Taking Initiative

Bobby Anguelov's Tech Blog

Basic Neural Network Tutorial : C++ Implementation and Source Code

APRIL 23, 2008 178 COMMENTS (HTTP://TAKINGINITIATIVE.WORDPRESS.COM/2008/04/23/BASIC-NEURAL-NETWORK-TUTORIAL-C-IMPLEMENTATION-AND-SOURCE-CODE/#COMMENTS)

i
36 Votes

So I've now finished the first version of my second neural network tutorial covering the implementation and training of a neural network. I noticed mistakes and better ways of phrasing things in the first tutorial (http://takinginitiative.wordpress.com/2008/04/03/basic-neural-network-tutorial-theory/) (thanks for the comments guys) and rewrote large sections. This will probably occur with this tutorial in the coming week so please bear with me. I'm pretty overloaded with work and assignments so I haven't been able to dedicate as much time as I would have liked to this tutorial, even so I feel its rather complete and any gaps will be filled in by my source code.

Introduction & Implementation

Okay so how do we implement our neural network? I'm not going to cover every aspect in great detail since you can just look at my source code. I'm just going to go over the very basic architecture. So what critical data storage do we need for our neural network?

- Our neuron values
- Our weights
- Our weight changes
- Our error gradients

Now I've seen various implementations and wait for it… here comes an OO rant: I don't understand why people feel the need to encapsulate everything in classes. The most common implementation of a neural network I've come across is the one where each neuron is modeled as an object and the neural network stores all these objects. Now explain to me that advantages of such an approach? All you're doing is wasting memory and code space. There is no sane reason to do this apart from trying to be super OO.

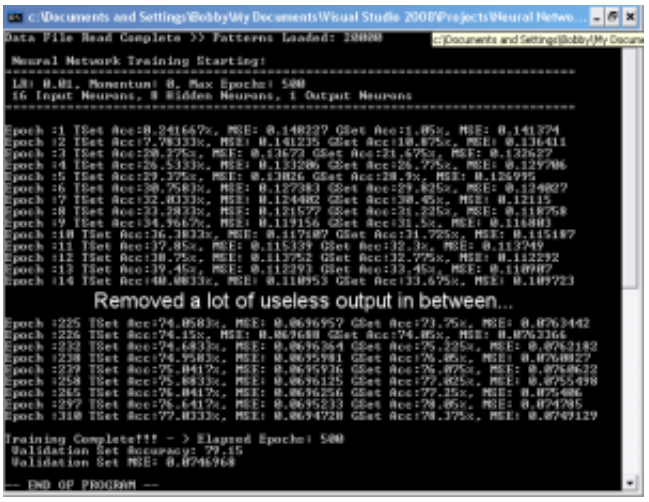
The other common approach is to model each layer as an object? Again what the hell are people smoking? I guess i can blame this on idiot professors that know the theory but can't code their way out of a paper bag. Like for example my father is a math's professor and is absolutely brilliant in regards to developing algorithms and just seeing elegant solutions to problems but he cannot write code at all. I've had to convert his original discrete pulse transform Matlab code into c++, and in the process of converting/optimizing it, I ended up developing a whole new approach and technique that wasnt possible to achieve directly from the maths – the so called fast discrete pulse transform.

I also tend to be a bit of a perfectionist and am firm believer in occam's razor (well a modified version) – "the simplest solution is usually the best solution". Okay enough ranting – the point is dont use classes when you dont need to! Useless encapsulation is a terrible terrible mistake, one that most college students make since its drilled into them during their years of study!

So below is how I structured my neural network and afaik it's as efficient as possible. If anyone can further optimize my implementation please do! I'd love to see your techniques (I love learning how to make code faster).

A neural network is a very simple thing and so must be implemented very simply. All the above values are just numbers so why can't I just use multi-dimensional arrays. The short answer is that I can and I do. Storing each of those sets of number in multi-dimensional arrays is the most efficient and simplest way of doing it with the added benefit that if you use the index variables i/j/k for input/hidden/output loops respectively, then your code will almost exactly resemble the formulas in the theory tutorial and in this one.

