Neural Networks and the Brain
a.k.a. Will I Graduate?

Laura Elisa Celis

Friday, April 28th

1 Introduction

Biological neurons, namely those found in the brain, are interconnected in vast and complex networks. They work by taking an input, and if the input is "strong enough", passing the message on to other neurons. Each neuron may have several inputs, and it is only when the combination is right that the message continues on to other neurons. Amazingly enough this is all the structure we need for the complex processing in our brains to work! An artificial neural network attempts to simulate how the brain works in the hopes of being able to quickly and efficiently process information in an equivalent manner. In addition, artificial networks can be trained in a way that allows them to "learn", again simulating brain function.

Here I explore the world of neural networks and the role of data in the learning process. I determine that in some cases, we may not need to train a network as rigorously as is often assumed. By making a base class of assumptions, potentially similar to those base assumptions we as people often make in real life, the network can learn enough to make reasonable judgements about the way in which the world works.

2 Neural Networks

To begin our discussion, I give a brief introduction into what a neural network actually is, and the way in which it is often constructed and trained. This will allow me to discuss my efforts in more detail.

2.1 Topology

The first thing to consider in a network is its topology - namely how it is put together. Clearly we are going to need some number of *inputs*. These inputs are the information we know, and what we will base our prediction on. To give an example, weather predictions are often made using neural networks with 19 inputs. Additionally, in order for the network to be useful, it must have some number of *outputs*. In the weather example there may be several outputs such as temperature, wind speed, and probability of rain. However many useful networks produce simply one output.[StatSoft, 2004]

Additionally, a network will have a series of *hidden layers* between the inputs and the outputs. Information always flows from the inputs to the outputs through these layers with no backtracking. The purpose of the layers is to allow us to take into account relationships between the different input variables. If the inputs are orthogonal (i.e. no input value depends on another), then such layers are unnecessary. However, this is rarely the case. Thus, the hidden layers can group different combinations of the input variables together in a more meaningful manner.[StatSoft, 2004]

The way we get from the inputs to the outputs is through a series of *transfer functions*. Any non-input node will have a transfer function associated with it. This function takes n inputs and returns a scalar value which becomes the input for the next layer. Thus, a given layer will have m transfer functions, each with n inputs, and the next layer will have ℓ transfer functions each with m inputs, etc. These functions can vary widely depending on their use, however are commonly some form of step or sigmoid function that take as input a weighted sum of the n variables[1]. The one thing all transfer functions have in common is that they are differentiable. This is important because we often rely on this fact when training a network.[StatSoft, 2004]

2.2 Learning

Similar to people, a network learns through experience. A network is trained using data where both the input and the output is known. This data is called the *training set*. In weather networks, data from the previous two to five years is often used as the training set. By running these inputs through the network, and then adjusting the transition functions based on the disparity between the output and the known outcome, the network can learn to predict outcomes more correctly. This process is repeated on the same training set many times, usually until the network has converged to

[1]This is described in more detail in Section 2.2.1

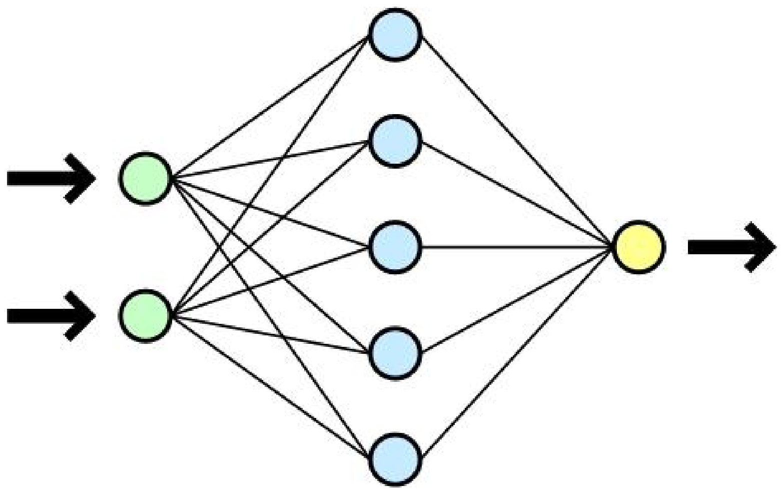
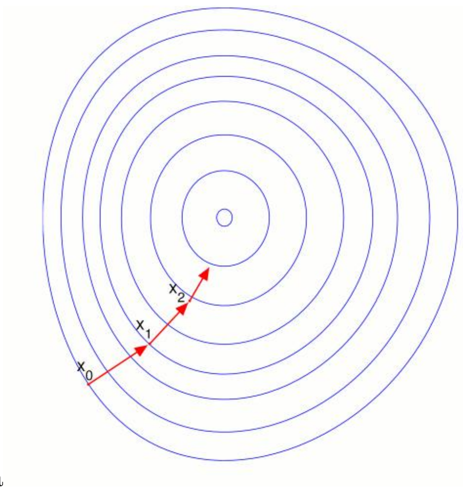


