Handwritten characters recognition

Nicolas Denier

1 Introduction

Text recognition and more specifically handwriting recognition is an important problem to look at. A lot of old scripts and paperworks was done by hand and need now to be digitized, it can help blind people to gain autonomy, or a bank to handle cheques. In any cases, it is a difficult task because of the variations in strokes, the proximity of characters, the unfixed size of characters, words, etc. This report will focus on the first step of this complex challenge: identifying characters.

1.1 Dataset

The dataset is composed of 3410 labeled images of handwritten characters, in which 62 classes are represented:

- lowercase alphabet: from a to z
- uppercase alphabet: from A to Z
- digits from 0 to 9

Each image is 900 × 1200 × 3 (height, width, RGB) with values going from 0 to 1. They are cleaned and the characters are black in a white background. The RGB channels are thus useless, only one is necessary.

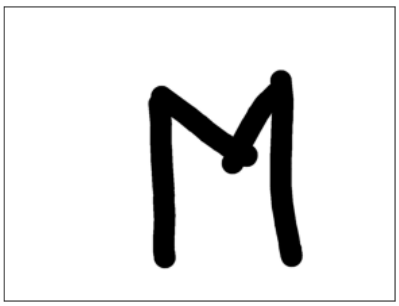

Fig. 1: Sample of the dataset: handwritten M

A CSV file containing labels and path to each corresponding image is provided to easily link labels and images.

1.2 Objectives

The first objective is to train a neural network that will be able to identify the different classes better than at random (a random prediction will have a $1/62 \approx 1.6\%$ accuracy). Then an interesting milestone would be to reach 80% of accuracy, and ideally with a quite fast computing time, let's say less than 5 min train with Google's GPUs (when Google is not deciding to restrict their use).

2 First model

The data is prepared in batches of size 64, each image is transformed to grayscale and resized to (90, 120, 1) and split randomly into 80% training and 20% test sets (test set is also validation set).

The chosen base model is constituted of 3 consecutive 2D convolutional layers of increasing depth (with ReLU as activation), with a 2D max pooling after each one. The result is then flattened to a dense layer and finally another

dense layer with as many nodes as classes (62) and a softmax activation. This model was already used for image recognition, it is expected to give at least 40% accuracy and reach the first objective (better than random).

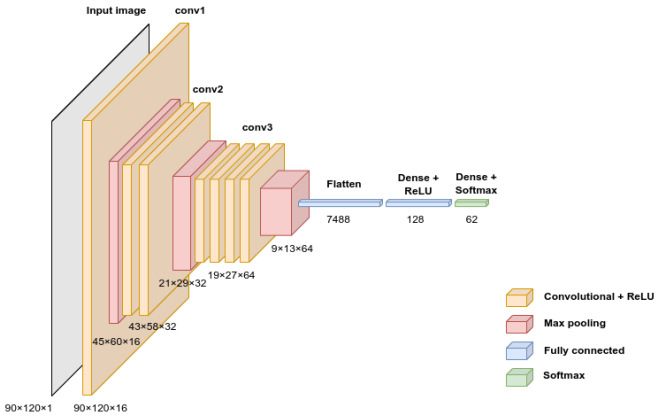

Fig. 2: Base model architecture

For the compilation, an Adam optimizer is selected, with a learning rate for now at 1×10^{-3}, the loss is sparse categorical crossentropy.

Trained for 10 epochs, a maximum accuracy of 56% is obtained. An important overfitting is observed, the model can definitely be improved.

