A Neural Network Approach for Optical Character Recognition

TNM095
2015-11-09
Kristofer Janukiewicz

Abstract— To accomplish an optical character recognition through the use of a multilayer perceptron neural network. The OCR trained neural network is used on handwritten characters through a real-time application using a web-camera.

1 Introduction

The goal of this report is to give an understanding of the methods used for the optical character recognition. This technique can be used for understanding how an artificial intelligence understands handwritten characters while being trained with digital characters from a database. The report also mentions how these methods can be used in a real-time applications and suggests how it can be improved.

2 Implementation

This project is made in C++ with the library OpenCV 2.4.10 [1] on an ordinary Ultrabook-computer with the Windows 7 operating system. The code is compiled and run in Visual Studio Professional 2013. The neural network used in this project is the multi-layered perceptron (MLP) [2]; included in the OpenCV 2.4.10 library. The database of character images are downloaded from The Chars74K dataset [3].

3 Method

The project is divided into three phases:

- **Image processing** - This is the part where each image for every character is processed and made into a string of bits, see illustration in figure 2. When all the desired images have been processed; the result is then sent to the next step: Training.
- **Training** - In this part, the MLP-neural network is initiated. Using the processed images from the previous part; the neural network is trained, tested and exported to the next step.
- **Output** - This part handles the real-time application, computer vision methods, reading the characters and letting the neural network guess the characters in each frame.

3.1 Image processing

The image processing stage is all about extracting the necessary data from our database of images. Each image in the database is a 128x128 pixels PNG-image of a character. Since there are 1016 different fonts, each character has a maximum of 1016 inputs in the database for each font. This is illustrated in figure 1.

If, for example the neural network should be able to recognize the characters: A, B & C, it can read in a maximum of 3048 images from the database.

To turn each image into valuable information for the neural network; it is necessary to threshold each image, crop out the necessary data and resize the image. The threshold technique used in this project is a simple segmentation threshold; the threshold is used to prevent noise and to achieve a zero-one representation of the character data. Cropping out the character is done by finding the most upper, rightmost, downwards and leftmost 1 (in the image); creating a rectangle

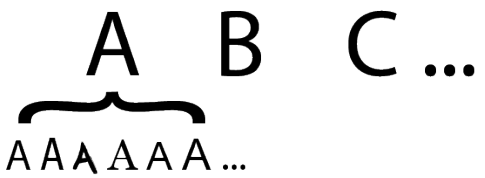

Figure 1: *For each character and for each font, the database contains an image. Since there is 1016 fonts, the database contains a maximum of 1016 images for every character.*

exactly around the character and then deleting the zeros outside the rectangle.

The cropped out image is then scale-resized into a small patch (preferable 8x8, 16x16 or 32x32) and then saved as a string of ones and zeros (the size of this string is dependable on the size of each patch. e.g: $8x8 = 256$ long string of ones and zeros which is a 1 bit/pixel representation of the resized image). The resizing is done to minimize the amount of data for each letter, as too large images wont do in real-time guesses with the neural network. The whole procedure is illustrated in figure 2.

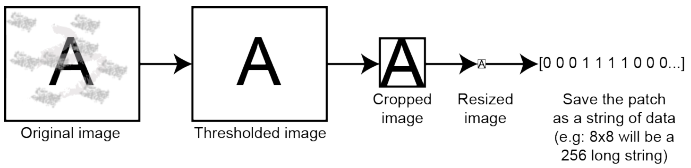

Figure 2: Remove the noise with a threshold. Crop the image to contain the character. Resize the image to a small patch. Save the image as a string of ones and zeros in the training set.

When each image has been processed; the training set and testing set are initiated. The training set will include all the saved and processed images while the testing set can include lesser, equally or even more of the training set images. The idea behind the testing set is to let the neural network, in the next step, train itself with the images from the training set against the testing set to increase its accuracy (as the neural network will know if it is right or wrong with each guess).

3.2 Training

With the creation of the training data set and the testing data set, as gigantic matrices including the data information of each letter, it is time to initiate the neural network.

