**Supervised learning in artificial neural networks**

*A brief introduction*

Borgersen Gustav  
Mälardalens Högskola  
Brahegatan 4 A  
722 16 Västerås  
+4670 400 23 99  
gbn05001@student.mdh.se

Karlsson Linus  
Mälardalens Högskola  
Brahegatan 4 A  
722 16 Västerås  
+4673 631 66 08  
lkn05007@student.mdh.se

**ABSTRACT**

This paper will present two different approaches to supervised learning in artificial neural networks. The back propagation algorithm and Genetic algorithms. We will also give a brief introduction to multi layered perceptrons, neural networks. The main part focuses on how, why and when back propagation and genetic algorithms should be used and a brief introduction as to how they both work. The article will end with a summary, conclusion and some points for discussion on the subject.

1. INTRODUCTION

Artificial Neural Networks (ANNs) is a machine learning technique which is inspired by the function of a human brain. The human brain consists of billions of neurons, connected with synapses in a very complex network. In the same fashion we build the artificial neural network, where perceptrons are connected in a network from input to output layer. The connections are weighted and the job for each perceptron is to calculate some output value depending on the weighted input values. We will discuss the functionality of each perceptron and the connections later in this paper.

ANNs is very well suited for approximation of unknown complex functions (and their derivatives), pattern recognition, and much more. The problem is to find a learning algorithm that in a fast and reliable way trains the network to function in a desirable way. Training in this context is when we update the weights and tries to find values that give us good results. For more general information on artificial neural networks see Rumelhart [7].

There are different ways to train a network depending on situation and application. Unsupervised learning is used when we don’t have a good way to know if a certain input should map to some distinct output. For example, in a chess game you can not know if a particular move will result in victory, it depends on the whole series of moves. In this situation we use unsupervised learning and let the network train itself by playing a lot of chess games.

In supervised learning, however, we have a set of training data, the set contains of some input examples connected with the correct output, and the output value is often referred to as the target value. There is also a problem with over fitting where a network is trained to specifically to the training set. Some of these concepts, concerning supervised learning, will be explained more in depth later on the paper.

In this paper we will focus on two main techniques to train artificial neural networks in a supervised way, the classic back propagation and genetic algorithms.

Back propagation is a method where we with help of derivates and the mean square error of the output makes a gradient search to find new values to the weights. This algorithm is one of the oldest and most used, and we will describe its functionality, advantages and drawbacks later in this paper.

Genetic algorithms are another approach when training artificial neural networks. This method is inspired by the evolution theory. In this algorithm we first build a population that consists of solutions to the apparent problem. Then 'survival of the fittest' is applied together with some random mutations and, as in nature, the strongest and best solution is found after some generations. This will also be explained further in this paper.

2. Supervised learning methods

The perceptron

The perceptron, or neuron, that is used in today’s artificial neural networks was first conceived of by Rosenblatt in the late 1950s [3]. The perception takes an input vector consisting of one to n inputs and then presents an appropriate output. The way it does this is by first calculating a weighted sum of the inputs, adds an bias if there is a bias, and then passes this result through a non-linearity. The weights are the most important part of the perceptron and they are updated so that an input gives the desired output when passed through the non-linearity. Rosenblatt used a hard linearity for this purpose but today it is more common to use the sigmoid non-linearity [2].

\[ f_{sy} = \frac{1}{1+e^{-\beta y}} \]  

This function is continuous and therefore differentiable, which is important if we want to use the back propagation learning algorithm. Also the fact that its output varies between 0 and 1 as its input goes from $-\infty$ to $\infty$ is useful if we want to be able to use our output as a probability value.
The perceptron can be used to express a number of logical functions including AND, OR and COMPLEMENT. However it cannot be made to function as an XOR. To implement more complicated functions, such as XOR, we use multilayered networks of perceptrons which will be discussed in more depth in the next part.

