Hand-Printed Character Recognizer using Neural Networks

95.407A Project

By: Shahzad Malik (219762)

Thursday, April 27, 2000

Table of Contents

1 Introduction

With the recent popularity of small, hand-held personal digital assistants (PDAs), the commercial need for accurate hand-printed character recognition technology has exploded. Clearly, attaching a keyboard to a pocket-sized device is not feasible, and devices such as the PalmPilot make use of an input device that resembles a pen (which is immediately intuitive to almost any user). Thus if a user could use this pen-based device to draw alphanumeric characters on the screen in a similar fashion to writing on a piece of paper, the need for a keyboard or numerous buttons could be completely eliminated.

The process of recognizing such handwriting from pixel information falls into a field of artificial intelligence called pattern or image recognition. Lots of work has been done in this field recently, and most techniques for pattern and image classification make use of neural networks. This project implements such a neural network in order to "learn" to recognize general features of hand-printed characters. The trained network can then be fed new inputs, which it then attempts to recognize and categorize properly (see Figure 1).

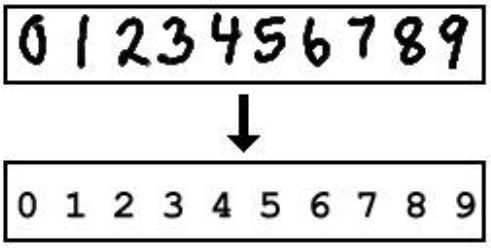

Figure 1 - Hand-printed character recognition

2 Feedforward Backpropagation Network

Typical pattern recognition systems are designed using two passes. The first pass is a feature extractor that finds features within the data which are specific to the task being solved (eg. finding bars of pixels within an image for character recognition). The second pass is the classifier, which is more general purpose and can be trained using a neural network and sample data sets. Clearly, the feature extractor typically requires the most design effort, since it usually must be hand-crafted based on what the application is trying to achieve.

One of the main contributions of neural networks to pattern recognition has been to provide an alternative to this design: properly designed multi-layer networks can learn complex mappings in high-dimensional spaces without requiring complicated hand-crafted feature extractors [LECU1995b].

Thus, rather than building complex feature detection algorithms, this project focuses on implementing a standard backpropagation neural network, which is fully connected. Figure 2 shows the typical layout of such a network as it relates to our task of classifying character images.

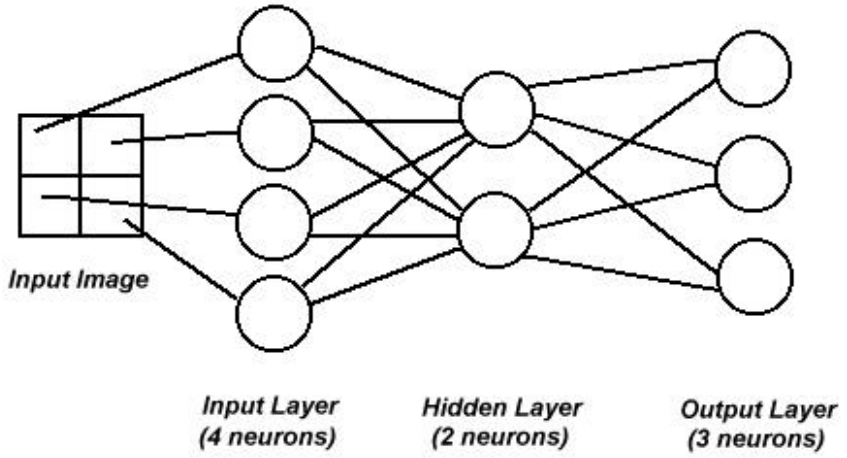

Figure 2 - Feedforward Backpropagation Neural Network for classifying 2x2 images

The idea behind this network is to map input vectors (in our case, pixels) to output vectors (in our case, ASCII characters). The process by which this mapping occurs is by assigning weights to each of the edges in Figure 2. These weights can initially be set to random values, and the neural network will automatically make adjustments to them based on a set of training data. This is the process of the feedforward backpropagation mechanism. Known inputs are fed into the neurons at the input layer, which are then activated and pass the activation information to the hidden layers, which pass their activations to other optional hidden layers, and then ultimately the output layer. At this point, the *resulting* output at the output layer is compared to the *desired* output. The

amount of error at each neuron is then propagated backwards through the network, whereby adjustments to the weights are made accordingly.

Each pixel in the input image is first normalized into a greyscale colour value between 0 and 1, where 0 means a fully white pixel, and 1 means a fully black pixel. Assuming RGB pixels, where each component is between 0 and 255:

$$\textbf{pixel\_input} = 1.0 - (((R + G + B) / 3.0) / 255.0)$$

Now assuming that the network has been fully connected and initialized with random weights, each of the [0,1] pixels are fed into their own input neuron. So for the 2x2 image in Figure 2, we have 4 neurons. Thus for an nxn image, we will need nxn neurons.

The network in Figure 2 is a 3 layer configuration, with the required input and output layers, as well as a single hidden layer. [RAO1993] mentions that choosing the number of hidden layers is a difficult task with no hard rules or guidelines. However, the *size* of a hidden layer is related to the features or distinguishing characteristics that are to be discerned from the data [RAO1993]. The network in Figure 2 has 2 neurons in the hidden layer, which was chosen arbitrarily to keep the diagram simple. The number of hidden layers and their sizes will be experimented with in the implementation, and discussed in the Results section.

