Experiments with BP Variations

Joseph A. Del Rocco
University of Central Florida
EEL 6812
April 29th, 2011

Abstract – **Multi-layer perceptron neural networks (MLP NNs) are discussed throughout this paper. In addition, the MLP back-propagation (BP) learning algorithm, the most well-known and commonly used training algorithm for MLP NNs, proposed in the 1980s and responsible for re-invigorating research on layered NNs, is discussed and experimented with. Three separate variations of error function minimization techniques (GD, GDM, and RPROP) are discussed and then tested by use of a custom-built MLP generation program. MLP models supported by the program include standard feed-forward networks of roughly any number of layers and layer nodes. Successful tests performed against four different classification problems, and one encoder problem, demonstrate the ability of MLPs to generalize effectively after being trained with the back-propagation learning algorithm. Although all three minimization techniques performed well, there are inherent strengths and weaknesses to each.**

I. INTRODUCTION

Neural networks (NNs) are a collection of neurons (*nodes*), each of which perform simple operations, are densely interconnected, and which can be thought of as simulated biological neurons in the human brain. The "strength" (or effectiveness) of the interconnections (*weights* or simulated synapses), reveals the effect that one node has on another node. There are an infinite number of possible NN node-and-weight configurations (termed *models* or networks), each with their own advantages and disadvantages. These models resemble the functionality of the human brain in other ways as well, that knowledge is acquired via learning process, and that the knowledge learned is stored within the interconnection weights [8].

One of the major benefits of NNs includes the ability to model non-linear relationships of input/output data [8], which is obviously useful if the functional relationship is complex and unknown. Simply feeding training data to a well-designed NN enables the possible discovery of relationships that can be generalized across similar, yet unseen, data. In other words, NNs can learn, and can then be used in live situations.

One practical example comes from *LeCun* et al. [5], known throughout pattern recognition literature as the *handwritten digit recognition problem*. The NNs here were able to learn patterns in handwritten digits so well, that they could be used in a production environment such as the post office, to automatically read the zip code digits, handwritten onto letters.

Other benefits of NNs include the ability to be used without in-depth knowledge of the subject matter on which they are being used, as well as fault tolerance, in the sense that some number of nodes can be removed from the model without dramatically degrading the overall effectiveness (though this is more obvious with large distributed systems), inherent parallelization and recurrent learning. This is meaningful for scientists who straddle the line between disciplines; biologists can use efficient, fault-tolerant NN software to model mathematical relationships as easily as computer scientists can use them to model biological relationships. Regarding parallelization, a vast amount of calculations can be tackled in concert per layer of nodes, provided that the model is setup in a certain fashion. Certainly there are configurations that are so intricately interconnected that little-to-no parallelism is exploitable.

Neural networks have been actively researched since the 1950s, initially because of the parallels with neurobiology and the fact that the human brain operates on the order of milliseconds [8], versus computers, which have become increasingly faster. The goal was (is?) to be able to simulate a much faster human brain. Given that the average adult human brain has ~100 billion neurons and ~300 trillion synapses, this has yet to be realized in model environments.

In terms of the *types* of NNs, there are many to choose from, and in general they have been developed over many years of research. A single-layer perceptron (SLP) refers to a network with only two interconnected layers, the *input* and *output* layer, each of which can have an arbitrary number of nodes. A *multi-layer perceptron* (MLP) can have any number of intermediary layers (termed "hidden" layers) between the input and output layer, each of which also consists of an arbitrary number of nodes. How the interconnections (weights) are organized, is completely up to the NN model designer. A *standard feed-forward* network, is one in which each layer of nodes is connected to an immediately higher-indexed layer, while specifically not connected to lower layers, and there are no connections between nodes within the same layer. Thus the information presented in the input layer flows through all the layers of the network in a *feed-forward* manner to the output layer, with no loops or cycles in between. In addition to SLP and MLP, there are various other types of NNs, although their internal structure can be different than indicated so far, including *fuzzy ARTMAPs* (FAM), *radial-based functions* (RBF), *evolutionary neural networks* (ENN), *probabilistic neural networks* (PNN), etc.

The focus of this paper is on MLP NNs, and the most commonly used learning mechanism for this type of network, *back-propagation* (BP). The back-propagation algorithm is heavily referenced throughout NN literature, given that its emergence re-invigorated NN research in the 1980s. Many "improvements" have been proposed, or at least observations noted, to the BP learning method, as it has been applied to thousands of MLP model configurations over the years. This paper attempts to explain BP MLP NNs in-depth, as well as provide some insight into some of the more common variations of weight minimization functions. A program has been implemented which tests these variations and the results are presented below.

This paper is organized as follows: Section II discusses the basics behind a single neuron, Section III discusses MLP structure, Section IV explains the back-propagation learning

algorithm and mathematical definitions, Section V notes considerations that should be kept in mind when applying any learning method, though some of which are specifically applicable to BP here, Section VI details a recent implementation of MLP with BP variations, Section VII shows an example of learning over the simple (but important) XOR model, Section VIII describes the classification problems tested in this paper, Section IX details the results, and Section X is the conclusion. References follow.

