7-2012

A Regression-based Training Algorithm for Multilayer Neural Networks

Christopher W. Sherry

A Regression-based Training Algorithm for Multilayer Neural Networks

by

Christopher W. Sherry

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Supervised by

Dr. Zack Butler
Department of Computer Science
B. Thomas Goliano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York
July 2012

Approved By:

Dr. Zack Butler
Associate Professor, Department of Computer Engineering
Primary Adviser

Dr. Leon Reznik
Professor, Department of Computer Science
Reader

Dr. Joe Geigel
Associate Professor, Department of Computer Science
Observer

Acknowledgements

Contents

List of Figures

List of Algorithms

List of Tables

Abstract

Artificial neural networks (ANNs) are powerful tools for machine learning with applications in many areas including speech recognition, image classification, medical diagnosis, and spam filtering. It has been shown that ANNs can approximate any function to any degree of accuracy given enough neurons and training time. However, there is no guarantee on the number of neurons required or the time it will take to train them. These are the main disadvantages of using ANNs. This thesis develops an algorithm which uses regression-based techniques to decrease the number of training epochs. A modification of the Delta Rule, combined with techniques established for regression training of single-layer networks, has resulted in much faster training than standard gradient descent in many cases. The algorithm showed statistically significant improvements over standard backpropagation in the number of iterations, the total training time, the resulting error, and the accuracy of the resulting classifier in most cases. The algorithm was tested on several datasets of varying complexity and the results are presented.

1. Introduction

Artificial neural networks are mathematical models that can represent complex non-linear functions in multidimensional spaces. These networks are built from simple structures composed of several inputs, weights associated with the inputs, and an activation function. These artificial neurons operate by summing the products of each input by its associated weight and passing the result through the activation function. As such, each neuron represents a line or plane in the input space. By assembling a network of these neurons, the model aggregates the activation functions and composes a non-linear curve or surface.

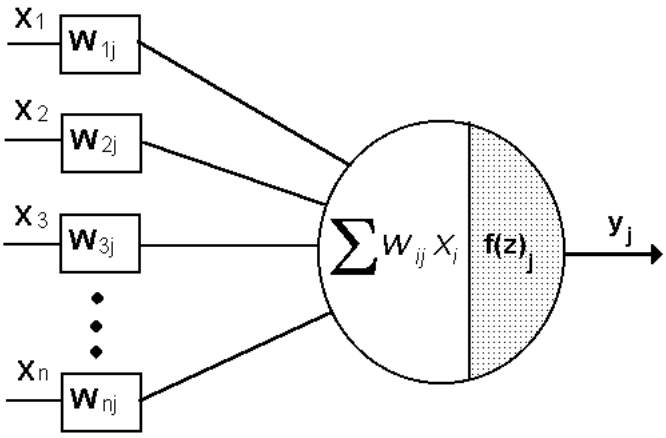


Figure 1.1: Diagram of a neuron.

It has been shown that these models are capable of representing any function to any degree of accuracy if given enough nodes and training time. This is particularly useful in pattern classification tasks since the non-linear curve that the model represents can be used to separate data from different classes. Artificial neural networks have been successfully applied to speech recognition, image classification, medical diagnosis, spam filtering and many other domains.

The central element in a neural network is the artificial neuron. Inspired by biology, this basic unit is composed of a series of weights and a function which combines the inputs

and weights into an output value. The typical neuron simply multiplies each input by its corresponding weight, and sums all the products before passing the resulting sum through an activation function.

$$output = f(\sum_{i=0}^{n} input_i \times weight_i) \tag{1.1}$$

For a neuron to produce a desired output when fed specific inputs, the weights, which essentially scale the importance of the inputs before being processed by the activation function, must be set accordingly.

The procedure for determining these weights is usually non-trivial, especially in multilayer neural networks where the number of neurons and connections between neurons can become large, and therefore the number of weights can grow to be very large. Typically, an iterative approach is used to adjust weights over time to converge on the network's desired behavior. The most common approaches to training belong to the gradient descent family of algorithms (backpropagation, scaled conjugate gradient, Levenberg-Marquardt, etc.). These algorithms update weights to reinforce connections which modify inputs with a strong correlation with the outputs. They use the local gradient of a global performance metric, such as least-squared error, to determine how to adjust the weights. The basic backpropagation algorithm is shown in Algorithm 1.

The adjustments backpropagation makes to each weight at each iteration are usually very small and governed by a parameter called the learning rate. By making weight updates small, the knowledge represented by the network can be accumulated gradually and incrementally. The learning rate scales the adjustments made at each iteration so that one iteration's learning does not undo the learning of any previous iterations. Small training steps, however, while preserving previous weight updates, can cause the algorithms to get stuck at local rather than global minima.

Algorithm 1 The basic backpropagation algorithm [11]

Initialize weights randomly
Initialize err, $threshold$, and $maxEpochs$
while $epoch < maxEpoch$ **and** $err > threshold$ **do**
    for each example (x, y) in the training set **do**
        / Propagate the inputs forward to compute the outputs */*
        for each node i in the input layer **do**
            $a_i \leftarrow x_i$
        end for
        for $\ell = 2$ to L **do**
            for each node j in layer ℓ **do**
                $in_j \leftarrow \Sigma_i w_{i,j} a_i$
                $a_j \leftarrow g(in_j)$
            end for
        end for
        / Propagate deltas backward from output layer to input layer */*
        for each node j in the output layer **do**
            $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
        end for
        for $\ell = L - 1$ to 1 **do**
            for each node i in layer ℓ **do**
                $\Delta[i] \leftarrow g'(in_j) \Sigma_j w_{i,j} \Delta[j]$
            end for
        end for
        / Update each weight using deltas */*
        for each weight $w_{i,j}$ **do**
            $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$
        end for
    end for
end while

While backpropagation can be accelerated with techniques such as, among others, dynamic learning rates (momentum) and estimations of good starting weights, gradient descent still can take a long time to converge. The number of iterations required to produce an optimal set of weights is often very large. A reduction in this number is often the best way to speed up training.

1.1 Problem Statement

Backpropagation provides a stable way to train multilayer neural networks but can take a long time to achieve desired results. Neural network users have to wait for training to finish to determine the quality of the resulting network. Using slow traditional methods prevent rapid iteration to improve up network architectures should training not produce a good model. Reducing total training time would allow more time for experimentation with architecture and data to reach the best solution possible or simply to complete research or implementation faster.

Research in the mid 90's to early 00's showed that regression based approaches for training single layer neural networks greatly outperformed traditional methods of training. This thesis seeks to test the hypothesis that similar regression methods to those previously used on single layer neural networks can be applied to multilayer neural networks with similar results in improving performance.

1.2 Approach

This thesis investigates a multilayer neural network training algorithm based on regression of the network's error function. The goal of this investigation is to reduce multilayer neural network training times through a reduction in the number of training epochs. This research built upon work done on regression-based training algorithms for single-layer neural networks.

The number of weights in a fully connected neural network grows rapidly as the number of neurons in the network increases. This affects the running time of any training algorithm used to determine the proper values of the weights since there are more weights to evaluate at each iteration of training. By replacing multiple iterations of simple gradient descent calculations with a single, more complex, but more effective regression iteration, total training time can be reduced.

