h e p i a

Haute école du paysage, d'ingénierie
et d'architecture de Genève

OCR Project Report

Authors: Sergey Platonov

Adrien Lescourt, Valerie Duay

April 2020

DESCRIPTION

First experience with Machine Learning and Computer Vision concepts.The goal of this project is to create an application that allows for handwritten digit recognition. We have a certain liberty with the tools we use for communication and infrastructure, but we have to use Tensorflow Keras for the ANN modelisation. Below you will find an explanation on how the application is set up, and what choices have been made. I shall start with the general architecture of the app, followed by a part covering the ML components.

Architecture

Front-End

The FE of the application is coded using the PyQt5 library, allowing for creation of custom GUI. The communication with the BE happens using Python's requests library with POST requests on predefined routes.

UI

The GUI consists of a number of elements that are necessary for our use case:

- **Canvas**: allows for drawing out handwritten digits, which are then easy to extract.
- **Submit Button**: sends the canvas content to the appropriate route, depending on other criteria.
- **Erase Button**: erases the content of the canvas.
- **Fit Button**: sends a signal to the BE that triggers the ANN training procedure.
- **Label Input**: allows us to label our data.
- **Result Labels**: displays the results of our manipulations.
- **Variable Inputs**: allow us to control the hyperparameters of our ANN from the FE.
- **Train/Test Radio Buttons**: mode selection.

These elements are all defined in the MainWindow class (except for Canvas), as well as their functionalities.

Error Handling

There are two error handling methods employed in this application, hard and soft. The hard error handling occurs in two scenarios, triggering a pop-up error message:

- submitting an image into the dataset without labeling or wrongly labeling it.
- sending the "fit" signal without giving all of the hyperparameters.

The soft error handling is performed in a quiet manner - the faulty operation is not performed. Also, when the FE is awaiting a response from the BE, the UI becomes unresponsive. Any answer (positive or negative) from the BE unblocks the UI.

Depending on the "mode" the user wants to be in (train or test), only the elements that are of importance for that role are displayed, contributing to error avoidance and cleaner UX.

UX

Even though this is a relatively short project, I decided to still give some attention to user experience. As mentioned above, only the relevant content is displayed to the user depending on the mode. Other examples of UX optimisation are:

- After an image has been added to the dataset, the canvas is wiped - allowing for faster composition
- The results of the training and evaluation are displayed in a minimalistic manner, allowing for quick overview of the operation.
- After an evaluation, the canvas is not wiped, giving the user the opportunity to rethink whether he/she had written a 6 or a 0.

Back-End

The BE of the application is handled by Python's Flask library, allowing easy route definitions and communication flexibility.

Routes

The choice of routes has been guided by the different uses of the application, and the variables that needed to be transported over the network in each case. I wouldn't dare calling it a REST API, as it does not do that. The principle is similar though, so let's call it an AFE API (Add, Fit, Evaluate).

-
- Add an image to the dataset: image as JSON string, with the label value in the path.
- Fit the model: empty string in the data (it's a trigger), with the hyperparameters in the path.
- Evaluate an image: image as JSON string.

The route path strings are constructed in the FE, based on different criteria that describe the user's intentions.

ML

The general idea behind the image analysis has been the following: an image needs to be collected, normalized (preprocessed) and classified before it can be used for training. The same steps apply for evaluation, minus the classification. Since this is my first experience in ML, I decided to leave the hyperparameters open for modification from the Front-End, allowing for experimentation and learning.

Libraries

- Keras for ANN
- OpenCV for preprocessing
- sklearn for dataset manipulations
- imutils for experimentation with paths
- matplotlib for plotting

Image Preprocessing

In order to allow for handwitten digits coming from sources of different canvas sizes, I implemented the following steps in my preprocessing routine (in this order):

- **inversion**: as we are interested in white on black.
- **centering**: calculates the center of mass of the image and it's offset from the center of the area. The image is then shifted such that the center of mass is in the center. This allows us to correct for images drawn in weird places, which could mess up the next step.
- **stretching**: A rectangle exists that can enclose the drawn image. The coordinates of such rectangle are calculated and the the surface inside is stretched over the entire surface. This compensates for different digit sizes.
- **resizing**: the goal of this project is to work with 20x20 image arrays. As such, a resize is needed.
- **blur**: the image is blurred, generalizing the general shape of the figure. An example of why this is important could be a poor sighted person such as myself: without glasses, two similar objects appear blurred and this can lead to confusion. This effect is what we are after - remove unique traits, confusing a machine into thinking two digits are same if they are similar enough.
- **binary thresholding**: the side-effect of blurring is that we find ourselves with a whole spectrum of grayscale values. Since we are only interested in pure black and white, we only select the regions that are "important" enough, and light them up, dimming the rest.

Thus, we end up with our normalized image array.

Classification / Labeling

Once our image is normalized, we then want to either insert it into our dataset, or to evaluate it. For evaluation, no classification is required. As for appending to the dataset, we need to label it correctly if we want to find it back afterwards. The label sent through the http request is a digit (0-9). We take our image, and insert it into the digits directory, inside the corresponding sub-directory named the same as the label. As for the name generated: I decided to just name it after the number of elements already present in the sub-directory. The images are saved as in the PNG format.

ANN Model

In order to be able to play with the values, I decided to let the following hyperparameters be configurable from the FE:

- batch size.
- the size of the test set in % of the whole.
- number of epochs.
- the number of neurons in the middle layer.

