all the characters of the same word were in the same font. In order to use this assumption and improve my model, I used the SynthTextCharactersDatasetTest that I mentioned before, and predicted the font for every character in every word. Then, for every word I took the font that was predicted by the most characters in this word (tie are decided by the minimum label).

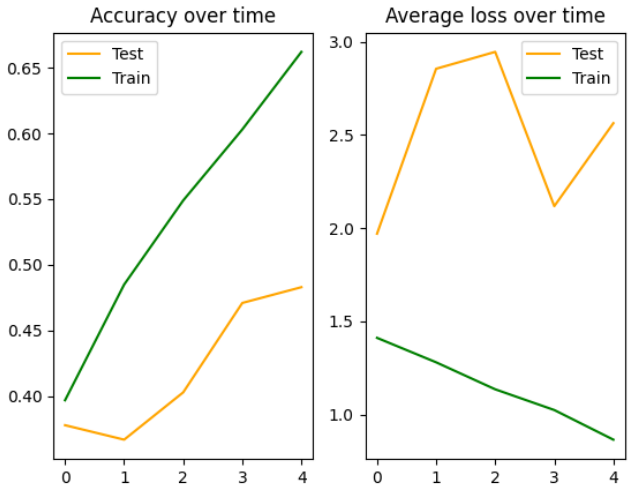
Creating the predictions .csv file

The final step was to create the predictions .csv file in the required format. The script that does it using the word majority font decision process that I described before can be found here.

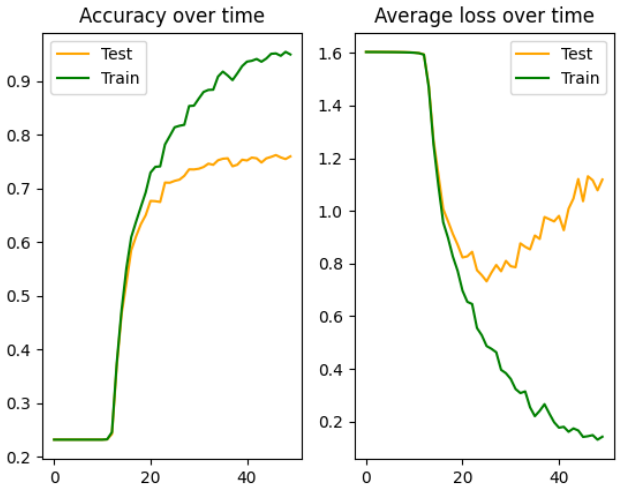
Results

In this section I will show some results of my model.

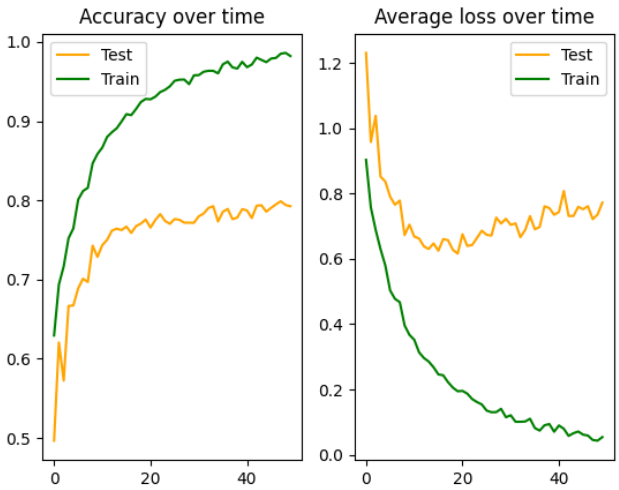
This is a run of 5 epochs on the ResNet 32 model. We can see that it is not very accurate, and it is also really slow (this run took around 30 minutes on a GPU, and it is only 5 epochs). This run reached 48.3% accuracy. From now and on, the graphs will show the transfer learning model I chose and not the ResNet 32 model.



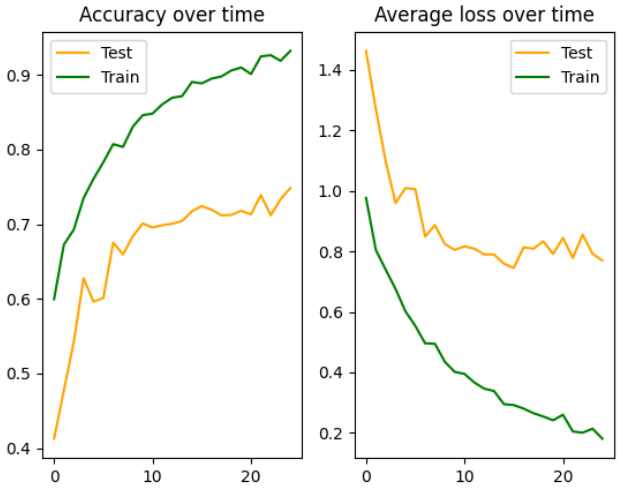
In this run we can see the overfitting very clearly. We see that starting at epoch ~20 the average loss on the validation set start to rise. This model reached 71.3% accuracy.