**The Multi Layer Perceptron (MLP)**

If we want to use a neural net to explain more complex logical operators, for example the XOR, we must use a multilayer network. These networks work by letting the output of one layer act as the input for another layer [1]. Usually when we speak of multilayered nets we do not count the input as a layer. And sometimes we do not count the output either. The important thing is the numbers of hidden layers in the net. For example a net consisting of an input layer which outputs to a layer that outputs directly to our output layer is said to be a 2 layered net or a net with one hidden layer. See figure 1.

![A MLP with one hidden layer.](image)

A 2-layered perceptron is able to represent any logic function, since any arbitrary logical function can be described by two layers consisting of ANDs and ORs [4].

**Supervised learning**

Supervised learning is used when we have a set of training data. This training data consists of some input data that is connected with some correct output values. The output values are often referred to as target values. This training data is used by learning algorithms like back propagation or genetic algorithms that we will look into later. Back propagation uses the target values to calculate the mean square error of the artificial neural network and genetic algorithms use target values when calculating the fitness levels of an individual in a population.

But the goal of the learning algorithm is not to create a neural network that outputs perfect values for the training data; the mission is to give good values for input data that is from the real world and not from the training set. When we train the network to hard against the training set we tend to learn the ‘noise’ in the measured data and not the underlying structure, we don’t see the ‘whole picture’, this is called overfitting. If we split our training data into two pieces and uses one for training and the other part for validation we will see that both the error for the training data and the error of the validation data will decrease at first, then at some point we starts to over fit the network and we see that the training data still gets better values but the validation error get bigger again, this is the point of overfitting and this is when we should stop the training, see figure 2.

![Both the training and validation sets errors decreases at first, until we reach the point of overfitting.](image)

Overfitting also occurs easily when the network is too large and flexible for the task at hand, therefore it could be avoided by trying different network architectures. To find a good size and architecture for an artificial neural network is not an easy progress, and there are numerous ways of attacking the problem. Further reading on the subject can be found in ‘Intelligent Systems’ [10].

**Back Propagation**

One way that we could train our neural network is by use of the back propagation algorithm. A method which relies on the fact that the linearity we choose for our perceptrons is continuous and differentiable. So now let us take another look at equation number 1 and how we can calculate an appropriate adjustment to our weight by using the current error in output and the derivate of the function, put them together and perform a gradient search [2]:

\[ w_{l,j} = w_{l,j} + \eta (e_{l+1,j} f'(x)) \]

Where \( w_{l,j} \) is the output of a perceptron, \( j \) in the layer \( l \) and \( f \) is the sigmoid non-linearity. \( w_{l,j} \) is the weight with which the output of the perceptron \( i \) in layer \( l-1 \) connects to the chosen perceptron \( j \) in this layer. \( e_{l+1,j} \) is the output that is weighted according to \( w \); this is all summed up and then ran through the sigmoid. If we are to use this in our back propagation algorithm the function must be differentiable. Luckily this function has a simple derivate:

\[ f'(x) = \frac{dx}{dx} = f(x) f'(x) = f(x)(1-f(x)) \]

As a common learning algorithm we use a gradient search algorithm which updates the weights in search of a minimum in the sum-of-squared-error function:
The number of nodes in the output-layer is defined as NL and dqxp is the desired response given for the chosen training example. To minimize this error we must update the weights and this is done according to:

\[
wl,j.k+1=wl,j.k+\mu\frac{\partial J}{\partial w,l.j}(w)\frac{\partial J}{\partial w,l.j}(k)
\]  

(6)

This, in turn, equals:

\[
w,l.j.k=1P\frac{\partial J}{\partial w,l.j}(wl,j.kw)
\]  

(7)

Where the learning rate of the system is represented by the symbol \( \mu \), and is a positive, often small, constant. To implement this as a learning algorithm we must now find a way to express a partial derivative to \( J \) with respect to each individual weight in the whole network. This is done using the chain rule for any arbitrary weight in layer 1 as follows:

\[
\frac{\partial J}{\partial w,l.j}(i,j)=\frac{\partial J}{\partial w,l.j}(i,j)
\]  

(8)

The last term in this equation can be rewritten according to equation 2, and then rewritten to assist substitution from equation 3 like this:

\[
\frac{\partial J}{\partial u,l.j}(i,j)=\frac{\partial J}{\partial w,l.j}(i,j)
\]  