If a neural network's nodes all use continuous, differentiable activation functions, then the network's error will be a continuous, differentiable function of the network's weights. If this error function is known, it is a simple enough task to find global optima using the first and second derivative tests. However, the error function is rarely known, and few assumptions are made about it to attempt to approximate it. Thus, gradient descent algorithms examine local features of the error function and move in the direction towards a minimum. Because these algorithms examine and exploit local features, they have the tendency to converge on local rather than global minima and get stuck there. Regression estimates the error function globally, and as such can avoid getting stuck at local minima because global minima can be observed directly. Thus, in addition to reduced training time, accuracy of the network can also be improved.

Algorithms that take advantage of regression techniques have been proposed for single-layer networks. However, they are not in popular use. This is because multilayer networks have more utility in practice than single-layer networks. By extending these algorithms to work for multilayer networks, they become more useful for application on real world problems.

2. Supporting Work

Andersen and Wilamowski[1] proposed an algorithm to train a one layer neural network using regression information. Similar to a least-squared error linear discriminant, the weights are updated directly based on the patterns that are misclassified. Their work showed significant improvements over backpropagation in both the number of epochs needed to train the network and the number of floating point operations used during training. The Andersen and Wilamowski algorithm is designed for single-layer neural networks, but as this thesis shows, it can be used powerfully as a subroutine to train multilayer networks.

Castillo *et al.*[4] also used a direct approach to training weights in a neural network. While Andersen and Wilamowski worked on single-layer networks, Castillo *et al.*. train a network with a hidden layer in addition to the output layer. Their method uses linear regression based on minimizing least squares error to train the output layer directly while still using a gradient descent algorithm to compute weight updates for the hidden layer. By calculating the output weights directly, the algorithm decreases the degrees of freedom in the optimization and thus decreases training time by reducing the number of optimized variables. This work also showed significant experimental speedups when compared to standard gradient-descent-only methods.

Castillo *et al.*[2] had also previously done work on single-layer neural networks. They showed that it is possible to train a one-layer neural network with non-linear activation functions using a system of linear equations and/or linear programming with constraints to minimize the error function. This method was also shown to be significantly faster than standard gradient descent algorithms.

2.1 The Andersen and Wilamowski algorithm

The Andersen and Wilamowski single-layer training algorithm (A&W henceforth) uses the pseudoinverse of the input matrix and the difference the outputs are from the targets to

update the weights.

To update the weights, it first determines "how wrong" the weights are by simply taking the difference between the targets and the outputs that are generated by feeding the inputs through the single-layer network. Those outputs are then scaled by the derivative of the activation function. This element-wise scaling ensures that the algorithm moves in the direction of the gradient towards a good solution. The resulting matrix is right-multiplied with the pseudoinverse of the input matrix and added to the weights.

This procedure moves the weights in the direction of the minimum-norm solution or least-squares solution to the equation $inputs \times weights = targets_{\aleph}$ where $targets_{\aleph}$ are the 'deactivated' targets (targets that have been passed through the inverse of the activation function). It makes the adjustments incrementally so that it doesn't overshoot solutions better than the minimum-norm or least-squares solution.

Algorithm 2 The Andersen and Wilamowski algorithm

1: Initialize $weightMatrix$, $epoch$, $maxEpochs$, $error$, and $threshold$
2: **while** $(epoch < maxEpochs)$ && $(error > threshold)$ **do**
3: $\quad outMatrix \leftarrow sigmoid(inputMatrix \times weightMatrix)$
4: $\quad delMatrix \leftarrow targetMatrix - outMatrix$
5: $\quad delMatrix \leftarrow delMatrix \div sigmoidPrime(outMatrix)$
6: $\quad weightMatrix \leftarrow weightMatrix + (inputMatrix^{\dagger} \times delMatrix)$
7: $\quad$ Calculate new $error$
8: $\quad$ Increment $epoch$
9: **end while**

Note that the weight matrix is initalized to small random values, the † operator in line 6 denotes the pseudoinverse of a matrix, and that the ÷ operator in line 5 denotes element-wise division of two matrices. It should also be noted that the pseudoinverse of the input matrix can and should be precalculated since it is an expensive cubic time complexity operation and the $inputMatrix$ never changes for a single layer.

3. Multilayer Regression Training

A single layer of a neural network can be expressed as the matrix equation:

$$Ax = B \tag{3.1}$$

where A is a matrix such that each row represents on example in the training set, x is the weight vector (or matrix for multiple neurons), and B is the matrix of desired outputs. Since A and B are known prior to training, if there is a unique solution for linearly sepearble data, it is trivial to solve for x, but this is rarely the case. For more complex systems, such as multilayer neural networks, there is no known closed solution for determining x. Thus an iterative approach is taken.

Andersen and Wilamowski use the pseudoinverse of the input matrix to update the weights of their single-layer network. This works for a single-layer network because the inputs and outputs to the one layer are both known. In a multilayer neural network, the inputs to the first layer, and the outputs to the final layer are known, but the intermediate values are not. For Andersen and Wilamowski's approach to be extended to multilayer neural networks, a way to approximate the 'desired inputs' is needed.

Backpropagation uses the Delta Rule to estimate the impact that hidden weights have on the output of successive layers. The Delta Rule assumes that the inputs to a layer are correct and modifies the weights accordingly. This implies that all of the weights in all previous layers are correct (since they generated the 'desired outputs/inputs'). The Delta Rule then uses these assumptions to update all the weights of the neural network based on their estimated effect on the gradient of the error function. By changing the weights, however, the outputs of each layer is changed, which means the inputs to the successor layers are also changed. (Note: Since the Delta Rule uses the derivative of the activation function in the calculation of the error gradient, this method is agnostic of the choice of activation function as long as the chosen function is differentiable.)

What if, however, these assumptions are inverted to assume that the weights in the current layer are correct, but all other weights need adjusting? Changing the inputs to a layer accomplishes this since those inputs are a function of all of the previous weights. By changing the previous layer's target outputs for each layer, you generate both the desired inputs and desired outputs for hidden layers. Now the Andersen/Wilamowski method can be applied to adjust the weights to achieve the new desired outputs.

The proposed approach will modify the Delta Rule to estimate the 'desired inputs' such that the pseudoinverse of the estimated matrix and the calculated outputs can be used in the same manner as Andersen and Wilamowski's method to directly update the weights of hidden layers as well as output layers.

3.1 Modified Delta Rule

The Delta Rule that is used in backpropagation training locally approximates the partial derivative of the error function with respect to a single weight. Once a delta is calculated using equation 3.2, it can be used to update the individual weight the delta was derived from.

$$\Delta w_{ji} = \alpha(t_j - y_j)g\prime(h_j)x_i \tag{3.2}$$

In equation 3.2, α is a small constant called the learning rate. It controls the speed of learning and improves convergence. If α is high, initial training error drops quickly, but training may not be able to reach the target error threshold either due to the presence of a local minima or the weight updates overcompensating and moving far past the target. If α is too low, however, training procedes at a crawl and the target error threshold may not be achieved in a reasonable amount of time. The remaining variables t_j, y_j, h_j, and x_i are the target output for node j, the actual output of node j, the weighted sum of node j, and input i respectively. Since $g(x)$ is the activation function, $g\prime(x)$ is the derivative of the activation function.

