Character Recognition with Neural Networks
– A MATLAB Tutorial –

Martin H. Luerssen
School of Informatics & Engineering
Flinders University of South Australia
GPO Box 2100 Adelaide 5001 Australia
Martin.Luerssen@infoeng.flinders.edu.au

Part of the assignment work for the topic *COMP3007 Artificial Intelligence* involves the training and evaluation of artificial neural networks on the problem of classifying characters in the *UCI Artificial Characters Database*. This tutorial is designed to give students the necessary background to accomplish this task. Essential MATLAB commands are introduced to the reader, as are the Neural Network toolbox (version 4) and various aspects of the UCI database. We also present a complete run-through of the required data pre-processing, neural network training, and results interpretation, with detailed instructions and hands-on examples.

1 Setup

Before continuing, please ensure that you have a copy of the UCI Artificial Characters Database in your home directory. It is available from the *UCI machine learning repository* at http://www.ics.uci.edu/~mlearn/MLSummary.html, but you can also find it in the `neural/artichar` subdirectory of the COMP3007 directory.

```
cp -R /opt/teaching/comp3007/neural/artichar .
```

Copying the entire directory requires around 18MB of disk space in your account. You can save 11MB by deleting several unnecessary files.

```
rm m* *.zip
```

You also need access to MATLAB R13 or R14 with the Neural Networks Toolbox Version 4.X. If you are using other versions of MATLAB or the toolbox, please note that some of the instructions, particularly in section 3, may not be correct. MATLAB can be started from any UNIX terminal and is also present on the PCs in the PC lab. In a UNIX shell, type `matlab` if you want the graphical user interface or `matlab -nodesktop` if you want to work in text mode (e.g. over Telnet). Be aware that simulating neural networks is computationally expensive, so try not to do it on a terminal in a full lab. Using the PCs is preferable in this respect, but you cannot use the `convert` program (see section 3.1) directly on a PC.

2 MATLAB

MATLAB (*Mat*rix *Lab*oratory) is a programming language and a development environment for matrix-based computation. It integrates an editor, an interpreter, and numerous visualisation tools in a single, powerful application. MATLAB can be extended

with *toolboxes*, which implement various algorithms commonly used in science and engineering. Of particular interest to us here is the *Neural Network Toolbox*, which constitutes one of the most comprehensive neural network packages currently available. While it is quite easy to create and run neural networks with this toolbox, some elementary knowledge of MATLAB is necessary. The best way to learn MATLAB is to use it, so you should follow the examples in the next few subsections and then experiment a little on your own.

