Artificial Intelligence - Methods and Applications

Assignment 1 - Othello

UMU-57205
Cyril Bos (CAS: cybo0001, CS: ens17cbs)

Teachers: Ola Ringdahl,
Juan Carlos Nieves Sánchez,

Contents

This report explains our implementation of the perceptron algorithm and presents the results of our tests.

1 General Information, usage and results

The main file `faces.py` and the training files used to make our tests are at the root of the ZIP file. The assignment was developed in `Python 3(.6)` and uses only standard library packages. We respected the specifications for the execution, so after extraction on the CS linux environment the following command should succesfully work.

```
1 python3 faces.py training-A.txt facit-A.txt test-B.txt > result.txt
```

Our program begins by printing comments (beginning with '#') of the percentage of correct classifications and the value of the mean of squared errors for each iteration. Currently, our tests lead to a percentage of correct classifications around 90%, both on the sub training set and the unknown set.

The algorithms given in this report are written in a Python-ish language.

2 Implementation of the perceptron

2.1 Perceptron structure

The perceptron we designed uses 4 output nodes, each of them associated with one of the 4 emotions. Thus, when classifying an image, we consider the output node returning the biggest activation level as the activated one. The idea is that the output node of the correct emotion gets as near as possible to the value 1, and the other output nodes as close as possible to the value 0.

The activation function used is the sigmoid function $sigmoid(x) = \frac{1}{1+e^{-x}}$. The reasons we chose it is that the sigmoid function is monotonous, the image of any value is between 0 and 1, and its S shape makes it favour extreme values of stimuli. Hence, the resulting activation levels will have a high value of difference.

Each of the output nodes is linked to 401 input nodes, one for every pixel in the images, and the bias input node linked to a virtual pixel of constant grey level 1. This bias method helps the training by adapting the weighting of the activation function to the left (closer to 0) or to the right (closer to 1).

There is a single weight associated to each link. The weights are initialized to 0. The weights are updated using the `Perceptron.learn()` method, which is the algorithm described in the assignment slides (p. 17):

```
1     def learn(self, image):
2         for output_node in self._output_nodes:
3             desired_output = output_node.emotion == image.emotion
4             activation_level =
                 output_node.get_activation_level(image)
5             error = desired_output - activation_level
6             for link in output_node.input_links:
7                 delta = self._learning_rate * error *
                     link.input_node.get_activation_level(image)
8                 link.weight += delta
```

The training algorithm respects the specifications of the slide number 18. Here is the condition for the training iterations to stop:

```
1 while iteration < min_iteration or (
2                         squared_error_mean >
                             squared_error_mean_threshold and
                             iteration < max_iteration)
```

Meaning that we oblige the network to train a minimum amount of iterations but we also prevent overfitting. Also, the training stops earlier than the overfitting limit when the mean of squared errors drops below a certain value.

