Data Mining Technology for Business and Society

Homework 3

1643257 Valerio Guarrasi
1651051 Davide Aureli

DATA GENERATOR

For each character in string.printable[:-6] we decided to create a clean image for each font type taken from the font file, without bothering if it was bold or italics. Since in the font folder there was a weird Desktop file, we put a try and except to skip that file. Using the Pillow library we tried to position the image in the center of the matrix to prevent bad augmentation in the second part. Instead of doing translation in the augmented part, we decided to initialize 18 different positions of the clean character so that we could control its movement. To create the matrix, we used as mode “L”, which means that the numbers that we can represent go from 0 to 255 (based on the gray scale). We wrote a black letter on a white background of dimension 40. While we were creating the clean images, we created 4 lists (one for each characteristic: fonts, char, bold, italics) to help us for creating the Y_trains (to use them in different .py files, we pickled them). We saved all the images(matrices) in list called npy_file, since we then converted and saved it in a .npy file. So the dimension of the array was (2632,64,64,1): there were 2632 images (94 characters*28 possible fonts with bold and italics), 64*64 is the number of pixels for each image and 1 is the number of channels (grey scale).

DATA AUGMENTATION

Now we want to create our X train. After loading the file that we created before (clean_data.npy), we decided to augment each image 18+6*1*18 times. Beyond the augmented images, we also putted in the training set the one clean image per character of a certain font.
The augmentation procedures that we applied are 6:

- Rotation: we choose randomly an angle between -35 and 35 degrees. We decided to apply this technique to the clean image because we thought that, over the range of these angles, the character could have been mistaken with another one.
- Scaling: here to apply the scaling we had to reshape the image from 64*64*1 to 64*64. we choose randomly a zoom level between 0.4 and 2 selecting only the even values. This is then used to zoom in (if the scale value is bigger than 1) or to zoom out (if the scale value is lower than 1). If we zoom out, to obtain a 64*64 image, we added a padding of zeros of pad_width = (64 - dimension of the scaled image)/2. This is why we choose only even numbers. Instead, when zoom in, to obtain a 64*64 image, we have to cut the borders of dimension (dimension of the scaled image - 64)/2.
- Rotation and Scaling: we applied the scaling procedure to the rotated image created before.
- Salt: replaces random pixels with either 1.
- Rotation and Salt: we applied the s procedure to the rotated image created before.
- Rotation, Scaling and Salt: we applied the s procedure to the scaled and rotated image created before.

All of these procedures, since they are random, we applied them on the clean image 1 time, since we already have 18 for each one. Since the only augmentation that transforms in a 0/1 scale is the salt and pepper, before appending the augmented image in the training set, we had to divide all the pixels by 255. We applied these augmentation procedures because we thought that they were the most related to our context, for example flipping a letter may lose the characteristic of that character. Since the tensorflow library was too slow, we decided to use the skimage library. Like before, we created a .npy file (dirty_data.npy) with our X train. To create the Y training set, we took the old lists (fonts, char, bond, italics) and repeated each element by 18+6*1*18. Then we pickled these new 4 lists (fonts_train, char_train, bold_train, italics_train).

MODEL
Loading all of the Y training sets, with OneHotEconder, we transformed all of the Y training lists of strings to a matrices of zeros and ones. The keras library changes the order to alphabetical. Then we also loaded our X training set created in augmentation step. With a pipeline procedure and the functional API, our best model has this structure:

- convolutional layers: 64 filters, 3 kernel size, relu activation and with padding
- pooling layer : 2 pool size
- drop out = 0.25   (repeat first 3 steps for 5 times)
- flat
- Neural Network:
    - output layer for font: number of possible fonts as neurons, softmax activation
    - output layer for char: number of possible characters as neurons, softmax activation
    - output layer for bold: 2 neurons(since it is yes or no), sigmoid activation
    - output layer for italics: 2 neurons(since it is yes or no), sigmoid activation
- Compile: adam optimizer and 4 loss functions (font: categorical_crossentropy, char: categorical_crossentropy, bold: binary_crossentropy, italics: binary_crossentropy)
- Fit the model: 10 epochs, 256 batch_size

We applied the pooling steps because then the size of our NN would be excessively large, in such a way we reduce the dimensionality. We added a neuron dropout because it serves to put a penalty when the model overfits. The parameter that passes is the probability of killing a neuron. We have to do the flatting, with the pulling we have reduced the first two dimensions, but now we have to reduce the third. The goal is to output only one vector. We decided to not have too big epochs, since we noticed that after a few runs the accuracy converged. Then we saved the model architecture and weights in the model.json and model.h5 files. Then in the test set we wanted to normalized the training set to do a good comparison with our training set, but at the end all of the images were already ina 0/1 range. The choice of our optimizer was given by: https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-deep-learning-5acb418f9b2.