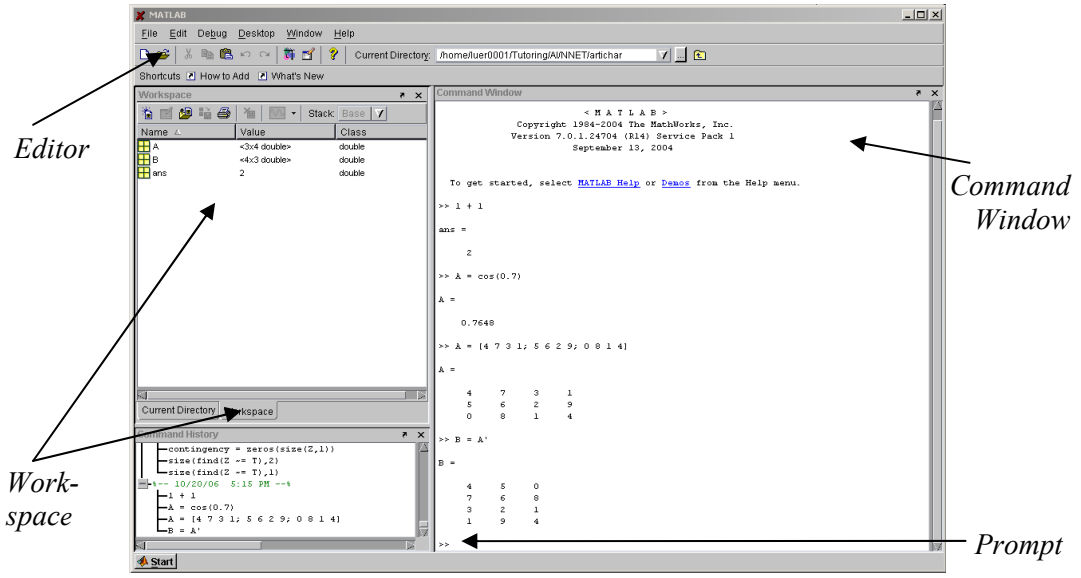

Figure 1: The graphical user interface of MATLAB with the most significant features highlighted.

2.1 The Basics

The >> prompt in the *Command Window* is where statements written in the MATLAB programming language can be entered for evaluation. They can be mathematical expression such as

```
>> 1 + 1
```
(i.e. you have typed 1+1 at the prompt)

to which MATLAB will unsurprisingly respond with

```
ans =
     2
```

Variables and functions also exist in MATLAB, for example

```
>> A = cos(0.7)
A =
    0.7648
```

assigns the return value of the cosine function to variable A. By default, MATLAB echoes the value of the assigned variable (as shown in green above). If you like to avoid this, use a semicolon after your statement.

```
>> A = cos(0.7);
```

The variables that have been defined are visible in the *Workspace* (click on the Workspace tab if the Launch Pad is in its place). You can delete all the presently defined variables with the `clear` command.

```
>> clear
```

2.2 Into the Matrix

Data in MATLAB, including the data that will be passed into our neural networks, is typically handled as matrices. A matrix is a rectangular array of numbers, and a matrix with R rows and C columns is referred to as an RxC matrix. The following is a 3x4 matrix:

$$\begin{bmatrix} 4 & 7 & 3 & 1 \\ 5 & 6 & 2 & 9 \\ 0 & 8 & 1 & 4 \end{bmatrix}$$

In MATLAB, you can define this matrix and store it in a variable of your, such as A.

```
>> A = [ 4 7 3 1; 5 6 2 9; 0 8 1 4];
```

The spaces between numbers here separate each column, whereas the semicolon indicates that a new row should begin.

Sometimes it is useful to exchange the rows of a matrix with its columns. This is called *transposing* the matrix, and we can do this in MATLAB with the apostrophe (‘) operator.

```
>> B = A'
B =
     4     5     0
     7     6     8
     3     2     1
     1     9     4
```

The value in a particular row and column can be accessed by using a 2-dimensional index, where the first element is the row and the second is the column.

```
>> B(2,3)
ans =
     8
```

You can also take the entire row or column by using a colon (:) instead of a numerical index.

```
>> B(2,:)
ans =
     7     6     8
```

A 1xC matrix (such as this) or an Rx1 matrix may also be referred to as a *vector*.

2.3 Help!

All MATLAB functions, including those in the toolboxes, are described in *help files*, which are accessible with the following commands:

`help` *name*	–	prints the help file for *name* in the Command Window
`helpwin` *name*	–	shows the help file in a browser (with hyperlinks)
`lookfor` *name*	–	returns a list of all the commands containing the word *name* in their respective help files

If you just type `help` (or `helpwin`), MATLAB will give you a basic overview of the help topics, as shown below.

```
HELP topics
matlab/general        -  General purpose commands.
matlab/ops            -  Operators and special characters.
matlab/lang           -  Programming language constructs.
... (various toolboxes)
nnet/nnet             -  Neural Network Toolbox
... (various other toolboxes)
```