Figure 1: This is an example of a neural net with two inputs, one output, and one five node hidden layer. The transitions are represented by edges. *This figure taken from [Wikipeia, 2006]*

the point that the error is no larger than some preset *goal*, or until it has run though the training set some number of *epochs*. [Wikipeia, 2006]

There are many different ways in which a neural network is trained. Most use a general method called *backpropagation* where each output node calculates its error and adjusts its transfer function accordingly. Then each output node allocates blame for the source of its error to the nodes in the previous layer. These neurons then calculate their error based on said blame, and the process repeats, propagating backwards through the network. [Wikipeia, 2006] To frame this more mathematically, the training uses the gradient of the error in order to determine how to proceed towards an optimal configuration. [Grudic, 2006]

It is also important to note that there are two large distinctions between types of training methods. Some are *stochastic*, or on-line, and others are *standard* or batch algorithms. In a standard gradient descent algorithm, we adjust the transfer functions according the gradient as determined by whole of the data in the training set. Thus every datum is considered before the network is adjusted. In a stochastic algorithm the transfer functions are adjusted after every time a training datum is run. For large training sets a stochastic method is much more efficient. Additionally, the stochastic method can be made to approximate the batch method arbitrarily closely. Thus, stochastic algorithms tend to be the method of choice when it comes



ma

Figure 2: This depicts the level curves of the function, and the gradient at a given point. The gradient can then direct us towards local maxima or minima as appropriate. *Image taken from [Wikipeia, 2006]*

to network training. [Weisstein, 2006, Wikipeia, 2006, Grudic, 2006]

2.2.1 Gradient Descent

As mentioned earlier, the way in which a network learns is usually using a type of gradient descent algorithm. A general *gradient descent* algorithm approaches a local minimum by computing the gradient of the error of a given network, and using that to then minimize said error. This is depicted in figuregrad.[Weisstein, 2006] As mentioned above, one of the most common form of transition function is the logarithmic sigmoid function, which is defined as $\sigma(x) = \frac{1}{1+e^{-x}}$. This is a nice function both because it gives us a smoother step (see Figure 2.2.1), and because its derivative is simply $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. This means it is easy to work with in the context of a gradient descent algorithm.[Weisstein, 2006, Grudic, 2006]

During this discussion I use a standard gradient descent algorithm with a logarithmic sigmoid function as the transfer function. The input to the sigmoid function will simply be a weighted average of the inputs to the node. This example gives the basic idea of how a gradient descent algorithm works. This basic concept is generalizable to other gradient descent methods and differentiable transfer functions. [Grudic, 2006]

Let us consider a single sigmoid unit with n inputs as depicted in Fig-

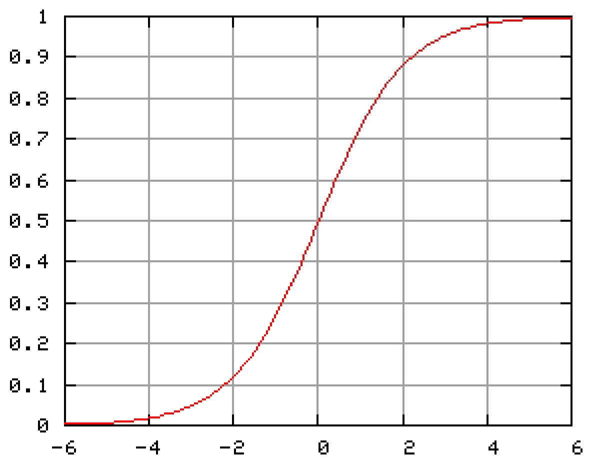


Figure 3: This is a generic sigmoid function. *Image taken from [Wikipeia, 2006]*

