Fundamentals of Artificial Intelligence - 5dv121

Assignment 2 - Face recognition in a neural network

Jakub Jasinski - `id14jji`
Simon Asp - `id14sap`

October 2016

Teachers: Thomas Johansson och Alexander Sutherland

Table of contents

1 Introduction 4

2 Problem specification 4

3 Compiling and running 5

4 Algorithm description 5

4.1 Backpropagation 6

4.2 Network training algorithm 7

4.3 Solution 8

5 System description 9

5.1 ImageLoader 9

5.2 Image 9

5.3 OutputNode 9

5.4 Network 9

5.5 Faces 10

6 Discussion 10

6.1 Our solution 10

6.2 General notes 10

7 Source code 11

7.1 ImageLoader 11

7.2 Image 13

7.3 OutputNode 14

7.4 Network 17

7.5 Faces 18

8 Figures and links **19**

1 Introduction

This report discusses an implementation of a neural network made for face recognition. The implementation is written in Java and uses pixel matrices that represents face images. The assignment is a part of the Fundamentals of Artificial Intelligence course given at Umeå University. Implementation is based on theory given at the lectures.

2 Problem specification

The task is to create a neural network capable of classifying four types of images. The network can use a given set of images to train on and it must be able to tell the right answer 65% of the times.

The images are made of 20x20 pixels and has a grey scale of 32 levels. Each image has a category, i.e a face, which can be any of the types: happy, sad, mischievous or mad.

In **Figure 1** we can see an example of these images which are also rotated randomly to give an extra level of difficulty.

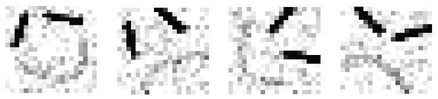

Figure 1: *Example of the images shown to the network. Happy, Sad, Mischievous and Mad*

300 images of the type described above are used to train the network and 100 images drawn from the same set as the training images are used to test if the network did reach satisfying results.

The type of network used is a perceptron network, which is a linear classifier and has one layer of neurons. The algorithm used is Backpropagation.

3 Compiling and running

The solution for this assignment is located at the servers of the department of computer science in Umeå. To access and run the files, one should go to `/id14sap/edu/5DV121/lab2`.

To compile the program, go to `src/` and run the command

```
javac -Xlint:unchecked -d ../out Faces.java
```

To run the program navigate to the output folder `/out` and run

```
java Faces ../Images/training-file.txt
../Images/training-facit.txt
../Images/test-file.txt
```

It is possible to use other image files as long as they are represented in the right way.

4 Algorithm description

The perceptron network consists of neurons which takes several inputs and calculates an activation level depending on what the values of the inputs are and what *weight* each input has. The input in this case is the pixel array that represent one image, and the weights are numbers that gets updated over time.

The activation level is a measure of how much that particular neuron *activated* given the input, meaning how much it "thinks" that the input it receives belong to the same category that the neuron represents. This category should be set upon creating the network, so that each neuron represent either happy, sad, mischievous or mad. The activation level should be stronger as the network gets trained.

In **Figure 2** we can see a neuron that receives several inputs with corresponding weights. The different weights correspond to a particular input but it's important to note that the weight should stay when new input is received, since the weights are used by the neuron every time to determine a new activation level from what it previously "learned". The weights gets changed with each iteration and is the main component in achieving a training behaviour.

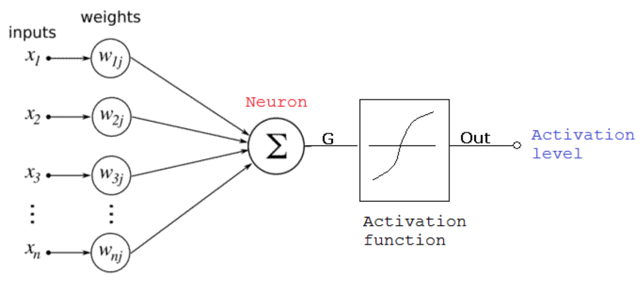

Figure 2: *A neuron receiving input and calculating it's activation level with an activation function.*

The activation level is calculated by summing all the input values times the weight values and running them by an activation function. This can be seen in **Figure 3** where *a* is the activation level.