So if you wish to know something about the Neural Network Toolbox, you can type `help nnet/nnet` (or just `help nnet`, because the help function explores all subcategories automatically). This gives you a list of the functions that make up the Neural Network Toolbox.

```
  Neural Network Toolbox
  Version 4.0.4 (R14SP1) 05-Sep-2004

  Graphical user interface functions.
    nntool   - Neural Network Toolbox graphical user
               interface.
... (various commands)
  New networks.
    network  - Create a custom neural network.
    newc     - Create a competitive layer.
    newcf    - Create a cascade-forward backpropagation
               network.
    newelm   - Create an Elman backpropagation network.
    newff    - Create a feed-forward backpropagation
               network.
... (more commands)
```

Typing help `netff` will provide you with instructions on how to use the `netff` function, which creates feed-forward backpropagation networks. These instructions are quite sufficient if you already know what you're doing, but otherwise they may seem a bit concise. It is therefore a good idea to also have a look at the proper documentation of the Neural Network Toolbox, which you can find in the Help menu or by pressing F1 and choosing the Neural Network Toolbox in the Help Navigator on the left-hand side of the Help window.

2.4 Scripting

Sequences of commands can be saved as a *script*. You can use the M-file editor to edit scripts, but any text editor is fine. Scripting is particularly convenient if you intend to use more complex programming constructs such as `while` loops and `if` statements. A script can also be encapsulated into a function, so that parameters can be passed and values returned from the call to the script.

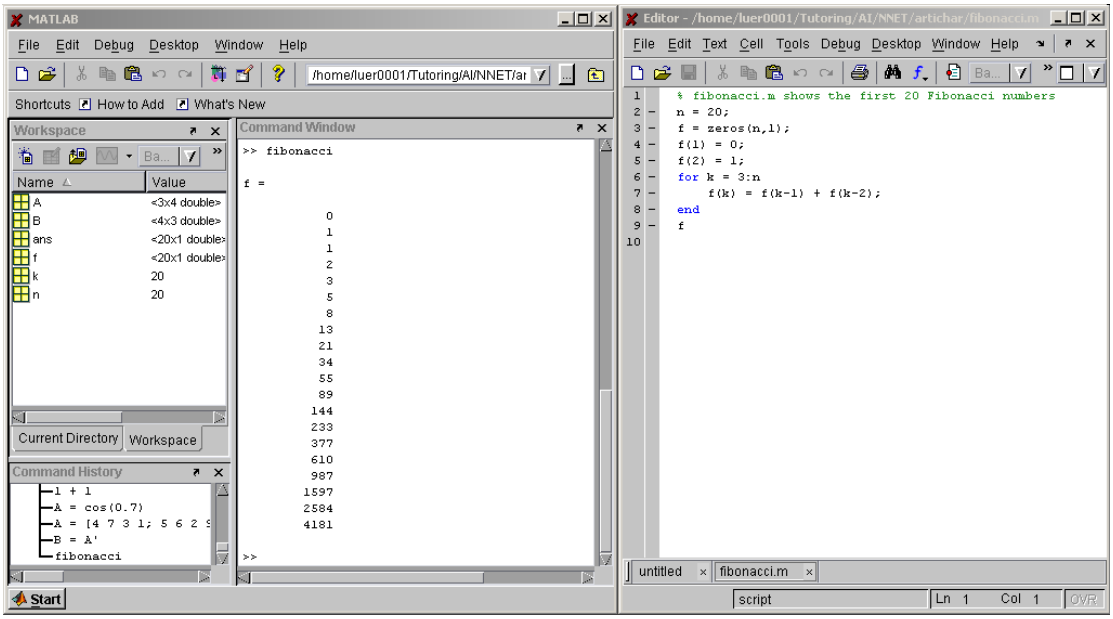

Figure 2: Example script that prints the first 20 numbers in the Fibonacci sequence.

A script has to be saved with a *.m* extension. You can then execute the script by calling its name, e.g. if the script is named fibonacci.m (as in Figure 2), you can type `fibonacci` at the prompt. However, you need to make sure the script is located in the working directory of MATLAB. The current working directory is revealed with the `pwd` command (as in a UNIX shell).