Since the multilayer regression algorithm estimates the inputs to each layer, the Delta rule needs to be modified to calculate the change in the inputs rather than the change in the weights. With input deltas, the algorithm can move the inputs to make it easier for a layer to separate classes. Then the weights can learn the decision boundary to achieve the network's desired results.

$$E = \frac{1}{2}\sum_{j}(t_j - y_j)^2 \tag{3.3}$$

Equation 3.3 defines the error of a network. The Delta Rule examines the partial derivative of the error with respect to each weight to adjust the weights. By taking the partial derivative of the error with respect to each input and intermediate activation value, we can estimate the desired inputs to each layer, so that training via the A&W method can procede. The new rule is derived by first writing the partial derivative of the error with respect to the desired input.

$$\frac{\partial E}{\partial x_i} = \frac{\partial \frac{1}{2}\sum_j (t_j - y_j)^2}{\partial x_i} \tag{3.4}$$

For the purposes of illustration the summation is then expanded and the derivative of the sum is changed into a sum of derivatives. The constant is also factored out.

$$\frac{\partial E}{\partial x_i} = \frac{1}{2}[\frac{\partial (t_1 - y_1)^2}{\partial x_i} + \cdots + \frac{\partial (t_j - y_j)^2}{\partial x_i}] \tag{3.5}$$

Next, the chain rule is applied to simplify the derivative of the difference term by putting it in terms of y.

$$\frac{\partial E}{\partial x_i} = \frac{1}{2}[\frac{\partial (t_1 - y_1)^2}{\partial y_1}\frac{\partial y_1}{\partial x_i} + \cdots + \frac{\partial (t_j - y_j)^2}{\partial y_j}\frac{\partial y_j}{\partial x_i}] \tag{3.6}$$

Now that the difference term is easily derivable, the derivative is taken and the new constant that is created is factored out of the sum and combined with the current constant.

$$\frac{\partial E}{\partial x_i} = -[(t_1 - y_1)\frac{\partial y_1}{\partial x_i} + \cdots + (t_j - y_j)\frac{\partial y_j}{\partial x_i}] \tag{3.7}$$

The summation is recombined and the chain rule is applied again to put the partial derivative of y in terms of x.

$$\frac{\partial E}{\partial x_i} = -\sum_j (t_j - y_j)\frac{\partial y_j}{\partial h_j}\frac{\partial h_j}{\partial x_i} \tag{3.8}$$

The derivative of y in terms of h, the sum weighted sum of a node, can now be evaluated. This is simply the derivative of the activation function.

$$\frac{\partial E}{\partial x_i} = -\sum_j (t_j - y_j)g'(h_j)\frac{\partial h_j}{\partial x_i} \tag{3.9}$$

The remaining unevaluated term is the partial derivative of the weighted sum with respect to an input. Since the weighted sum is a linear combination of variables, most of which are held constant when considering only the ith term, the derivative simply becomes the weight.

$$\frac{\partial E}{\partial x_i} = -\sum_j (t_j - y_j)g'(h_j)w_{ji} \tag{3.10}$$

The resulting expression is multiplied by the learning rate, α, to achieve the final formula for the modified Delta Rule.

$$\Delta x_i = \alpha \sum_j (t_j - y_j)g'(h_j)w_{ji} \tag{3.11}$$

3.2 Multilayer regression algorithm

The new regression-based multilayer neural network training algorithm presented here uses the A&W algorithm as a subroutine to train each layer. To utilize the A&W algorithm, however, the inputs and outputs to each layer being trained need to be known prior to the

execution of the A&W subroutine. The inputs to the first layer are known and the outputs to the last layer are known for the training set however, the target values of the intermediate input/output connections between layers are unknown. This makes the use of the A&W algorithm tricky.

Each layer of a multilayer neural network transforms and projects input features into higher or lower dimensional spaces to separate the classes in the data to make it easier for later layers to identify what data belongs to which class. Ideally, the transformations would linearly separate the data to allow the final layer to simply classify the data. Because these transformations are often complex and the specifics of them are unknown prior to training, the training algorithm of a neural network automatically learns these transformations. Since the A&W algorithm requires the inputs and outputs of the currently training layer to be known, the outputs of each transformation must be estimated. Note that since the outputs of each layer are the inputs to the subsequent layer, or conversely that the inputs to each layer are the outputs from the previous layer, once the inputs or outputs to a layer are estimated, they can be fixed for that iteration of training and used in the training on previous/next layer's training.

The presented algorithm works backwards, estimating each layer's desired inputs using the modified Delta Rule and then training the layer with the A&W algorithm. Since the inputs to each layer are being fixed as the algorithm works its way backwards, it operates under the assumption that minimizing each layer's error towards its desired outputs, the entire network's error will be minimized. This turns out to be a somewhat flawed assumption due to the non-linearity to which each layer's error is subject. It was observed that each layer could have a small error, but the aggregate over the whole network was large. This is due to the coordination of intermediate inputs and outputs. Unless the previous layer can exactly learn its desired outputs, the error associated with its learning gets propagated to the successive layers. So while each layer can have a small error while transforming its desired inputs towards target outputs, it most likely will not recieve the exact inputs it is expecting. In spite of this, the network as a whole does converge on the target error.

Algorithm 3 Regression training algorithm

```
1: Initialize weights, epoch, maxEpochs, error, and threshold
2: while epoch < maxEpochs do :
3:     Pass the training set through the network and record the outputs of each layer.
4:     Calculate the error of the network.
5:     if error < threshold then
6:         Stop training
7:     end if
8:     for each layer i = n : 2 do
9:         Use the modified Delta Rule to adjust the inputs to the layer.
10:        Record the new inputs as the expected outputs to layer i-1.
11:        Train layer i with A&W method with adjusted inputs and expected outputs.
12:    end for
13:    Train layer 1 with A&W method with the inputs and layer 1's expected outputs.
14:    Increment epoch
15: end while
```

3.3 Normalization

The modified Delta Rule looks at a layer's error, weights, and outputs and determines which direction to move the inputs to that layer such that the error is decreased. Adjusting the inputs using the modified Delta Rule, however, tends to produce numbers for the desired inputs to the layer that are larger or smaller than the activation function of the previous layer can produce. Therefore, the inputs must also be normalized into the range of the previous layer's activation function before the layer is trained. Each input to the layer is then recorded for propagation to earlier layers.

The normalization process also has a benefit of breaking a cycle in the logic of the training. The basic premise of the regression algorithm is to estimate what input values the layer wants to be able to produce the desired outputs. The inputs are then set to these desired values and the weights are adjusted to use the new input values. If the input values were chosen based on the weights and the error of the layer, however, the weights shouldn't need

to be changed. The normalization procedure, however, squashes the inputs into the range of the activation function, and changes what weights are needed to produce the desired outputs.

Algorithm 4 Input normalization algorithm used in Algorithm 3.

for all $feature$ **in** $inputs$ **do**
    $max \leftarrow argmax(feature)$
    $min \leftarrow argmin(feature)$
    if $min < 0$ **then**
        Add $|min|$ to each $feature$
        $max \leftarrow max - min$
    end if
    if $max > 1$ **then**
        Divide each $feature$ by max
    end if
end for

The normalization of the inputs to hidden layers happens after the inputs are estimated based on the current weights and the error of the layer. Since the estimation procedure doesn't take into account the range of the previous layer's activation function, the normalization process corrects any out-of-range violations.

The normalization procedure looks at the columns of the input matrix and finds the maximum and minimum value in each column. It then shifts the data in each column by the minimum value for that column. Next the data in each column is scaled by the maximum of that column. The shifting and scaling are only necessary if the values are out of range. Therefore, those operations are only applied if the minimum value is below zero or the maximum value is above one respectively. Since the normalization transformation is linear, the structure of the data is maintained and the classification goals are not obfuscated.

