Select Theme

Introduction to the Matlab Neural Network Toolbox 3.0

The Matlab Neural Network Toolbox (NNT) is an all-purpose neural network environment. Everything but the kitchen sink is included, and most of it has somehow been incorporated in the network object. Trying to understand this object and its properties can be a bewildering experience, especially since the documentation is of the usual Matlab quality (which is a Bad Thing^{TM}).

Note: This tutorial describes version 3.0 of the NNT. The newest version is 4.0, available since Matlab 6.0 (R12).

The purpose of this document is to try to explain to all those interested how to build a custom feed-forward network starting from scratch (i.e. a `blank' neural network object). It consists of the following sections:

INTRODUCTION

Matlab's Neural Network Toolbox (NNT) is powerful, yet at times completely incomprehensible. This is mainly due to the complexity of the network object. Even though high-level network creation functions, like **newp** and **newff**, are included in the Toolbox, there will probably come a time when it will be necessary to directly edit the network object properties.

Part of my job is teaching a neural networks practicum. Since we wanted the students to concern themselves with the ideas behind neural networks rather than with implementation and programming issues, some software was written to hide the details of the network object behind a Matlab GUI. In the course of writing this software, I learned a lot about the NNT. Some of it I learned form the manual, but most of it I learned through trial-and-error.

And that's the reason I wrote this introduction. In order to save you a lot of time I already had to spent learning about the NNT. This document is far from extensive, and is naturally restricted to my own field of application. Therefore, only feed-forward networks will be treated. I do think however that reading this can give you a firm enough background to start building your own custom networks, relying on the Matlab documentation for specific details.

All Matlab commands given in this document assume the existence of a NNT network object named `net'. To construct such an object from scratch, type

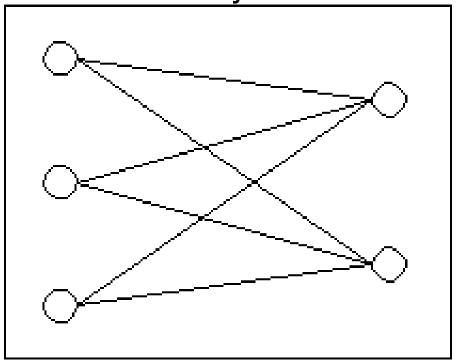
```
>> net = network;
```

which gives you a `blank' network, i.e. without any properties. Which properties to set and how to set them is the subject of this document.

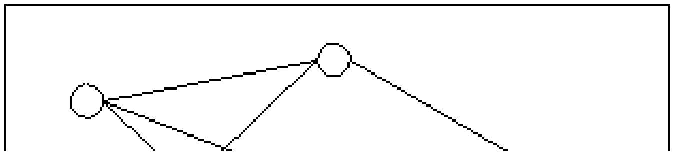
Any errors, omissions etc. are completely my responsibility, so if you have any comments, questions or hate mail, please send them to me at portegie@science.uva.nl.

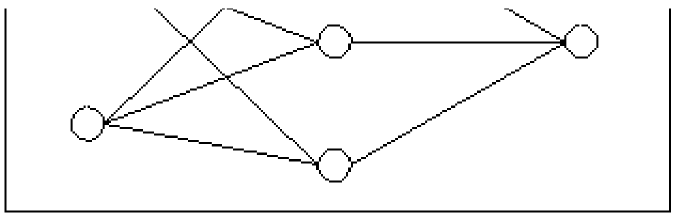
NETWORK LAYERS

The term `layer' in the neural network sense means different things to different people. In the NNT, a layer is defined as a layer of neurons, with the exception of the input layer. So in NNT terminology this would be a one-layer network:



and this would be a two-layer network:





We will use the last network as an example throughout the text.

Each layer has a number of properties, the most important being the transfer functions of the neurons in that layer, and the function that defines the net input of each neuron given its weights and the output of the previous layer.

Constructing Layers

OK, so let's get to it. I'll assume you have an empty network object named `net' in your workspace, if not, type

```
>> net = network;
```

to get one.