You can also list the directory contents with `ls` and change the working directory with `cd` (e.g. `cd /opt/teaching/comp3007/neural/artichar`). A script named *play.m* is already present in the `artichar` directory. It will train and simulate a neural network, but it is in fact not a very useful script, since it was written for an earlier version of MATLAB.

2.5 You don't like me?

If the above introduction to MATLAB left you confused or perhaps yearning for more, there are several excellent tutorials available on the Internet. You may want to check the following out:
http://www.mathworks.com/academia/student_center/tutorials/launchpad.html
http://www.cyclismo.org/tutorial/matlab/
http://www.indiana.edu/~statmath/math/matlab/gettingstarted/index.html

3 Neural Networks

An *artificial neural network* is a circuit composed of many simple processing units referred to as *neurons*, which are inspired by their (much more complex) biological counterparts in the brain. Each neuron receives inputs from several other neurons; it then performs some basic processing on these inputs and passes the result on to other neurons. Each connection typically has a *weight* associated with it, which affects the strength of the signal that passes along this connection. With the right choice of neurons, connections, and weights, a neural network can do any processing you like, but

how would you come up with the right choice? Neural networks are so popular because researchers have devised ways of automatically learning certain aspects of network design, particularly the value of the weights, just by showing examples of the problem to the network.

In order to properly understand neural networks, you really need to read up on them. There are several good books in the library, I suggest:

- Neural network design / Martin T. Hagan, Howard B. Demuth, Mark Beale
  Central Library – 006.32 H141n
- Neural networks : a comprehensive foundation / Simon Haykin
  Central Library – 006.32 H419n

Additionally, you can find plenty of information on neural networks by searching the Internet, although the quality of the results is frequently poor. Neural network tutorials specific to MATLAB are also available, but be aware that some might refer to older versions of the relevant toolbox. Here are some suggested neural network links:
http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html
http://www.dacs.dtic.mil/techs/neural/neural_ToC.html
http://www.avaye.com/files/articles/nnintro/nn_intro.pdf
A valuable resource is also:
http://www.faqs.org/faqs/ai-faq/neural-nets/part1/preamble.html

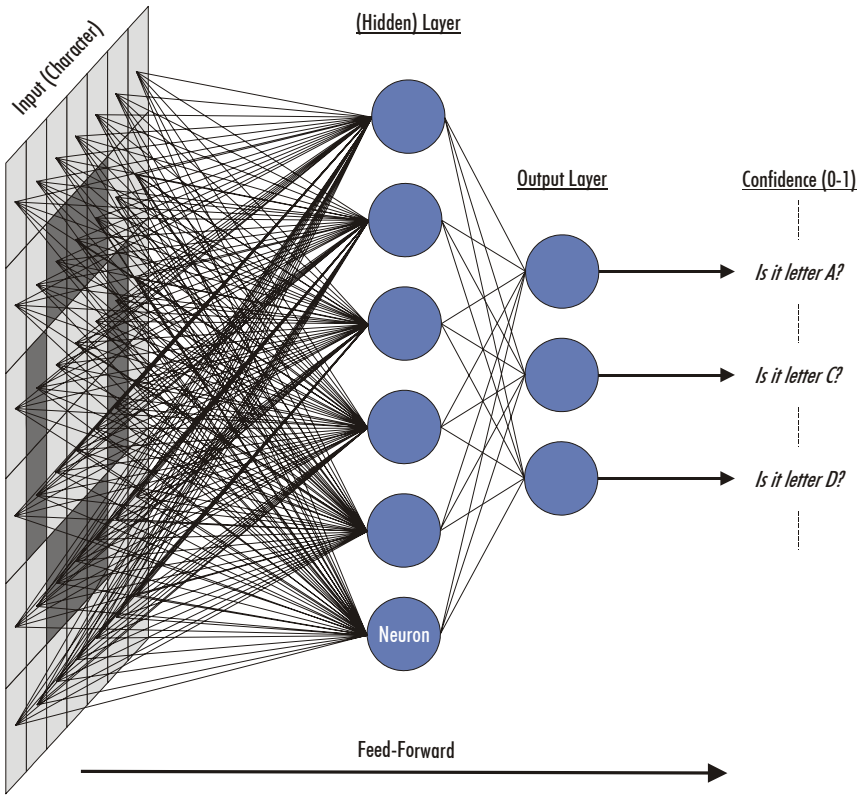

Figure 3: A neural network for recognising 3 different alphabetic letters.