4. Experiments

To determine if this approach gains any advantage over backpropagation, experiments were performed to compare training times over well known datasets. The University of California, Irvine (UCI) Machine Learning Repository has a wide variety of such datasets, including the famous Iris dataset. Benchmarks were be collected to compare the algorithms on datasets of varying size and complexity.

For consistency and simplicity, the sigmoid function was used as the activation function during these experiments.

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{4.1}$$

The sigmoid function was used for its non-linearity with asymptotes approaching $y = 0$ at negative infinity and $y = 1$ at positive infinity.

4.1 Datasets

Neural network training datasets are typically classified as one of three types: artificial, realistic, and real. Artificial or synthetic datasets have characteristics that are exactly known. The datasets tend to be small with discrete variables and the entire dataset is defined and provided. Realistic datasets are modeled after real-world data. Typically realistic datasets are created when collection of real data is difficult or expensive, for example in medical

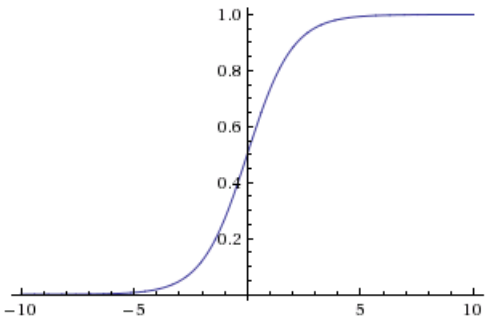


Figure 4.1: Plotted sigmoid function.

applications. Realistic dataset tend to be well behaved and can sometimes be generated algorithmically as needed. Real datasets are collected from real-world events. These datasets may not be governed by well-defined or well-behaving distributions, and therefore tend to be harder for machine learning algorithms to digest. Their values are usually continuous with a higher likeliness of outliers and the distributions underlying them may be dynamic and change over time.

Since no single category of dataset can provide a complete picture of an algorithm's performance, datasets from each category were chosen to be tested on. These datasets have varying numbers of examples and features and were used to test the multilayer regression training algorithm and to compare its performance with backpropagation. The details of the datasets are outlined in this section.

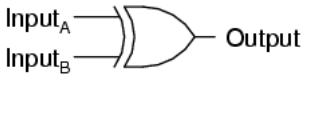
4.1.1 XOR

Number of inputs: 2

Number of outputs: 1

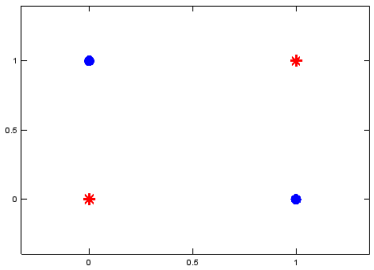
Number of examples: 4

Description: The XOR dataset represents the function of the logical exclusive disjunction operation. The output of an XOR gate is 1 if exactly one of the inputs is a 1. The XOR dataset is a very simple example of non-linear separability and a classic test of the validity of a neural network training algorithm.



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

(a) An XOR gate and truth table.



(b) Plotted XOR dataset.

Figure 4.2: The XOR dataset.

4.1.2 3-bit Parity

Number of inputs: 3

Number of outputs: 1

Number of examples: 8

Description: Parity is calculated by taking the exclusive or of all of the bits or, equivalently, adding all of the bits modulus 2. Essentially, parity is counting the number of bits that are on and determining if the count is odd or even. The output is 0 if the parity is even and 1 if the parity is odd.

4.1.3 5-bit Majority

Number of inputs: 5

Number of outputs: 1

Number of examples: 32

Description: The majority dataset counts the zeros and ones in the input bitstring and determines which the bitstring has more of. The output is 0 if most of the bitstring is 0's and 1 otherwise.

4.1.4 3-bit Decoder

Number of inputs: 3

Number of outputs: 8

Number of examples: 8

Description: The decoder data set takes a bitstring and determines the decimal value. Instead of a continuous range, each possible value for the fixed-length bitstring is given its own class and output node.

4.1.5 Iris

Number of inputs: 4

Number of outputs: 3

Number of examples: 150

Description: This is perhaps the best known database to be found in the pattern recognition literature. It was first used in R. A. Fisher's 1946 paper The Use of Multiple Measurements in Taxonomic Problems, which is still widely cited. The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are NOT linearly separable from each other. The inputs are measurements of sepal and petal lengths and widths.

4.1.6 Scale Balancing

Number of inputs: 4

Number of outputs: 3

Number of examples: 625

Description: The goal of this dataset is to provide examples such that a system can learn the function behind determining which direction a scale will tip. It is based on a psychological experiment done with children in 1978. The inputs are the weight on each side of the scale and how far each weight is placed from the pivot. The outputs are the three classes representing the resulting motion of the scale: tip to the left, balanced, and tip to the right.

Each of the classes is represented by an output node that should have a value of 1 if that class is chosen and 0 otherwise.

4.1.7 Inflammation Diagnosis

Number of inputs: 6

Number of outputs: 2

Number of examples: 120

Description: This dataset was intended to train an expert system to diagnose two diseases of the urinary system: acute inflammation of the urinary bladder and acute nephritis of renal pelvis origin. Only one of the six inputs was a continuous variable; the rest were yes/no values indicating the presence of non-quantitative symptoms. These values were translated to 0's and 1's for noes and yesses respectively. The outputs also were translated from yesses and noes representing the diagnosis of the two diseases.

4.1.8 Banana-shaped Data

Number of inputs: 2

Number of outputs: 1

Number of examples: 200

Description: The banana-shaped data is so named for the appearance of two classes of curved interlocking data. The points are sampled along a circle or ellipse and offset by a small random amount to add some variance. The classes are not linearly separable but do not overlap. This makes banana-shaped data a good test for a multilayer neural network's ability to learn non-linear decision boundaries.

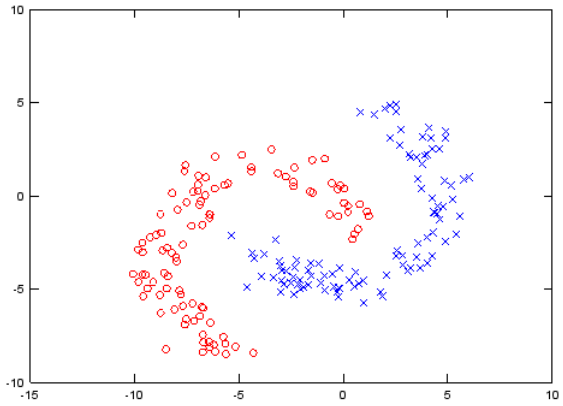


Figure 4.3: Two classes of interlocking banana-shaped data.

4.2 Methodology

Both backpropagation and the multilayer regression algorithm were run ten times on each dataset. The network structure for each dataset was fixed for both algorithms and the initial weights were randomized before each training run. The number of training epochs, the total training time, the final error, and the number of misclassified patterns were recorded for each run. The average and standard deviation for each metric was then calculated as well as the average time it takes for a single iteration. The raw data for all experiments can be found in Appendix A.

The Encog machine learning library implementation of backpropagation was used to train the neural networks during the backpropagation experiments. Encog is an advanced neural network and machine learning framework that has been in active development since 2008. The Encog library contains a configurable implementation of backpropagation, but for these experiments, the algorithm was limited to its simplest form. Dynamic learning rates were turned off and the learning rate was fixed at 0.05 for all experiments for both backprop and regression. The result is the backpropagation algorithm shown in Algorithm 1 with $\alpha = 0.05$.

