Neural Network OCR

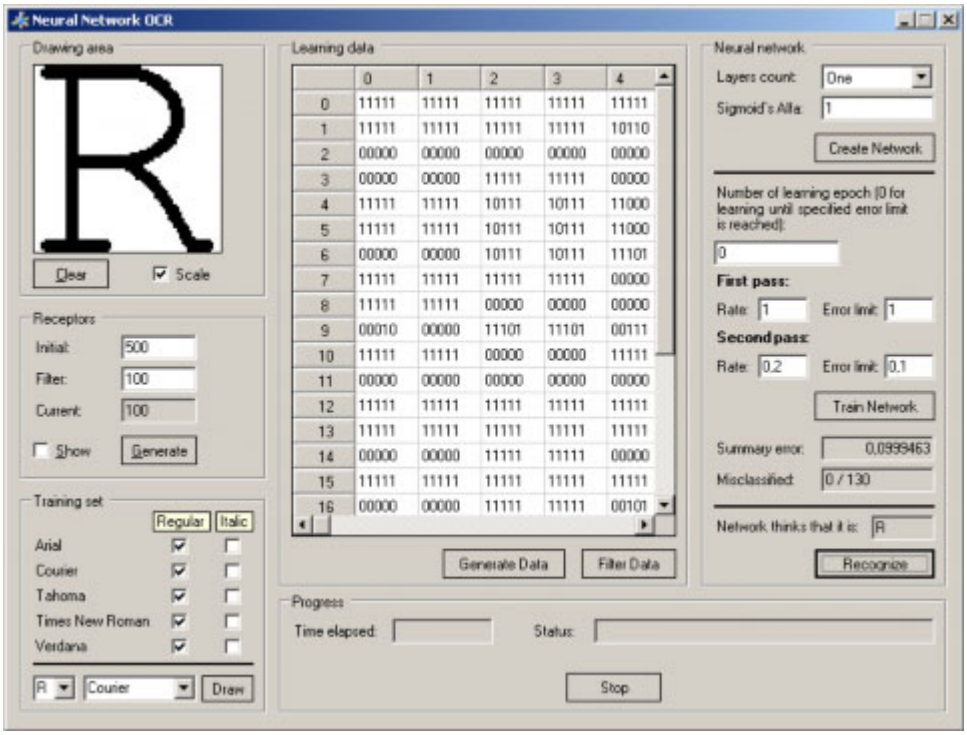
Andrew Kirillov

12 Aug 2005 GPL3

Some ideas about optical character recognition using neural networks.

Download demo - 173 Kb

Download source - 191 Kb



Introduction

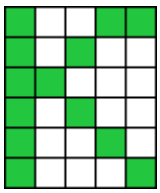
There are many different approaches to optical character recognition problem. One of the most common and popular approaches is based on neural networks, which can be applied to different tasks, such as pattern recognition, time series prediction, function approximation, clustering, etc.

In this article, I'll try to review some approaches for optical character recognition using artificial neural networks. The attached project is aimed as a research project, so don't try to find here a ready solution for scanned document processing.

Popular approach

The most popular and simple approach to OCR problem is based on feed forward neural network with backpropagation learning. The main idea is that we should first prepare a training set and then train a neural network to recognize patterns from the training set. In the training step we teach the network to respond with desired output for a specified input. For this purpose each training sample is represented by two components: possible input and the desired network's output for the input. After the training step is done, we can give an arbitrary input to the network and the network will form an output, from which we can resolve a pattern type presented to the network.