So now that the major architecture design decision has been mentioned, here's what the end result looks like:



(http://takinginitiative.files.wordpress.com/2008/04/training.png)

Initialization of Neural Network

Okay so how do we initialize our neural network? Well again its super simple, we set all the values of the inputs, deltas and error gradients to 0. What about the weights?

The weights between the layers be have to randomly initialized. They are usually set to small values often between in the range of [-0.5, 0.5], but there are many other initialization strategies available for example, we can normally distribute the weights over the number of inputs but changing the range to: [-2.4/n, 2.4/n] where n is the number of input neurons.

Technically you can set the weights to a fixed value since the weight updates will correct the weights eventually but this will be terribly inefficient. The whole point of setting the initialization range is to reduce the number of epochs required for training. Using weights closer to the required ones will obviously need less changes than weights that differ greatly.

If for example you have an implementation where the data for training coming in is very similar, for example an image recognition system that takes in various silhouettes and their classification and trains using that data. After a few training sessions with different data you can observe the range in which the final weights are found and initialize the weights randomly within that range, this technique will further reduce the number of epochs required to train the network.

Training Data and Training Problem

So we've created our neural network but what good does that do without data to train it on? I've selected a letter classification problem as an example. I downloaded a letter recognition dataset from the UCI machine learning repository. (http://archive.ics.uci.edu/ml/datasets/Letter+Recognition (http://archive.ics.uci.edu/ml/datasets/Letter+Recognition)).

I'll be using this test data to explain the basics of training your neural network. The first thing is to format the dataset, in our case we had a list of 16 attributes and the letter those attribute represent. To make life simple the first problem I selected was to distinguish the letter 'a' from the rest. So I simply replaced the letter with a 1 for an "a" and a 0 for everything else, this effectively reduces the output to a single Boolean value.

Another problem i've included is a vowel regonition problem, where the output neuron is 1 for (a,e,i,o,u) and 0 for all other letters. This is a more complex problem and as such the accuracy achieved will be much lower.

So you can already see that we have 16 inputs (the attributes) and a single output, so most of our NN architecture is done, how many hidden neurons do we need? Well we don't know, we have to play around with different numbers of hidden neurons till we get a good result.

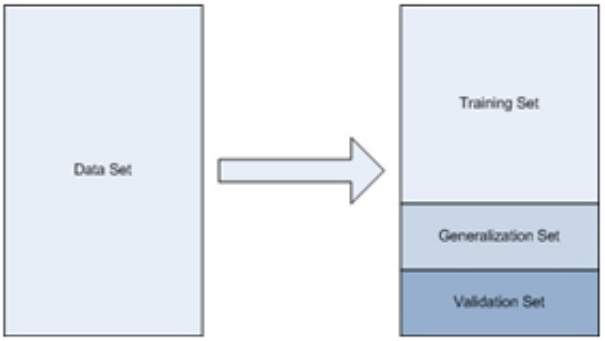
I've also noticed that there seems to be an accuracy ceiling for every NN's architecture. A good method to find out how many hidden neurons you need would be to: train the network several times (10+) and see what the average accuracy is at the end and then use this accuracy as a measure to compare different architectures.

Something that was asked in the comments section was what is the range for the input values, well the sigmoid functions is continuous over the range (-inf, inf) so any values will work but it a good idea to keep the values within the active region (region of greatest change / steepest gradient) of your activation function.

Okay moving along, so we've "formatted" our data the next step is to split it up so that we can train our neural network with it.

The Training Data Sets

So in our case we have this massive data set of 20000 patterns (entries), we can't just stick all of the data into our network since the network will learn that data and we have no way of checking how well the network will do with unseen data. This problem is referred to as over-fitting, basically the network starts remembering the input data and will fail to correctly handle unseen data. I don't want to go into too much detail here as this is something of an advanced research topic. Once you are familiar with neural networks you can read up on over-fitting on your own.



(http://takinginitiative.files.wordpress.com/2008/04/dataset1.png)

So we don't want the network to memorize the input data, so obviously we'll have to separate our training set. Classically we split the data set into three parts: the training data, the generalization data and the validation data. It is also often recommended to shuffle the initial dataset before splitting it to ensure that your data sets are different each time.