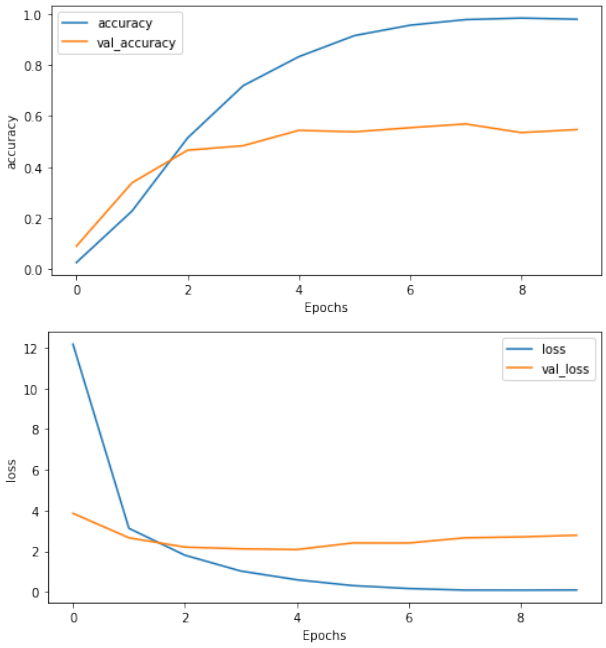

Fig. 3: Base model performances

3 Improvements

3.1 Adding complexity

To have a better model, adding layers can be tried. Let's add a similar 2D convolutional layer, with more filters (128) at the end of the model (just before the flattening). Also, the kernel sizes are changed to have first bigger kernels and then smaller ones.
A better accuracy is reached: around 71%, it is still overfitting.

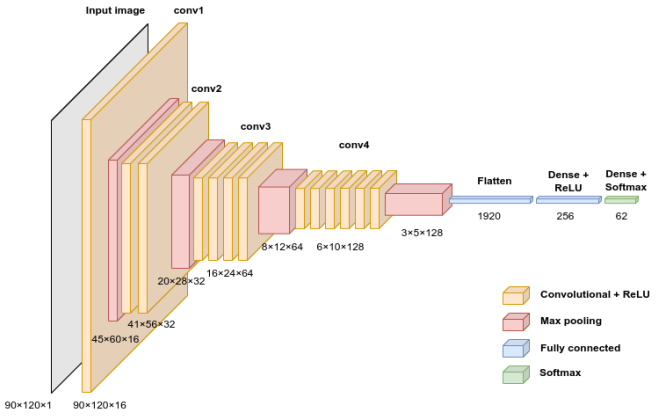

Fig. 4: Improved model architecture

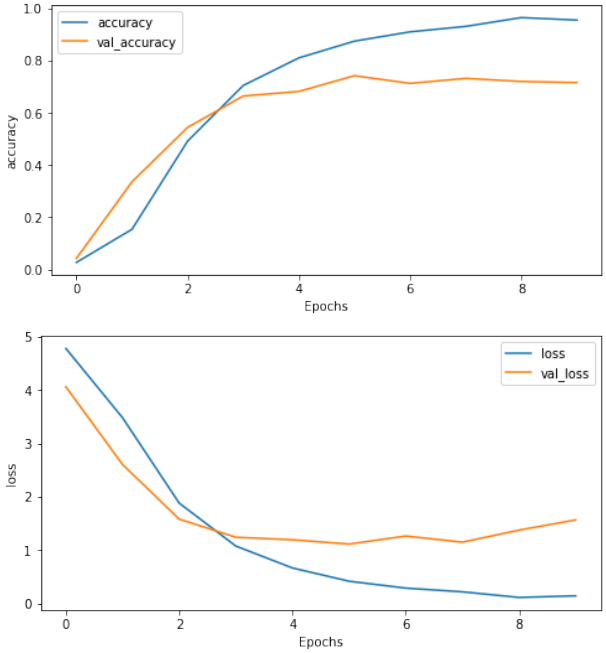

Fig. 5: Adding one layer

3.2 Dropout

In order to reduce overfitting, a few different methods can be used. Dropout is one of them, by disabling random nodes the model will better generalize. In return, the training time will be longer to achieve similar results.

Applied between several layers, with dropout rates between 0.3 and 0.5, the accuracy is 70% after 40 epochs. From the curve, we can see that the overfitting is indeed reduced. The validation accuracy is even over the training one, this is because dropout only apply during training. It can be observed that at the end, the learning curves seems not to have reached a plateau yet, so more epochs could lead to better scores. It looks like it has difficulties to start, as the accuracy stays below 0.1 until the 7^{th} epoch.