ure 2.2.1. These inputs can be represented as a vector x that depends on the input to the network as a whole. The output of the sigmoid unit is $\sigma(s)$ where $s = \Sigma_{i=1}^{n} w_i x_i$ and w_i is the weight assigned to the input entry x_i. For a given training example d in the set of all training data D, the error for this unit is $E_d = \frac{1}{2}(t_d - \sigma(x_d))^2$ where t_d is the expected output. Recall that the logarithmic sigmoid function has the nice property that $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.

Given this information, let us compute the gradient of the error sigmoid unit. Notice that each entry is a partial derivative that looks like:

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2.$$

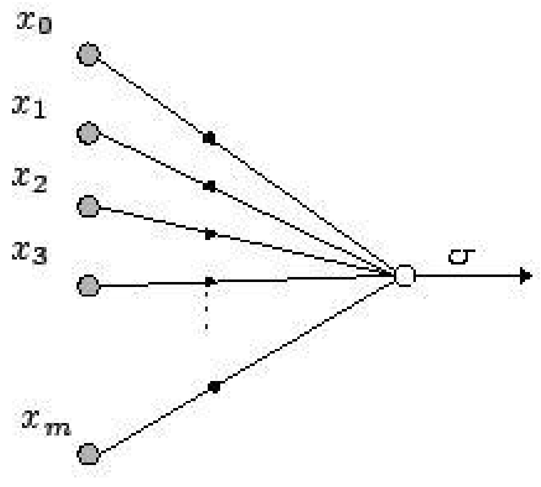


Figure 4: This is a depiction of a single sigmoid unit with m inputs. *Image taken from [Wikipeia, 2006]*

Let us solve this equation.

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d-\sigma(x_d))^2 \\
&= \frac{1}{2}\sum_{d\in D}2(t_d-\sigma(x_d))\frac{\partial}{\partial w_i}(t_d-\sigma(x_d)) \\
&= -\sum_{d\in D}(t_d-\sigma(x_d))\frac{\partial\sigma(x_d)}{\partial w_i} \\
&= -\sum_{d\in D}(t_d-\sigma(x_d))\frac{\partial\sigma(x_d)}{\partial x_d}\frac{\partial x_d}{\partial w_i} \\
&= -\sum_{d\in D}(t_d-\sigma(x_d))\sigma(x_d)(1-\sigma(x_d))x_{i,d}
\end{aligned}$$

This gives us the gradient of the error for this unit! [Grudic, 2006]

We can now adjust the weights that correspond to the inputs to our unit, where $\vec{w}=\vec{w}-r\Delta E$. Where r is the learning factor (discussed in Section 2.2.2). Because we have adjusted the weights in a way that will minimize the error, we have taken a step towards a minima of the error function, and thus will fit the data more accurately.

In order to generalize this to more complicated networks, we simply need a way of assigning the blame for the error and thus propagate the corrections back through the network. This is usually done by letting the error of a node i on the input x_d to be $\delta_i = \sum_{k \in K} w_{i,k}\delta_k$ where K is the set of all nodes that use the output of h as an input. Thus, the error is assigned to the nodes in a way that is proportional to their influence. Now, δ_i can replace $(t_d - \sigma(x_d))$ in the equation for the gradient of E, which allows us to adjust the weights of the inner layers.

