Umeå University
Department of Computing Science

Fundamentals of artificial intelligence
5DV121

Happy, Sad, Mischievous or Mad?

Thomas Ranvier *Valentin Lecompte*

supervised by
Thomas Johannson
Carl-Anton Anserud
Anders Broberg
Adam Dahlgren Lindström
Ola Ringdahl

24 October, 2018

Contents

Introduction

This report explains how we designed and implemented a single layer perceptron in order to classify given images of faces to their right emotion.

Chapter 1

Data pre-processing

1.1 Reading and stocking the datas

The first thing that we need to do is to read the datas and stock them in adapted data-structures in order to have easy and efficient access to all of them.

To do so we've created a function that will read the datas given the files names, preprocess them and organize them into a dictionary.

That dictionary is named 'datasets' and contains three datasets:

- datasets['train_x'] and datasets['test_x'] both contain the images of the training and test sets.
- datasets['train_y'] contain the keys of the training set.

1.2 Preprocessing of the images

1.2.1 Results without data preprocessing

Once we had all our datas easily accessible in our variables we wanted to know what success rate could a classic multilayer perceptron reach.

To test it we used the scikit-learn function 'MLPClassifier' with the 'lbfgs' solver. We created a function allowing ourselves to run x times the classifier on different permutations of the datas. That function then computes the mean of the success rates and returns it.

Without any data pre-processing and using an MLP with one hidden layer containing 102 perceptrons we reached a mean success rate of about 71%.

1.2.2 Automatic rotation of the images

Then we noticed that many images were actually not oriented with their 'forehead' on top. We thought about it and figured out that there is a way to automatically determine if the head is in the right direction. Indeed if the eyes

are on top the upper half of the array containing the image will have a sum inferior to the bottom half (or higher, depending if you stock the reverse of the images or not). We used that fact to create a function that creates rotates the initial image by 90 degrees three times and then return the image with the lowest upper half sum.

We can then apply this function to every images in our datas and all of them will be oriented in the right way afterward.

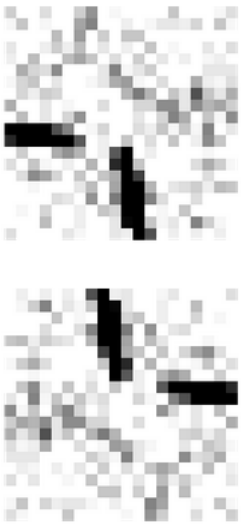

Figure 1: Auto rotate before/after on image 11

Of course we could not orientate the images in angles inferior to 90 degrees without deforming the image too much, so the images are not totally straight but it is good enough.

After this first modification on the images we ran our evaluation function and obtained a mean success rate of about 79%.

1.2.3 Reshape of the images

This modification is very simple: the images are emoticons and emoticons are not supposed to be squares, they are supposed to be round. That means that all the pixels in the corners of the square are just noise that is not supposed to exist. With this modification we simply got rid of it by reshaping all the images with a circular mask.

Figure 2: Circular reshape before/after on image 11

After this modification we ran our evaluation function and obtained a mean success rate of about 83%.

1.2.4 Blurring the images

Then we tried to sharpen the image using the 'scipy' library, but it did not improve the results. On the other hand we discovered that to blur the images was giving better results.

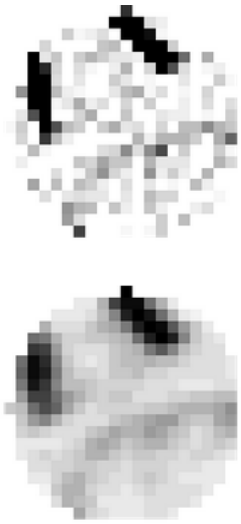

Figure 3: Blur before/after on image 1

The blur function takes every pixels of the image and compute the mean between this pixel and its adjacent neighbours, the computed value then replaces the pixel value. By doing so the image gets blurred and the noise get flattened.

The circular mask is applied after blurring the image so we do not add any noise to the image during the blurring process.

We reached a mean success rate of about 88% by adding this modification.

1.2.5 Abandoned leads

After those modifications we tried to keep improving our results. To do so we've tried several things.

We tried to use the contours of the images to reinforce the important areas of the images. To automatically find the contours we used the 'find_contours' from the 'skimage' package.