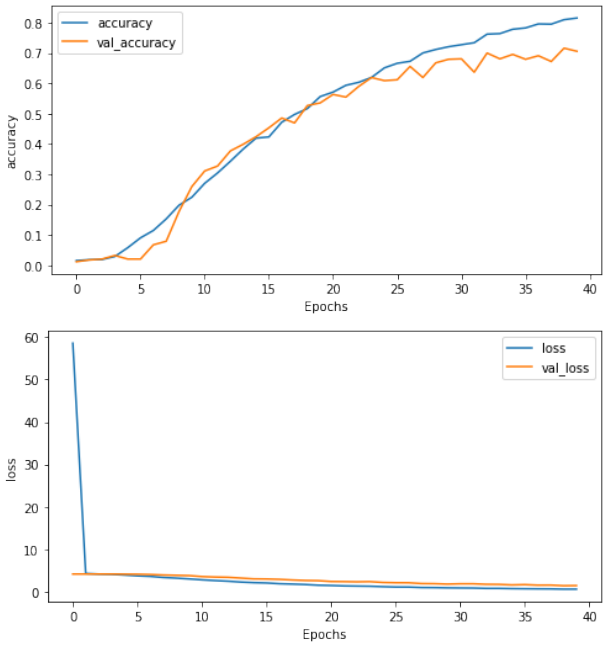

Fig. 6: Dropout is helping reducing overfitting

3.3 Data augmentation

Another way to prevent overfitting, and to have a more robust model, is to apply random transformations to images, or adding noise to make the model look at the overall features and not focus on each single pixel. For that, a random rotation and a random zoom are applied, with small parameters so it stays relevant data (rotating a character to 180° will not help to classify it). Also, a gaussian noise is added, with a standard deviation of 0.3.

The result is a longer training time, the learning curve is more chaotic but is goes up to 73%.

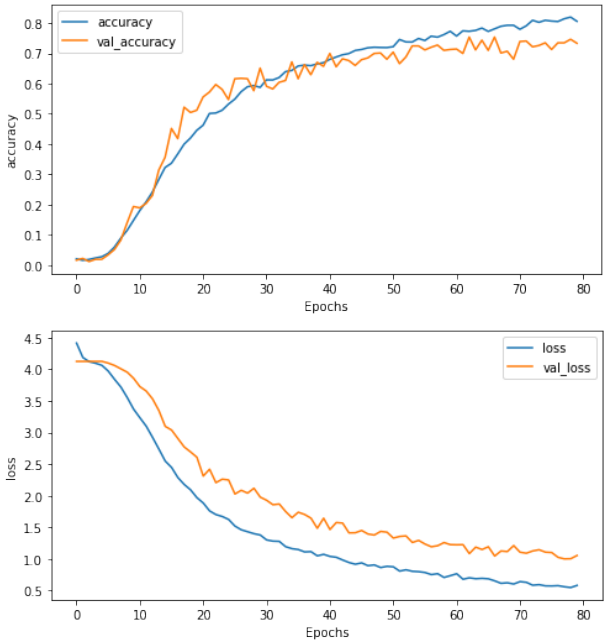

Fig. 7: Image augmentation makes it harder to learn

3.4 Adjusting learning rate

Now, the model is acceptable, but it still can be improved. the learning rate will be tuned to not be stuck in a plateau nor overshooting the lowest gradient. For that, a callback is implemented, called learning rate scheduler, and it will decrease exponentially the learning rate after a certain epoch (here 50). Another callback, early stopping, is added to stop the training if nothing improves for a few epochs.

Slightly better results are obtained, the model stabilize around 76% accuracy after 100 epochs (for a training time around 2 min). However, it is not easy to find the right ways to tune the learning rate. A higher one at the beginning was expected to reduce start time but it has not changed.