2.2.2 Improvements on Training

When training, there is a *learning factor* $r < 1$ that describes the rate at which we allow a network to learn. This prevents massive fluctuation in the behavior of the network.[Bernacki and Wlodarczyk, 2006] One way to adjust the generic gradient descent algorithm is to use something called *adaptive learning* where the learning factor is varied throughout the course of training. Thus it can begin large while we initially converge towards some minima, and then get smaller as it fine tunes the transfer functions. [Matlab, 2005]

Another factor that is sometimes used is what is called *momentum.* When training with momentum, we adjust the weights and transition function based not only on the current gradient, but also on what our previous adjustment was - thus we have momentum in that direction! The idea behind this is that it allows us to take bigger steps towards the minima while still checking partway and correcting the direction if necessary. [Matlab, 2005]

Additionally, it is important to note that most algorithms employ some form of randomization. This done in order to nudge the network in some direction, which may allow it to find new (and better!) minima. This is often done by simply allowing the initial weights to be set randomly, thus causing the training to converge to different local minima. If there is considerable variation in results and no way to determine the best, then the average of such networks can be used as a predictive measure. [Grudic, 2006]

Finally, it is interesting to note that there are other error functions that can be defined and used when training a network. This completely depends on the applications, but they can do things such as penalize large weights, or train on slopes in addition to values. [Grudic, 2006]

2.2.3 Problems with Training

However, it is important to note that training is not always perfect. One of the main problems that can occur is when we *overfit* the data. As seen in Figure 2.2.3, we see how by matching our training data too closely we may end

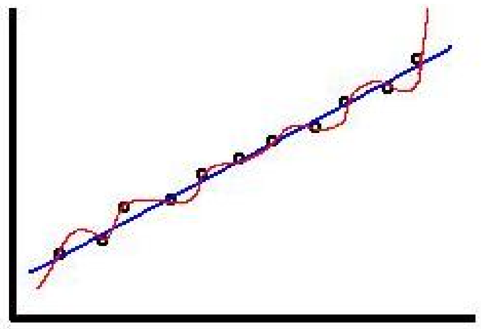


Figure 5: It is possible to achieve a fit for the data which minimizes the measurable error. However, we run the risk of overfitting. In this example a simple linear fit is more appropriate.

up with something that does not simulate our system as effectively.[Gurney, 1997]

3 The Project

The purpose of this project was to learn about neural networks, but more importantly to discuss the idea of how much information we really need to learn. I chose to build a neural network that would predict the probability of a student graduating. This is naturally a rather important concern, but was also one for which I had no access to data. Thus, my intent was to train the network by making a series of obvious observations, and seeing if this was enough for it to successfully predict the success rate of a given student.

3.1 Setup

The network takes as input the answers to eight questions on a scale from 0 to 1. For each question a 0 means no/never/none, and a 1 means yes/always/all. The questions are as follows;

- Are you a senior?
- Did you turn in an application for graduation?
- Do you go to class?
- Do you do your homework?

- Do you drink?
- Do you play video games?
- How many of this semester's credits do you need to graduate?
- How many of this semester's classes do you need to graduate?

Thus the network has 8 inputs, and a single output which predicts the probability of graduation.

The topology of the network contains a singly hidden layer of twelve nodes. While the rigorous data that I took was with this topology, it was not the only one I tried. I first constructed the naïve case with no hidden layer. As expected, this did not allow for much flexibility, and the predictions were rather crude. I ran the network with a hidden layer containing eight nodes. Again, as expected this did not add much to the predictive power because it does not allow for many correlations to be built between the input information. However, it is interesting to point out that I also tried the network with a hidden layer of fifteen nodes. Contrary to intuition, this did not seem better than the twelve node network, and actually the predictions varied even more widely. There seems to be a point for certain networks where there is too much redundancy and the network itself becomes more difficult to train. This seems to be what was happening here, so the twelve node network became the logical choice to use for the remainder of the project.

The network trains on all possible binary inputs. Because we must have outputs to train, I used common senses to decide which of our "students" should graduate. If either of the first two questions was a 0, they automatically do not graduate because they either not a senior, or did not apply for graduation, or both. Otherwise, if both of the last two questions are 0 they automatically graduate because they do not need any more credits or classes. Of the four remaining questions if two or more were answered "incorrectly" (i.e. I never go to class and am always drinking) then the student does not graduate. Otherwise, they are home free. Naturally this is a very large oversimplification of what it takes to graduate - but that is the point! The goal is to see if these crude assumptions can train a network that performs reasonably well.