The Encog library automatically detects multi-core CPU architectures and creates multiple threads to parallelize training. Since the regression based training is not parallelized,

the backpropagation training runs were forced to run on a single core by using the affinity flag when running the programs from the Microsoft Windows command prompt. This ensures that both algorithms are running on a single CPU and thus their performances are easier to compare.

The Encog library also uses mean-squared error (MSE) as the default error metric for neural networks. Therefore, for the sake of consistency, MSE was also used by the regression algorithm.

4.3 Validity

Before the regression algorithm can be evaluated for speed-ups over backpropagation, it must first be tested to ensure that it does learn complex, non-linear decision boundaries correctly. The correctness of the algorithm is difficult to prove, however, the soundness of the logic behind the algorithm and the effectiveness of the implementation can be observed empirically.

The XOR, parity, majority, and decoder datasets were used to test the multilayer regression training algorithm's validity. If an algorithm can not learn these synthetic datasets, it is unlikely that the algorithm will perform well on real or realistic data. The bitwise data was also used to test validity since the entire sets were known, small, and well defined.

4.4 Performance

Once the validity of the algorithm had been established through testing on the simple synthetic problems, experimentation broadened to examine the performance of the algorithm in comparison to backpropagation. All datasets were used in performance testing and the training was set to achieve an error rate of less than 0.01. An additional maximum number of iterations stopping condition was used on a few problems for the sake of time. The inflammation diagnosis and banana-shaped datasets, for example, were allowed to train for

a maximum of 100,000 iterations when training with backpropagation and 1,000 iterations for regression. This forced the algorithm to stop when it might not have otherwise.

5. Results and Discussion

The results of the experiments are presented in this section. The figures in this section summarize the collected data over four metrics: training time, number of epochs, misclassified patterns, and network error. As illustrated in the figures and tables, the presented regression-based algorithm out-performed backpropagation over most of the datasets for each metric. Note that the training time, epochs, and misclassified patterns charts are on a log-scale axis.

5.1 Validity

The validity of the regression-based algorithm was confirmed by training multilayer networks to learn the synthetic XOR, 5-bit majority, 3-bit parity, and 3-bit decoder datasets. The algorithm was observed to correctly train a network to learn these datasets entirely. Since the algorithm successfully learned the complex non-linear decision boundaries of these datasets, experimentation moved on to the performance phase.

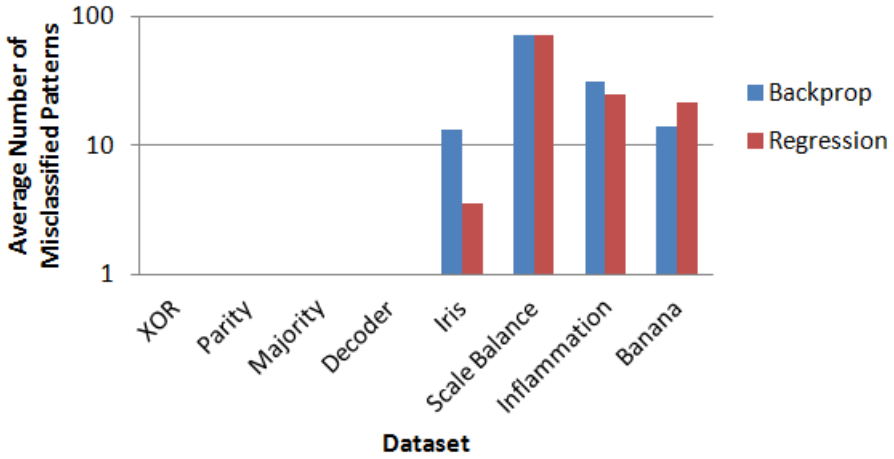


Figure 5.1: Average number of misclassified patterns per dataset.

5.2 Performance

The goal of investigating a regression-based algorithm was to reduce training time while maintaining or even improving on the accuracy of the resulting network. When considering the errors of the networks that resulted from training, the regression-trained networks achieved lower resulting error than backpropagation on all but two datasets: Scale Balancing and Banana-shaped Data. The number of misclassified patterns by the regression-trained networks, however, were still equal or close to the number misclassified by backpropagation over all datasets, including those which regression did not achieve better error than backpropagation. This indicates that the regression algorithm is possibly producing a more accurate model of the data. A few drastically misclassified outliers in the regression training could drive up the error while still classifying most patterns correctly.

Each measure for both methods was tested for statistically significant differences to validate the observed improvements were actual improvements and not just due to noise between trial runs. T-tests were performed on the data collected for each measure for each data set to determine if the regression algorithm statistically outperformed backpropagation and these values are shown in table 5.3. Using a confidence interval of 95% ($p = 0.05$) the training time, number of iterations, and error were found to be statistically different in most cases. The number of misclassified patterns, however, was not statistically different for most datasets. In general, the training times went down for the regression algorithm and the number of misclassified patterns stayed the same or improved. This was the goal of the investigation, to decrease training times without negatively impacting accuracy, and this goal was met within a 95% confidence interval.

The regression-based algorithm greatly out-performed backpropagation in terms of training time. There were only two datasets where regression training was slower: Iris and Scale Balance. The cost of extra time on the Iris dataset paid off, however, with fewer misclassified patterns from the resulting network. There was no difference, though, in the number of misclassified patterns over the Scale Balance dataset. Initially, the size of the Scale Balance dataset was the suspect for its decreased performance since it had more than

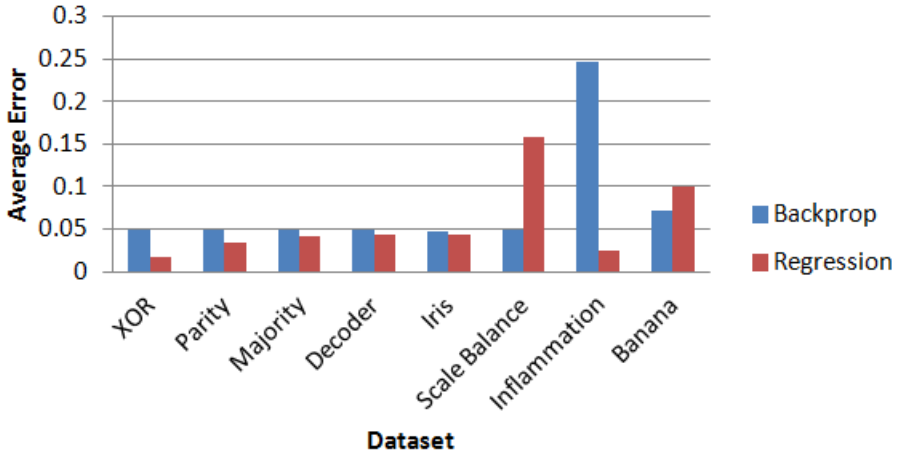


Figure 5.2: Average error per dataset.

three times the number of examples than the next largest dataset and the matrix operations are of cubic complexity. However, reduction in the size of the dataset by using only the first 500 or 375 examples did not improve the performance ratio between the two training methods which should be expected, based on the performance of the other datasets, if the number of examples was the offending variable.