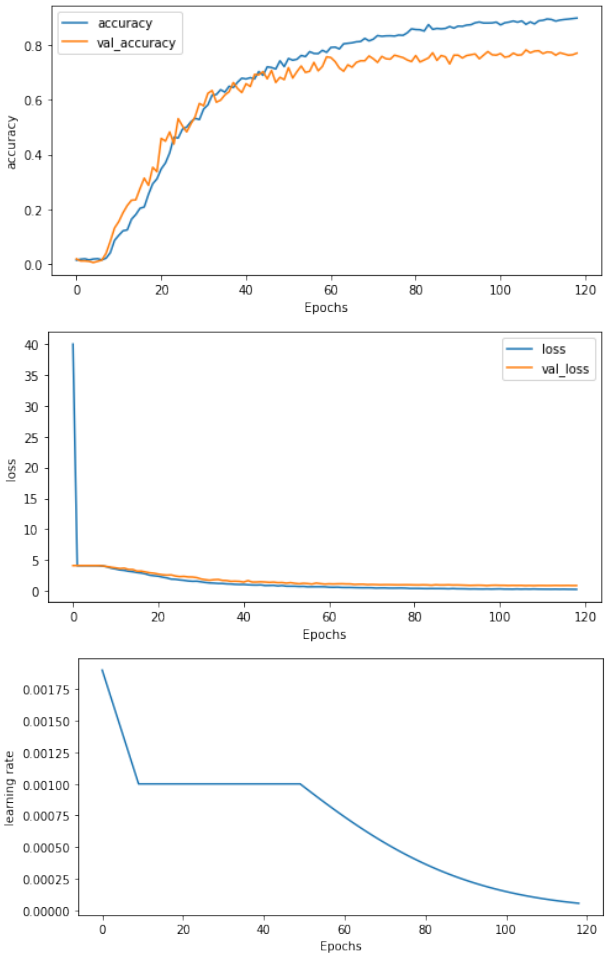

Fig. 8: The effect of small changes to learning rate are not obvious

4 Conclusion

4.1 Observations

By observing the learning curves of the model, and adding small improvements step by step, a simple model has been augmented sufficiently to be used in a real situation. The trade off of a better model is a longer training time, more power and resources usage, but it is not a linear relation. Also, a very complex model will not be better if it is not correctly designed.

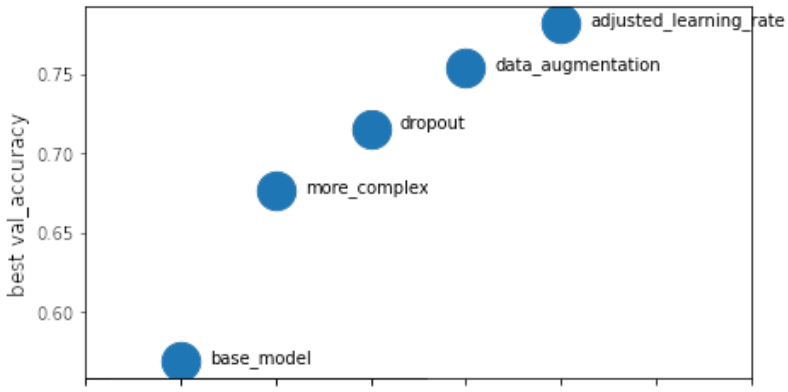

Fig. 9: Improvements over models

The learning rate can have an important impact, but it is difficult to tune. Similarly, small parameters change like dropout rate or convolution kernel size are not always resulting to explicit modifications on the model behavior.

Taking a look at the misidentified characters is interesting to get ideas on how to improve the model. In some cases, the data itself is too scrambled, an even an human can guess it wrong. In other situations, it can be deduced that the model should less take into account the serifs, that can make a letter looks like another. Finally, for some samples, the character seems perfectly identifiable but for some reason the model is guessing something else. That the case of the I in *Fig. 10* that is identified as an x.

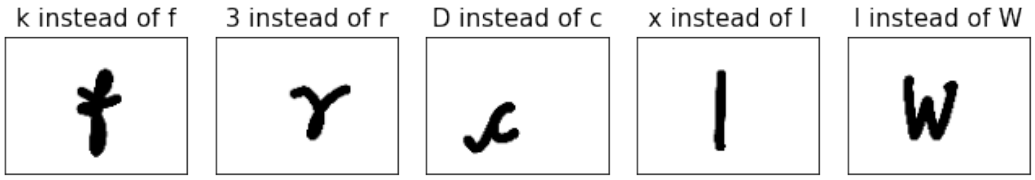

Fig. 10: A few erroneous predictions

4.2 Next steps

This was the first step of a full handwriting recognition pipeline, where several consecutive characters are identified, and some natural language processing is used to correct each character regarding the surrounding ones.

The dataset used here is nice to have a basic experience of characters classification, but it is very clean compared to real life data: completely white background, each character is alone and well centred, etc. A handwritten word is already more complex, because of the uncertain length and the links between each character.

5 References

GitHub repository: https://github.com/NicolasDenier/Handwriting-recognition

Dataset: https://www.kaggle.com/code/dhruvildave/starter-english-handwritten-characters/data

Introduction to handwriting recognition: https://nanonets.com/blog/handwritten-character-recognition/amp/

A database of words and sentences for next steps: https://fki.tic.heia-fr.ch/databases/iam-handwriting-database

Comparison of learning rate schedulers: https://neptune.ai/blog/how-to-choose-a-learning-rate-scheduler

Architecture diagrams base: https://github.com/kennethleungty/Neural-Network-Architecture-Diagrams