(9)

And now for the substitution from equation 3:

\[
\frac{\partial J}{\partial w,l.j}(i,j)=\frac{\partial J}{\partial u,l.j}(i,j)
\]  

(10)

Now it is time to rewrite equation number 8:

\[
\frac{\partial J}{\partial w,l.j}(i,j)=\frac{\partial J}{\partial u,l.j}(i,j)
\]  

(11)

Now we can measure the sensitivity of the final error with respect to the output of the perceptron \( u,l.j \) by looking at the term \( \frac{\partial J}{\partial u,l.j} \). The perceptron also influences the sensitivity of preceding layers, therefore we can write the sensitivity as a function of the perceptrons in the next highest layer as follows:

\[
\frac{\partial J}{\partial u,l.j}(i,j)=m=1NL+1\frac{\partial J}{\partial p,l.w}(i,m\cdot u,l.j)
\]  

=\( m=1NL+1\frac{\partial J}{\partial p,l.w}(i,m\cdot u,l.j) \)

\[
=\frac{\partial J}{\partial p,l.w}(i,m\cdot u,l.j)
\]  

(12)

\[
=\frac{\partial J}{\partial p,l.w}(i,m\cdot u,l.j)
\]  

(13)

We can continue these calculations for higher and higher layers until we reach the output layer. The output layer is where we can derive the sensitivities of the perceptrons in the last non-output layer and then we just keep backing up until we reach the input layer again. We get the sensitivity of the last non-input layer by use of this formula which is derived from equation 5:

\[
Jw=p=1PJp\]  

(4)

Where \( P \) is the number of training patterns and \( Jpw \)is the total error for that particular pattern, \( p \). \( Jpw \) can be expressed like this:

\[
Jpw=12q=1NL\sum_{xp}dqxp2
\]  

(5)

By use of this equation we can now calculate more appropriate weights from the input layer and all the way to the output layer. This however is a bit confusing since we are supposed to learn by knowing the current error in the output. Actually our calculations go from input to output but once we get there we adjust our weights all the way back to the input once again. In combination with equation 7, equation 11-14 can now be iterated for a gradient search, tough in most cases equation 7 is replaced by the approximation shown in equation 15 before iteration is started.

\[
w,l.j,k+1=wl,j.k-\mu\frac{\partial J}{\partial w,l.j}(wl,j.kw)
\]  

(15)

Starting out the weights are most often set to some low random value, or in some situations we might have some prior knowledge that allows us to use better starting values. In most situations we do use random weights even if we are biased, since this might help us escape local minimums. The weights are updated in accordance with the learning rates that we also usually want to keep to a low value. Sometimes the learning rate is not a constant number but changes as we go along. The closer to a minimum we come the smaller we want it to be. This however is very hard to implement since we do not know where we will find any minimums. A constant low learning rate might lower the efficiency of the search but it will find a minimum for the error curve.

When we have come this far, all we have to do is repeat the last for steps until our condition for the solution is met. These conditions can vary depending on what type of task the network is constructed for and also the cost for failure and rewards for success. Sometimes we let the net back propagation algorithm run until the total error goes under a certain threshold value, other times we might have to run it until we find a minimum for the error, which would be when the gradient is zero. Also in some situations we might run it for a set number of iterations, this could come in handy when comparing different algorithms to see which needs the fewest number of iterations for a particular system.

One off the biggest drawbacks with back propagation is that it has no inbuilt way of avoiding local minimums. This can sometimes be ignored but sometimes it is a critical error. As with any pure gradient search the hill-climbing problem gives rise to a series of problems. There are different solutions to the problems, varying in complexity, for example repeating the search with new randomized weights to avoid heading towards the same bad result again. There are also other ways to do this without increasing the number of iterations. Or we could combine the use of extra momentum terms to the search which also uses the last gradient when updating weights [10].