As a complement to that, I create and save a plot that describes the most important metrics of my model over the epochs. The trained model is saved, as well as the label data. This can be found in the model directory.

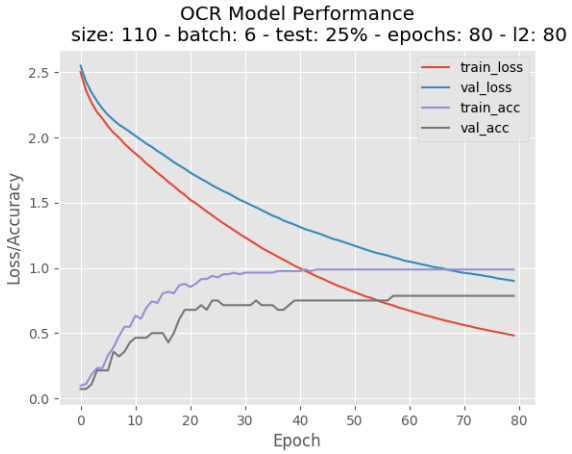
The model I use is Sequential, with all the layers densely connected. The hidden layer has a sigmoid activation function, as required for the categorical crossentropy loss function in the end. The latter is used in problems that attribute one label and one label only to a given data item. Also, I use a stochastic gradient descent optimizer (SGD), because I read that it boosts the higher values and lowers the lower ones. This seemed important as our dataset is very small (100 total samples), and we need all the accuracy we can get.

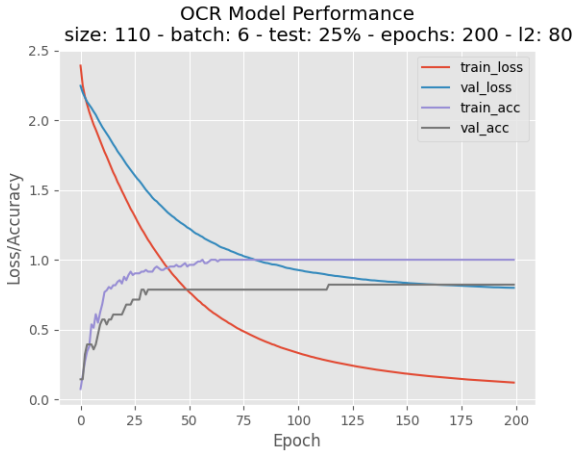
Performance

For the performance evaluation in the FE display, I decided to use the f1 metric, as it seemed more reliable for this context. Precision and Recall both rely on specific sensitivity on false negatives and positives, and f1 offers an "entre deux".

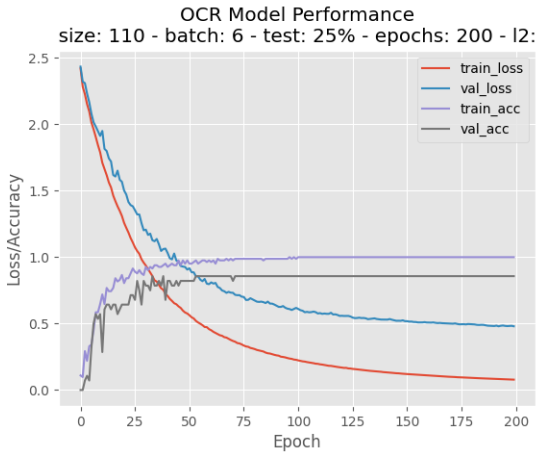
For each model build, we can consult the graph that shows us the performance of said training. The hyperparameters from FE are also listed in the title, allowing us to notice how those parameters affect the result.

I don't feel confident enough yet to make an entire analysis of my model - it has been tuned and tested by hand every time. I did notice an interesting relationship between the training loss and evaluation loss curves: once they start diverging, we are at risk of overfitting. Sadly, this occurs often with my dataset, and I don't yet have enough theoretical knowledge on how to adequately compensate for that. **NOTE: L2 is the amount of neurons in the hidden layer.**

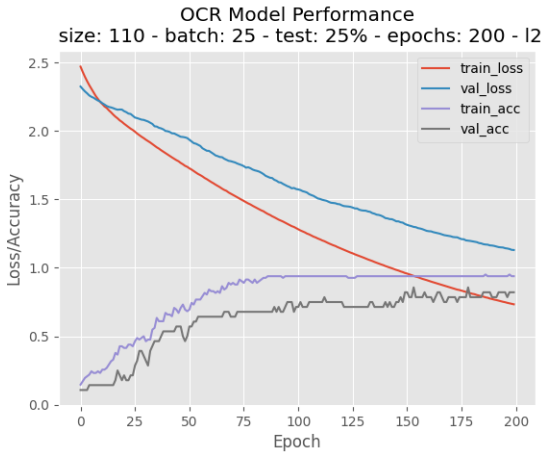





OCR Model Performance
size: 110 - batch: 6 - test: 25% - epochs: 80 - l2: 512
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch


OCR Model Performance
size: 110 - batch: 6 - test: 25% - epochs: 200 - l2: 512
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch


OCR Model Performance
size: 110 - batch: 25 - test: 25% - epochs: 80 - l2: 80
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch


OCR Model Performance
size: 110 - batch: 25 - test: 25% - epochs: 200 - l2: 80
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch