RMS Minimization and Backpropagation for Feedforward Neural Networks

Eduardo Gutarra

February 1, 2011

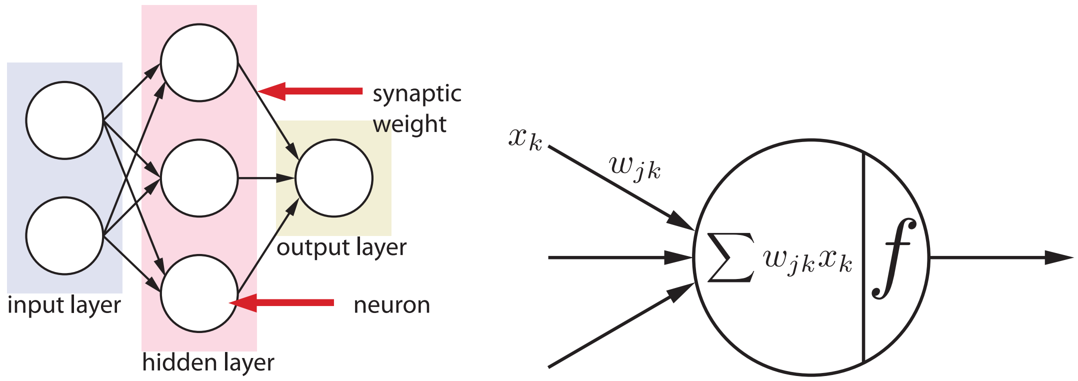
1 Introduction

Artificial neural networks are computational models that mimic the architecture, structure and/or functional aspects of biological neural networks such as the human brain. They are comprised of multiple processing elements called neurons which are interconnected through links. Often, these links have weights associated to them called synaptic weights. These weights scale the signals received from different neurons, allowing the network to process patterns and generate an output pattern. The synaptic weights are free parameters that may be changed, allowing the neural network to change its behavior. In artificial neural networks, neurons are often grouped together in layers or slabs and a neural network may be composed of 1 or more of these [3]. As an example, a feedforward neural network is illustrated in Figure 1a.

The entire behavior of the neural network is determined by the individual behaviors of neurons that comprise it. Each neuron gathers input from the external environment or other neurons. The neuron then generate a signal that may be input to other neurons or the final output. This process continues throughout the network until a response is produced on the external environment. Neurons modulate and aggregate the input to calculate an activation value. The activation value may be the same or it could be a function of the aggregation. The activation value then is passed as a parameter to an activation function, which determines the output of the neuron (See Figure 1b).

Neural networks possess important advantages and capabilities as computational models. Among these include their ability to: solve problems of nonlinear nature, perform input-output mapping through a learning process, adaptation to new situations, and their inherent parallelism [2]. One of the greatest advantages of using a neural network is that we do not program it with a configuration to solve a problem. It is actually programmed to learn and adapt to solve a problem. Neural networks have two types of learning supervised and unsupervised learning. Supervised learning consists in giving a network a set of points and allowing to correct its prediction by giving it the expected output. Unsupervised learning is often used when the network cannot be provided with a target output, and therefore works with evolutionary models. We focus on supervised learning for this report.

Even though biological neurons are 6 orders of magnitude slower than the integrated circuits found in today's computer processors. They work as a massively parallel system allowing efficient solutions to complex problems such as facial recognition. These problems are still a challenge today for today's artificial neural networks, and other computational models. Artificial neural networks have been successfully applied in applications such as: Function approximation or regression analysis and Classification, including pattern and sequence recognition among others.



(a) A feedforward neural network with labeled constituents

(b) A neuron j with input x_k from with synaptic weight w_{jk}. The neuron integrates the inputs modulated by their respective weights, and calculates an output with activation function f

Figure 1: A feedforward neural network and a magnified view of its processing element the neuron

Our interest in this report is confined to a class of neural networks which emulate the process of learning. This report examines two different algorithms for training feedforward neural networks. One algorithm is the Back-propagation algorithm and the other is the Root Mean Square (RMS) Minimization algorithm.

2 Feedforward Neural Networks

In a feedforward neural network only neurons of adjacent layers are interconnected with synaptic weights. Each layer of the neural network has connections to the next layer and there are no connections oriented backwards. The feedforward neural network begins with an input layer which may be connected to a hidden layer or directly to an output layer. The first layer is the input layer. It connects directly to the external environment and captures the input patterns presented to the network. The last layer is the output layer which produces the output pattern to the external environment. All other layers are considered hidden and may or may not be present.

The processing of a feedforward neural network begins when an external pattern made is copied to the input layer. The neurons of the input layer communicate the pattern to the following layers through synapses. The pattern is then received by neurons of non-input layers and modulated by the weight of their connections. We denote weights as W_{jk}, where j is the neighbor neuron, and k is the neuron. Each neuron receives stimulation from other neurons, except the input layer which captures the pattern. Once the inputs are modulated, they are integrated and an output value is determined by the activation function. Often the activation function only takes as parameter the integration of the modulated inputs. However, it may also be a take as parameters: the previous time step's activation value and the integration of modulated inputs. The output generated by each neuron is gathered as input by the other connected neurons, and the process repeats recursively (see Algorithm 1).

```
Input: Set of inputs from the environment
Output: Set of output values calculated by the feedforward neural network
foreach layer from the first non-input layer to the output, do
    foreach unit on the current layer, do
        Set the accumulated input value for this unit to zero
        foreach input connection to this unit, do
            Compute the modulated input across this connection
            Add the modulated input to the accumulated input
        end
        Convert the accumulated input to its corresponding output
        Store the output value for the unit in the layer structure
    end
    Return the output values from the top-most layer structure
end
```

Algorithm 1: The Feedforward Algorithm (Taken from [3])

3 Data Structure and Algorithms

To work with the neural network, we represent it using a data structure. This structure can be used by the training algorithms to store information relevant to neurons and layers. We examine two different algorithms to train the feedforward neural network. One algorithm is the Back-propagation algorithm and the other is the Root Mean Square (RMS) Minimization algorithm.

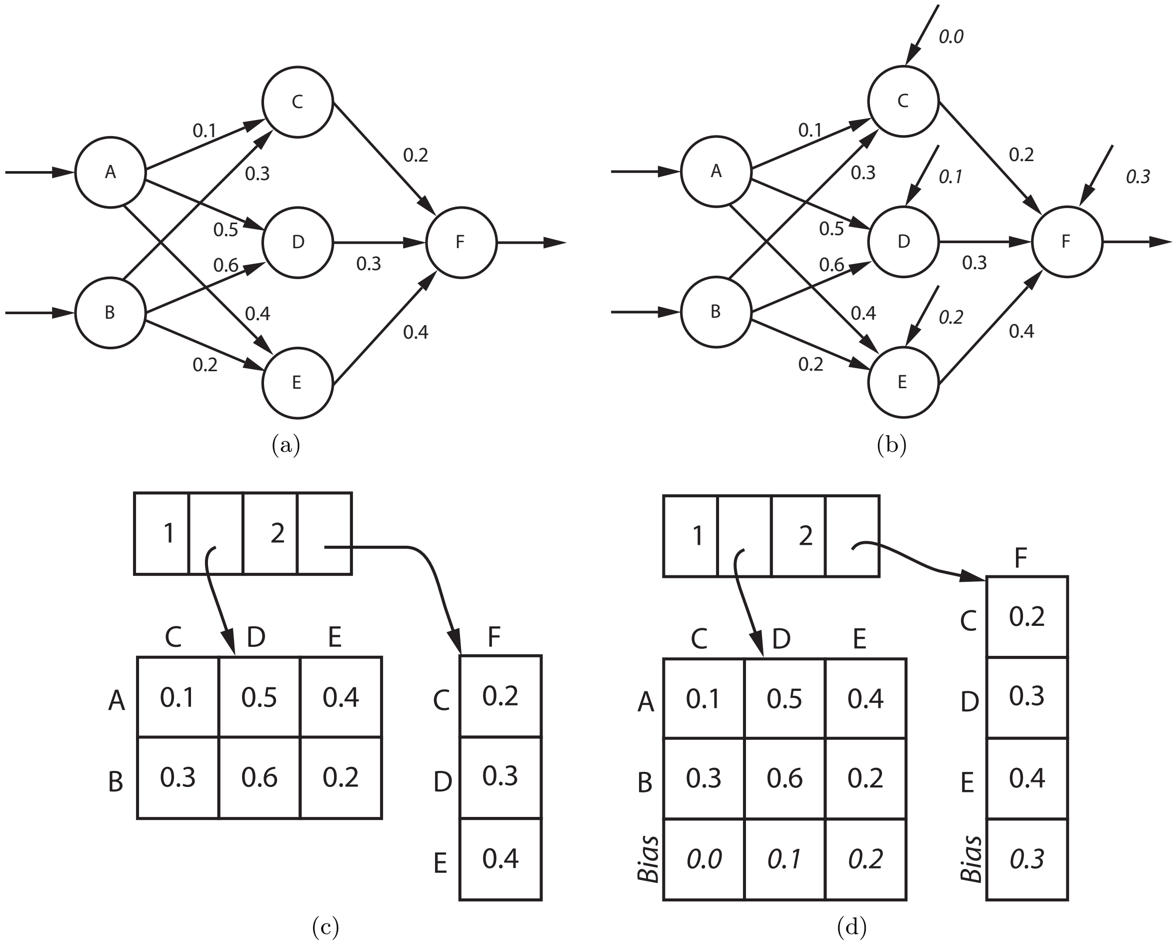
3.1 Data Structure

We do not have to implement neural networks to study them, it is possible to simulate their execution to solve problems in a computer with a single processing element. We build data structures in order to represent the neural network. A neural network can be thought of as a directed graph, where the neurons are nodes, and the synaptic weights are the arcs of the graph. Because we focus on feed-forward networks, our graphs can further be specified as directed acyclic graphs. These graphs do not require a full matrix to represent connections between the neurons. Instead we use an array of matrices where each matrix stores the synaptic weights of the departing connections from the current neuron in the current layer to the neurons of the next layer (see Figures 2a and 2c).

We may also add an additional free parameter that does not link to another neuron, and has a constant input. We call this parameter the bias, and it allows the activation functions to translate along the y-axis. We include the bias as one more row of elements in each weight matrix (see Figures 2b and 2d).

3.2 Backpropagation training algorithm

Back-propagation is a method of supervised learning. It is used to train our feedforward neural network. To use the back-propagation algorithm we provide it with both example inputs and target outputs. The outputs generated by the neural network are then compared against the target outputs for a given example. Using the target outputs, the back-propagation training algorithm then calculates errors and adjusts the weights of the various layers backwards from the output layer



(c)

	C	D	E
A	0.1	0.5	0.4
B	0.3	0.6	0.2

	F
C	0.2
D	0.3
E	0.4

(d)

	C	D	E
A	0.1	0.5	0.4
B	0.3	0.6	0.2
Bias	*0.0*	*0.1*	*0.2*

	F
C	0.2
D	0.3
E	0.4
Bias	*0.3*

Figure 2: Feedforward neural networks (a and b) with their corresponding computer representation (c and d). Figures (a) and (c) represent a neural network without bias parameters. Figures (b) and (d) represent a neural network with bias parameters

to the input layer. The degree of adjustment in the weights and bias can be scaled. The scaling factor used is called the learning rate η, and it determines how quickly the neural network attempts to learn the mapping of input to output.