Figure 4: Contours on image 1

Using those contours we can create a mask.

Figure 5: Contours mask of the image 1

Then we tried to apply this mask to the initial image.

Figure 6: Image 1 with contours mask

We also tried applying the same treatment twice.

Figure 7: Image 1 with two contours mask

We tried our evaluation function on each of those steps and none of those improved the results.

Chapter 2

Development of the neural network

2.1 Single layer perceptron

We implemented a single layer perceptron in order to classify the given images into their right categories.

Our neural network is constituted of only one output layer containing four perceptrons (or neurons). Each one of these perceptrons stands for one category of images, happy, sad, mischievous or mad.

Each perceptron possess an array containing 400 weights, each of those weight matches with one pixel of the given image.

In order to process an output when one perceptron is given an image it multiplies each pixel value with the corresponding weight, sum everything and passes the obtained result through the function $sigmoid(x) = \frac{1}{1+e^{-x}}$.

The *sigmoid* function is called an activation function, it returns a result that is always contained between 0 and 1. The important characteristic of an activation function is that it provides a smooth transition as input values change, a small change in input produces a small change in output.

To use this neural network we have to create an instance of 'Neural_network' and then call the fit function and passing it the training datas $datasets['train\_x']$ and $datasets['train\_y']$.

That function automatically creates the output layer neurons depending on how many different categories there is in the targets of the dataset.

It will then start the training and learning process for a number of epochs [1] between 'min_epochs' and 'max_epochs'. That loop will end when it reached 'max_epochs' or when the reached success rate is above or equal to the user

[1] An epoch is one forward pass and one backward pass of the entire training set into the neural network.

expectations. The loop cannot stop before reaching the 'min_epochs' value.

At each epoch it will split the datas to fit into a training set and a test set. Then the 'learn' is called for every images contained in the training set. That function will compare the desired output for each perceptron of the output layer with their real output. It will then update the weights of each perceptron using the computed error, the weights will go up if the error is positive and down conversely. After the weights update the network is evaluated in its actual state with the test set and the success rate of that process is returned.

Bellow is the pseudo-code of the 'learn' function with x being the image and y the corresponding key.

Algorithm 1 Learning function

```
1:  procedure LEARN(x, y)
2:      for all perceptron in output layer do
3:          desired_output = (neuron_index + 1 == y)
4:          neuron_output = perceptron.compute_output(x)
5:          error = desired_output − neuron_output
6:          for all weight in perceptron.weights do
7:              weight += learning_rate ∗ error ∗ x[weight_index]
8:          end for
9:      end for
10: end procedure
```

Once the neural network have been fit to the training datas we use the function 'predict' on each test image and display the predicted classification on the user screen.

With our current neural network and our used parameters we usually reach a success rate around 90% on the test datas.

2.2 Used parameters

The basic parameters that we determined as being the best ones for our implementation are the following:

- $Learning\ rate = 0.05$
- $Training\ proportion = 0.66\ (2/3)$
- $Max\ epochs = 100$
- $Min\ epochs = 15$
- $Success\text{-}rate\ threshold = 0.90$

2.3 Multilayer perceptron

At first we tried to implement a multilayer perceptron neural network. It seemed like a good choice since the used 'MLPClassifier' of the 'scikit-learn' library is one and gives correct results.

But it adds a lot of new parameters to be aware of and it was very difficult to obtain a working result, usually the network outputs were always the same no matter the given input.

It is only later that we decided to implement a really easier version of a neural network by abandoning the idea of an hidden layer.

Since the results of our final implementation are even better than those obtained with the 'MLPClassifier' we can see that with those input datas an hidden layer is just a waste of resources anyway.

Conclusion

Our implementation reaches a success rate around 90% for the test datas. To keep improving that score we could keep researching image preprocessings that make the neural network classify the images in the right categories easier.

Another idea could be to split the images in two parts, one containing the eyes and a second one containing the mouth and give them to two perceptrons. The first would classify the eyes as bad or good and the second one would classify the mouths between happy and sad. That way we would then determine which emotion correspond to the combination of the two results.

A reason for which such idea could really improve the results is that it would 'multiply' by two the number of training datas. Indeed there is only two possible states for the eyes and same for the mouth where there is four possible state in the datas with the complete emotions.

List of Algorithms

List of Figures