3.2 Implementation

The code for this project was written in Matlab using the Neural Network Toolbox. From poking around I found that Matlab had the most techniques implemented, and allowed for a wide variety of networks. While the code is short, there are many ways with which to request the same sequence of events. This was the most efficient and relevant method I could find, and will outline it here because it is all you need to know to create and train a typical neural network in Matlab, and thus I think it is valuable information.

The network is created using a call to `newff(X, N, F, T)`. Given x input nodes, `X` is a $2 \times n$ matrix where each row corresponds to an input variable, and the columns describe the range for each one. Hence if `X = [0 5]`, then I have a one input network where the input will be in the range between 0 and 5. The next two inputs, `N` and `F`, are vectors whose length is the number of hidden layers plus one. Each entry in `N` is the number of nodes at that layer. The last entry is the number of outputs. Each entry in `F` is the type of transition function used in that layer. The final input `T` determines the type of training that will be used for the network.[Matlab, 2005]

The two types of transfer functions I used are `tansig` which is a hyperbolic tangent sigmoid function, and `purelin` which is a simple linear combination. The generic sigmoid function described earlier can be obtained using `logsig`.[Matlab, 2005] My reasoning behind my choices was that `tansig` is a bit flatter than the usual `logsig`, and this allows for a wider range of inputs to have some form of impact on the result (as opposed to something closer to a step function and is therefore "all or nothing"). My choice for `purelin` to determine the output was because I wanted a wide range of possible outputs. The sigmoid functions would tend to give outputs near the fringe of my 0-1 spectrum, while I wanted to be able to have people with say a 62 percent chance of graduation.

For testing purposes I use two different kinds of training schemes. The first is Matlab's `traingdx` which is a gradient descent algorithm that uses momentum and adaptive learning. The second training scheme is Matlab's `trainlm` which is a mixed algorithm that interpolates between the typical gradient descent and Gauss-Newton methods. This is the well known Levenberg-Marquardt algorithm that is often used in practice because of its fast rate of convergence.[Matlab, 2005, Wikipeia, 2006] In both cases the goal was 0.01, with a maximum of 500 epochs. The results for both are discussed in the following section.

3.3 Results

The verification data was taken from a total of 10 Harvey Mudd College students, professors, and alumni. While I ran many more trials, this was real data about real people, and thus the only appropriate way to measure whether or not the network was behaving in a reasonable manner.

Because of randomization, different training sessions can actually result in a different network because we may converge to different local minima. Because I was seeing a good deal of variation it became inadequate to simply find the prediction from a single network. Thus, I trained 100 networks on the same data for each algorithm, and averaged the predictive result of the verification data on each one. These averages along with the inputs and standard deviations are found in Appendix A.

An interesting thing to note is that the the `tainlm` algorithm almost always converged to the point where the error in the training data met the 0.01 goal. However, the `traindgx` algorithm rarely met this goal and most often would run the 500 epochs and reach an error of about 0.1. However, the average standard deviation for `traingdx` is 0.093 while the average standard deviation for `trainlm` is 0.156! Thus, while it seems that the `trainlm` algorithm is better, it in fact overfit the data and the more naive `traingdx` algorithm outperformed it (in terms of consistency) in practice.

The real measure, however, has to be in the *validation set* which tells us weather or not our predictions were valid. Because it is not the end of the semester I do not know for sure whether or not the people in my training sample are actually going to graduate. However, I can tell you that the four bottommost (one of whom is a senior!) are not planning to graduate this semester. Hopefully, everyone else will.

However, in general the predictions seem to be common-sense reasonable, though a bit harsh. I tend to believe that most people will graduate with the exception of extreme cases, but here we see more of a spread which indicates who is most likely to graduate. This, however, is mostly because the network is set up to give such a spread. As mentioned earlier, in order to be able to detect more about how the network was functioning the output used a simple linear combination instead of some kind of sigmoid function, thus reducing the polarity of the outcomes.

The main issue, however, is my lack of real data. Not only do I not know who is actually going to graduate, but I also only have data that people have self-report. As any psychologist will tell you, this is not necessarily an accurate portrayal (for better or worse!) of a subject's true inclinations. Thus, in order to truly determine if the basic observations are enough for a network to truly learn, we must use real data as opposed to that which is self-reported. However, if those results confirm what we see here, perhaps we do not actually need much concrete information to learn!