It could be that the non-bounded nature of the Scale Balance data makes it more difficult for the regression to learn. Other datasets like Iris and Inflammation have continuous inputs that can be modeled with a probability distribution. While not explicitly bounded, the data in these sets is more concentrated and less likely to go beyond a certain range of values. The Scale Balance dataset, which can use any positive values, is explicitly boundless and extrapolation beyond the values contained in the training set may be 'confusing' the algorithm.

The poor performance in some instances indicates that datasets exist that are more difficult for the regression-based algorithm to learn than other training methods. This is also indicated by the results of the experiments using the Banana-shaped data. This dataset appeared difficult for both backpropagation and regression alike, with both algorithms reaching their maximum allowed number of training epochs and neither achieving the desired

error threshold. Further experimentation is needed, however, to tease out the exact characteristics which determine if a problem will be difficult for regression training.

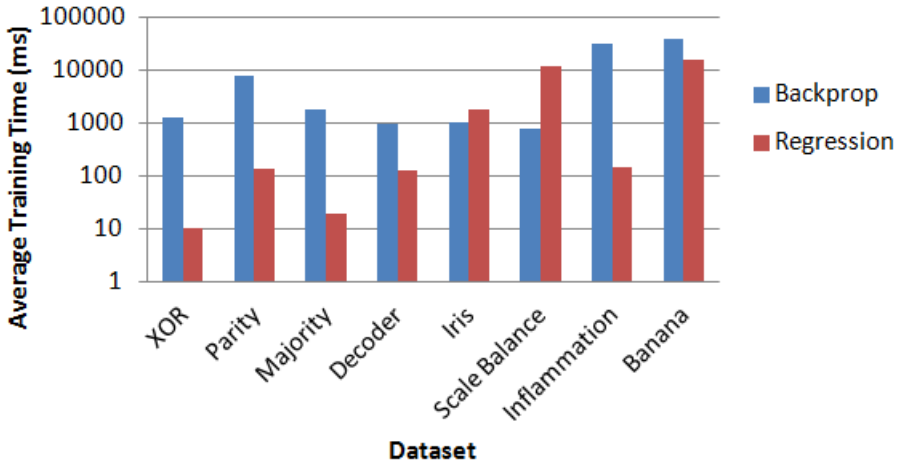


Figure 5.3: Average training time per dataset.

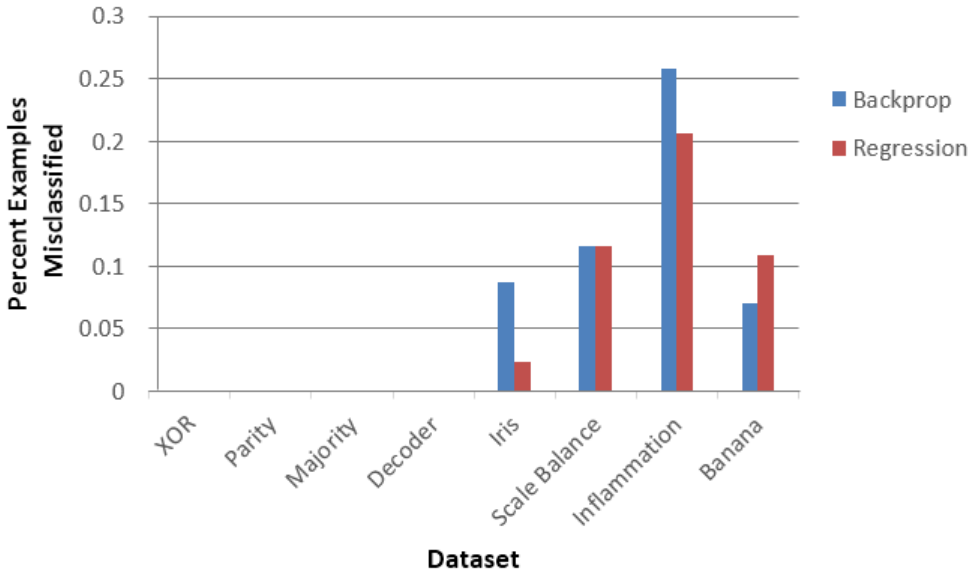


Figure 5.4: Average percentage of missed patterns per dataset.

The regression-based algorithm's gains in training time over backpropagation can be attributed to the decreased number of training epochs. In general, the number of epochs used by regression training was orders of magnitude less than backpropagation. The average time used per iteration, however, was greater for the regression algorithm. The matrix operations that are used in each iteration of regression training have cubic time complexities and so a greater time per iteration is expected. If the matrix algorithms and other aspects of the algorithm are upgraded to state-of-the art or can be further optimized, the time per iteration will decrease and thus performance will further increase.

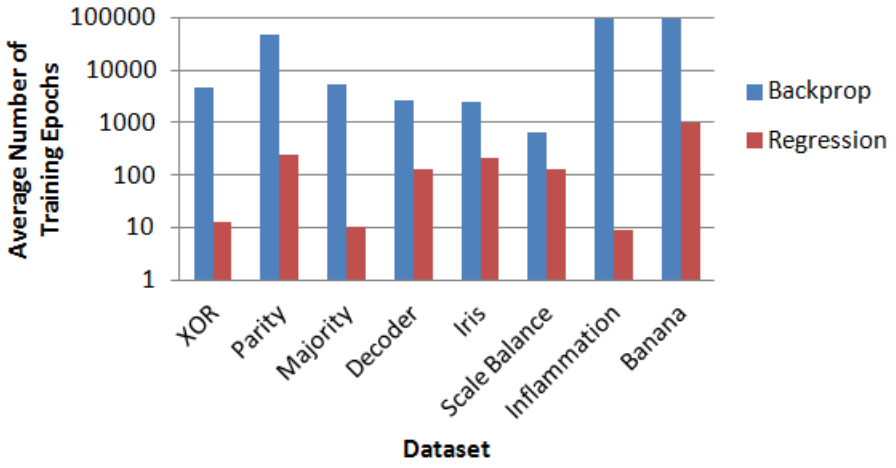


Figure 5.5: Average epochs per dataset.

Dataset	Training time (ms)	Epochs	Missed patterns	Error
XOR	1,261.1	4,790.8	0	0.05
Parity	7,925.5	47,568.1	0	0.05
Majority	1,767.2	5,564.9	0	0.05
Decoder	977.6	2,587.5	0	0.05
Iris	1005.3	1,550.7	13.1	0.0476
Scale Balance	797.2	653.6	72.2	0.0492
Inflammation	31,005.2	100,000	31	0.2465
Banana	40,261.8	100,000	14	0.071

Table 5.1: Backpropagation training results averaged over the 10 trials for each dataset.

Dataset	Training time (ms)	Epochs	Missed patterns	Error
XOR	10.1	12.9	0	0.0177
Parity	136.4	252.5	0	0.0339
Majority	18.8	10.4	0	0.0416
Decoder	129.8	130.7	0	0.0442
Iris	1,844.4	205.2	3.5	0.0434
Scale Balance	12,199.8	129.1	72.6	0.1573
Inflammation	144.2	8.8	24.7	0.0245
Banana	15,843.2	1,000	21.8	0.1005

Table 5.2: Regression training results averaged over the 10 trials for each dataset.