The activation function serves as a threshold saying how much the neuron should activate, and can be a simple step function or a smoother sigmoid function. More on this in the next section.

$$a_i = Act\left(\sum_j x_j w_{j,i}\right)$$

Figure 3: *The activation level is calculated by summing the input* ***x*** *times the weights* ***y****, and putting it in an activation function* ***Act***

4.1 Backpropagation

This is a common training algorithm for neural networks and works by calculating an error from the actual output and the expected output, and then sending this information back into the input, by updating the weights.

As explained briefly above, the weights are updated at each iteration of the training phase. This happens in the end of each training cycle when the output activation level is calculated. By knowing which category the input belongs to, and which category the neuron belongs to, an error can be determined as shown in **Figure 4**.

Here the expected output is *y*, and the actual output is *a* (the activation level). The α is the learning rate of the system which is a constant. Too low and the training will take too long time. Too high and the network might be *overtrained* resulting in bad generalization of the network.

$$e_i = y_i - a_i$$
$$\nabla w_{ji} = \alpha e_i x_j$$
$$w_{ji} \leftarrow w_{ji} + \nabla w_{ji}$$

Figure 4: *The backpropagation weight update. The new weight is a sum of the old weight and the change in weight,* Δ *w*

Expected output
The expected output *y* is determined by a fix *performance standard*, which in this case is either 0 or 1. If the expected output is 0, it means that we don't expect the neuron to activate at all, and 1 is to activate fully.

If the image is of category A and the neuron is representing category B, we tell the neuron that the expected value is 0, since the image is of the wrong category. If it is the right category (both the image and the neuron represent category B for instance), we tell the neuron that we expect it to output 1 as it's activation level.

This gives us the possibility to adjust the weights as in **Figure 4**, and the network will soon learn which kind of input belongs to the right category since you supervised the network and told it which input it just looked at, and given it a chance to tweak it's weights and in time output a correct categorization.

4.2 Network training algorithm

To train the whole network to the images given, a training algorithm is executed. This goes through the whole list of 300 images and shows them to all the neurons.

1. **Order the training set randomly**
2. **For each image in the training set**

   (a) **For each neuron in the network**
      i. **If the image category is the same as the neuron’s category**
         A. **Set the expected value to 1**
      ii. **Else**
         A. **Set the expected value to 0**
      iii. **Execute the training algorithm for the image with the expected value**
      iv. **Calculate the neuron’s error to the power of 2 and put in a total error sum**

3. **Calculate a mean squared error from the total error sum**
4. **Repeat from step 1 as long as the mean squared error is greater than a certain value**

4.3 Solution

The network created for this particular task consists of four nodes which are called output nodes. These nodes represent one of the four faces (happy, sad, mischievous or mad), and are determined with numbers, i.e 1, 2, 3 or 4. The weights of each node is randomized at first and a bias weight of value 1 is also used.

The pixel values of each image is normalized by dividing them by 32 to give values between 0 and 1. Also, pixel values over 5 are replaced with a value of 0, since they are classified as noise and should not be present in the image.

When testing if the network did reach results over 65% a pre-made program is used, which was given as material on the course.

The training phase keeps on going as long as the mean squared error is greater than 0.01, a number we found out gave good results after testing the network with the given test program. If we make the training phase go on for a longer time we get worse results due to over training.

The learning rate have been set to 0.01, as we found it to give pleasing results and not take too much time.

5 System description

Our implementation is constructed with the following classes:

5.1 ImageLoader

This class is responsible for loading data from given text-files, since images are represented as matrices filled with numbers. The class ignores rows that contains irrelevant symbols and saves each matrix data into index of an *ArrayList*. Saved data includes image name, pixel values and result of which face image represents.

The last-mentioned is included in a different text-file then image names and theirs pixel values. Therefore they need to be added separately. This class takes care of everything and has functionality of returning complete lists of images that can be used later on in implementation.

This class also filters out noise in the image by removing pixels values above 5, and replacing them with a value of 0.

5.2 Image

Creates Image objects we use to store data of each image mentioned in *ImageLoader* class. The class has functionality of returning, getting and setting individual values.

5.3 OutputNode

Manages activation levels. Contains an `ArrayList` with weights and is responsible for adding and updating them in learning phase. Has functions that triggers node training and testing.

5.4 Network

Network is a class where we create four different OutputNodes(Happy, Sad, Mischievous and Mad). This class is simply linking together components for training and testing we created in OutputNode. It shows all the images for all nodes we created and trains them based on value of mean squared error. It is also responsible for examination of test-file.

