Neural Nets

To give you an idea of how new this material is, let's do a little history lesson. The origins of neural nets are typically dated back to the early 1940's and work by two physiologists, McCulloch and Pitts. Hebb's work came later, when he formulated his rule for learning. In the 1950's came the *perceptron. The perceptron* is what we called a linear neural network- it became clear that the perceptron could only classify certain types of data (what we now call linearly separable data). This shortcoming led to a stop in neural net research for many years- It was not until the 1980's that neural net research really took off from a coming together of many disparate strands of research- There was a group in Finland led by T. Kohonen that was working with self-organizing maps (SOM); there was the work from the 70s with Stephen Grossberg, and finally several researchers were discovering methods for training new networks that could be put in parallel (back-propagation).

It was in the 80's and 90's that researchers were able to prove that a neural network was a **universal function approximator**- That is, for any function with certain nice properties, we are able to find a neural network that will approximate that function with arbitrarily small error. It is interesting to note that the use of a neural network could be used to solve Hilbert's 13th Problem.

"Every continuous function of two or more real variables can be written as the superposition of continuous functions of one real variable along with addition."

Coming to present day, research is aimed at something called **deep neural nets** that perform **automatic feature extraction** and we'll discuss those once we've looked at the typical neural net.

The term "neural network" has come to be an umbrella term covering a broad range of different function approximation or pattern classification methods. The classic type of neural network can be **defined** simply as a connected graph with weighted edges and computational nodes. We now turn to the workhorse of the neural network community: The feed forward neural network.

General Model Building

We assume that we have p data pairs, $(\mathbf{x}^{(1)}, \mathbf{t}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{t}^{(2)}), \cdots, (\mathbf{x}^{(p)}, \mathbf{t}^{(p)})$ (the letter t is for *target*), and we are looking to build a function F so that ideally,

$$F(\mathbf{x}^{(i)}) = \mathbf{t}^{(i)} \qquad \text{for } i = 1, 2, \ldots, p$$

However, we will typically allow for error, so we want to distinguish between the network output and the desired target. Let $\mathbf{y}^{(i)}$ denote the output of the model so that now

$$\mathbf{y}^{(i)} = F(\mathbf{x}^{(i)}) \quad \Rightarrow \quad \text{Error}_i = \|\mathbf{t}^{(i)} - \mathbf{y}^{(i)}\|^2$$

If we assume that $\mathbf{y}^{(i)}$ depends on parameters, then we might state it as an optimization problem- Find the function F that minimizes the error function:

$$E = \frac{1}{2}\sum_{i=1}^{p} \|\mathbf{t}^{(i)} - \mathbf{y}^{(i)}\|^2$$

so that E is a function of the parameters in F, and we would go about determining the values of the parameters that minimize the error, and that will include differentiating E.

Example: Line of Best Fit

Here's an example of what we mean. Suppose we have 4 points:

$$(-1,1),(0,1/2),(1,1),(2,3)$$

and we want to find a model of the form: $y = a_0 + a_1x + a_2x^2$. Then in this problem, the $x-$values are the first values in the ordered pairs, $t-$values are the second in the ordered pairs, and to find the y's, we substitute in our values:

$$\begin{aligned} y^{(1)} &= a_0 - a_1 + a_2 \\ y^{(2)} &= a_0 \\ y^{(3)} &= a_0 + a_1 + a_2 \\ y^{(4)} &= a_0 + 2a_1 + 4a_2 \end{aligned}$$

The error function will then be:

$$E = (1-(a_0-a_1+a_2))^2 + \left(\frac{1}{2} - a_0\right)^2 + (1-(a_0+a_1+a_2))^2 + (3-(a_0+2a_1+4a_2))^2$$

We see that the error function is a function of the parameters a_0, a_1, a_2. Using Calculus, we can find the minimum by using an algorithm like gradient descent. FYI, if you want to try it out in this case, you should find that

$$a_0 = \frac{17}{40}, \quad a_1 = \frac{1}{40}, \quad a_2 = \frac{5}{8}$$

In a feed-forward neural network, what changes is the model for y. We will see what parameters the neural network uses to build its function.

But first, we look at some biology.

The Feed-Forward Neural Network

One neuron has the form in Figure 1 where information flows from left to right in the following way:

- Present real numbers $x_1, \ldots, x_n$ to the "input layer", as shown.