3.1 Character Recognition

Character recognition involves the translation of handwritten or typewritten text captured by a scanner into machine-editable text. This technology has long been used by libraries to make old documents available electronically. The task is made difficult by variations in how the characters are drawn. Neural networks are particularly adept at overcoming this problem, because they can learn to extract the features that are most relevant to the recognition of the characters. Our goal hence is to create a neural network, as shown in Figure 3, that takes a picture of a character as an input and tells us which letter of the alphabet it represents.

The source of our character "pictures" will be the UCI Artificial Characters Database. If you have completed the setup in section 1, you should already have the database in your own `artichar` directory. The database contains artificially distorted pictures of characters of the alphabetic letters A, C, D, E, F, G, H, L, P, and R. Each character is represented by a set of line segments stored in files in the `learning` and `testing` subdirectories. However, lines are not a suitable data format to feed into a neural network. We need something as in Figure 4: a set of points/pixels. Fortunately, we can easily convert from lines into pixels by using the `convert` program located in the `artichar` directory.

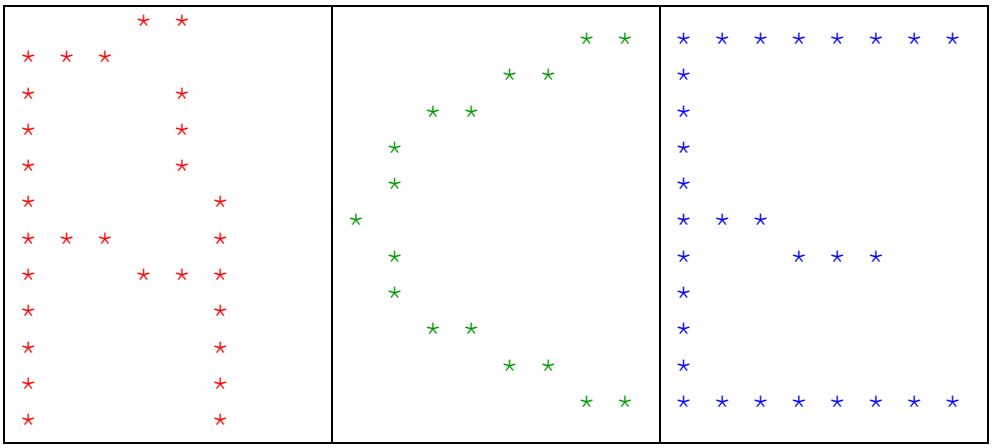

Figure 4: You can use the `showpat` program to show the characters of your pattern file.

`Convert` generates two files: `pattern` and `target`. The pattern file contains all the characters you selected to convert, with each row holding a 12x8 (=96) bit array indicating the pixels that define our character. The target file specifies the identity of each character in the pattern file, so, for example,

```
1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
```

indicates that the first three rows/characters of the pattern file represent the letters A (1st letter of the available alphabet), D (3rd letter), and F (5th letter).

Several instances of pattern and target files are already located in `artichar` and its subdirectories, but you should generate your own, so you know what letters your network is trying to recognise. It is also important to create separate sets for training and testing (and validation, if you like to be thorough), each of which should hold differ-

ent characters of the same letters (i.e. different ranges, see below). http://www.faqs.org/faqs/ai-faq/neural-nets/part1/section-14.html gives a proper explanation of the different uses of training, testing, and validation sets.