Let's assume that we want to train a network to recognize 26 capital letters represented as images of 5x6 pixels, something like this one:



One of the most obvious ways to convert an image to an input part of a training sample is to create a vector of size 30 (for our case), containing "1" in all positions corresponding to the letter pixel and "0" in all positions corresponding to the background pixels. But, in many neural network training tasks, it's preferred to represent training patterns in so called "bipolar" way, placing into input vector "0.5" instead of "1" and "-0.5" instead of "0". Such sort of pattern coding will lead to a greater learning performance improvement. Finally, our training sample should look something like this:

```
float[] input_letterK = new float[] {
    0.5f, -0.5f, -0.5f,  0.5f,  0.5f,
    0.5f, -0.5f,  0.5f, -0.5f, -0.5f,
    0.5f,  0.5f, -0.5f, -0.5f, -0.5f,
    0.5f, -0.5f,  0.5f, -0.5f, -0.5f,
    0.5f, -0.5f, -0.5f,  0.5f, -0.5f,
    0.5f, -0.5f, -0.5f, -0.5f,  0.5f};
```

For each possible input we need to create a desired network's output to complete the training samples. For OCR task it's very common to code each pattern as a vector of size 26 (because we have 26 different letters), placing into the vector "0.5" for positions corresponding to the pattern's type number and "-0.5" for all other positions. So, a desired output vector for letter "K" will look something like this:

```
// 0.5 is placed only in the position of "K" letter
float[] output_letterK = new float[] {
    -0.5f, -0.5f, -0.5f, -0.5f, -0.5f,
    -0.5f, -0.5f, -0.5f, -0.5f, -0.5f,
     0.5f, -0.5f, -0.5f, -0.5f, -0.5f,
    -0.5f, -0.5f, -0.5f, -0.5f, -0.5f,
    -0.5f, -0.5f, -0.5f, -0.5f, -0.5f,
    -0.5f};
```

After having such training samples for all letters, we can start to train our network. But, the last question is about the network's structure. For the above task we can use one layer of neural network, which will have 30 inputs corresponding to the size of input vector and 26 neurons in the layer corresponding to the size of the output vector.

```
// pattern size
int patternSize = 30;
// patterns count
int patterns = 26;

// learning input vectors
float[][] input = new float[26][]
{
    ...
    new float [] {
        0.5f, -0.5f, -0.5f,  0.5f,  0.5f,
        0.5f, -0.5f,  0.5f, -0.5f, -0.5f,
        0.5f,  0.5f, -0.5f, -0.5f, -0.5f,
```

```
        0.5f, -0.5f,  0.5f, -0.5f, -0.5f,
        0.5f, -0.5f, -0.5f,  0.5f, -0.5f,
        0.5f, -0.5f, -0.5f, -0.5f,  0.5f}, // Letter K
    ...
};
// learning ouput vectors
float[][] output = new float[26][]
{
    ...
    new float [] {
            -0.5f, -0.5f, -0.5f, -0.5f, -0.5f,
            -0.5f, -0.5f, -0.5f, -0.5f, -0.5f,
             0.5f, -0.5f, -0.5f, -0.5f, -0.5f,
            -0.5f, -0.5f, -0.5f, -0.5f, -0.5f,
            -0.5f, -0.5f, -0.5f, -0.5f, -0.5f,
            -0.5f},                 // Letter K
    ...
};

// create neural network
AForge.NeuralNet.Network neuralNet =
  new AForge.NeuralNet.Network(new BipolarSigmoidFunction(2.0f),
                                       patternSize, patterns);
// randomize network`s weights
neuralNet.Randomize();

// create network teacher
AForge.NeuralNet.Learning.BackPropagationLearning teacher = new
   AForge.NeuralNet.Learning.BackPropagationLearning(neuralNet);
teacher.LearningLimit    = 0.1f;
teacher.LearningRate    = 0.5f;

// teach the network
int i = 0;
do
{
    teacher.LearnEpoch(input, output);
    i++;
}
while (!teacher.IsConverged);

//
System.Diagnostics.Debug.WriteLine("total learning " +
                                       "epoch: " + i);
```

In the sample above a complete neural network training procedure for pattern recognition task is provided. On each learning epoch all samples from the training set are presented to the network and the summary squared error is calculated. When the error becomes less than the specified error limit, then the training is done and the network can be used for recognition.