As mentioned before, the neural network used in this project is a multilayer perceptron (MLP). By default; the MLP is built as three distinct layers with different size and function; the input layer, the hidden layer and the output layer. Each layer consists of a set amount of neurons. Each neuron has several input links from neurons in the

• *Kristofer Janukiewicz is with Linköping University, Sweden, e-mail: krija286@student.liu.se.*

previous layer and one output link to a neuron in the next layer. In MLP the neurons in the input layer are outputted to the neurons in the hidden layer. The neurons in the hidden layer are then outputted to the neurons in the output layer, this procedure is illustrated in figure 3. The MLP can consist of several hidden layers.

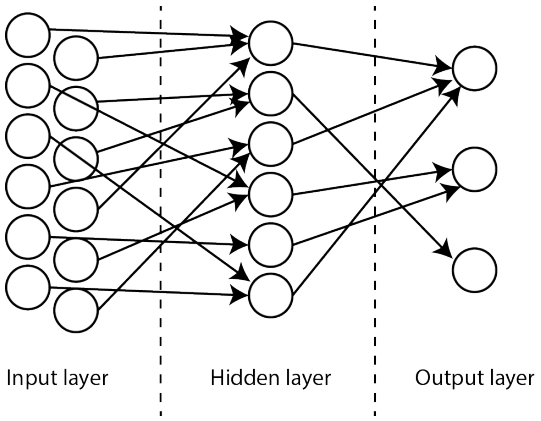

Figure 3: *The neurons (the circles) in the hidden layer get the input information from the neurons in the input layer. The neurons in the output layer get the input information from the hidden layer. The input to each neuron is calculated with a weight-function* (1) *and the output from a neuron is calculated with the activation function* (2)*.*

Every input to a neuron is weighted and summed (1), this value represents the decision that the neuron will take. The result is then sent through an activation function (2), which will be the output of a neuron and an input to a neuron in the next layer.

$$u_i = \sum_{j}^{n} (w_{i,j}^{n+1} \cdot x_j) + w_{i,bias}^{n+1} \tag{1}$$

Where the w is the individual weight in each neuron, n is the amount of inputs to the neuron i, j is the current input (sums n inputs), the x is the input value from the previous neuron and an added bias-term [2]. The weights in each neuron is computed with the train-algorithm in OpenCV.

$$f(u_i) = \beta \cdot \frac{(1 - e^{-\alpha u_i})}{(1 + e^{-\alpha u_i})} \tag{2}$$

Where α and β are constants. The u_i is the summed weight from the input to the neuron i. The result of this function is then sent as an input to the neuron in the next layer. The activation function is also called a symmetrical sigmoid function [2].

In this project, the following amount of neurons in each layer was used:

- **The input layer** - The size of all dimensions in each resized image patch. E.g: 256 neurons for 8x8 patches.
- **The hidden layer** - The size of a dimension in the resized image patch. E.g: 8 neurons for 8x8 patches.
- **The output layer** - The amount of characters used. E.g: 9 neurons for 9 characters.

The neural network is then trained with the training data set. The training builds up the connections, weight functions for every neuron. This is the part where the testing data set steps in. The newly trained neural network is going to be tested against the testing data set in order to test its accuracy; as it will be known (since this is a test) if it guessed right or wrong on each guess against the training data set. The result of each guess (the maximum weightage to a character) is then stored in a classification matrix (the experience of the neural network) and can be printed out too see how successful the neural network is at guessing a character.

The experience of the neural network together with all the weights and indexes are then stored into an XML-file; which makes the neural network portable and simple to read (since the training can take a lot of time) in the real-time application.

3.3 Output

Since the neural network is now trained, tested and ready to use from the previous stage; it is simply loaded into the output-application (as an XML-file). There is no more interaction with the neural network until the characters has been detected in the frame.

Since the application is run in real-time with a web-camera; the character detection only uses simple computer vision techniques to detect every character on each frame. E.g. using advanced noise reduction techniques would slow down the application.

To detect a character; the current frame is turned to gray-scale, figure 4.