Back-propagation is often used to train feedforward neural networks but it can also be used to train other types of networks, and likewise feedforward networks may be trained with other methods. In this report, we only examine back-propagation when it is used to train a feedforward neural network. Algorithm 2 illustrates the process in further detail.

3.3 RMS Minimization Algorithm

Another algorithm we use to train the feedforward neural network is the Root Mean Square (RMS) Minimization algorithm. For this algorithm we also provide example inputs and target outputs. We run the feedforward algorithm with each individual example generating the corresponding outputs of the neural network. After generating the outputs we calculate the root mean square between outputs given by the neural network and target output provided by the example. We keep this calculation as RMS_{old}.

Next, a small change is made to one of the weights of the neural network, and the feedforward algorithm is called again to calculate new outputs for the neural network. We take again the root mean square between outputs and targets which we call RMS_{new}. We then calculate a slope by taking the difference between RMS_{old} and RMS_{new}, and divide by the change made to the weight. We use this slope to determine what change needs to be made to improve the RMS between the output of our network and the target from the example. Unlike the back-propagation method, this method performs a $\text{num}(E)$ feedforward passes, where $\text{num}(E)$ is the number of examples used to train the network. The process is illustrated in further detail in Algorithm 3.

4 Experiments and Results

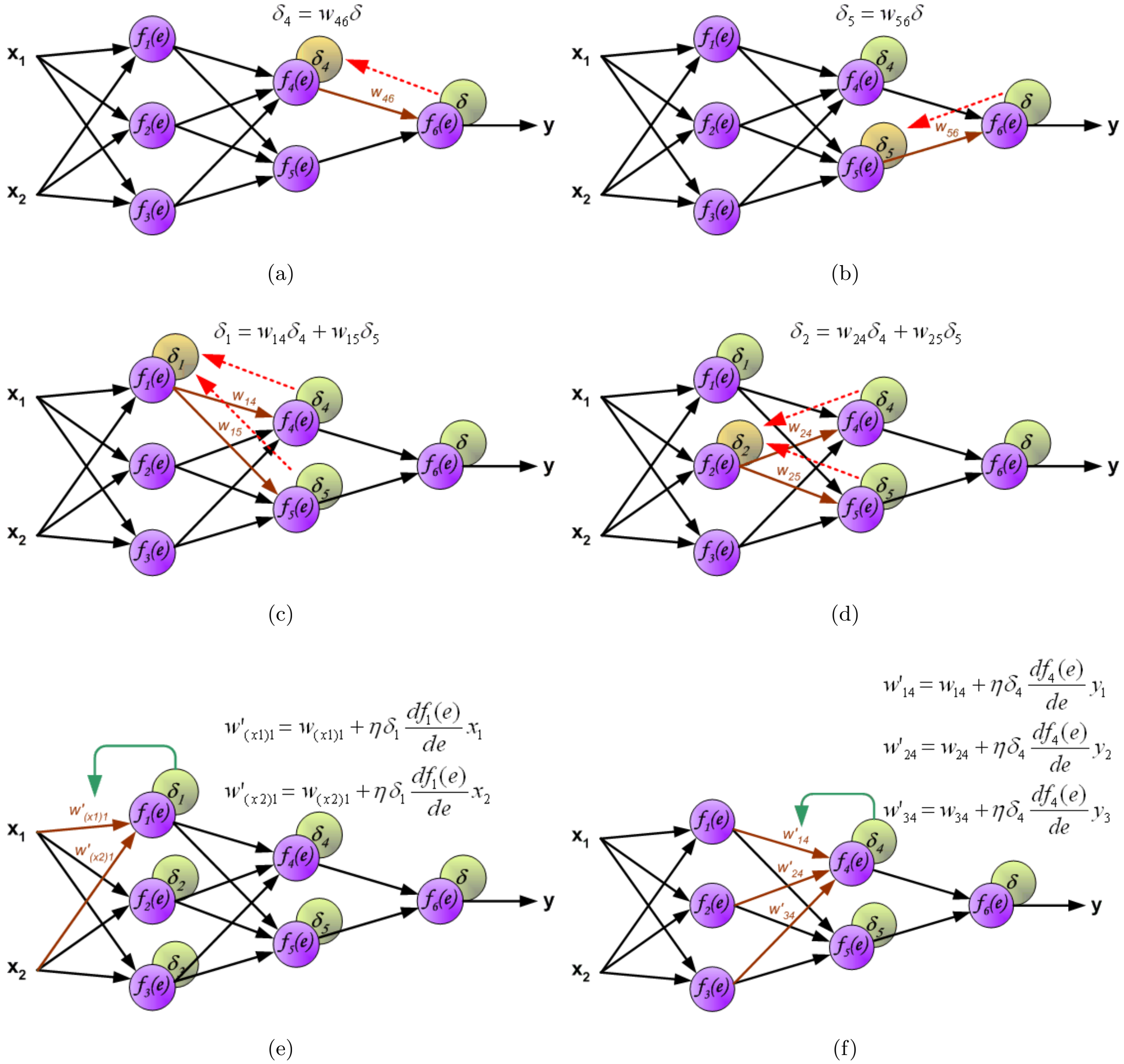
A set of system experiments are performed where we test the RMS and back-propagation training methods. We compare the time complexity between both training methods as well as their accuracy after training is completed. The set of examples to train the network are generated by taking two numerical inputs and adding them to generate the expected output. This data is normalized so that its range is between -1 and 1. The feedforward neural network is configured with 2 neurons in the input layer, 3 in the middle layer and 1 in the output layer. The neurons in the middle layer use a hyperbolic tangent activation function, and the neurons in the output layer use a linear function. In our procedures we change different parameters of the neural network, and observe the behavior in time complexity and training accuracy.