How to recognize something? We need to input to the trained network and get its output. Then we should find an element in the output vector with the maximum value. The element's number will point us to the recognized pattern:

```
// "K" letter, but a little bit noised
float[] pattern = new float [] {
            0.5f, -0.5f, -0.5f,  0.5f,  0.5f,
            0.5f, -0.5f,  0.5f, -0.5f,  0.5f,
            0.5f,  0.5f, -0.5f, -0.5f, -0.5f,
            0.5f, -0.5f,  0.5f, -0.5f, -0.5f,
            0.5f, -0.5f, -0.5f,  0.5f, -0.5f,
            0.3f, -0.5f, -0.5f,  0.5f,  0.5f};

// get network's output
float[] output = neuralNet.Compute(pattern);
```

```
int i, n, maxIndex = 0;

// find the maximum from output
float max = output[0];
for (i = 1, n = output.Length; i < n; i++)
{
    if (output[i] > max)
    {
        max = output1[i];
        maxIndex = i;
    }
}

//
System.Diagnostics.Debug.WriteLine(
  "network thinks it is - " + (char)((int) 'A' + maxIndex));
```

Another approach

The approach described above works fine. But there are some issues. Suppose we train our network using the above training set with letters of size 5x6. But, what should we do, if we need to recognize a letter, which is represented by an 8x8 image? The obvious answer is to resize the image. But, what about the image, which contains a letter, that is printed with 72 font size? I don't think we'll get a good result after resizing it to 5x6 image. OK, let's train our network using 8x8 images, or even 16x16 to get high accuracy. But, 16x16 images will lead to an input vector of size 256, which will be more performance consuming to train the neural network.

Another idea is based on using the so called receptors. Suppose we have an image with a letter of arbitrary size. In this approach we'll form an input vector by using not the pixel values of the image, but by using the receptors values. What are these receptors ? Receptors are represented by a set of lines with arbitrary size and direction. Any receptor will have an activated value ("0.5" in input vector) if it crosses a letter and deactivated value ("-0.5" in input vector) if it does not cross a letter. The size of an input vector will be the same as receptors count. By the way, we can use a set of short horizontal and vertical receptors to achieve the same effect as in the case of using pixel values for images of small size.

The advantage of this method is that we can train neural network on big images even with small amount of receptors. Resizing an image with letter to 75x75 (or even 150x150) pixels will not lead to a bad image quality, so it will be much easer to recognize. But, on the other hand, we can always easily resize our receptors set, because they are defined as lines with two coordinates. And another big advantage is that we can try to generate a rather small receptors set, which will be able to recognize the entire training set using only most significant letter's features.

But, there are some disadvantages. The described approach can be applied only to OCR task. It's not possible to recognize complex patterns, because in this case we'll need too many receptors. But, as we are doing research in OCR task area, it will not disturb us very much. There is another question, which is much harder and which requires more research. How to generate the receptors set? Manually or randomly? How to be sure that the set is optimal?

We can use the next approach for receptors set generation: first, we'll randomly generate a large set of receptors, and then we'll choose a specified amount of best receptors. How to resolve if the specified receptor is good or not ? We'll try to use entropy, which is well known to us from the information theory. Let me explain it with a small example:

Here is a table, which contains some training data. Here we can see five types of objects, which are represented by rows, and three receptors, which are represented by columns. Each object has five different variants. Let's describe, for example, the first value of the table: "11101". It means, that the first receptor is crossing the first variant of the first object, it also crosses the second, third and fifth variants, but it does not cross the fourth variant. Does it clean? OK, let's look at the fifth row, first column: a receptor crosses the first, third and fifth variants, and it does not cross the second and the fourth variants.