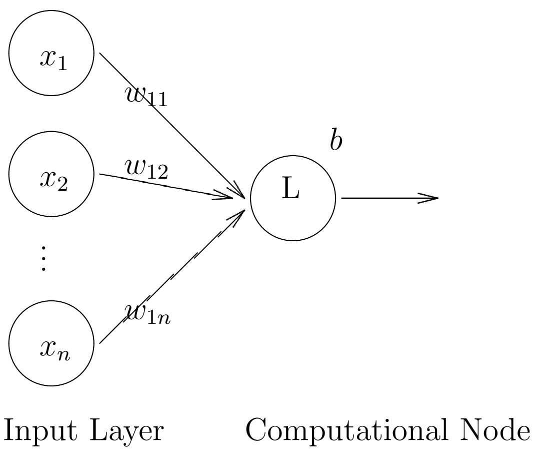


Figure 1: One Neuron. Information flows from left to right.

- The "w_{1j}" corresponds to a "weight". A weighted edge corresponds to multiplication by w_{1j}. In general, weight w_{ij} corresponds to the edge going FROM node j TO node i.
- At node L, the sum of the incoming signals is taken, and added to a value, b. We think of b as the "resting state" of the cell.
- A function $\sigma(x)$ is applied. This mimics the action of a neuron- If the stimulus reaches a certain point, the cell fires. This function (sigma for "sigmoidal") is a general model of that (to be discussed later).
- The result is passed along the axon.

Mathematically, the end result is:

$$\boldsymbol{x} \mapsto \sigma \left[(w_{11}, w_{12}, \ldots, w_{1n}) \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + b \right]$$

or

$$\boldsymbol{x} \mapsto \sigma(w_{11}x_1 + w_{12}x_2 + \ldots + w_{1n}x_n + b)$$

We can connect multiple computational nodes together to obtain a *neural network*, as shown in Figure 2.

If we use k neurons, then acting on them all at once, we can right the initial computation as:

$$\mathbf{x} \to W_1\mathbf{x} + \mathbf{b}_1$$

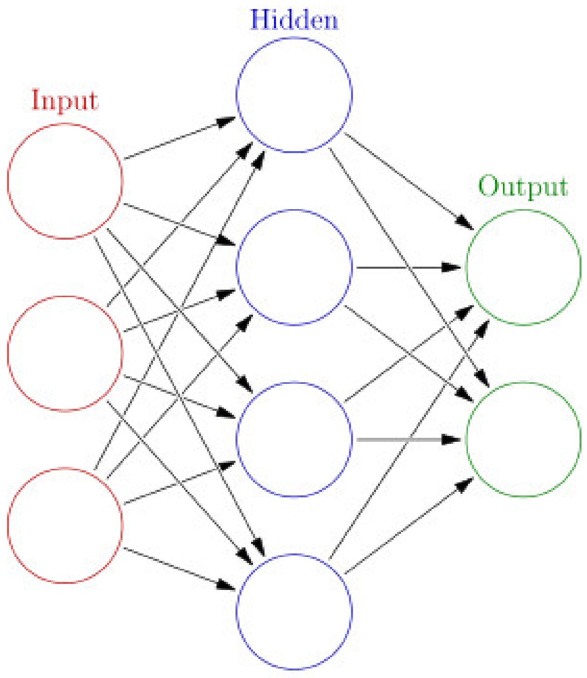


Figure 2: The three layer neural net. Shown are the input layer, the hidden layer, and the output layer.

Putting it all together, we could write the function F explicitly:

$$F(\mathbf{x}_i) = W_2\left(\sigma\left(W_1\mathbf{x}_i + \mathbf{b}_1\right)\right) + \mathbf{b}_2$$

so that, with σ defined, F becomes a function of the *weights* W_1, W_2, and the biases $\mathbf{b}_1, \mathbf{b}_2$.

Defining the Network Architecture

We have constructed what many people call a two layer network (although I typically say it is three layers- Some people don't count the input layer as a real layer):

- The first "layer" is called the *input layer*. If $\mathbf{x}_i \in \mathbb{R}^n$, then the input layer has n "nodes".
- The next layer is called the *hidden layer*, and it consists of k nodes (where k is the number of neurons we're using). The mapping from the input layer to the hidden layer is performed by our first affine map, then σ is applied to that vector.
- The last layer is called the *output layer*, and if $\mathbf{y} \in \mathbb{R}^m$, then the output layer has m nodes.