4.1 Time Complexity

We tested the time complexity of both algorithms in different aspects that may be configured in the neural network. We look at time as we vary the number of neurons in the middle layer, the number of iterations, and the number of points in the example. Experimentally, we notice that both algorithms exhibit linear growth in time when we change the number of iterations or examples (see Figures 4b and 4c). The algorithms however differ in that RMS Minimization grows quadratically whereas back-propagation grows linearly when the number of neurons in the middle layer is varied

```
Input: Set of examples $E$
/* $e$ is a single example                                              */
/* $I(e)$ set of inputs in a single example                             */
/* $T(e)$ set of target outputs for a given example                     */
/* $O(e)$ are the target outputs of the example                         */
/* Each iteration of this loop we an epoch                              */
foreach example $e$ in a set of examples $E$ do
    Calculate $O(e)$ for $I(e)$ with feedforward (refer to Algorithm 1)
    Call function CalculateOutputDeltas($O(e)$, $T(e)$)
    Call function CalculateInternalDeltas
    Call function UpdateWeights
end
CalculateOutputDeltas($O(e)$, $T(e)$):
Get output values $O(e)$ from the output layer neurons
foreach individual output value $O(e)_i$ do
    Calculate error $\epsilon$ as $O(e)_i$ - $T(e)_i$
    Calculate $\delta_{O(e)_i} = \partial f(O(e)_i) \times \epsilon$
    Add $\delta_{O(e)_i}$ to set of deltas $\Lambda$
end
CalculateInternalDeltas:
Let $\Lambda_{i+1}$ be the next layer's set of deltas
foreach non-output layer $i$ from the penultimate to the first do
    foreach neuron $j$ in this layer do
        Initialize error $\epsilon$ as 0.0
        foreach neuron $k$ of the next layer do
            Calculate $\epsilon$ as $\epsilon + \Lambda_{i+1,k} W_{ijk}$
        end
        $\Lambda_{i,j} = \partial f(\epsilon \times$ neuron $j$'s output$)$
    end
end
UpdateWeights:
/* $\eta$ is the learning rate                                          */
foreach layer $i$ do
    foreach neuron $j$ in this layer do
        foreach neuron $k$ of the next layer do
            Calculate $\Delta W_{ijk}$ as $\Lambda_{i,j} \times$ neuron $j$'s output
            $W_{ijk} \leftarrow \eta \times \Delta W_{ijk}$
        end
    end
end
```

Algorithm 2: The Back-propagation Algorithm


$\delta_4 = w_{46}\delta$
$\delta_5 = w_{56}\delta$
$\delta_1 = w_{14}\delta_4 + w_{15}\delta_5$
$\delta_2 = w_{24}\delta_4 + w_{25}\delta_5$
$w'_{(x1)1} = w_{(x1)1} + \eta\delta_1 \frac{df_1(e)}{de} x_1$
$w'_{(x2)1} = w_{(x2)1} + \eta\delta_1 \frac{df_1(e)}{de} x_2$
$w'_{14} = w_{14} + \eta\delta_4 \frac{df_4(e)}{de} y_1$
$w'_{24} = w_{24} + \eta\delta_4 \frac{df_4(e)}{de} y_2$
$w'_{34} = w_{34} + \eta\delta_4 \frac{df_4(e)}{de} y_3$
x_1
x_2
y
$f_1(e)$
$f_2(e)$
$f_3(e)$
$f_4(e)$
$f_5(e)$
$f_6(e)$
δ
δ_1
δ_2
δ_4
δ_5
w_{46}
w_{56}
w_{14}
w_{15}
w_{24}
w_{25}
$w'_{(x1)1}$
$w'_{(x2)1}$
w'_{14}
w'_{24}
w'_{34}

(a)

(b)

(c)

(d)

(e)

(f)

Figure 3: An illustration of the Back-propagation algorithm. (Taken from [1])

```
Input: Set of examples $E$, constants $\alpha$ and $\epsilon$
/* We call each iteration of this loop an epoch                          */
Set $\Delta W$ to 0.01
foreach layer i from the first non-input layer to the output, do
    foreach layer i from the first non-input layer to the output, do
        foreach neuron j on the current layer, do
            foreach  connection k to a neuron of the next layer, do
                Calculate $\mathrm{RMS_o}$ as CalculateRMS($E$)
                Set $W_{ijk}$ to $W_{ijk}$ + $\Delta W$
                Calculate $\mathrm{RMS_n}$ as CalculateRMS($E$)
                Calculate $\Delta RMS$ as $\frac{\mathrm{RMS_n}-\mathrm{RMS_o}}{\Delta W}$
                if  $\Delta RMS \neq 0.0$ then
                    Calculate $\Delta W_{ijk}$ as $\alpha \times \Delta W_{ijk} - \epsilon \times \Delta RMS$
                    Calculate $W_{ijk}$ as $W_{ijk} + \Delta W_{ijk}$
                end
            end
        end
    end
end
CalculateRMS($E$): /* $e$ is a single example                            */
/* $I(e)$ set of inputs in a single example                              */
/* $T(e)$ set of target outputs for a given example                      */
/* $O(e)$ are the target outputs of the example                          */
Set $\epsilon$ to 0.0
foreach example e in the set of examples E do
    $O(e) \leftarrow$ Call feedforward algorithm with $I(e)$ (see Algorithm 1)
    foreach  output $O(e)_i$ obtained from the neural network do
        Calculate $\epsilon$ as $\epsilon + (O(e)_i - T(e)_i)^2$
    end
end
return $\frac{\epsilon}{\mathrm{num}(E)-1}$ ;          /* num($E$) is the total number of examples in $E$ */
```

Algorithm 3: The RMS Minimization Algorithm

	Calls	Estimated # of ops.
Feedforward	num$(E)i$	$\propto c$
CalculateOutputDeltas	num$(E)i$	$\propto o$
CalculateInternalDeltas	num$(E)i$	$\propto b$
UpdateWeights	num$(E)i$	$\propto c$
Total		num$(E)i(2c+o+b)$

Table 1: Estimated number of operations performed at each stage of the Back-propagation algorithm. Here, c is the number of connections, i is the number iterations, o is the number of outputs, and num(E) is the number of examples.