Dataset	Training time (ms)	Epochs	Missed patterns	Error
XOR	7.26E-8	1.74E-7	-	0.0003
Parity	0.0104	0.0196	-	0.0096
Majority	4.85E-8	1.07E-7	-	0.0012
Decoder	4.41E-15	1.79E-9	-	0.0005
Iris	0.1556	0.0061	0.0004	0.1556
Scale Balance	0.0005	0.1349	0.8751	3.86E-18
Inflammation	1.48E-24	2.29E-38	0.1535	4.99E-11
Banana	4.64E-19	-	4.54E-7	1.01E-6

Table 5.3: T-test results ($p = 0.05$) for each metric of each dataset between backprop and regression training.

6. Future Work

The presented regression-based algorithm has shown some success over backpropagation in training multilayer neural networks to classify raw data. Future work on this regression-based algorithm could include further optimization and parallelization, a stochastic batch training version, or an even more in depth analysis of the performance metrics. Each of this is discussed in the following sections.

6.1 Optimization and Parallelization

The matrix multiplication and inversion operations used by the regression algorithm are naive ones. As such they have $O(n^3)$ running time. The fastest known running time for these operations are $O(n^{2.373})$. The faster matrix algorithms can be substituted for a potentially substantial speedup. Faster algorithms tend to have larger coefficients, however, and an in-depth analysis could be done of when or with what size datasets it is beneficial to use the faster matrix operations.

There are also several parallelizable loops, including those in the matrix operations, that can be threaded to achieve speedups as well. An investigation of which parallel structures and how many threads should be used for maximal speedups could be performed. There are parallel versions of backpropagation already and so a comparison of speedup ratios can also be performed.

6.2 Stochastic Batch Training

Since the matrix multiplication and inversion operations are of cubic complexity, as datasets get large, the running time of the regression algorithm grows rapidly. To attempt to reduce training time, small batches from the training can be used to update the weights. By randomly selecting small batches, the matrix operations should remain fast while moving the

weights in the correct direction. A validation set representative of the population could then be used to stop training when the network has learned the patterns. The size of the batches and even methods for selecting those batches could be experimented with and their effects determined.

6.3 Deeper Analysis

This thesis provided an initial investigation into a new regression-based training algorithm for multilayer neural networks. Good science involves not only introducing new results, but having them verified, confirmed, and expanded on. Thus, future work could include a deeper and broader analysis of the algorithm, possibly with comparison to other algorithms than backpropagation, finer tuning of training parameters, and/or larger, more complex, or otherwise interesting datasets.

A. Training Data

This appendix contains the raw data collected during the experiments with each dataset for both training algorithms as well as averages and standard deviations of the collected data.

A.1 Backpropagation training data

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	1001	3549	0	0.05
2	1610	6694	0	0.05
3	1052	3864	0	0.05
4	1442	5199	0	0.05
5	1152	4791	0	0.05
6	1139	4801	0	0.05
7	1375	5245	0	0.05
8	865	3064	0	0.05
9	1528	5568	0	0.05
10	1447	5133	0	0.05
Average	1261.1	4790.8	0	0.05
Std. Dev.	251.2867	1059.308	0	7.3E-18
Miliseconds per iteration	0.2632			

Table A.1: Backpropagation: XOR results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	12253	75169	0	0.05
2	5168	28196	0	0.05
3	4742	25387	0	0.05
4	4314	23302	0	0.05
5	7055	42601	0	0.05
6	3759	18595	0	0.05
7	4759	25911	0	0.05
8	5311	29590	0	0.05
9	28454	189966	0	0.05
10	3440	16964	0	0.05
Average	7925.5	47568.1	0	0.05
Std. Dev.	7645.097	52797.37	0	7.3E-18
Miliseconds per iteration	0.1666			

Table A.2: Backpropagation: Parity results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	1873	6296	0	0.05
2	1278	3898	0	0.05
3	2136	6298	0	0.05
4	1300	3851	0	0.05
5	1875	6256	0	0.05
6	1915	6307	0	0.05
7	1872	6294	0	0.05
8	2198	6300	0	0.05
9	1329	3885	0	0.05
10	1896	6264	0	0.05
Average	1767.2	5564.9	0	0.05
Std. Dev.	340.4973	1164.238	0	7.3E-18
Miliseconds per iteration	0.3176			

Table A.3: Backpropagation: Majority results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	1080	3115	0	0.05
2	994	2740	0	0.05
3	930	2311	0	0.05
4	966	2647	0	0.05
5	802	1893	0	0.05
6	1059	3100	0	0.05
7	1031	2277	0	0.05
8	1005	2609	0	0.05
9	978	2684	0	0.05
10	931	2499	0	0.05
Average	977.6	2587.5	0	0.05
Std. Dev.	79.1555	371.2604	0	7.3E-18
Miliseconds per iteration	0.3778			

Table A.4: Backpropagation: Decoder results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	886	1385	16	0.0488
2	740	924	13	0.0474
3	915	1423	20	0.0482
4	1375	2562	14	0.0493
5	2218	4537	20	0.044
6	610	518	10	0.0489
7	917	1395	19	0.0481
8	580	648	7	0.0487
9	653	819	8	0.048
10	1159	1296	4	0.0441
Average	1005.3	1550.7	13.1	0.0476
Std. Dev.	493.5999	1196.433	5.7242	0.0019
Miliseconds per iteration	0.6483			

Table A.5: Backpropagation: Iris results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	695	371	77	0.0489
2	1009	852	71	0.05
3	312	155	78	0.0496
4	423	252	79	0.0489
5	3062	3462	63	0.0492
6	584	340	77	0.0496
7	587	322	77	0.0483
8	260	130	64	0.0492
9	354	192	63	0.0491
10	686	460	73	0.0493
Average	797.2	653.6	72.2	0.0492
Std. Dev.	826.4436	1008.257	6.5625	0.0005
Miliseconds per iteration	1.2197			

Table A.6: Backpropagation: Scale balancing results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	30867	100000	31	0.2465
2	31434	100000	31	0.2465
3	31164	100000	31	0.2465
4	31084	100000	31	0.2465
5	30943	100000	31	0.2465
6	30987	100000	31	0.2465
7	30924	100000	31	0.2465
8	30866	100000	31	0.2465
9	30532	100000	31	0.2465
10	31251	100000	31	0.2465
Average	31005.2	100000	31	0.2465
Std. Dev.	246.9596	0	0	0
Miliseconds per iteration	0.3101			

Table A.7: Backpropagation: Inflammation diagnosis results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	40493	100000	14	0.0712
2	40273	100000	14	0.0708
3	40435	100000	14	0.0711
4	39813	100000	14	0.0709
5	39592	100000	14	0.0711
6	40290	100000	14	0.0708
7	39750	100000	14	0.0712
8	41408	100000	14	0.0709
9	40864	100000	14	0.0714
10	40060	100000	14	0.0709
Average	40261.8	100000	14	0.07103
Std. Dev.	473.5027	0	0	0.0002
Miliseconds per iteration	0.4026			

Table A.8: Backpropagation: Banana results

A.2 Regression training data

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	3	1	0	0
2	24	46	0	0.0377
3	4	4	0	0.0215
4	24	15	0	0.0412
5	16	25	0	0.0403
6	5	4	0	0.0289
7	2	1	0	0
8	8	12	0	0
9	7	11	0	0.0014
10	8	10	0	0.0059
Average	10.1	12.9	0	0.0177
Std. Dev.	8.2926	13.7473	0	0.0181
Miliseconds per iteration	0.782946			