Let's start with defining the properties of the input layer. The NNT supports networks which have multiple input layers. I've never used such networks, and don't know of anybody who has, so let's set this to 1:

```
>> net.numInputs = 1;
```

Now we should define the number of neurons in the input layer. This should of course be equal to the dimensionality of your data set. The appropriate property to set is **net.inputs{i}.size**, where **i** is the index of the input layers. So to make a network which has 2 dimensional points as inputs, type:

```
>> net.inputs{1}.size = 2;
```

This defines (for now) the input layer.

The next properties to set are **net.numLayers**, which not surprisingly sets the total number of layers in the network, and **net.layers{i}.size**, which sets the number of neurons in the ith layer. To build our example network, we define 2 extra layers (a hidden layer with 3 neurons and an output layer with 1 neuron), using:

```
>> net.numLayers = 2;
>> net.layers{1}.size = 3;
>> net.layers{2}.size = 1;
```

Connecting Layers

Now it's time to define which layers are connected. First, define to which layer the inputs are connected by setting **net.inputConnect(i)** to 1 for the appropriate layer **i** (usually the first, so i = 1).

The connections between the rest of the layers are defined a connectivity matrix called **net.layerConnect**, which can have either 0 or 1 as element entries. If element (i,j) is 1, then the outputs of layer j are connected to the inputs of layer i.

We also have to define which layer is the output layer by setting **net.outputConnect(i)** to 1 for the appropriate layer **i**.

Finally, if we have a supervised training set, we also have to define which layers are connected to the target values. (Usually, this will be the output layer.) This is done by setting **net.targetConnect(i)** to 1 for the appropriate layer **i**. So, for our example, the appropriate commands would be

```
>> net.inputConnect(1) = 1;
>> net.layerConnect(2, 1) = 1;
>> net.outputConnect(2) = 1;
>> net.targetConnect(2) = 1;
```

Setting Transfer Functions

Each layer has its own transfer function which is set through the **net.layers{i}.transferFcn** property. So to make the first layer use sigmoid transfer functions, and the second layer linear transfer functions, use

```
>> net.layers{1}.transferFcn = 'logsig';
>> net.layers{2}.transferFcn = 'purelin';
```

For a list of possible transfer functions, check the Matlab documentation.

WEIGHTS AND BIASES

Now, define which layers have biases by setting the elements of **net.biasConnect** to either 0 or 1, where **net.biasConnect(i) = 1** means layer **i** has biases attached to it.

To attach biases to each layer in our example network, we'd use

```
>> net.biasConnect = [ 1 ; 1];
```

Now you should decide on an initialisation procedure for the weights and biases. When done correctly, you should be able to simply issue a

```
>> net = init(net);
```

to reset all weights and biases according to your choices.

The first thing to do is to set **net.initFcn**. Unless you have build your own initialisation routine, the value 'initlay' is the way to go. This let's each layer of weights and biases use their own initialisation routine to initialise.

```
>> net.initFcn = 'initlay';
```

Exactly which function this is should of course be specified as well. This is done through the property **net.layers{i}.initFcn** for each layer. The two most practical options here are Nguyen-Widrow initialisation ('initnw', type 'help initnw' for details), or 'initwb', which let's you choose the initialisation for each set of weights and biases separately.

When using 'initnw' you only have to set

```
>> net.layers{i}.initFcn = 'initnw';
```

for each layer **i** and you're done.

When using 'initwb', you have to specify the initialisation routine for each set of weights and biases separately. The most common option here is 'rands', which sets all weights or biases to a random number between -1 and 1. First, use

```
>> net.layers{i}.initFcn = 'initwb';
```

for each layer **i**. Next, define the initialisation for the input weights,

```
>> net.inputWeights{1,1}.initFcn = 'rands';
```

and for each set of biases

```
>> net.biases{i}.initFcn = 'rands';
```

and weight matrices

```
>> net.layerWeights{i,j}.initFcn = 'rands';
```

where **net.layerWeights{i,j}** denotes the weights from layer j to layer i.

TRAINING FUNCTIONS & PARAMETERS

The difference between train and adapt

One of the more counterintuitive aspects of the NNT is the distinction between **train** and **adapt**. Both functions are used for training a neural network, and most of the time both can be used for the same network.

What then is the difference between the two? The most important one has to do with incremental training (updating the weights after the presentation of each single training sample) versus batch training (updating the weights after each presenting the complete data set).