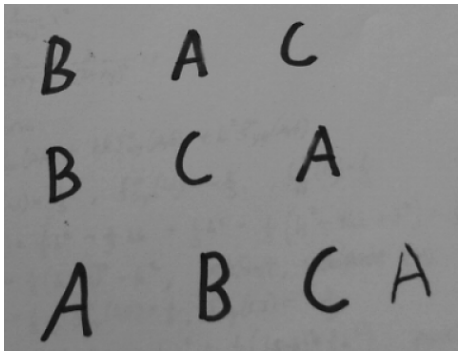

Figure 4: *The current frame is gray-scaled, all colors are removed.*

Then a sobel-filter is used on the gray frame, this is used to detect high frequency areas (e.g. black characters on a white papper), figure 5.

Figure 5: *Using a sobel-filter returns all the high frequency areas in the frame. The characters and the noise in the background-paper is partly visible in this image.*

The next step uses Otsu's thresholding on the result from the sobel-filter, to differentiate the high frequency areas (the characters) from the low frequencies, figure 6.

The last step is to use a morphology operator, preferable a **closing**, to remove some more noise and to fill in guessed characters, figure 7.

Since it is now possible to tell where the characters can be in the frame, every guessed character is cut out from the frame into smaller

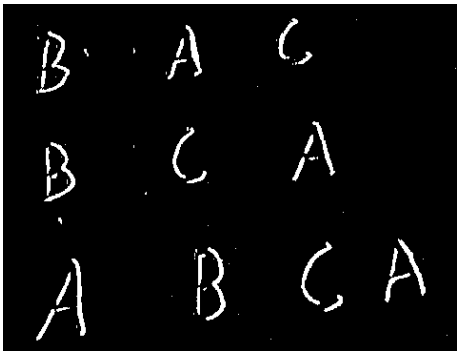

Figure 6: *With Otsu's thresholding technique, almost all the background noise is removed.*

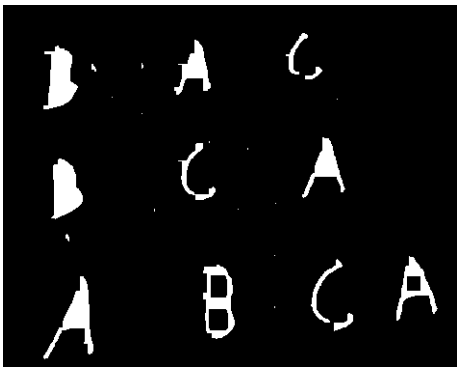

Figure 7: *Using a closing morphology operator, some characters have been filled and are therefore easier to find (consistency/contours) in the frame.*

separate images, figure 8. Too small character guesses are filtered away since they may be noise.

Figure 8: *As the position of each character has been found in the frame. It is cut out into smaller rectangles.*

Every cut-out image of a guessed character is then going to the same process as in the input stage (when a character image is read to the training set). Which results in smaller patches, of each cut-out character, and stored as vectors of strings containing zeros and ones.

This is where the neural network starts guessing characters on the cut-outs from the frame and it is performed in the same way as in the training stage (where it guessed against the testing set), the difference here is that the neural networks does not know the true answer and will not adapt if it guesses wrong. Going through the weights and the network, it returns an index. This index is the guessed character in that cut-out. This is done on every character, on every frame in real-time, see figure 9 for the result.

4 Result

Two tests were done on the correct guesses of the neural network and is presented in the table 1 and 2. As can be seen in the tables; by learning from digital fonts, a MLP neural network has a high success rate of reading handwritten characters in real-time.

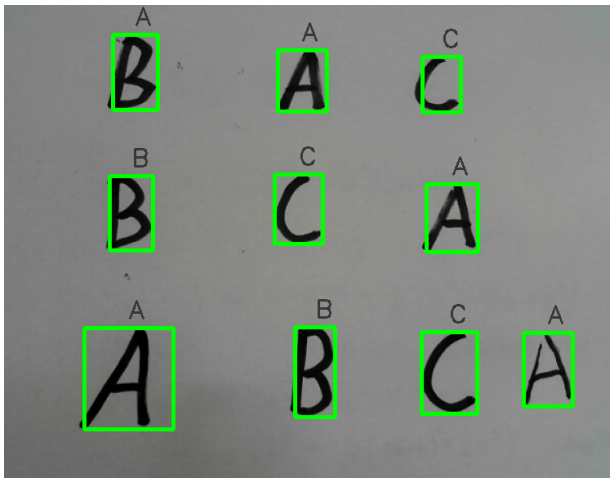

Figure 9: *The result of the trained MLP neural network with three characters. Each character in the frame is found and it guesses right on 90% of the characters. The correct guesses increases with tweaking of the computer vision algorithms.*