Table A.9: Regression: XOR results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	99	113	0	0.0107
2	230	605	0	0.0331
3	88	111	0	0.0386
4	70	35	0	0.0427
5	110	171	0	0.0011
6	248	544	0	0.0401
7	178	431	0	0.0447
8	113	147	0	0.036
9	94	181	0	0.0418
10	134	177	0	0.0497
Average	136.4	252.5	0	0.0339
Std. Dev.	61.594	200.664	0	0.0156
Miliseconds per iteration	0.5402			

Table A.10: Regression: Parity results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	6	3	0	0.0433
2	3	1	0	0.036
3	6	3	0	0.0409
4	6	3	0	0.0462
5	5	2	0	0.0411
6	23	5	0	0.0491
7	105	77	0	0.0427
8	21	4	0	0.0483
9	7	3	0	0.0309
10	6	3	0	0.0375
Average	18.8	10.4	0	0.0416
Std. Dev.	31.0691	23.4246	0	0.0057
Miliseconds per iteration	1.8077			

Table A.11: Regression: Majority results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	121	115	0	0.0469
2	143	122	0	0.0411
3	146	154	0	0.0461
4	100	92	0	0.0404
5	82	50	0	0.0451
6	104	79	0	0.043
7	118	132	0	0.0445
8	161	181	0	0.0488
9	59	36	0	0.0381
10	264	346	0	0.0479
Average	129.8	130.7	0	0.0442
Std. Dev.	56.2609	87.8143	0	0.0035
Miliseconds per iteration	0.9931			

Table A.12: Regression: Decoder results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	309	24	5	0.0475
2	2186	244	2	0.0269
3	685	56	5	0.0432
4	2442	275	2	0.0477
5	744	71	4	0.0486
6	2816	326	5	0.0457
7	538	46	3	0.0472
8	617	59	4	0.0485
9	5727	678	1	0.0292
10	2380	273	4	0.0496
Average	1844.4	205.2	3.5	0.0434
Std. Dev.	1663.497	202.4653	1.4337	0.0083
Miliseconds per iteration	8.9883			

Table A.13: Regression: Iris results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	13486	141	72	0.1545
2	6945	72	78	0.1561
3	25133	269	69	0.1584
4	3328	33	68	0.1545
5	22315	239	71	0.1585
6	6507	69	67	0.1548
7	11893	126	70	0.1594
8	12240	129	80	0.1583
9	10518	111	76	0.1591
10	9633	102	75	0.1597
Average	12199.4	129.1	72.6	0.1573
Std. Dev.	6827.084	73.7646	4.4272	0.0021
Miliseconds per iteration	94.4988			

Table A.14: Regression: Scale balancing results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	130	9	7	0.0393
2	298	14	37	0.0001
3	128	5	9	0.0141
4	31	1	17	0.0124
5	276	28	34	0.0468
6	86	4	26	0.005
7	154	7	31	0.0452
8	33	1	10	0.0036
9	115	6	39	0.0492
10	191	13	37	0.029
Average	144.2	8.8	24.7	0.0245
Std. Dev.	90.1823	8.0526	12.7806	0.0196
Miliseconds per iteration	16.3864			

Table A.15: Regression: Inflammation diagnosis results

Trial	Training time (ms)	Epochs	Missed patterns	Error
1	15968	1000	25	0.1104
2	15750	1000	21	0.1077
3	15779	1000	23	0.1107
4	16060	1000	20	0.0917
5	15760	1000	24	0.1014
6	16030	1000	20	0.0947
7	15929	1000	19	0.0892
8	15706	1000	21	0.0968
9	15738	1000	23	0.107
10	15712	1000	22	0.0951
Average	15843.2	1000	21.8	0.1005
Std. Dev.	138.1093	0	1.9322	0.008
Miliseconds per iteration	15.8432			

Table A.16: Regression: Banana results

B. Modified Delta Rule Derivation

$$
\begin{aligned}
E &= \frac{1}{2}\sum_j (t_j - y_j)^2 \\
\frac{\partial E}{\partial x_i} &= \frac{\partial \frac{1}{2}\sum_j (t_j - y_j)^2}{\partial x_i} \\
&= \frac{\partial \frac{1}{2}[(t_1 - y_1)^2 + \cdots + (t_j - y_j)^2]}{\partial x_i} \\
&= \frac{1}{2}[\frac{\partial (t_1 - y_1)^2}{\partial x_i} + \cdots + \frac{\partial (t_j - y_j)^2}{\partial x_i}] \\
&= \frac{1}{2}[\frac{\partial (t_1 - y_1)^2}{\partial y_1}\frac{\partial y_1}{\partial x_i} + \cdots + \frac{\partial (t_j - y_j)^2}{\partial y_j}\frac{\partial y_j}{\partial x_i}] && \text{(via chain rule)} \\
&= \frac{1}{2}[\text{-}2(t_1 - y_1)\frac{\partial y_1}{\partial x_i} + \cdots + \text{-}2(t_j - y_j)\frac{\partial y_j}{\partial x_i}] \\
&= -\sum_j (t_j - y_j)\frac{\partial y_j}{\partial x_i} \\
&= -\sum_j (t_j - y_j)\frac{\partial y_j}{\partial h_j}\frac{\partial h_j}{\partial x_i} && \text{(via chain rule)} \\
&= -\sum_j (t_j - y_j) g'(h_j)\frac{\partial h_j}{\partial x_i} \\
&= -\sum_j (t_j - y_j) g'(h_j) w_{ji} \\
\Delta x_i &= \alpha \sum_j (t_j - y_j) g'(h_j) w_{ji}
\end{aligned}
$$

Bibliography

[1] T.J. Andersen and B.M. Wilamowski. A. Modified Regression Algorithm for Fast One Layer Neural Network Training. In *World Congress of Neural Networks*, volume 1, pages 687–690, 1995.

[2] E. Castillo, O. Fontenla-Romero, B. Guijarro-Berdiñas, and A. Alonso-Betanzos. A global optimum approach for one-layer neural networks. *Neural Computation*, 14(6):1429–1449, 2002.

[3] R. A. FISHER. The use of multiple measurements in taxonomic problems. *Annals of Human Genetics*, 7(2):179–188, 1936.

[4] O. Fontenla-Romero, D. Erdogmus, J.C. Principe, A. Alonso-Betanzos, and E. Castillo. Accelerating the convergence speed of neural networks learning methods using least squares. In *11th European Symposium on Artificial Neural Networks (ESANN)*, number April, pages 255–260. Citeseer, 2003.

[5] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[6] H.Zarzycki J.Czerniak. Application of rough sets in the presumptive diagnosis of urinary system diseases. *Artifical Inteligence and Security in Computing Systems*, ACS'2002 9th International Conference Proceedings:41–51, 2003.

[7] N.B. Karayiannis and a.N. Venetsanopoulos. Fast learning algorithms for neural networks. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(7):453–474, July 1992.

[8] D Klahr and R S Siegler. The representation of children's knowledge. In *Advances in child development and behavior (*, pages 61–116, 1978.

[9] Nan-Ying Liang, Guang-Bin Huang, P Saratchandran, and N Sundararajan. A fast and accurate online sequential learning algorithm for feedforward networks. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 17(6):1411–23, November 2006.

[10] Saman Razavi and Bryan a Tolson. A new formulation for feedforward neural networks. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 22(10):1588–98, October 2011.

[11] S Russell and P Norvig. Artificial intelligence: A modern approach, 2003.

[12] B.M. Wilamowski, S. Iplikci, O. Kaynak, and M.O. Efe. An algorithm for fast convergence in training neural networks. *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, 3(2):1778–1782, 2001.