Convert: Usage Guide

Usage:
`convert -2c chars-to-convert range-begin range-end directory`

chars-to-convert: any combination of a, c, d, e, f, g, h, l, p, and r

range-begin, range-end: the range of characters of each of the above letters that you want to have converted; so 10 25 would choose characters no. 10 to no. 25 of a, c, d … etc.

directory: location to read characters from, either `learn/` or `test/`; there are 100 instances of each letter in `learn/` and 500 in `test/`. There is nothing special about either directory despite their different names, so you can pick any you like.

Examples:

`./convert -2c acdfgr 1 20 learn/`
will convert characters 1-20 of the letters A, C, D, F, G, and R from directory learn/

`./convert -2c dgp 100 300 test/`
will convert characters 100-300 of the letters D, G, and P from directory test/

NB: You don't need to use a UNIX shell to call convert (although you will need to at least run MATLAB on a UNIX terminal). In MATLAB you can call any UNIX command by putting an exclamation mark (!) in front of the command, so *!./convert -2c dgp 100 300 test/* will do just the same as the example above.

3.2 Creating the Neural Network

The first step in using a neural network is to create one. You can create a network using the functions listed under *New Networks* when you type `help nnet`. For now, we will focus on a standard neural network, the feedforward backpropagation network. A new instance of this network can be constructed with the `newff` function. As `help newff` informs us,
`NEWFF(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)`
has six arguments:

PR is an Rx2 matrix of min and max values for R input elements. We have 96 input elements (see section 3.1) consisting of 0s and 1s, so PR = [zeros(96,1) ones(96,1)].

[S1 S2...SNl] is a vector of layer sizes, i.e. the number of neurons in each stage of the network. For our chosen type of network, it is a good idea to have a layer in addition to the outputs. We'll just make a guess and put 4 neurons into this layer. The output layer, the last value in the vector, has to be equal to the number of different letters we expect, as each output indicates a separate letter (see Figure 3). So the vector is [4 6] if we assume we called on *!./convert -2c acdfgr 1 20 learn/* earlier.

{TF1 TF2...TFNl} is a special kind of matrix, a *cell array* – hence the curly brackets. It specifies for each layer what kind of processing the neurons are expected to do; see *Transfer Functions* in the help documentation for more on this. Let's pick 'logsig' for all layers = {'logsig' 'logsig'}.

BTF is the training function, where you again have multiple choices. The default is 'trainlm', which is very memory hungry, so we'll be using 'traingdx' instead.

BLF is irrelevant for this task, so we will leave it as default.

PF, the error measure, should be consistent between your experiments and/or with your method of interpreting the results, so we will leave it as default here as well.

With all the network arguments having been decided upon, we can create our network and store it in a variable called `mynet`.

```
>> mynet = newff([zeros(96,1) ones(96,1)], [4 6], {'log-
sig' 'logsig'}, 'traingdx');
```

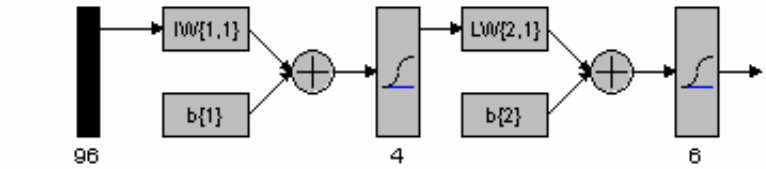

Figure 5: According to the MATLAB `nntool`, the `mynet` network looks like this, with 96 inputs, 4 neurons in the hidden layer, and 6 outputs.