If the brain truly works in a similar manner, this would mean that we can learn by observing broad trends. This may not be the most accurate form of learning - but it would be enough to function in society. While showing that this sort of learning works in an artificial network proves nothing about biology, it still raises some interesting psychological questions about assumptions, bias, and prejudice that would be interesting to explore.

4 Future Work

Given more time I would have liked the opportunity to experiment more in depth with the different kinds of transition and learning techniques and develop a more formal rubric to determine weather or not the network is effective. While some of it will be verified on May 14th, I would like to have more concrete data with which to check how the network is behaving. Additionally, I was only able to get a sample of 10. It would be great to get many more people involved.

I think the most interesting correlation here goes back to the relationship between artificial and biological neural networks. If we can learn artificially by making basic logical assumptions, this seems to be more correlated to the way our minds would actually work. In some cases we think we have learned or can predict very complicated reactions, but how much data do we really have? There is no one measuring the error in our mental conclusions and then fine tuning them against a preset mold. Thus, in some ways, this method of training seems more accurate in terms of simulating the way in which people think. However, I do not know much about cognitive science, but it would be interesting to look into that area a bit more, and see if such theories are supported.

5 Acknowledgements

I would like to thank Professor Young for his fantastic Scientific Computing class which gave me the background and confidence to tackle this problem.

Additionally, I would like to thank Susanna Ricco for suggesting the book *Neural Networks for Pattern Recognition*, and providing me with the information on neural networks for weather prediction.

Finally, I would like to thank those who volunteered to provide me with test data as they bravely faced the possibility of a computer telling them they would not walk in our graduation ceremony.

A Results

This table depicts the data taken from ten members of the Harvey Mudd Community. This included seniors, non-seniors, alumni, and faculty. As you can see from the responses, there is a fair amount of variation in the lifestyle of the different people polled. The Average columns give the average output given by 100 networks trained by the given algorithm, and the corresponding StDev column gives the standard deviation thereoff.

As a reminder, the input answers the following questions on a 0 to 1 scale:

- Are you a senior?
- Did you turn in an application for graduation?
- Do you go to class?
- Do you do your homework?
- Do you drink?
- Do you play video games?
- How many of this semester's credits do you need to graduate?
- How many of this semester's classes do you need to graduate?

Input								Average - gdx	StDev - gdx	Average - lm	StDev - lm
1	1	1	1	0.25	0.5	0.2	0.4	0.97660	0.09354	1.19261	0.37964
1	1	0.7	0.9	1	0	0.1	0.1	0.89322	0.11002	0.99731	0.18461
1	1	0.5	0.9	0.5	0	0.2	0.8	0.72577	0.10109	0.86449	0.16018
1	1	0.95	0.9	0.7	0.8	0.4	0.5	0.55853	0.08120	0.57916	0.18511
1	1	0.8	0.8	0.7	0.3	0	1	0.55818	0.07194	0.63446	0.14480
1	1	1	0.9	1	1	0.5	1	0.22879	0.12640	0.10163	0.14319
1	1	0.5	0.4	1	0.1	1	1	0.09064	0.13316	0.03885	0.12523
0	0	1	1	0.25	0	1	1	0.02420	0.06150	-0.02620	0.07427
0	0	1	0	0.5	0	0	0	-0.02271	0.06621	0.01622	0.09965
0	0	0	0	1	0	0	0	-0.04295	0.08666	-0.00248	0.06134

As you can see, both algorithms agreed mostly if not exactly on the probability of graduation. More discussion of these results is given in Section 3.3.

B Code

I wrote a main function which created a network, trained it, and predicted the probability of you graduating given your input. I present the different sections of the code below. All work is original.