We did not need to stop with only a single hidden layer- Some researchers like to use multiple hidden layers as a default neural network- And in fact, that is what is at the heart of "deep learning". We'll stick with only one hidden layer for now.

Definition: The *architecture* of the neural network is typically defined by stating the number of neurons in each layer. For example, a $2-3-4$ network has one hidden network of three neurons, and maps $\mathbb{R}^2$ to $\mathbb{R}^4$.

The parameters of a neural network

To define a three layer neural network in the form $n-k-m$, we should first set up the transfer function σ. Although we could define a different σ for every neuron, we typically will use the same transfer function for all the neurons in a single layer.

Once that is done, then we have to find matrices W_1, W_2 (and more, if we use more layers) and the bias vectors $\mathbf{b}_1, \mathbf{b}_2$. Altogether, this makes $(nk+k)+(mk+m)$ parameters. Ideally, we would have much more data than that in order to get good estimates. In any case, we want to minimize the usual sum of squared error:

$$E(W_1, W_2, \mathbf{b}_1, \mathbf{b}_2) = \frac{1}{2}\sum_{i=1}^{p} \|\mathbf{t}^{(i)} - \mathbf{y}^{(i)}\|^2$$

where $\mathbf{y}^{(i)}$ is the output of the neural net.

The process of finding the parameters is called *training the network.* Matlab has some good built-in algorithms for training that we'll use.

The transfer function

In a neuron, the incoming signals to the cell body must usually surpass some lowest trigger value before the signal is sent out. A graph of this would be a step function, where the step is at trigger.

This is not a good function using notions from Calculus because the voltage function is not continuous and not differentiable at the trigger . We replace the step function by any function that is:

- Increasing.
- Differentiable
- Has finite horizontal asymptotes at $\pm\infty$.

Such a function generally looks like an extended "S"- We call it a *sigmoidal function.*

There are many ways we could define a sigmoidal, but here are some standard choices (going from most to least used):

- $$\sigma(r) = \frac{1}{1+\mathrm{e}^{-r}}$$

  Matlab calls this the "logsig" function.

- $$\sigma(r) = \arctan(r)$$

  Matlab does not use this one.

- $$\sigma(r) = \tanh(r) = \frac{\mathrm{e}^{2r}-1}{\mathrm{e}^{2r}+1}$$

  Matlab calls this one "tansig".

Exercises with σ

1. Compute the limits as $x \to \pm\infty$ for the two types of sigmoidal functions that Matlab uses. Show that they are also monotonically increasing functions.

2. Let $\sigma(x) = \frac{1}{1+\mathrm{e}^{-\beta x}}$. Show that

   $$\sigma'(x) = \beta\sigma(x)(1-\sigma(x))$$

3. If $\boldsymbol{x} \in \mathbb{R}^n$ and our targets $\boldsymbol{t} \in \mathbb{R}^m$, and we use k nodes in the hidden layer, how many unknown parameters do we have to find?

   As you are constructing your network, keep this number in mind. In particular, you should have at least several data points for each unknown parameter that you are looking for.

4. We have to be somewhat careful when our data is badly scaled. For example, complete this table of values:

x	0	0.5	1	10	40	100
$\tanh(x)$						
`logsig`(x)						

   What do you see as x becomes very large? This phenomenon goes by the name of *saturation.*

5. Some people like to scale the sigmoidal function by an extra parameter, β, that is $\sigma(\beta x)$. Show by sketching what happens to the graph of the sigmoidal (either the `tansig` or `logsig`) as you change β.

   It is not necessary to scale the sigmoidal, as this is equivalent to scaling the data instead (via the weights).

6. Show, using the definition of the hyperbolic sine and cosine, that the hyperbolic tangent can be written as:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$

7. Show that the hyperbolic tangent can be computed as:

$$\tanh(x) = \frac{2}{1 + \mathrm{e}^{-2x}} - 1$$

   (Matlab claims that this version is faster, but warns about possible numerical error)

Using Matlab to build a Net

For the type of network we're using, the relevant Matlab command (from the Neural Network Toolbox) are: `feedforwardnet`, which initializes the network `train`, which trains the network, and `sim`, which simulates the network using new data.