Table 1: The result of correct guesses using different amount of patch-size and training images on three characters: A, B and C as seen in figure 9. The B in the top left corner was never guessed correctly in any test.

Patch-size	Characters	Training images	Correct guesses
8x8	3	3	90%
8x8	3	200	83%
8x8	3	500	87%
8x8	3	1016	77%
16x16	3	3	90%
16x16	3	200	75%
16x16	3	500	90%
16x16	3	1016	90%
32x32	3	3	90%
32x32	3	200	85%
32x32	3	500	87%
32x32	3	1016	55%

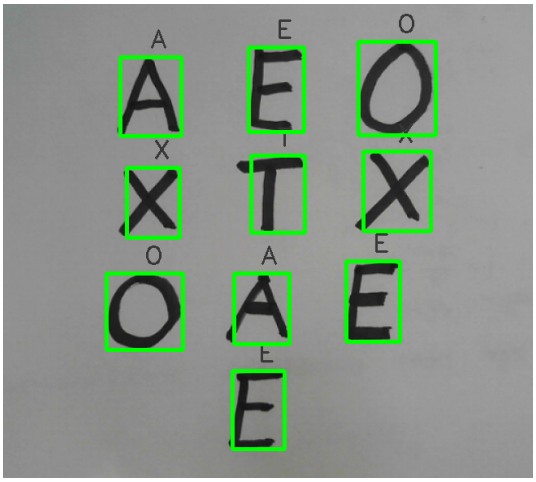

Figure 10: *A good guess by the neural network when it had learned five different characters. It had an easier time recognizing distinct looking characters.*

Table 2: The result of correct guesses using different amount of patch-size and training images on five distinctive characters: A, E, O, T and X, as seen in figure 10. A higher correct guess percentage was achieved with five characters than with three characters. This is due too that in this test; all characters looked distinctive.

Patch-size	Characters	Training images	Correct guesses
8x8	5	3	91%
8x8	5	200	85%
8x8	5	500	95%
8x8	5	1016	72%
16x16	5	3	98%
16x16	5	200	97%
16x16	5	500	90%
16x16	5	1016	90%
32x32	5	3	95%
32x32	5	200	97%
32x32	5	500	97%
32x32	5	1016	88%

5 Discussion

When creating the training and the testing set in the preprocessing-stage. A resize size can be chosen for each character stored into the sets. Depending on the amount of characters that is desired to learn into the neural network; a larger size is recommended. This is because a higher grade of detail is needed to differentiate each character. For a few and very different characters; an 8x8 size is recommended for a fast result. But for more characters; a 16x16 or 32x32 size is recommended. Larger sizes does, of course, slow down the real-time application. A larger size does not always return a better result either due to that the complexity of each character is increased in the training set; which will then result in that the neural network will have a harder time to find the correct character. Adding more neurons and hidden layers may improve that.

The accuracy in the real-time application can be increased with tracking. Since the neural network was struggling when letters had a high complexity (e.g. B and E), but guessed right in the majority of the frames during the run. Using an index-tracking between each frame, keeping a history of the neural network guesses for each object/index and then assign each character with the majority of the guesses would probably increase the accuracy of each tracked character in the frame. With a tracking method; the guesses would be more adaptable by being calculated depending on the majority of the guesses on each found character.

References

[1] OpenCV 2.4.10, OpenCV dev. team, `http://opencv.org/documentation/opencv-2-4-10.html`, 2015-11-09

[2] Neural Networks, OpenCV dev. team, `http://docs.opencv.org/2.4/modules/ml/doc/neural_networks.html`, 2015-11-03

[3] The Chars74K image dataset, T.deCampos, `http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/`, 2015-11-09