- The training data is what we used to train the network and update the weights with so it must be the largest chunk.
- Generalization data is data that we'll run through the NN at the end of each epoch to see how well the network manages to handle unseen data.
- The validation data will be run though the neural network once training has completed (i.e. The stopping conditions have been met), this gives us our final validation error.

The classic split of the dataset is 60%, 20%, 20% for the training, generalization and validation data sets respectively. Let's ignore the validation set for now; the training set is used to train the network, so if you can remember from the last tutorial for every piece of data (pattern) in the training set the following occurs:

- Feed pattern through NN
- Check errors and calculate error gradients
- Update weights according to error gradients (back propagation)

Once we have processed all the patterns in the training data set, the process begins again from the start. Each run through of all the patterns in the training set is called an epoch. This brings us to the question how do we decide how many epochs to run?

Stopping Conditions

There are several measures used to decide when to stop training. I'm going to list the various measures, what they mean and how to calculate them.

- **Maximum Epochs Reached** – The first measure is really easy, all it means is that the NN will stop once a set number of epochs have elapsed.

- **Training Set Accuracy** – This is the number of correctly classified patterns over the total number of patterns in the training set for an epoch. Obviously the higher the accuracy the better. But remember that this is the accuracy on previously seen patterns so you can use this alone to stop your training.
- **Generalization Set Accuracy** – This is the number of correctly classified patterns over the total number of patterns in the generalization set for an epoch. This gets calculated at the end of an epoch once all the weight changes have been completed. This represents the accuracy of the network in dealing with unseen patterns. Again you can't use this measure alone since this could have a much higher accuracy that the training set error.
- **Training Set Mean Squared Error (MSE)** – this is the average of the sum of the squared errors (desired – actual) for each pattern in the training set. This gives a more detailed measure of the current networks accuracy, the smaller the MSE the better the network performs.
- **Generalization Set Mean Squared Error (MSE)** – this is the average of the sum of the squared errors (desired – actual) for each pattern in the generalization set.

$$MSE = \frac{\sum_{i=0}^{n}(desired\ value\ i - actual\ value\ i)^2}{n}$$

$$where\ n = number\ of\ patterns\ in\ set$$

(http://takinginitiative.files.wordpress.com/2008/04/mse.png)

So now we have these measures so how do we stop the network, well what I use is either the training and generalization accuracies or MSEs in addition to a maximum epoch. So I'll stop once both my training and generalization set accuracies are above some value or the MSE's are below some value. This way you cover all your bases.

Just remember that while your accuracies might be the same over several epochs the MSE may have changed, the MSE is a measure with a much higher resolution and often is a better choice to use for stopping conditions.

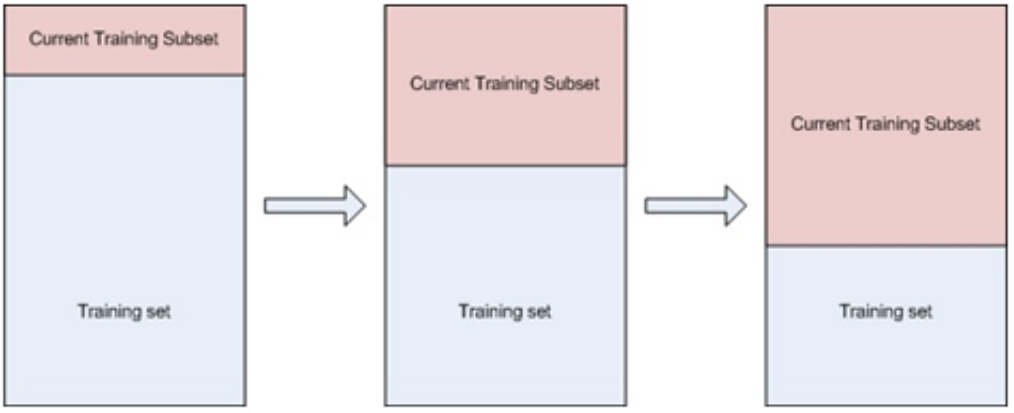
Now we come to an important point that people often overlook, every time you train the network the data sets are different (if you shuffled the initial dataset as recommended earlier) and the weights are random, so obviously your results will differ each time you train the network. Usually the only difference is the number of epochs required to reach the NN's accuracy ceiling.