3.3 Loading the Character Data

Before showing the characters to the network, they need to be loaded into MATLAB. For this you need valid pattern and target files; section 3.1 provided you with instructions on how to go about this. For the examples below I used pattern and target files from a call to *!./convert -2c acdfgr 1 20 learn/*. The resulting files can be read into MATLAB with the `load` function.

```
>> P = load('-ascii', 'pattern');
>> T = load('-ascii', 'target');
```

This reads a file called `pattern` into variable P and a file called `target` into variable T (if you are using different file names, substitute these here). Each row in the matrix P contains pixels of a character, and each row in the matrix T shows the iden-

tity of that character. However, the neural network training function requires each column (not row) to hold this information, so we need to transpose these matrices.

```
>> P = P';
>> T = T';
```

Another potential problem is that the target file always specifies which of the 10 letters a particular character represents, even if you're using fewer than 10 letters in your experiment. Previously, we proposed to create a network with as many outputs as letters used, so if you want to compare network outputs against the targets in T, then T must also be reduced to only those letters actually used.

```
>> T = T(find(sum(T')), :);
```

3.4 Training the Neural Network

Manually setting the weights of each link between each neuron would be very challenging and tedious. Fortunately, neural networks can be automatically *trained* so that a particular input pattern leads to a specific target output. Typically many pattern/target pairs are needed to train a network properly. We've loaded a fair number of those into MATLAB in the previous section. However, before we start training, we have to adjust some of the training parameters, which are stored in the *trainParam* structure array of the network.

```
>> mynet.trainParam
ans =
          epochs: 100
            goal: 0
              lr: 0.0100
          lr_dec: 0.7000
          lr_inc: 1.0500
        max_fail: 5
    max_perf_inc: 1.0400
              mc: 0.9000
        min_grad: 1.0000e-06
            show: 25
            time: Inf
```

The goal is 0, i.e. no error on the problem task, but this is unlikely to be achieved, as we are only training for a maximum of 100 epochs. Each epoch is a presentation of all the pattern/target pairs to the learning network. This is not quite enough for the 'traingdx' algorithm, however, so we will reset it to 500 epochs.

```
mynet.trainParam.epochs = 500;
```

That's it! We are now ready to train the network `mynet` using patterns P and targets T and storing the newly trained network back into the variable `mynet`.

```
mynet = train(mynet,P,T);
```

A curve of the performance of the network at each epoch is shown during training. Training is likely to continue until the 500th epoch, although the training algorithm may decide earlier that further training is not worthwhile.