2.2 Parameter values

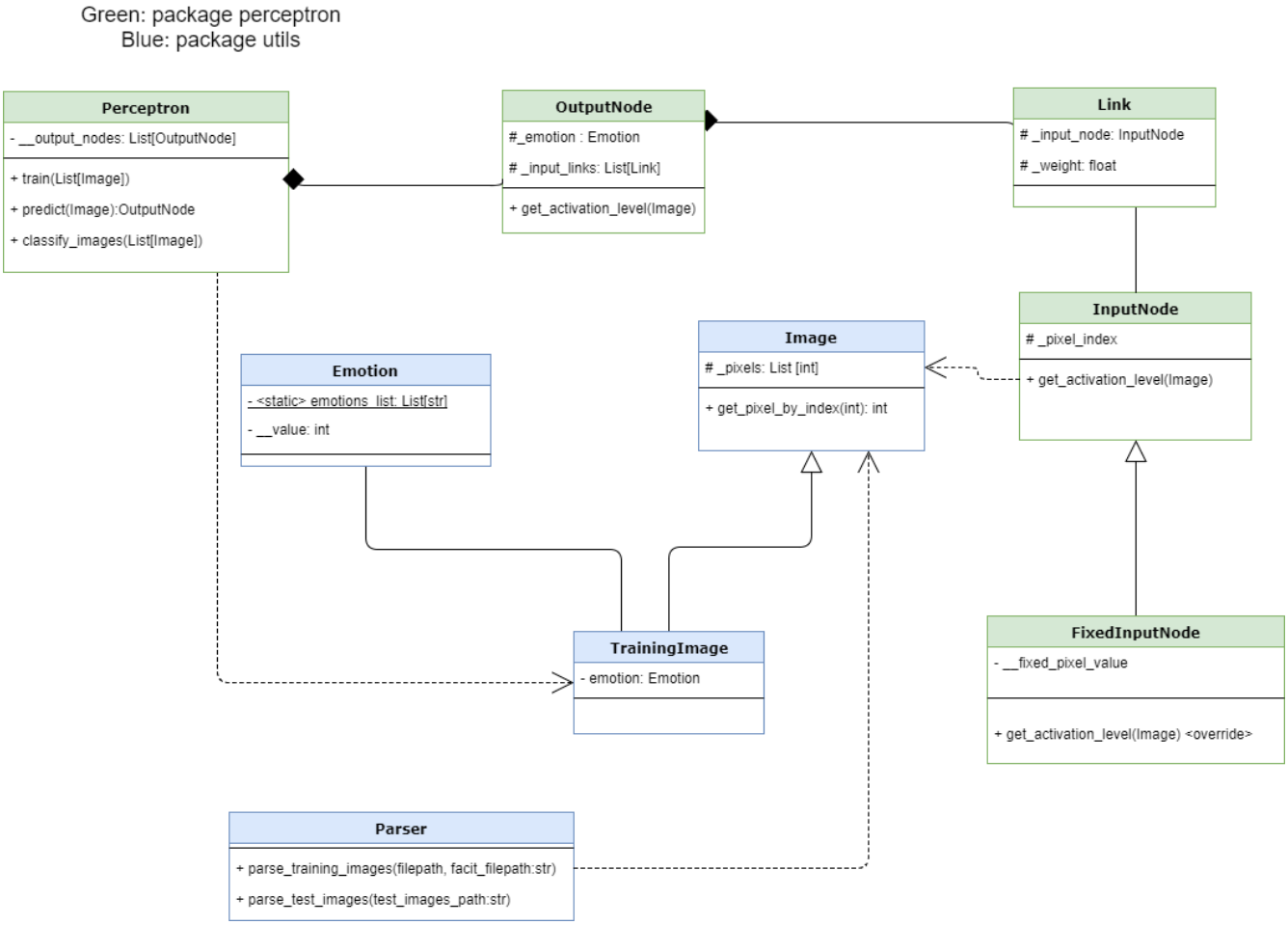
Based on our trials, we chose the values:

- 15 for the minimum number of iterations
- 100 for the maximum number of iterations
- 0.08 for the threshold of the average error
- 0.1 for the learning rate
- 2/3 (0.66) of images reserved for training and the remaining for performance check

2.3 Code structure

Our implementation is mostly written in OOP, apart from the main file `faces.py` and the file `utils.image_preprocessing.py`. There is a package `perceptron` that contains the classes used to implement the perceptron algorithm and a package utils that contains a Parser class, Image and TrainingImage classes, an Emotion class and a image preprocessing module. This module implements blur and rotation of images as explained in section 3.

Here is a UML class diagram of the whole structure:



OOP helps dividing the whole perceptron algorithm in several abstract components close to the vocabulary used. It makes the program a little heavier to compute than just using builtin types such as just an int value for Emotion, and some kind of tuple for links and weights, but it makes it easier to understand, to compare different implementations. Also the polymorphism and inheritance mechanisms to factorise code.