Advanced Data Partitioning Methods

There are several advanced concept I want to deal with here. Firstly when dealing with large datasets, pushing through a massive training dataset through the network may not be the best solution. Since the NN will have to process all the patterns in the training set over and over, and there could be tens of thousands of patterns, you can imagine how slow this will be.

So there are several techniques developed to partition the training set to provide better performance in regards to both time taken to train and accuracy. I'm only going to cover two basic ones that I've implemented into my data set reader found below.

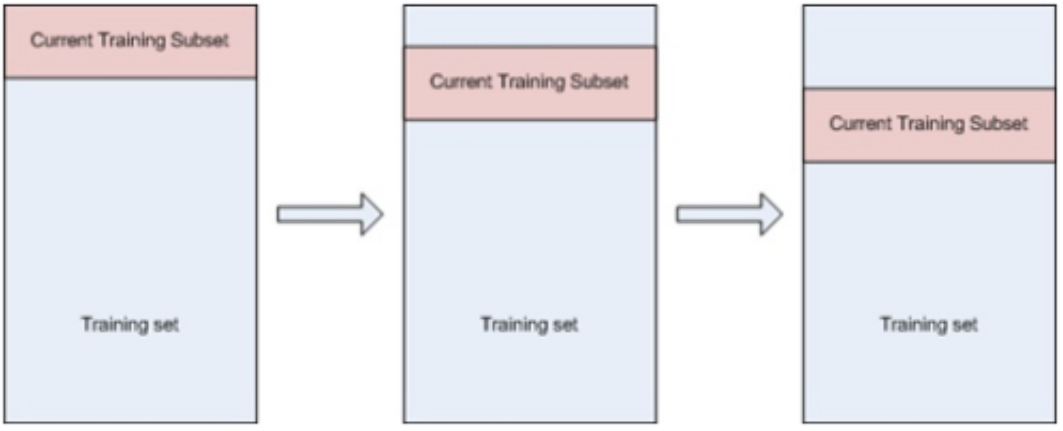
Growing Dataset:



(http://takinginitiative.files.wordpress.com/2008/04/growingdataset.png)

This approach involves training with a growing subset of the training set; this subset will increase with a fixed percentage each time training has completed (stopping conditions for the neural network have been met). This carries on until the training data set contains all the training patterns.

Windowed Data set:



This approach creates a window of fixed size and moves it across the dataset. This move occurs once training has completed for that window. You'll specify the window size and the step size. This method stops once the window has covered the entire training data set.

Momentum

Momentum is a technique used to speed up the training of a BPN. Remember that the weight update is a move along the gradient of the activation function in the direction specified by the output error. Momentum is just that, it basically keeps you moving in the direction of the previous step. This also has the added bonus that when you change direction you don't immediately jump in the opposite direction but your initial step is a small one. Basically if you overshoot you've missed your ideal point and you don't wish to overshoot it again and so momentum helps to prevent that.

The only change to the implementation is in regards to the weight updates, remember from part one that the weights updates were as follows:

$$w_{ij} = w_{ij} + \Delta w_{ij} \; and \; w_{jk} = w_{jk} + \Delta w_{jk}$$

where $\Delta w_{ij}(t) = \alpha . inputNeuron_i . \delta_j \; and \; \Delta w_{jk}(t) = \alpha . hiddenNeuron_j . \delta_k$

$\alpha - learning\ rate$

$\delta - error\ gradient$

Those weight updates now become:

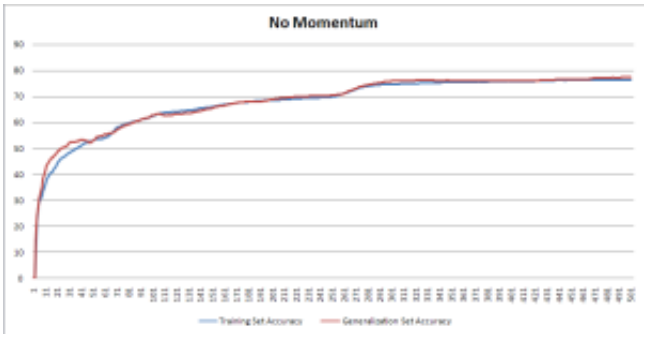
$$\Delta w_{ij}(t) = \alpha . inputNeuron_i . \delta_j + \beta . \Delta w_{ij}(t-1)$$

$$\Delta w_{jk}(t) = \alpha . hiddenNeuron_j . \delta_k + \beta . \Delta w_{jk}(t-1)$$