5.5 Faces

Faces is our Main class that creates network, setts arguments for input files and gets list of images from files. It also initializes network training and it's later examination.

6 Discussion

6.1 Our solution

We spent some time in the beginning trying to figure out the theory behind neural networks and it was beneficial. We did a lot of sketching and thinking on paper before implementing anything and it felt much easier to do the task afterwards. We also discussed this theory with other students from the course and that was when we figured out that only making four nodes which handles everything was the way to go, to save time.

The network could have been more generalized by introducing input nodes though. Only output nodes were used which received an image class instead of taking input from other nodes. The network could have worked on different types of input, but in this particular case, our solution worked good. It was faster to write it this way and that's why we went with it.

We did encounter some minor problems when we starting to implement the solution but we managed to work it out with some help from the teacher. This was mainly on how long the training phase should go on and how to calculate the mean squared error, nothing hard in general but we got quite dizzy when thinking of the network and forgot what the mean error really was in this case.

6.2 General notes

It is interesting to see how the network actually works and especially that it doesn't *know* anything about the input that it receives. The only thing it knows is that a given input belongs to a certain category and trains to recognize similar patterns. It doesn't know what the input *means*.

One might think that the network learns from actually looking at an image so many times that it finally "knows" what it is, and can express what it sees. But maybe that's how us humans work too, only that we have a much larger and more powerful set of neurons.

When we say that we know what an image represents, what do we actually know? Maybe we compare it to images we seen before, and maintain a representation for those kinds of images, much like the artificial network. Or are we so smart that we can figure it out without seeing an image "a thousand times"?

If we have never seen a dog, how could we then say what it is? Maybe that it's similar to another type of animal, which gives us a type of reasoning about the answer. But that should be possible to implement to a artificial network too, if it's complex enough.

7 Source code

The source code is also available at GitHub

7.1 ImageLoader

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;
/**
 * Created by Jakub on 10/5/16.
 */

/**
 * Class that loads the files and makes them into appropriate data
    structures, image lists. Also loads the facit file.
 */
public class ImageLoader {

    private ArrayList<Image> trainingImages = new ArrayList<>();
    private ArrayList<Image> testImages = new ArrayList<>();

    /**
     * This class takes in the training file and the training facit.
     * @param trainingFile The file that includes the training images
     * @param testFile The file that includes the test images
     */
    public ImageLoader(String trainingFile, String testFile) {
        try {
            this.trainingImages = loadFile(trainingFile);
            this.testImages = loadFile(testFile);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * The method to load a training, or test file with images. When it
    's loaded it gets put in an image list with pixel
     * values.
     * @param fileName the path to the image list file.
     * @return an array list of images.
     * @throws IOException
     */
    private ArrayList<Image> loadFile(String fileName) throws
    IOException {
        File textFile = new File(fileName);
        String str;

        ArrayList<Image> imageList = new ArrayList<Image>();

        // Loads the input file
        try (Scanner scanner = new Scanner(textFile)) {
            while (scanner.hasNextLine()) {
                str = scanner.nextLine();
                // Skipping unnecessary input lines
                if ((!str.startsWith("#") && (str.startsWith("Image")))
    ) {
                    String imageName = str;
                    ArrayList<Double> pixels = new ArrayList<>();

                    // Loads pixels as Doubles into pixel array and
    normalizes them, there's 400 pixels
                    for (int i = 0; i < 400; i++) {
                        str = scanner.next();
                        // Filter out the noise pixels
                        if(Double.parseDouble(str) <= 6.0) {
                            pixels.add(0.0);
                        }
                        else {
                            pixels.add(Double.parseDouble(str) / 32);
                        }
                    }

                    //Object image for storing the name, pixels and
    result
                    Image image = new Image(imageName, pixels, 0);
                    //List that has every image we have loaded
                    imageList.add(image);
                }
            }
            scanner.close();
        }
        return imageList;
    }
    /**
     * Puts the right category from the facit file in the training
    images-list.
     * @param fileName - the path to the facit file.
     * @throws FileNotFoundException
     */
    public void loadFacit(String fileName) throws FileNotFoundException
     {

        File textFile = new File(fileName);
        String str;
        int i = 0;
        try (Scanner scanner = new Scanner(textFile)) {
            while (scanner.hasNextLine()) {
                str = scanner.nextLine();