(see Figure 4a). Finally, in all our system experiments, back-propagation was faster than RMS Minimization.

To look at the time complexity analytically we estimate the number of operations made by both algorithms. The number of operations is proportional to the number of weights and biases in the neural network in many stages of the different algorithms. The number of weights between two layers of neurons is the product between the total number of neurons in two adjacent layers. Therefore the total number of weights for the entire network is $\sum_{i=1}^{n} N_{i-1}N_i$ where N_i is the number of neurons in each layer, and n is the total number of layers. For the back-propagation algorithm, the number of operations in a single epoch depends on the number of weights and biases. To account for the biases we count one bias for each neuron that is not in the input layer. Therefore the total number of biases is $\sum_{i=1}^{n} N_i$. We denote the total number of connections including weights and biases as $c = \sum_{i=1}^{n} N_{i-1}N_i + \sum_{i=1}^{n} N_i$.

For each iteration of the back-propagation algorithm we run the feedforward algorithm for every example in the set of examples E. The feedforward algorithm has a number of operations that is proportional to the number of connections in the neural network, which we denote as c. After the neural network outputs are generated, the algorithm proceeds to calculate an error between expected output and input with the **CalculateOutputDeltas** procedure. The number of operations in this method is proportional to the number of outputs of the neural network denoted o. The error is propagated backwards throughout the rest of the network with the **CalculateInternalDeltas** method. The number of operations made in this method is proportional to the number of deltas that we need to adjust the connections of the neural network. The number of deltas we must save is equal to the number of neurons that are in the output layer and middle layers which is also equal to the number of biases we keep. Therefore the number of operations in **CalculateInternalDeltas** is proportional to $\sum_{i=1}^{n} N_i$. Finally in the **UpdateWeights** method, the weights are adjusted with the deltas that were calculated in the **CalculateInternalDeltas** and **CalculateOutputDeltas** methods. Because the the adjustments are done for every weight and bias, the total number of operations in this last method is proportial to the number of connections. The analysis for back-propagation is summarized in Table **??**.

In each iteration of the RMS Minimization algorithm the method **CalculateRMS** is calculated twice for every connection. The two calculations are made to determine the weight or bias adjustment necessary to improve the error between the neural network's output and the expected output provided in the examples. The method **CalculateRMS** has a number of operations proportional to the number of examples in the set of examples. To calculate the output generated by the neural

	Calls	Estimated # of ops.
CalculateRMS	$2ci$	$\propto$ num(E)
Feedforward called by CalculateRMS	num(E)	$\propto c$
Total		$2c^2$num$(E)i$

Table 2: Estimated number of operations performed at each stage of the RMS minimization algorithm. Here, c is the number of connections, i is the number iterations, and num(E) is the number of examples.

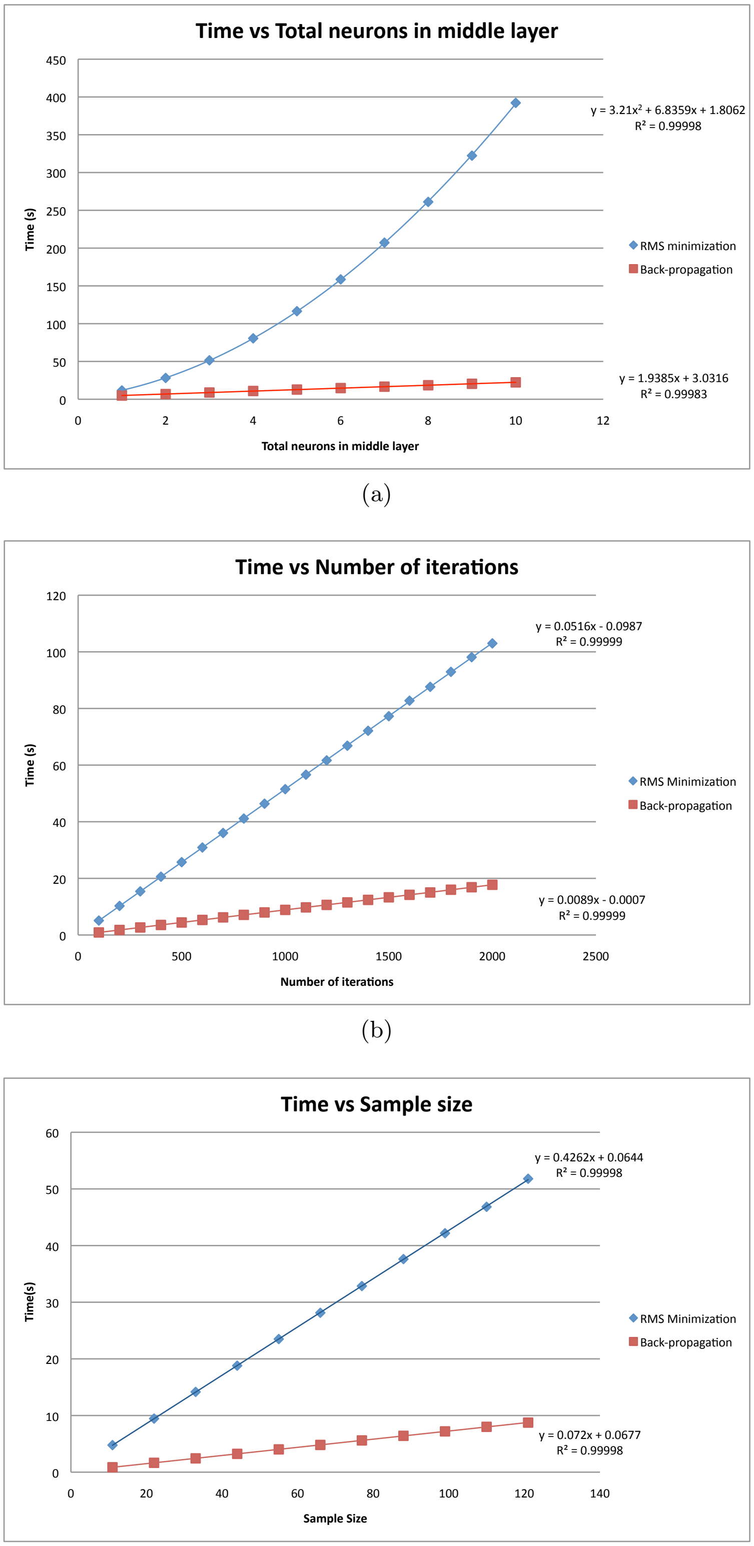
network the feedforward algorithm is called by the **CalculateRMS**. The feedforward algorithm is called by **CalculaterRMS** for every example in the set of examples. The number of operations performed in the feedforward algorithms is proportional to the number of connections. RMS Minimization exhibits linear growth to variations of all aspects of the neural network, but those related to the number of connections in the neural network. This means that when the number of neurons is modified in the neural network, it exhibits quadratic growth in time. The analysis for RMS minimization is summarized in Table **??**.