Back propagation is a strong algorithm for training a neural network of moderate size or preferably smaller. It can of course be used for bigger nets but then the performance rate goes down a bit. As long as you are aware of the problem with local minimums and take steps to avoid it, back propagation should yield quick and good results. Back propagation is also fairly easy to implement in a number of popular programming languages such as C, C++ and Java. This makes it a highly used way to train and implement networks in settings that allows for the use of supervised learning.
Genetic Algorithms

Another way to do supervised learning in artificial neural networks is to use genetic algorithms, GA. To understand how this works we must first look at how genetic algorithms work in general before we look in to their application in multi layered perceptrons.

Genetic algorithms is a way of coming up with a suitable hypothesis to a problem by first randomly create a large number of different hypothesis and then crossbreed these in a manner similar to evolution. Often this is done by representing the hypothesis with a vector of integers, alternatively a binary vector. Sometimes it can also be wise to use a combination of arbitrary integers and binary ones.

When we have decided on which type of vector we will use and how to interpret it, in other words what each unit in it represents, we create a large and randomly generated population. When this is done we must now try to measure every individual hypothesis’ fitness rating by devising some test that gives a good evaluation for each individual. In some cases this might be done by testing the hypothesis in its future application with a well-chosen number of known training sets. For a neural network we simply run some of our training data and measure actual output versus desired output and from that data we devise a fitness rating.

When all individuals in our population have been measured and assigned a fitness value it is time for mating. This is often done by choosing individuals in a stochastic manner where each individual’s chance of being chosen is equal to its relative fitness value:

\[
P_{\text{chosen for reproduction}} = \frac{\text{Individual fitness}}{\text{Total fitness}}
\]  

Total fitness is the sum of the individual fitness for each individual in the population. Now we set up something like a wheel of fortune where each individual has a slice equal to its probability for reproduction. We choose two individuals using this method then revise the wheel and keep on choosing until a predetermined number of individuals have been chosen for mating or some other condition has been met. Now we come to the actual paring up and mating of the individuals and this can be done in a number of ways.

First we can have any number of cross-over points which is the number of places in which we break up the individual’s string to interchange parts of it with its chosen mate [5]. For this we can use a binary key string that tells us where and how to interchange parts of our hypothesis vectors:

In figure 3 we can see how the binary cross-over key operates, and how, given the same parents, different designs of the interchange key have an impact on the offspring. Normally we chose this key randomly for each coupling but sometimes we can have a predetermined key that is always used. This can also be applied to a non-binary crossover with good results.

Now that we have two new individuals there is only one step left before we reinsert them in our population and that is mutation. Far from every new individual is mutated but with some probability we mutate offspring to see if a wholly new trait might be an asset for its fitness-value. Mutations are done by choosing one value, or some other number of values, and randomize them, or in the event of a binary value simply change it.

Now that our new individuals are complete we put them back in our population and restart the process again from scratch. This process is continued until the population’s best individual is a hypothesis good enough to satisfy our particular application.

Implementing GA in a network

To implement this in neural networks we simply set up our table of weights as a hypothesis, creating vectors where each number in the vector corresponds to a certain weight, as mentioned above.

In the vector we first add the weights from the first perceptron, then the next perceptron and so on, on till all weights are represented in the vector.

\[\text{Vweights} = Wp1, Wp2, Wp3 \ldots Wpn\]

Then we follow the steps of the algorithm and create a set of randomized sets of weights. These weights should, unlike when using back propagation, be totally random, so as to not point us towards a local error-minimum. It has been shown that the lack of diversity in a population can lead to unwanted results, such as local error minimums far from the optimal solution, and therefore mislead the GA in the wrong direction [6].

For a fitness function we can choose among a large number of different functions depending on our particular problem. For example, running the net with our individual set of weights and then compares the mean-squared-error, or just uses an arbitrary squared error from the results as a fitness value.