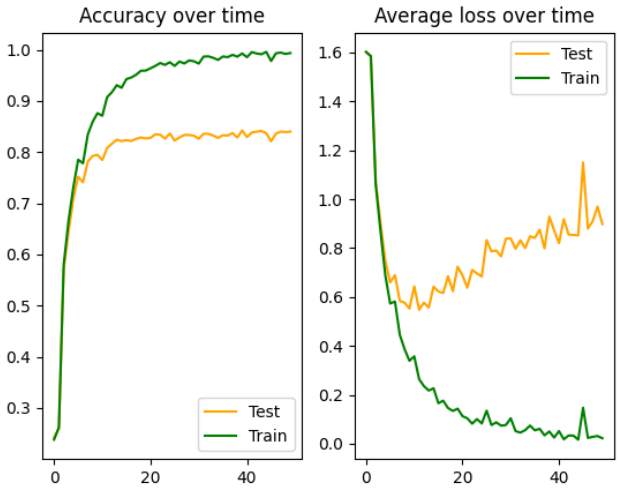
Here we can see the run when I decided to add the brightness and hue shifting transformation to the data augmentation layers. We can see that there's still some overfitting, but it is better, and also the accuracy on the validation set has improved. This model reached 80% accuracy.



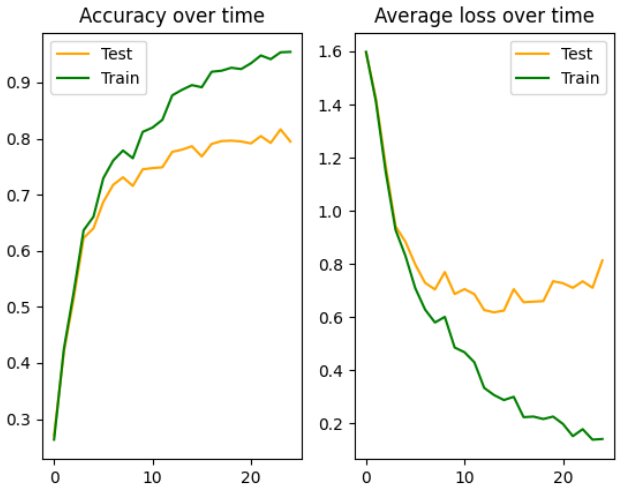
This is a run with more dropout layers that I tried in order to lower the overfitting. In this model I added a dropout layer, and raised the probability to 50%. We can see that it helped the overfitting, but also lowered the accuracy according to other runs, which is why I decided to use dropout layer with smaller probability. This model reached 72.4% accuracy in the validation set.



This is a really good run. In this run I improved the model by adding some more convolution layers to it. This run reached 84.2% accuracy. We can still se some overfitting here, but it is handled by the early stopping, which saves the model which lowers the average loss on the validation set the most. This model is the one of epoch ~15.



This run is the run of the best model I got, and this is the model that will be used for the test dataset. This run got 79.5% accuracy.



All these results are without the assumption that every character in a word is in the same font. Using this assumption, the accuracy of the best model jumps from 90.2% to **96.1%** on the train set and from 79.5% to **87.6%** on the validation set.

How to run the code

There are multiple scripts that you can run in the project. For all of them, you'll need to clone my font-classifier repo at the main branch. If you want to use my pre-trained model (for the testing scripts), make sure to download it from here and replace it with the empty file in this path: 'models\all_models_without_perms\FontClassifierModel_0.01_25_CrossEntropyLoss_SGD_32_0.8.pth'.

You'll also need to install some dependencies:

- numpy
- matplotlib
- h5py
- pytorch (Pay attention to install the CUDA version. The code will also work without it, but won't finish in reasonable time)
- open cv

Now you'll have multiple options:

- To train a new model, you can run the 'model_training.py' file. You can also edit the model's parameters in the code.
  To run this script, you'll need to have the train data in 'Project/SynthText_train.h5'.
  This script will create multiple files:
  - outputs/<model_name>.png – the accuracy and average loss graphs (as you saw in the results section).
  - outputs/<model_name>_permutation.txt – the permutation that was applied on the characters before the training.
  - outputs/results.csv – a file that will contain the results (test accuracy and average loss)
  - models/<model_name>.pth – the saved model weights.
- You can also run the 'finding_hyper_params.py' file to train multiple models with different hyper parameters. This script runs the 'model_training.py' file with all the combinations of the parameters that were given. You can change the output directory and the parameters options in the main function.