We'll use two concepts: inner entropy and outer entropy. Inner entropy is an entropy of a specified receptor for a specified object. The inner entropy will tell us how good is the specified receptor for recognizing the specified object, and the value should be as small as possible. Let's look at the second row and the first column where there is "11111". The entropy of the set will be 0. It's good, because this receptor will be 100% sure, that we are working with the second object, because the receptor has the same value for all the variants of the specified object. The same thing is true with the second row and the second column: "00000". Its entropy is 0 too. But, let's look at the fifth row and the first column: "10101". The entropy of the set is 0.971 and it's bad, because the receptor is not sure about the specified object. The outer entropy is calculated for the whole column, the more it is closer to the value of "1", the better it is. Why? If the outer entropy is small, then the receptor is useless, because it cannot divide patterns. The best receptor should be activated for one half of all the objects and deactivated for another half. So, here is the final formula for calculating the receptors usability: `usability = OuterEntropy * (1 - AverageInnerEntropy)`. The average inner entropy is just a sum of inner entropies of the receptor for all the objects divided by the amount of objects.

So, using the idea, initially we should randomly generate a big set of receptors, for example 500. Then we should generate temporary training inputs using these receptors. On the basis of the data, we can filter a predefined amount of receptors, for example we can save 100 receptors with the best usability. The filtering procedure will reduce the amount of training data and neural network's input count. Then, with the filtered training data, we can continue with the training of our network in the same manner as described in the above approach.

Test application

A test application is provided with the article, which tries to implement the second approach. How to use it? Let's try the first test:

- We need to generate initial receptors set. On application startup it's already generated, so we can skip this step, if are not planning to change the initial amount of receptors or the filtered amount.
- Select fonts, which will be used for teaching network. Let it be the regular Arial font for the first time.
- Generate data. In this step the initial training data will be generated.
- Filter data. In this step the initial receptors set as well as the training data will be filtered.
- Create network - a neural network will be created.
- Train network - neural networks training.
- Let's look at the misclassified value. It should be "0/26", which means that the trained network can successfully recognize all patterns from the training set.
- We can ensure this by using the "Draw" and "Recognize" buttons.

After performing all these steps we find that the concept is working! You can try a more complex test, choosing all regular fonts. But, don't forget to turn on the "Scale" option before data generation. The option will scale all images from the training set. Then you can set the error limit of the second pass to "0.5" for faster training or leave it "0.1" if you are not in a hurry. At the end of training you should get a misclassified value of "0 / 130". You can check that all images from the training set can be recognized.

You can even try to teach a network all fonts: regular and italic. You should use "Scale" option for it and you will need to play a little bit with the learning speed and error limit values. You can also try to use two-layered network. I was able to get a misclassified value of

"4/260" with only 100 receptors.

Conclusion

From the above tests, it looks like the second approach is able to perform the OCR task. But, all our experiments were made using the ideal training set. The application also allows to recognize hand drawn letters, but we should always use the "Scale" option in this case, still the result is not very outstanding. Possible future research can be done in the direction of better receptors set generation and filtering and image scaling.

But still, it works! And with some additional research and improvements we can try to use it for some real tasks.

License

About the Author

Andrew Kirillov
Software Developer IBM
United Kingdom

Started software development at about 15 years old and it seems like now it lasts most part of my life. Fortunately did not spend too much time with Z80 and BK0010 and switched to 8086 and further. Similar with programming languages – luckily managed to get away from BASIC and Pascal to things like Assembler, C, C++ and then C#. Apart from daily programming for food, do it also for hobby, where mostly enjoy areas like Computer Vision, Robotics and AI. This led to some open source stuff like AForge.NET, Computer Vision Sandbox, cam2web, ANNT, etc.

Going out of computers I am just a man loving his family, enjoying traveling, doing some sports, a bit of books, a bit of movies and a mixture of everything else. Always wanted to learn playing guitar, but it seems like 6 strings are much harder than few dozens of keyboard's keys. Will keep progressing ...

Comments and Discussions