                //Skipping unnecessary input lines
                if ((!str.startsWith("#") && (str.startsWith("Image")
    && (i <= 199)))) {
                    trainingImages.get(i).setFaceValue(Integer.parseInt
    (str.substring(str.length() - 1)));
                    i++;
                }
            }
        }
    }

    /**
     * Getter for the training images
     * @return an array list with the training images.
     */
    public ArrayList<Image> getTrainingImages() {
        return trainingImages;
    }

    /**
     * Getter for the test images
     * @return a list of test-images
     */
    public ArrayList<Image> getTestImages() {
        return testImages;
    }
}
```

7.2 Image

```
import java.util.ArrayList;


/**
 * Image
 * This class
 */
public class Image {
    private String imageName;
    private ArrayList<Double>pixels = new ArrayList();
```

```java
    private int faceValue = 0;

    public Image(String name, ArrayList pixels, int faceValue) {
        this.imageName = name;
        this.pixels = pixels;
        this.faceValue = faceValue;
    }

    public ArrayList<Double> getPixels() {
        return pixels;
    }
    public String returnName() {
        return imageName;
    }
    public int getFaceValue(){
        return faceValue;
    }
    public void setFaceValue(int x){
        this.faceValue = x;
    }
}
```

7.3 OutputNode

```java
import java.util.ArrayList;

/**
 * Created by Simon on 10/6/16.
 */

/**
 * An output node is a node in the network which calculates activation
    levels and updates the weights to each input
 * it receives.
 */
public class OutputNode {

    private Double activationLevel;
    private Double learningRate;
    private Double error = 0.0;
    private ArrayList<Double> weights = new ArrayList();
    private int faceValue;

    /**
     * Adds weights and adds the last bias weight
     * @param learningRate Specify learning rate for the node
     * @param numOfInputs Amount of pixels
     * @param faceValue Specifies which face the node represents
     */
    public OutputNode(Double learningRate, int numOfInputs, int
    faceValue) {
        this.learningRate = learningRate;
        this.faceValue = faceValue;
        // Create a weights list for this node with random values
    between 0 and 1
        for(int i=0; i<numOfInputs; i++) {
            this.weights.add(Math.random());
        }
        // Add a bias weight
        this.weights.add(1.0);
    }

    /**
     * Train the node with the given input image
     * @param img An image that will be iterated
     */
    public void train(Image img, int expectedValue) {
        this.calculateActivationLevel(img);
        this.updateWeights(img, expectedValue);
    }

    /**
     * Test the node with given test image. Calculates the activation
    level that is previously trained.
     * @param img An image that will be iterated, usually from the test
    -file.
     * @return Activation level of this node.
     */
    public Double test(Image img){
        this.calculateActivationLevel(img);
        return this.getActivationLevel();
    }

    /**
     * Calculates this OutputNode's activation level by summing each
    pixel value in the incoming image with the
     * corresponding weight, and then running it through the sigmoid
    activation function.
     *
     * @return The activation level of this OutputNode
     */
    private void calculateActivationLevel(Image img) {
        Double sum = 0.0;
        ArrayList imagePixels = img.getPixels();
        // Sum all pixel values times the weight of the pixel value
        for (int i = 0; i < imagePixels.size(); i++) {
            sum += (Double)imagePixels.get(i) * weights.get(i);
        }
        // Put the bias weight (last index of the list) to the sum
        sum += weights.get(imagePixels.size());
        // Run activation function
        this.activationLevel = activationFunction(sum);
    }

    /**
     * The sigmoid activation function
     * @param x Input value as a double
     * @return An activation of the input
     */
    private Double activationFunction(Double x) {
        return (1 / (1 + Math.exp(-x)));
    }

    /**
     * This is the learning method for this OutputNode. It updates the
    weights of each input pixel by calculating the
     * error between the expected value and the activation value.
     */
    private void updateWeights(Image img, int expectedValue) {
        ArrayList imagePixels = img.getPixels();
        this.error = calculateError(expectedValue);

        // For all the pixels in the image
        for (int i = 0; i < imagePixels.size(); i++) {
            Double deltaWeight = learningRate * error * (Double)
    imagePixels.get(i);
            Double weight = weights.get(i);

            // Update the weight
            weights.set(i, deltaWeight + weight);
        }
    }