PERFORMANCE
The performances in our validation set are:

Char: 0.5989
Font: 0.4440
Bold: 0.8216
Italics: 0.7101

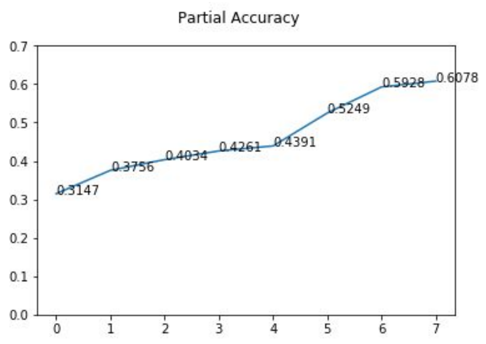
The performances in 33% of the test set are:

Char: 0.672518
Font: 0.340021
Bold: 0.763137
Italics: 0.757438
Partial Accuracy: 0.6078764390744329

COMPARISON

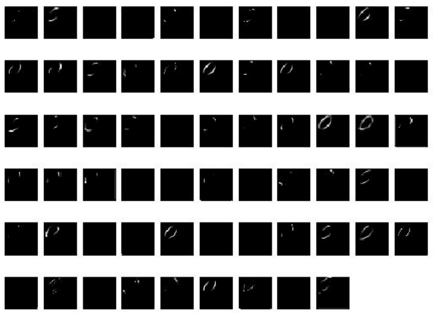
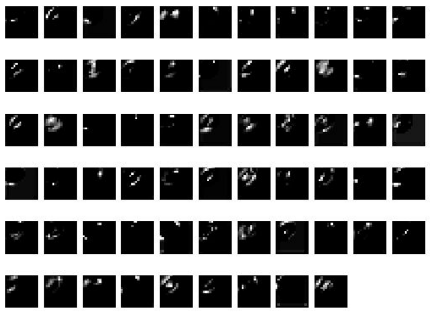
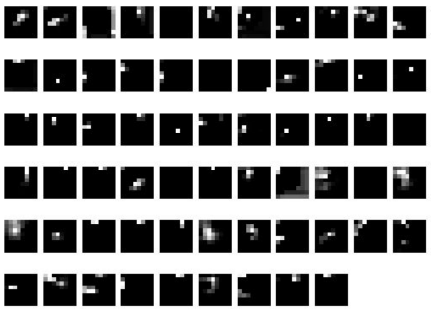
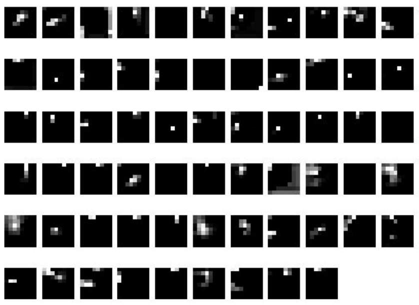
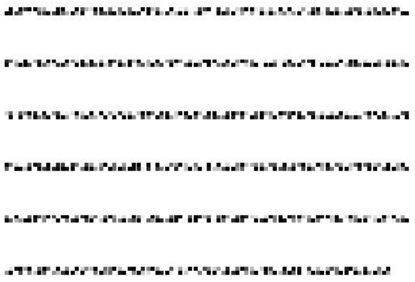
Here we show our progress. We noticed that by changing the model not a lot of improvements were given, so we concentrated more on the data and its augmentation. Here we can see that the changes on the data gave a lot more accuracy:

1. 60 text size, 2 conv
2. 50 text size, 2 conv with activation "relu"
3. conv with activation "relu"
4. 4 conv with activation "relu"
5. 4 conv with activation "relu" 64 nodes each instead of 32
6. 5 conv, data with translation
7. 40 text, different translation
8. more translation



EXTRA 1

In convolutional (filtering and encoding by transformation) neural networks (CNN) every network layer acts as a detection filter for the presence of specific features or patterns present in the original data. The first layers in a CNN detect (large) features that can be recognized and interpreted relatively easy. Later layers detect increasingly (smaller) features that are more abstract (and are usually present in many of the larger features detected by earlier layers). The last layer of the CNN is able to make an ultra-specific classification by combining all the specific features detected by the previous layers in the input data. Every network layer acts as a filter for the presence of specific features or patterns present in the original image. For detection by such a filter it is irrelevant where exactly in the original image this specific feature or pattern is present: the filters are especially designed to detect whether or not the image does contain any such characteristics. The filter is shifted multiple times and applied at different image positions until the entire image has been covered in detail (the filter can correct, if necessary, e.g. for scale, translation, rotation angle, color, transformation, opacity, out of focus, deviations of specific features present in the original image).

EXTRA 3
We decided to solve a image classification problem, we would like to tell for each image its class. We will have some images of cats and dogs and we will train convolutional neural network to predict if the image is a photo of a dog or of a cat. We have 2 different folders, one for the training set and one for the test set. Here again we created 2 different folders, one for the cats and one for the dogs. This is a subset of a Kaggle dataset, only 10000 images: 8000 in the training set(4000 of dogs and 4000 of cats) and 2000 in the test set(1000 of dogs and 1000 of cats). Here we used a sequential neural network. We saw that with only one convolutional layer we got an accuracy on the test set of 0.75 and on the training set 0.85, so overfitting. So we added another convolutional layer and we obtained an accuracy on the test set of 0.81 and on the training set 0.85, much better. You can check out everything in the Extra folder. To find the data go to: https://drive.google.com/drive/folders/16PtqlnoF9iGCLA1b7t0KWCDT52_X-eb8?usp=sharing