Example:

Build a 1-10-1 network to model a dataset that comes with Matlab (shown below). Matlab uses the percent sign for comments, which are not necessary to include.

```
[x,t]=simplefit_dataset;  %Data comes with Matlab

%Plot the data to see what you're building:
plot(x,t);

%Initialize the network using 10 nodes in the hidden layer.
net=feedforwardnet(10);
net=train(net,x,t);

% Test the net on new data:
xt=linspace(min(x),max(x),200);  %Create a vector of 200 points, evenly
                                 % spaced between the min and max of x.
yt=sim(net,xt);  %Simulate the network using the new data.

plot(x,t,'b*',xt,yt,'k-');  %Plot the results
```

Example: Sibling data

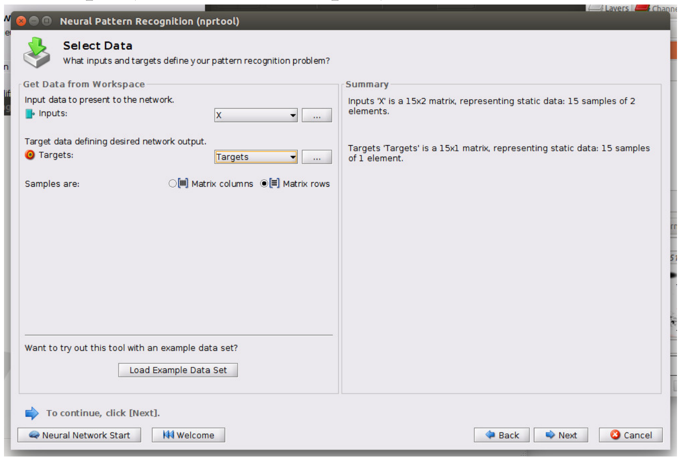
This is Example 7, page 874 of the text. Our textbook uses different software to build the model. Here is the Matlab version. The data is on our class website, *SiblingData.m*. It is a text file if you want to look at it.

"Six people live in Hooterville. Persons 1-3 are the Hatfield siblings, and persons 4-6 are the McCoy siblings. Two people who are not siblings are acquaintances. Can a neural net learn to correctly classify pairs of people as siblings or acquaintances?"

We load the data into Matlab by typing: `SiblingData` (be sure you have downloaded it- it should be in your home directory (or navigate to the appropriate directory using the navigation bar above the command window). To use Matlab's graphical interface, type the following in the command window:

```
nnstart
```

In this case, choose "Pattern recognition and classification". Once that window opens, select "next" to start. The dialog box you should see is **Select Data** (if not, let me know). Select the data that you loaded earlier (select X for the input, T for the output, and select "Matrix rows" as shown below):



Neural Pattern Recognition (nprtool)

Select Data
What inputs and targets define your pattern recognition problem?

Get Data from Workspace
Input data to present to the network.
Inputs: X
Target data defining desired network output.
Targets: Targets
Samples are: ( ) Matrix columns [✓ Matrix rows]

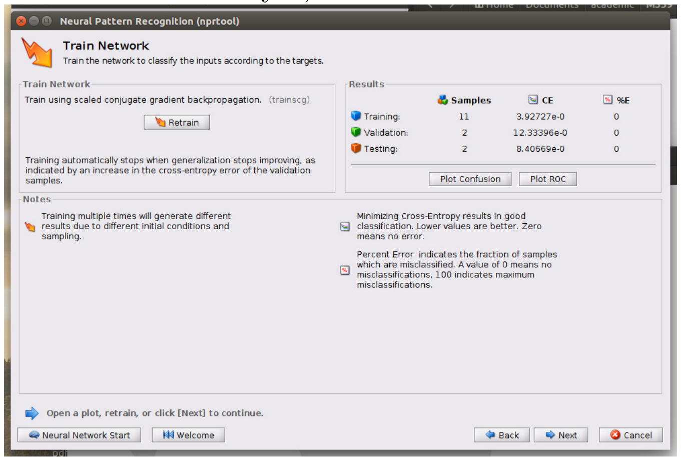
Want to try out this tool with an example data set?

Summary
Inputs 'X' is a 15x2 matrix, representing static data: 15 samples of 2 elements.

Targets 'Targets' is a 15x1 matrix, representing static data: 15 samples of 1 element.

To continue, click [Next].

Once the data is loaded, on the next screen select 5 for the number of nodes in the hidden layer, then "next". Now train the network:



Neural Pattern Recognition (nprtool)

Train Network
Train the network to classify the inputs according to the targets.