B.1 Will You Graduate?

```
% will_you_graduate.m    Elisa Celis
%
% This function predicts whether or not you will graduate!
%
% You must give it an input containing the answers to the folowing
% questions (if not yes-no, then a 1 indicates always, a 0 indicates never)
%
% are you a senior?
% did you submit an application for graduation?
% do you go to class?
% do you do your homework?
% do you drink?
% do you play video games?
% do you need your credits to graduate? (give percentage needed)
% do you need specific class to graduate?  (give percentage)
%

function pgrad = will_you_graduate(input)

l = length(input);

if l ~= 8
    'The lenght of the input array should be 8.'
else
% Create the training set (will train on all possible binary inputs)
P = training_set(l);
% Creat the appropriate responses to training set
% (note this is specific to our case)
```

```
T = training_output(l);

% Set up the neural network - the inputs are as follows:
%
% 1. the ranges for the input (one column per input variable).
% 2. the number of nodes in a layer (one entry per layer).
% 3. the types of transition function (one entry per layer).
% 4. the type of training to be used.
net = newff([0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1; 0 1], ...
             [12, 1], {'tansig', 'purelin'},'trainlm');

% Can replace with different training functions. Below are the ones I used:
%
% trainlm  - gradient descent & newton's method.
% traingdx - gradient descent w/ momentum and adaptive learning.


% Set up traning parameters, and train the network
net.trainParam.epochs = 500; net.trainParam.goal = 0.01; net =
train(net,P,T);

% Simulate the network on the given input
pgrad = sim(net,transpose(input));
end
```

B.2 Training Code

```
% Training_set.m    Elisa Celis
%
% This function gives the set we will train on - in this case, all binary
% possibilities given n variables.
%
% function P = training_set(n);
%

function P = training_set(n)

if n==1
    P = [0, 1];
else
```

```
    N = training_set(n-1);
    P = [zeros(1, 2^(n-1)) ones(1, 2^(n-1))
         N                 N           ];
end

% training_output.m    Elisa Celis
%
% This function gives the output we will train on. Not this function is
% very specific to our problem with 8 variables.
%
% function T = training_output(n);
%

function T = training_output(n)

P = training_set(n); T = ones(1, 2^n);

for (i = 1:2^n)
    % If you never go to class
    % OR never do your homework
    % OR always drink
    % OR always play video games
    %           .....you probably will not graduate.
    if ((1-P(3, i)) + (1-P(4, i)) + P(5, i) + P(6, i) >= 2)
        T(i) = 0;
    end
    % However, if you do not need any more credits
    % AND do not need any more classes
    %           .....you'll probably still graduate.
    if (P(7,i) == 0 && P(8, i) == 0)
        T(i) = 1;
    end
    % However, if you are not a senior
    % OR did not turn in your form
    %           .....you cannot graduate regardless.
    if (P(1,i) == 0 || P(2, i) == 0)
        T(i) = 0;
    end
end
```

B.3 Testing Code

```
% grad_ttest.m    Elisa Celis
%
% This function runs several trials of the will_you_graduate program on a
% single input in order to determine the consitency of a given response.
%

function [average, stdev] = grad_test(input)

num = 25; data = zeros(1, num); sum = 0.0;

for i = 1:num
    data(i) = will_you_graduate(input);
    sum = sum + data(i);
end

average = sum/num

stdev = 0; for i = 1:num
    stdev = stdev + (average - data(i))^2;
end

stdev = sqrt(stdev/num)
```

References

[Bernacki and Wlodarczyk, 2006] Bernacki, M. and Wlodarczyk, P. (2006). Backpropagation in neural networks. Available electronically at galaxy.agh.edu.pl/ vlsi/AI/backp_t_en/backprop.html.

[Bishop, 1995] Bishop, C. (1995). *Neural Networks for Pattern Recognition.* Oxford University Press.

[Grudic, 2006] Grudic, G. (2006). Neural networks. Available electronically at www.cs.colorado.edu/ grudic.

[Gurney, 1997] Gurney, K. (1997). Neural nets. Available electronically at www.shef.ac.uk/psychology/gurney/notes/index.html.

[Matlab, 2005] Matlab (2005). Helpfiles.

[StatSoft, 2004] StatSoft (2004). Available electronically at www.statsoft.com/textbook/stneunet.html.

[Weisstein, 2006] Weisstein, E. (2006). Mathworld. Available electronically at mathwolrd.wolfram.com.

[Wikipeia, 2006] Wikipeia (2006). Available electronically at en.wikipedia.org.