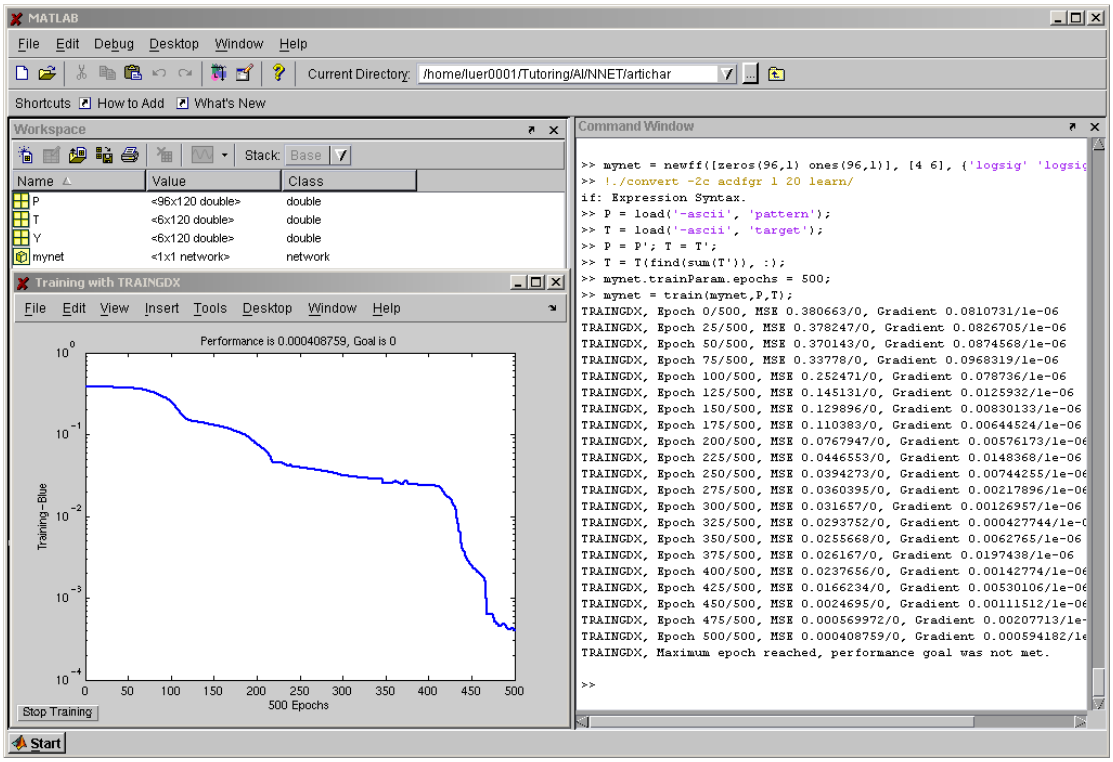

Figure 6: Creating and training our neural network.

3.5 Testing the Neural Network

Figure 6 indicates that our neural network trained well, as the error at the final epoch looks pretty low. So did our network learn to recognise the chosen letters? If a large network is trained for many epochs, it can simply memorise the correct answer for each character – and thus fail to recognise any new characters. This problem is known as *overfitting*. To properly evaluate our network, it is necessary to test it on a different set of characters than the one used for training.

```
>> !./convert -2c acdfgr 1 20 test/
```

(first 20 characters, but from the test/ directory now)

```
>> P = load('-ascii', 'pattern');
>> T = load('-ascii', 'target');
>> P = P';
>> T = T';
>> T = T(find(sum(T')), :);
```

The `sim` function presents all the patterns in P to `mynet`. The output of the network can then be stored in a variable Y.

```
>> Y = sim(mynet,P);
```

Neural networks produce real numbers as outputs, which are therefore rarely exactly 0 or 1. Let's be easy on the network and assume that the biggest output is the network's best guess of the presented letter. Y can be converted so that the biggest output is 1 and all other outputs are 0.

```
>> Y = (Y == ones(size(Y,1),1)*max(Y));
```

Now all that is needed is to compare Y against the actual targets T to see whether `mynet` got them right. This can be visualised as a *confusion matrix*, where each column represents the predicted instances of a character, while each row represents the actual instances of a character. The following script will create such a matrix and store it in variable C.

```
>> [I,x] = find(T);
>> [K,x] = find(Y);
>> C = zeros(size(Y,1))
>> for i = 1 : size(Y,2)
      C(I(i),K(i)) = C(I(i),K(i)) + 1;
   end
```

```
C =
                    Predicted
          A     C     D     F     G     R
    A    18     0     0     1     0     1
    C     0    18     0     0     2     0
    D     0     0    19     1     0     0
F   F     0     0     2    17     0     1
    G     0     3     0     0    17     0
    R     0     0     0     0     0    20
```

Actual

Since 20 characters of each letter were tested in our experiment, a perfect result would show all 20s in the main diagonal of the matrix, i.e. all the predicted letters are the actual letters. Our results aren't quite that perfect. For example, the 3 (under the 18) indicates that the network thought that 3 characters were Cs when they really were Gs. Recognition performance is often given as the precision and recall of the network, which can be computed from the confusion matrix; both are around 91% for `mynet`. You can read more about this topic here:
http://www2.cs.uregina.ca/~dbd/cs831/notes/confusion_matrix/confusion_matrix.html

4 Final Words

Can we get better results than the above? It's up to you to find out! Many different kinds of networks can be used for character recognition. Some don't require training (e.g. Radial Basis networks) and some don't need targets (e.g. Self-organising maps). There are also many parameters that can be varied, including the training function and variables, the transfer function, and the architecture of the network. For instance, networks with more neurons will train more easily, but are also more likely to overfit. Maybe you should train with more different characters instead? What is the best compromise? Go and explore!