It also gets pretty easy to add hidden layers with this architecture by adding a Node class with a recursive method get_activation_level(), though we did not as we know it is forbidden to do so.

3 Improving precision: image preprocessing

We tried and implemented several image preprocessing to improve the precision of the perceptron. Without preprocessing, our Perceptron reaches around 70% of correct classifications. The code we submitted reaches from 90 to 96% by blurring and rotating both the training set before the training and the set to classify before the classification.

3.1 Rotation

A big issue in the given images is that the faces are often rotated at different angles. We implemented a method that divides the image in 4 equal parts, extracts the 2 parts where the eyebrows seem to be, and rotate them in a way that these two parts are on the first top half of the image matrix. This way, every image will have the eyebrows and mouth features at almost the same location.

We detect the eyebrow center by iterating through every inner pixel of each part, calculating the sum of the grey levels around, and keeping the center pixel where this value is the maximum.

There are 4 cases for the rotation, depending on the eyebrows location:

- if the eyebrows are already at the right location, do nothing
- if the eyebrows are detected on the top right and top left quarter we rotate the image by 90 degrees counter clockwise,
- else if they are detected on the bottom right and bottom left quarters, we rotate by 180 degrees (such as illustrated on 1 below)
- else (if they are detected on the bottom left and top left quarters), we rotate by 270 degrees counter clockwise (or 90 degrees clockwise)

Here is an illustration of how the algorithm works (based on Image 10). The red point is the eyebrow center detected by the algorithm using the the 3x3 pixels mask visualised in yellow. This example could be interpreted as sad without the rotation, instead of the correct mischievous emotion.

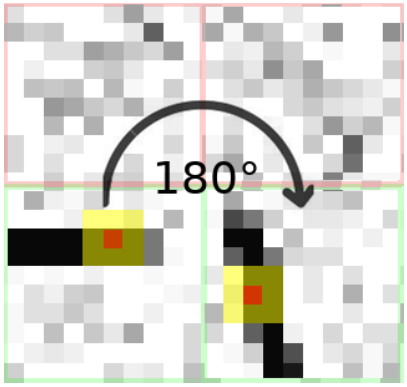


Figure 1: Visualisation of the rotation algorithm

Here is a pseudo code of the algorithm. The real implementation with all the details can be found in the file `utils/image_preprocessing.py`.

```
def rotate(image):
    quarters = split_image(image, 4)

    eyebrows = []
    for i in range(len(quarters)):
        if is_eyebrow(quarter):
            eyebrows.append(i)

    rotate_quarters(image, eyebrows)
```

We first thought of rotating the image by calculating the angle between the two eyebrows centers, but we did not have time. That would also lead to some information loss about the corners, which aren't important most of the time, but on some cases might help define the eyebrows or mouth features better.

3.2 Contrast

We tried to improve the performance by adding a simple linear contrast, making the dark pixels darker and the light pixels lighter. We thought it would remove some noise around the eyebrows and the mouth, but it did not work as expected. This may be explained because the area around the mouth is often noisy and the linear contrast may not improve the difference of grey level between the noise and the mouth, resulting in a less clear mouth shape.

The code for it was removed because unused.

3.3 Blur

In contrary to the contrast, blurring the image improves performance. The method we use is called low-pass filtering. It takes into account neighbors of each pixel. It replaces a given pixel value with the mean of surrounding pixel grey levels (in our case, a chunk of 3x3 pixel around one). The noise around the eyebrows and mouth is flattened because the filter smoothes the changes in intensity, reducing the importance of noise in the training.

4 Possible evolutions

Although the network already reaches high performance levels, some ideas may have deserved more exploration. The fact that a sad face has the same kind of curves than a reversed mischievous face may explain few errors of the predictions without rotation. We also could have split the image in two, one part with the smile and one part with the eyes, and the output nodes would process those parts instead of processing every pixel.