The output layer is where the output vector can be retrieved. Each neuron in this layer outputs a value between 0 and 1, which is guaranteed by the use of a sigmoid function when calculating each neuron's output [RAO1993]:

$$\textbf{1.0 / (1.0 + } e^{-x}\textbf{)}$$

where x is the value to be "squashed".

In order to train the network, a training set consisting of correct pairs of input and output vectors are chosen. For example, in Figure 2 we could have the pairs:

```
Pair 1: { (1,0,0,0), (0,0,0) }
Pair 2: { (1,0,1,0), (0,0,1) }
Pair 3: { (1,1,1,0), (0,1,0) }
```

where the first vector consists of the input pixels, and the second vector is a binary representation of the desired ASCII character.

If the full 256 ASCII codes aren't needed, a conversion table could be easily created. So for the above codes we could have:

```
(0,0,0) = A
(0,0,1) = B
(0,1,0) = C
```

Since the output neurons can give a value *between* 0 and 1, we can achieve these crisp values by simply rounding the values to the nearest integer.

3 Results

This section shows some implementation results. The training variables involved in the tests were: the number of cycles, the size of the hidden layer, and the learning parameter (which is a factor used during weight adjustment). The data set consisted of 20 handwritten digits of size 24x32 pixels, with the numbers from 0 to 9. Thus the input layer consisted of 768 neurons, and the output layer 4 neurons (since 4 bits can represent the 10 possible digits). Ideally, we'd like our training data to consist of thousands of samples, but this not feasible since this data was created from scratch. The testing and training split used is 50%.

Note: While the implementation supports multiple hidden layers, the following test results only make use of one hidden layer. Instead, the test runs modified the *size* of the hidden layer, since this is what affects feature detection [RAO1993].

3.1 Data Table:

Cycles	Hidden Layer Size	Learning Parameter	Training Accuracy (%)	Testing Accuracy (%)
100	3	0.1	50	32
100	3	0.25	68	48
100	3	0.5	10	10
100	6	0.1	71	46
100	6	0.25	50	47
100	6	0.5	40	36
100	12	0.1	100	61
100	12	0.25	49	36
100	12	0.5	55	42
100	24	0.1	100	43
100	24	0.25	100	67
100	24	0.5	34	25
1000	3	0.1	59	45
1000	3	0.25	20	18
1000	3	0.5	20	18
1000	6	0.1	100	64
1000	6	0.25	46	34
1000	6	0.5	10	10
1000	12	0.1	100	66
1000	12	0.25	21	19
1000	12	0.5	27	28
1000	24	0.1	100	74
1000	24	0.25	39	36
1000	24	0.5	52	35

3.2 Discussion:

From the results in Section 3.1, the following observations are made:

- A small number of neurons in the hidden layer (eg. 3) is insufficient to extract key features in the hand-drawn digits. In all cases with 3 neurons in the middle, both the training and testing accuracy fared poorly.
- A large number of neurons in the middle layer help the accuracy, however there is probably some upper limit to this which is dependent on the data being used. Additionally, high neuron counts in the hidden layers increase training time significantly.
- A low learning parameter causes the network to learn quite slowly, but helps the network converge to a solution quite well. However, too low learning parameters could increase the chances of reaching local minimums rather than global minimums.
- A high learning parameter seems to seriously affect the accuracy of the test classification, since the weights and objective function end up diverging (ie. no learning occurs [SARL2000])
- Accuracy is increased by increasing the number of cycles.

4 Conclusion

The backpropagation neural network discussed and implemented in this project can also be used for almost any general image recognition applications such as face detection and fingerprint detection. While the implementation of the fully-connected backpropagation network gave reasonable results toward recognizing hand-printed characters, it has some major deficiencies. The most notable is the fact that it cannot handle major variations in translation, rotation, or scale. While a few simple pre-processing steps can be implemented in order to account for these variances, in general they are difficult to solve completely [LECU1995a]. Additionally, fully-connected networks completely ignore the general topology of the input image, and instead rely on almost pixel-perfect feature detection.

One possible improvement to the network involve using localized connections between the hidden layers. [LECU1995a] discusses Convolutional Networks as such an alternative. These networks are a simple variation of backpropagation networks, whereby connections between hidden layers are localized to a group of pixels (rather than fully connected). This allows the first few hidden layers to become local feature detectors, thus eliminating translational variances. Additionally edges, end-points, and corners can be extracted automatically by the network.

5 References

[LECU1995a] LeCun, Y., Bengio, Y. Convolutional Networks for Images, Speech, and Time-Series. http://www.research.att.com/~yann/

[LECU1995b] LeCun, Y., Bengio, Y. Pattern Recognition and Neural Networks. http://www.research.att.com/~yann/

[RAO1993] Rao, V., Rao, H. Neural Networks and Fuzzy Logic. MIS Press, New York, 1995

[SARL2000] Sarle, W. Neural Networks FAQ. Newsgroup: comp.ai.neural-nets