Train Network
Train using scaled conjugate gradient backpropagation. (trainscg)

Training automatically stops when generalization stops improving, as indicated by an increase in the cross-entropy error of the validation samples.

Results

	Samples	CE	%E
Training:	11	3.92727e-0	0
Validation:	2	12.33396e-0	0
Testing:	2	8.40669e-0	0

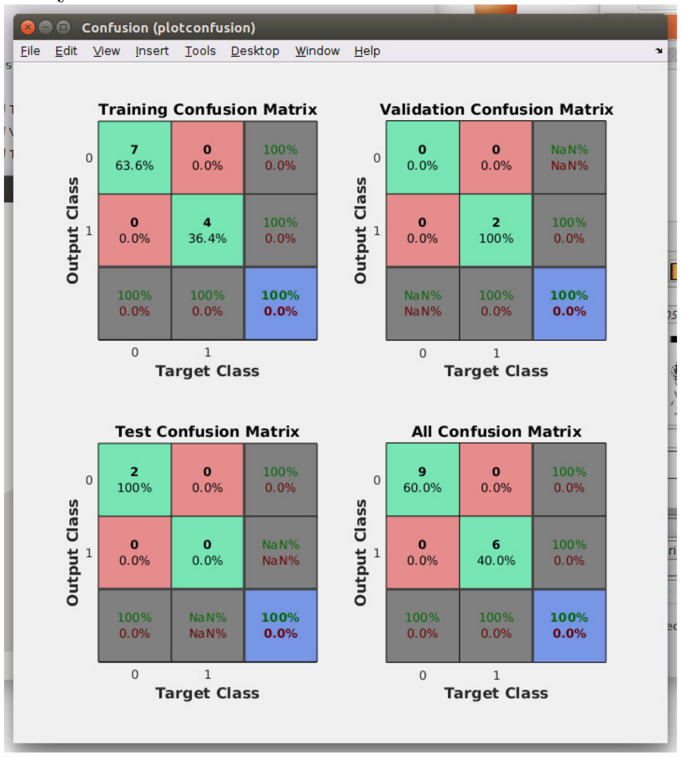
Notes
Training multiple times will generate different results due to different initial conditions and sampling.

Minimizing Cross-Entropy results in good classification. Lower values are better. Zero means no error.

Percent Error indicates the fraction of samples which are misclassified. A value of 0 means no misclassifications, 100 indicates maximum misclassifications.

Open a plot, retrain, or click [Next] to continue.

And it is common to use a "confusion matrix" to see how well the classification performed. The example shown gave 100% classification rate, but you may have to re-train the network several times to get it.


Confusion (plotconfusion)
File Edit View Insert Tools Desktop Window Help

Training Confusion Matrix

Output Class \ Target Class	0	1	
0	7 63.6%	0 0.0%	100% 0.0%
1	0 0.0%	4 36.4%	100% 0.0%
	100% 0.0%	100% 0.0%	100% 0.0%

Validation Confusion Matrix

Output Class \ Target Class	0	1	
0	0 0.0%	0 0.0%	NaN% NaN%
1	0 0.0%	2 100%	100% 0.0%
	NaN% NaN%	100% 0.0%	100% 0.0%

Test Confusion Matrix

Output Class \ Target Class	0	1	
0	2 100%	0 0.0%	100% 0.0%
1	0 0.0%	0 0.0%	NaN% NaN%
	100% 0.0%	NaN% NaN%	100% 0.0%

All Confusion Matrix

Output Class \ Target Class	0	1	
0	9 60.0%	0 0.0%	100% 0.0%
1	0 0.0%	6 40.0%	100% 0.0%
	100% 0.0%	100% 0.0%	100% 0.0%

At this point, we could do some analysis to see how robust our network is, but for now, let's try another training problem. Go ahead and close the windows in Matlab, and clear the memory:

```
close all
clear
clc
```

Example: Predicting Bankruptcy

This is Example 9, page 881 of our textbook. Our text also lists the data and gives an explanation. I have extracted that information to a text file suitable for Matlab, `Bankruptcy.m`. Download that, and to load it into Matlab, type `Bankruptcy` on the command line as before.

See if you can adjust parameters until you can get the 93% classification rate that our author claimed. (I actually don't think it will be quite that high- That percentage depends on how the data was used, and he gives no details)