The benefit of using genetic algorithms, compared to for example pure back propagation, is that they very rarely get stuck at a local error minimum. Given enough iterations, and assuming that we keep a high diversity in the population a genetic algorithm system will always give us an optimal solution, without having to take as
many precautions as in back propagation and other gradient search algorithms. If we have a problem with our population where a local error minimum solution takes up almost all of our solution space, we can add a penalty to the fitness of any individual who’s results in our tests comes to close to another individuals result. This helps us keep, or restore, diversity to our population. Genetic algorithms can be used with very good results in the training of neural networks, and also their ability to avoid getting stuck in local minimum, without us having to actively prevent it, adds to their allure as a good all around solution. Genetic algorithms also scale up with good results and therefore they can be used in neural nets whose size makes them unfit for back propagation. On the other hand in smaller nets and where the solution space is rather simple, the effort off creating a population of hypothesis and start the “natural selection” might result in a huge loss of performance compared to a quick and easily implemented back propagation algorithm, this is due to the large amount of processing needed before the iteration process can begin, though when this process start it takes a lot less time to compute every iteration.

3. SUMMARY

In this paper we have shown two different approaches to supervised learning. We have also explained the basic principles for supervised learning in artificial neural nets. We have described the perceptron and its functions. We have discussed a perceptron’s ability to simulate simple logical function. And if the output of an arbitrary number of perceptron is handled as inputs to other perceptrons input, we can simulate more complex functions. Interconnected layers of perceptrons constitute an artificial neural network which is later trained. The two algorithms we have shown both have their advantages and disadvantages and systems in which they excel. We have shown the basic back propagation algorithm which is derived from the sigmoid function used in every perceptron and genetic algorithms which use a form of simulated “natural selection” in a small population to train neural networks for which we have basic training data. Genetic algorithms are very good in large nets with complex solution space where as back propagation is shown to work really good in small systems.

The concept of supervised learning has been explained. And we have discussed how it differs from unsupervised learning. Also the problem with overfitting has been explained.

We have also shown the threat of local minimum in the error curve which is more of an issue in back propagation as opposed to genetic algorithms.

4. CONCLUSION

The process needed to start a genetic algorithm is a lot more time consuming than that of stating up a back propagating system. The time used in iterations, though, is a lot less in a genetic algorithm. A few iterations take less time to do in a back propagating system then in a genetic algorithm, while for a large number of iterations this is reversed and the opposite is true. This suggests that genetic algorithms are a better alternative in larger and more complex systems, but may not be suited for smaller system or system where the estimated logical functions are simple.

Avoiding local error minimums is also a factor we must take into account when choosing which algorithm to take in to accord. Both algorithms may be hampered and forced to extra iterations when faced with the possibility of a local error minimum. Tough as long as adequate steps are taken this should not turn out in favor of either system.

We have also seen how a too complex and flexible artificial neural network can actually be a drawback and increase the risk for overfitting. It can be very hard to choose an architecture appropriate for the particular problem. The choice of architecture is also important in designing our individuals in the genetic algorithms. This help show the importance of a thorough investigation and pre-study before to many conclusion about algorithm performance are drawn.

5. DISCUSSION

Given the subjects this article has addressed and the conclusions we have drawn there are some things on the subject that fall under the category of things that are interesting but would take up to much space, and, or be out of the scope of this article. We leave this things open for discussion. But would still like to address some questions which we find interesting and things which we have now found out are being developed for neural networks these days. This discussion is meant to ask questions that will broaden our interest in the field of neural networks in general and the supervised training of them in particular.

One way to combine Genetic algorithms and back propagation has risen in popularity in the later years. This system works by using a population of initially random individuals who are later trained to a certain degree before the evaluation process starts. The amount of training should be kept rather low. When the evaluation, selection and mating are done the new individuals are again trained and the process is repeated. This raises questions about the individuals learning rate as opposed to the populations.

What if we kept a population of complex individuals ready in static on hard drives to use when we needed a genetic algorithm search? If this population was composed of individuals with many chromosomes and also was set to maintain a certain number of individuals, would this in some way make the start up of a genetic search less of a pain or would we only be wasting processor power in the computer that maintains the population?

How can we design an adapting function for the learning rate in a back propagation system so that it varies in accordance with the error curve and helps us lesser the amounts of iterations needed to find a good solution?

If we train a system with one of our methods until its performance is accepted. Will it benefit from an application which lets it updates its weights further by use of feedback given to the system after it has been put in use. Or will this add risks of overfitting, and, or other problems.
6. REFERENCES