When using **adapt**, both incremental and batch training can be used. Which one is actually used depends on the format of your training set. If it consists of two matrices of input and target vectors, like

```
>> P = [ 0.3 0.2 0.54 0.6 ; 1.2 2.0 1.4 1.5]

P =

    0.3000     0.2000     0.5400     0.6000
    1.2000     2.0000     1.4000     1.5000

>> T = [ 0 1 1 0 ]

T =

     0     1     1     0
```

the network will be updated using batch training. (In this case, we have 4 samples of 2 dimensional input vectors, and 4 corresponding 1D target vectors).

If the training set is given in the form of a cell array,

```
>> P = {[0.3 ; 1.2] [0.2 ; 2.0] [0.54 ; 1.4] [0.6 ; 1.5]}

P =

    [2x1 double]    [2x1 double]    [2x1 double]    [2x1 double]

>> T = { [0] [1] [1] [0] }

T =

    [0]    [1]    [1]    [0]
```

then incremental training will be used.

When using **train** on the other hand, only batch training will be used, regardless of the format of the data (you can use both).

The big plus of **train** is that it gives you a lot more choice in training functions (gradient descent, gradient descent w/ momentum, Levenberg-Marquardt, etc.) which are implemented very efficiently. So when you don't have a good reason for doing incremental training, **train** is probably your best choice. (And it usually saves you setting some parameters).

To conclude this section, my own favourite difference between **train** and **adapt**, which is trivial yet annoying, and the reason for which completely escapes me: **the difference between passes and epochs**. When using **adapt**, the property that determines how many times the complete training data set is used for training the network is called **net.adaptParam.passes**. Fair enough. But, when using **train**, the exact same property is now called **net.trainParam.epochs**! If anybody can find any sort of ratio behind this design choice (or better, design flaw), please let me know!.

Performance Functions

The two most common options here are the Mean Absolute Error (**mae**) and the Mean Squared Error (**mse**). The mae is usually used in networks for classification, while the mse is most commonly seen in function approximation networks.

The performance function is set with the **net.performFcn** property, for instance:

```
>> net.performFcn = 'mse';
```

Train Parameters

If you are going to train your network using **train**, the last step is defining **net.trainFcn**, and setting the appropriate parameters in **net.trainParam**. Which parameters are present depends on your choice for the training function.

So if you for example want to train your network using a Gradient Descent w/ Momentum algorithm, you'd set

```
>> net.trainFcn = 'traingdm';
```

and then set the parameters

```
>> net.trainParam.lr = 0.1;
>> net.trainParam.mc = 0.9;
```

to the desired values. (In this case, **lr** is the learning rate, and **mc** the momentum term.)

Check the Matlab documentation for possible training functions and their parameters.

Two other useful parameters are **net.trainParam.epochs**, which is the maximum number of times the complete data set may be used for training, and **net.trainParam.show**, which is the time between status reports of the training function. For example,

```
>> net.trainParam.epochs = 1000;
>> net.trainParam.show = 100;
```

Adapt Parameters

The same general scheme is also used in setting **adapt** parameters. First, set **net.adaptFcn** to the desired adaptation function. We'll use **adaptwb** (from 'adapt weights and biases'), which allows for a separate update algorithm for each layer. Again, check the Matlab documentation for a complete overview of possible update algorithms.

```
>> net.adaptFcn = 'adaptwb';
```

Next, since we're using **adaptwb**, we'll have to set the learning function for all weights and biases:

```
>> net.inputWeights{1,1}.learnFcn = 'learnp';
>> net.biases{1}.learnFcn = 'learnp';
```

where in this example we've used **learnp**, the Perceptron learning rule. (Type 'help learnp', etc.).

Finally, a useful parameter is **net.adaptParam.passes**, which is the maximum number of times the complete training set may be used for updating the network:

```
>> net.adaptParam.passes = 10;
```

CONCLUSION

I hope this tutorial will help any of you struggling with the NNT. If you have any comments or questions, you can e-mail me at portegie@science.uva.nl.

Updates

October 13, 2000
    Corrected some weird typos.
May 11, 2000
    Corrected cell array syntax and some spelling errors. (Thanks to Homera Saeed).
Feb 22, 2000
    First Version