- To view your model's accuracy with the word assumption, you can run the 'model_testing_nb.ipynb' notebook. Pay attention to change the 'model_name' value to your model's name (the one that you assigned when you ran the two previous scripts) and the model type if you used a model which is not the 'FontClassifierModel'.
  This script will output the accuracy in the following form:

```
Test stats:
 With voting accuracy: 87.68020969855833%
 Without voting accuracy: 79.53800786369594%
Train stats:
 With voting accuracy: 96.09682175622542%
 Without voting accuracy: 90.26048492791612%
Overall stats:
 With voting accuracy: 94.413499344692%
 Without voting accuracy: 88.11598951507209%
```

- To test the model on the test set and generate the .csv file as I submitted, you'll need to run the 'model_testing.py' file.
  To run this script, you'll need to have the test set in 'Project - Test Set\SynthText_test.h5'. Pay attention to change the value of the 'model_name' variable to your model's name (the one that you assigned when you ran the two previous scripts) and the model type if you used a model which is not the 'FontClassifierModel'.

Pay attention that in all the scripts that uses your model, the weights file named '<model_name>.pth' must sit in the 'models' directory.

View my model's performance

You can view my model's performance on the given test set in the 'report/test_labels.csv' file. This file is in the required format.

Reflection

I really enjoyed coding this project and researching and learning about different concepts related to deep learning and computer vision.

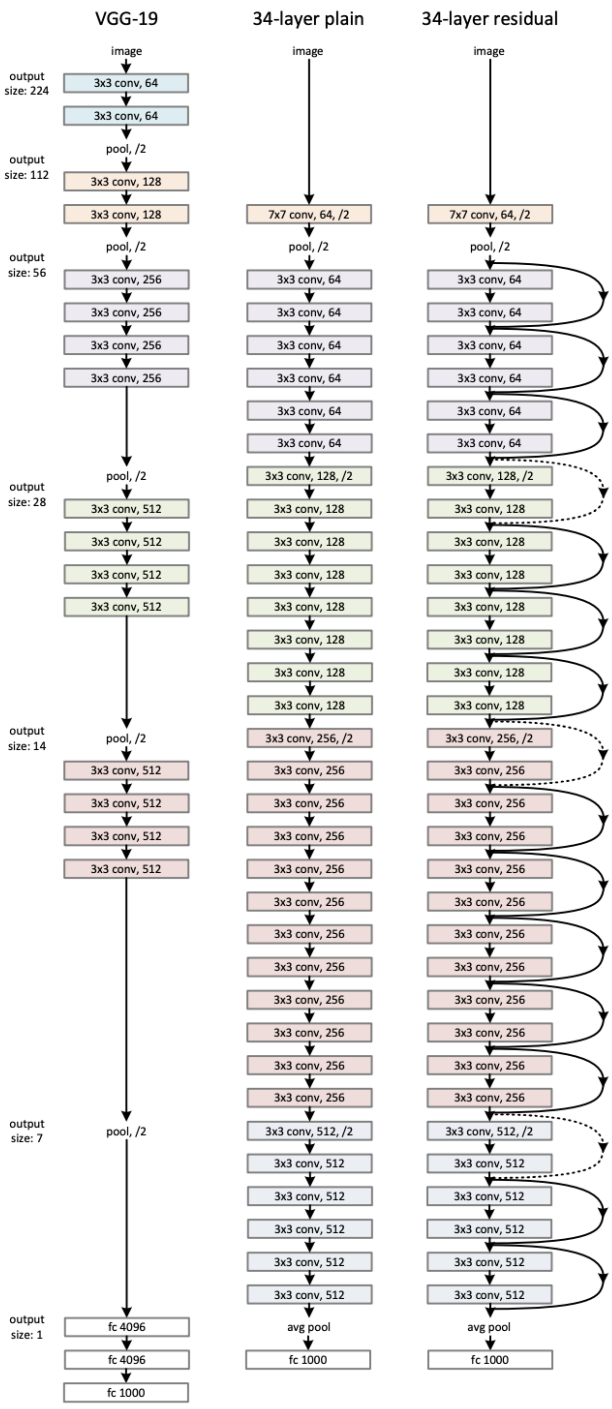
It was very fun to do a project at this scale, and try to face my problems during this project from different perspectives.

Appendices

These are the articles I read during my research about data preprocessing:

- https://www.isahit.com/blog/what-is-the-purpose-of-image-preprocessing-in-deep-learning
- https://www.v7labs.com/blog/data-preprocessing-guide
- https://www.freecodecamp.org/news/https-medium-com-hadrienj-preprocessing-for-deep-learning-9e2b9c75165c/amp/

This is the architecture of the ResNet32 model:


VGG-19
34-layer plain
34-layer residual
image
output size: 224
3x3 conv, 64
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
3x3 conv, 128
pool, /2
output size: 56
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
output size: 28
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 14
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 7
fc 4096
fc 4096
fc 1000
output size: 1
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 256, /2
3x3 conv, 512, /2
avg pool
fc 1000

This is the architecture of my model:

```
FontClassifierModel(
  (conv_layers): Sequential(
    (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (9): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1))
    (10): ReLU()
    (11): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (12): Dropout(p=0.25, inplace=False)
  )
  (linear_layers): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=8192, out_features=4096, bias=True)
    (2): ReLU()
    (3): Linear(in_features=4096, out_features=5, bias=True)
  )
)
```