For the RMS algorithm the number of operations in a single epoch depends on both the number of weights and the number of is squared because we run the feedforward algorithm for each change we do on a single weight.

4.2 Training Accuracy

In this section, we determine compare the training accuracy of the back-propagation and RMS minimization algorithms. To test the training accuracy of both algorithms, we chose a simple problem that takes as input 2 numbers and computes their sum as the output. To generate the sample data we use two input variables x and y that range from 0 to 10. We add them in all 121 possible combinations, and so the network is trained on a sample of 121 different sets of inputs and outputs.

We configured the neural network with 2 neurons in the input layer, 3 in the middle layer and 1 in the output layer. We use a hyperbolic tangent activation function for the neurons in the middle layer, and a linear activation function for the output neurons. The feedforward neural network was trained on 121 sample points for 10000 iterations. These sample points were normalized between -1 and 1 to be more suitable for the chosen activation function. For the back-propagation algorithm we set the learning rate $\eta = 0.25$. For RMS Minimization we set the constants $\epsilon = 0.03$ and $\alpha = 0.7$. Both algorithms were used to train the network to add the normalized sample input. 10 different training sessions were attempted, each one with its respective set of initial weights which we differentiate with a seed. The results of the different training sessions for RMS Minimization and Back-propagations are summarized in Table 3. Overall the back-propagation algorithm trained the neural network to a higher degree of accuracy. The expected output and the output produced by the neural networks after training with back-propagation and RMS minimization is illustrated in Figures5b and 5a respectively.

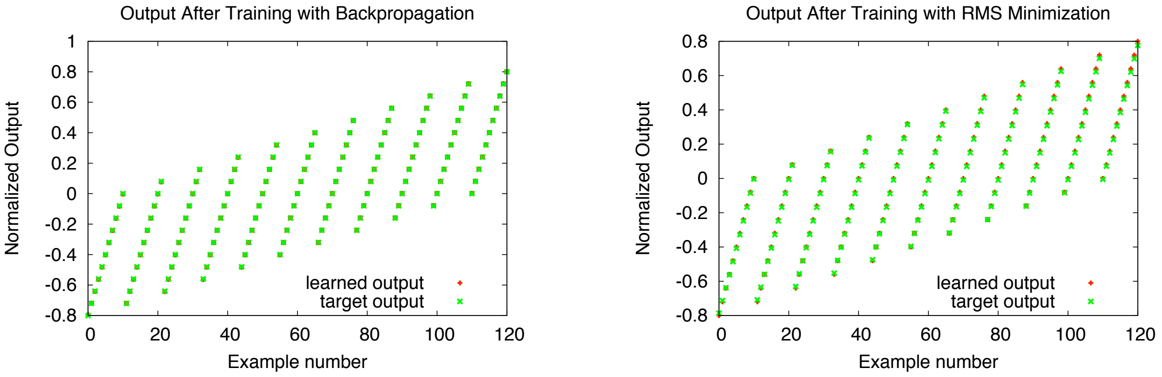


(a)

(b)

(c)

Figure 4: Comparison in times between back-propagation and RMS minimization training algorithms. In Figure 4a the the number of neurons in the middle layer is varied and time is measured. In Figure 4b the number of iterations is varied. In Figure 4c the size of the sample is varied



(a) (b)

Figure 5: Outputs for the best resulting neural network after training with Back-propagation (a) and RMS Minimization (b).

Seed	Error
9	4.61×10^{-5}
2	5.57×10^{-5}
10	6.03×10^{-5}
1	6.31×10^{-5}
7	9.52×10^{-5}
3	9.79×10^{-5}
5	0.000196554
4	0.000244331
6	0.000259482
8	0.000282861
0	0.000284188

(a) RMS Minimization Training Sessions

Seed	Error
5	1.46×10^{-7}
10	1.97×10^{-7}
4	2.70×10^{-7}
1	4.23×10^{-7}
8	4.53×10^{-7}
9	5.25×10^{-7}
6	5.59×10^{-7}
2	6.39×10^{-7}
7	9.65×10^{-7}
3	1.35×10^{-6}
0	1.53×10^{-6}

(b) Back-propagation Training Sessions

Table 3: RMS Error resulting from different initial random weight configurations for the neural network.