    /**
     * Calculates the error from this OutputNode in the form of ei = yi
     - ai
     * @return The error as a Double
     */
    private Double calculateError(int expectedValue) {
        return expectedValue - this.activationLevel;
    }

    /**
     * Returns this nodes activation level
     * @return The activation level as a Double
     */
    public Double getActivationLevel() {
        return this.activationLevel;
    }

    /**
     * Get the errorSum for this outputNode
     * @return the errorSum as a Double
     */
    public Double getError() {
        return this.error;
    }

    /**
     * Returns the weights of this node
     * @return An ArrayList with the weights of this node
     */
    public ArrayList<Double> getWeights() {
        return weights;
    }

    /**
     * Return the face representation that this node has
     * @return an integer representing a face (1=happy, 2=sad, 3=angry,
     4=mischievous)
     */
    public int getFaceValue() {
        return faceValue;
    }
}
```

7.4 Network

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;

/**
 * Created by Simon Asp on 2016-10-06.
 */

/**
 * The network is responsible for holding the four output nodes,
    training them and also examining the performance.
 */
public class Network {

    private ArrayList<OutputNode> outputNodes = new ArrayList<>();
    private final Double learningRate = 0.01;
    private Double mse = 0.0;
    private final int MATRIX_SIZE = 400;
    private Double totalError = 0.0;
    private int counter = 0;

    /**
     * Creates a network with four output nodes.
     */
    public Network() {
        // Add four outputNodes to this network
        outputNodes.add(new OutputNode(learningRate, MATRIX_SIZE, 1));
        outputNodes.add(new OutputNode(learningRate, MATRIX_SIZE, 2));
        outputNodes.add(new OutputNode(learningRate, MATRIX_SIZE, 3));
        outputNodes.add(new OutputNode(learningRate, MATRIX_SIZE, 4));
    }

    /**
     * Trains this network with the training images provided.
     * Creates a counter which is the size of all the images times the
    number of nodes (around 800)
     */
    public void trainNetwork(ArrayList<Image> images) {
        runSamples(images);
        do {
            runSamples(images);
            Collections.shuffle(images);
        }
        while (mse > 0.01);
    }

    /**
     * Shows all the images for all nodes and executes the training
    method on each node.
```

```
     * Also takes the errors that each node make and squares it, then
    puts it in a sum.
     * The counter is to keep track of how many times this network has
    showed the images to the nodes.
     */
    private void runSamples(ArrayList<Image> images) {
        int expectedFaceValue;

        for (Image img : images) {
            //System.out.println(img.returnName());
            for (OutputNode node : outputNodes) {
                counter++;
                // Set the expected value as 1 if the faceValue of the
    node is the same as the images facit, else 0
                expectedFaceValue = (node.getFaceValue() == img.
    getFaceValue() ? 1 : 0);
                node.train(img, expectedFaceValue);
                totalError += Math.pow(node.getError(), 2);
            }
        }
        mse = totalError / counter;
    }

    /**
     * This is the examination function that tests the nodes and prints
     out the results
     * @param images - an image list
     */
    public void examine(ArrayList<Image> images) {

        for (Image img : images) {
            HashMap<Double, Integer> values = new HashMap<>();
            for (OutputNode node : outputNodes) {
                values.put(node.test(img), node.getFaceValue());
            }

            Double maxKey = Collections.max(values.keySet());
            int faceValue = values.get(maxKey);

            System.out.println(img.returnName() + "␣" + faceValue);
        }
    }
}
```

7.5 Faces

```
import java.io.IOException;
import java.util.ArrayList;

/**
 * Created by Jakub on 10/5/16.
 */

public class Faces {
    /**
     * The main class that runs the face recognition.
     * @param args 0 = the training file, 1 = the training facit, 2 =
    the test file.
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {

        ImageLoader loader = new ImageLoader(args[0], args[2]);
        ArrayList imgList = loader.getTrainingImages();
        ArrayList testList = loader.getTestImages();

        loader.loadFacit(args[1]);
        Network network = new Network();

        network.trainNetwork(imgList);
        network.examine(testList);
    }
}
```

8 Figures and links

Figure 1 - From the course web
Figure 2 - from www.global-warming-and-the-climate.com - 2016.10.13
Figure 3 - from the course web
Figure 4 - from the course web