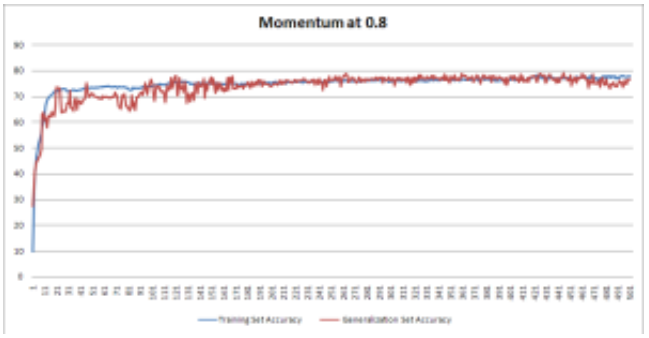
$where\ \beta - momentum\ constant$

(http://takinginitiative.files.wordpress.com/2008/04/momentum.png)

Momentum is usually set to a high value between 0 and 1. You'll notice that with momentum set to 0 the weight updates are identical to original ones. The effect of this momentum term is shown below for our training problem with the learning rate set to 0.01 and the momentum set to 0.8.



(http://takinginitiative.files.wordpress.com/2008/04/nomomentumgraph.png)



(http://takinginitiative.files.wordpress.com/2008/04/momentumgraph.png)

As you can see from the graphs the BPN will usually converge a lot faster with momentum added than without it and it also has the added benefit of allowing the back-propagation to avoid local minima's in the search space and to traverse areas where the error space doesn't change. This is evident from all the fluctuations seen in the graph.

The momentum formula's shown above are also known as **the generalized delta rule.**

Batch / Online learning

The last thing I want to go over is the difference between batch learning and stochastic or on-line learning.
Stochastic learning occurs when the neuron weights are updated after each individual piece of data is passed through the system. The FFNN therefore changes with every piece of data and is in a constant state of change during training. This is the way we've been doing it up to now.

Batch Learning on the other hand stores each neuron weight change when it occurs, and only at the end of each epoch does it update the weights with the net change over the training set. This means the neural network will only update once at the end of each epoch. Implementation wise this change is minor as all you need to do is just store the weight changes (the delta w values) and just update the weights at the end of the epoch.

The effects of this I'll leave up to you guys to discover for yourselves. I can't spoon feed you everything.

Source Code and Implementation

In the zip file below you'll find the complete visual studio 2k8 project for the following:

- My neural network class (has CSV logging, and has supports for momentum and batch/stochastic learning)

- My CSV data reader class (loads CSV files, has several data partitioning approaches built in)
- The test data files for the above problem
- A test implementation of the above training problem

My code is highly commented and very easy to read so any questions I haven't answered should hopefully be answered by the code. I hope this has helped you guys understand neural networks.

Neural Network Project and C++ Source code : nnImplementation.zip (http://dl.dropbox.com/u/18209754/Blog/nnImplementation.zip)
My First Neural Network Tutorial : Theory of a Neural Network (http://takinginitiative.wordpress.com/2008/04/03/basic-neural-network-tutorial-theory/)

UPDATE:

I've updated the source code to use a better archictural design, its cleaner, simpler and easier to understand: nnImplementationV2.zip (http://dl.dropbox.com/u/18209754/Blog/nnImplementationV2.zip)

FILED UNDER ARTIFICIAL INTELLIGENCE, NEURAL NETWORKS TAGGED WITH ARTIFICIAL INTELLIGENCE, BACK-PROPAGATION, NEURAL NETWORK

About Bobby
I'm a programmer at Ubisoft. My work interests include Animation and Artificial Intelligence. All opinions are my own!

178 Responses to *Basic Neural Network Tutorial : C++ Implementation and Source Code*