4.3 Learning the Damped Harmonic Oscillation

To test further the back-propagation training algorithm we chose a more complicated problem. The problem consists in predicting the behavior of a mass attached to the end of a spring. The spring is connected to a fixed object. Ideally without the effect of friction and damping, the system becomes a perfect oscillator when force is applied. Yet we add damping to this system, making the oscillation decrease over time. The oscillatory movement of this spring-mass system can be described with the following equation:

$$x(t) = Ae^{-\alpha t} \cos \beta t + \phi. \tag{1}$$

Here, the position x is determined by the amplitude A, the time t, and the constants α and β. Using this formula we calculate the position of the mass, for a given time. We chose an amplitude $A = 1$, and set the constants α and β to -3 and 20 respectively. We let $\phi = 0$ for the first experiments, where we test the network on the same data it has learned.

Because we want the neural network to learn the behavior of this curve, we feed it the following 2 inputs: The position in the previous time step $x(t-1)$, and the position for the current time step $x(t)$. As the target output we have the position of the mass in the next time step, $x(t+1)$.

We configured the neural network with 2 neurons in the input layer, 3 in the middle layer and 1 in the output layer. The hyperbolic tangent function was used as the activation function for the neurons in the middle layer, and for the output neuron a linear activation function was used. The feedforward neural network was trained on 100 sample points for 40000 iterations with a learning rate of 0.25 using the back-propagation algorithm.

The weights of the neural network were initialized randomly. The neural network was trained using 10 different randomized sets of weights. The weights for the neural network producing the least error with the target output were saved. The different training sessions are summarized in Table 4. The training sessions are sorted from best configuration being seed 9, to worst, seed 0. The change in error for each iteration in the training session seeds 9 and 0 is illustrated in Figure 6. Logarithmic scale is used for this plot because the error descends dramatically in the first iterations and slows down for later iterations. We observed that seed 0 reaches a minimum in an earlier iteration than seed 9. Conclusively one can observe that some initial configurations will reach a local minimum before all iterations are completed. This is a common problem when training neural networks, therefore different starting configurations should be attempted to increase the chances of finding the global minimum.

We configure the network to use the best set of weights found during the 10 training sessions. It is then tested with the same input data it was trained with to see how well it has learned. Initially we used perfect data, in which we supplied the input parameters to the network, $x(t)$ and $x(t-1)$ for all the time steps t. Given those parameters the neural network predicts $x(t+1)$. We then compare graphically (see Figure 7a), the predicted output of the neural network with the expected output obtained through equation 1. To further test the network, we give it initially positions $x(0.01)$ and $x(0)$. The network then predicts an $x(0.02)$ position. It then uses its own output and previous information to predict $x(0.03)$. This process continues recursively until all positions for 100 time steps are generated (see Figure 7b). Comparing Figures 7a and 7b we notice that the predicted output from the neural network deviates more from the expected output as time is advanced. **How do I explain that there is some error accumulation**.

Previously, we tested the network with data points that were taught before or that were within the bounds marked by the points the neural network had been shown. Finally, we test the neural

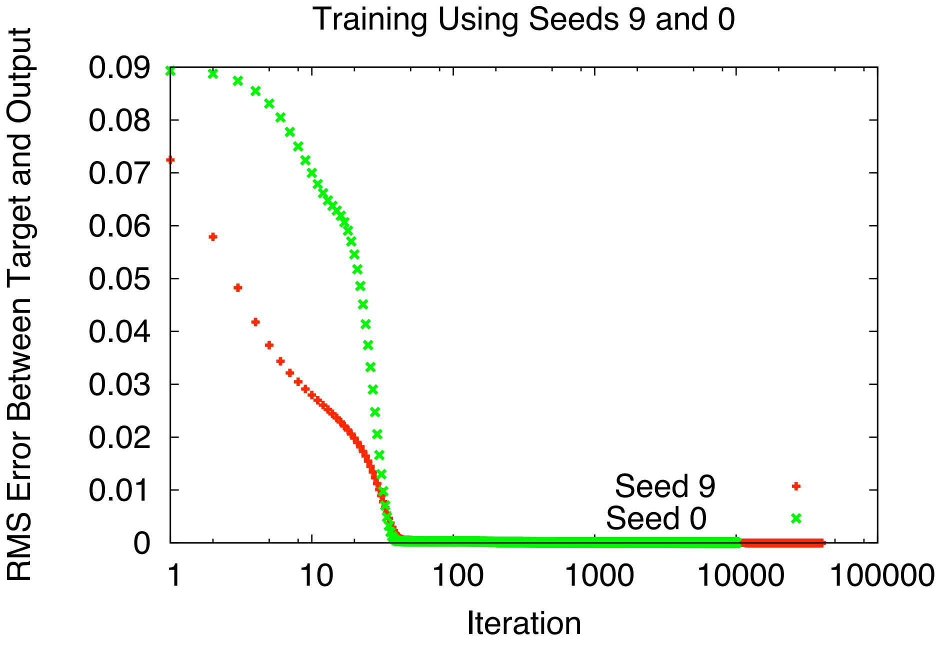
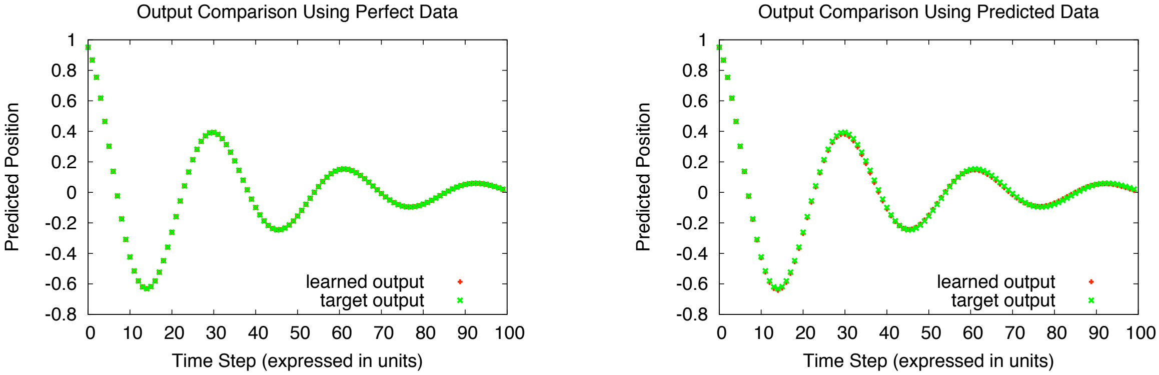


Figure 6: Training session comparison for seeds 0 and 9. We observe that seed 0 reaches a local minimum before it completes all iterations

Seed	Error
9	9.27 $\times 10^{-8}$
8	1.40 $\times 10^{-7}$
3	2.63 $\times 10^{-7}$
4	5.55 $\times 10^{-7}$
5	2.91 $\times 10^{-6}$
2	3.75 $\times 10^{-6}$
7	6.04 $\times 10^{-6}$
10	1.27 $\times 10^{-5}$
6	1.36 $\times 10^{-5}$
1	1.40 $\times 10^{-5}$
0	1.63 $\times 10^{-5}$

Table 4: RMS error resulting from different initial random configurations for the feed forward neural network. The seed determines the configuration of the neural network. Notice that seed 9 produced the best set of weights for approximating the neural network to the target output.



(a)

(b)

Figure 7: Output comparison between perfect data (a) and predicted data (b) as input for the neural network. It can be observed in figure (b) that the error is greater and increases as time is increased. Yet, the network can rely on its own generated data relatively well.

network on a different initial condition. We let $\phi = 10$, and by doing so we expose the neural network to some data it has not encountered before. Furthermore, the data is not within the bounds of data points that were taught to the neural network. The neural network makes a prediction for the unknown data points when $t > 0.5$. A comparison between the neural networks prediction and the expected output using equation 1 is illustrated in Figure 8. Based on this illustration, we observe that the network is capable of making a sane prediction on data points that it has not previously encountered.

5 Conclusion

To summarize the results of the experiments we arrived to the following 5 main conclusions:

The Back-propagation algorithm has a time complexity of $O(n)$, which is faster than the RMS Minimization algorithm with a time complexity of $O(n^2)$ when varying the number of neurons in the middle layer.

The Back-propagation algorithm attained a higher degree of accuracy than the RMS Minimization algorithm for the addition experiment.

As observed in some of the experimental results, the learning algorithms could get stuck in local minima unable to arrive to better solutions. We noticed this in both the Back-propagation and RMS Minimization algorithms.

The Back-propagation demonstrates in these experiments to be more scalable than the RMS Minimization algorithm in terms of numbers of neurons in the network.

Finally, the back-propagation neural network demonstrates ability to approximate solutions to non-linear problems even on data not previously shown to it. However, it is important to note that its accuracy wanes as it progressively continues to generate other points. Mostly due to accumulating errors from previous calculations.

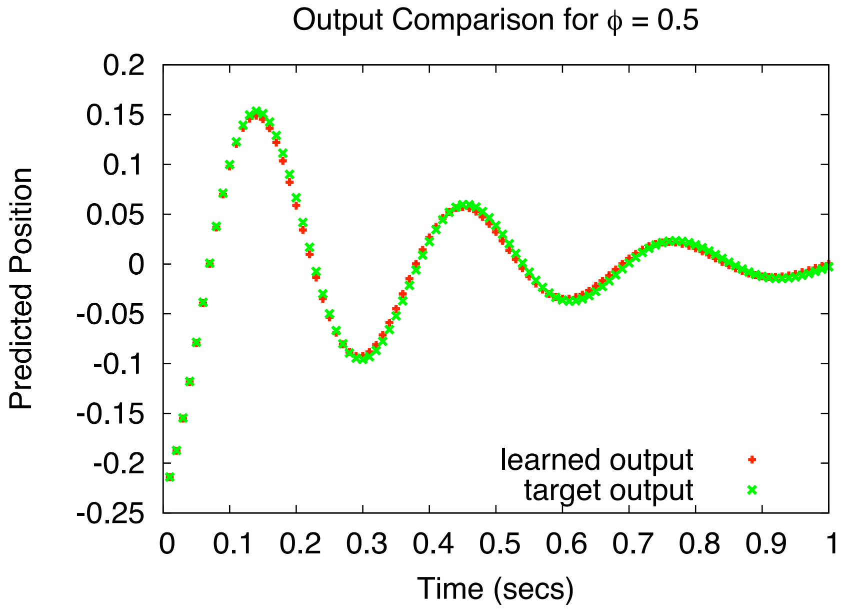


Figure 8: Comparison between predicted output and expected output for the neural network when $\phi = 10$. Unknown data to the neural network starts when $t > 0.5$

References

[1] Mariusz Bernacki. Backpropagation. available from `http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html`.

[2] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1994.

[3] David M. Skapura. *Building Neural Networks.* ACM Press, New York, 1996.