II. NEURONS

A NN is composed of nodes and weights, as mentioned above. The neurons (nodes, processors) are essential to the functionality of the network in the sense that they take in some information, process it and produce outputs which are propagated through the network via weights, funneled into the calculations of other nodes, or interpreted as part of the final output of the network. Note this is very similar to actual neurons in neurobiology, which also process information and then transmit electrochemical signals through synapses.

The node model used in NNs is composed of three main pieces: a set of interconnecting weights between the node and other nodes, an *adder* (summation) function that combines the information converging into the node to produce a *net input*, and an *activation* (or *squashing*) function used to limit the amplitude of the output in some fashion. The most commonly used output ranges are either [0, 1] and [-1, 1].

The adder function sums up all the converging weights multiplied by the output values of the nodes from which they are emanating. If the weights are emanating from the *input layer* of the NN, then the pattern input is used as the emanating node output.

The standard neuron (NN node) model is shown below, where net_j refers to the net input into *node j*, calculated from the weights emanating from the nodes in the layer below, times their output values. In this case, the bottom layer output values are not true outputs of neurons themselves, but a set of values from an input vector stuffed into a faux node. When patterns are presented to the NN, the pattern input is inserted into this bottom layer of faux nodes. $g(\ )$ represents the *activation function*, which is limiting the net_j in some fashion depending upon the function itself. y_j refers to the ultimate output value of the neuron.

Note the *bias node* (in orange) which always outputs some constant *threshold* value (typically 1).

Popular activation functions include, but are certainly not limited to, *threshold* functions, *piecewise-linear* functions, the *sigmoid* function, and the *hyperbolic tangent* function.

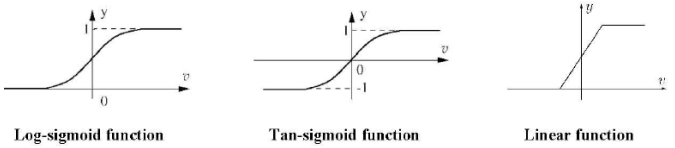

Figure 1: Examples of activation functions. Image borrowed from http://www5.in.tum.de.

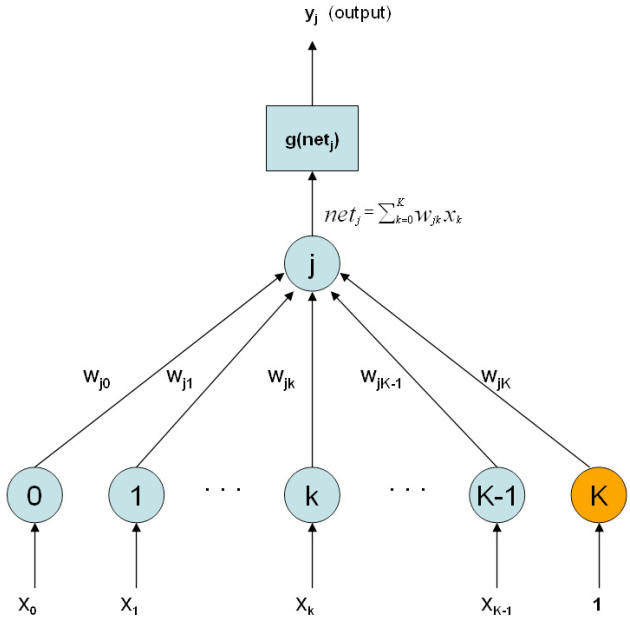

Figure 2: A typical neuron model.

III. MLP

A multi-layered perceptron (MLP) network is one which has support for some number of "hidden" (or intermediary) layers between the input and output layers. Technically speaking there can be any number of hidden layers, although it has been proven theoretically that only one is necessary. These middle layers are called hidden because they are not directly related to the input or output values of the network, but evolve their own values.

Like any neural network, the goal of MLPs is to be able to generalize across unseen data after being trained over a subset of similar data. There are generally two phases to using this network, a training phase, and a testing phase. During the former, for some number of training epochs (iterations, rounds), a set of *training patterns* (input/output pairs) are presented to the network and an output is produced. The goal is to have this actual output match the output of the pattern. This way we know that the network has learned some functional relationship between the input and the output of that particular pattern. Of course we want the network to be able to do this across all training patterns. Once this training phase is complete to satisfaction, the MLP network can be tested against unseen (but similar) data to hopefully be able to do the same thing. This is referred to as *generalization*, since the network is able to generalize over unseen data.

The main reason for the evolution of MLP NNs was the analysis which led to the discovery that SLP NNs were limited in the type of problems they could solve. Specifically [Minsky, Papert 1969] showed that SLPs could not properly classify something as simple as the XOR problem. This discovery actually deeply affected the research of NNs, since SLPs were thought to be limited in scope, and there was no known effective training method for MLP NNs. In fact it

wasn't until the mid-1980s the *back-propagation* learning algorithm was proposed.

The system that accounts for how the output layer nodes represent a final "answer," or actual output when presented with a pattern, is known as an encoding scheme. If there is only one output node, then the range of the value of that particular node determines the solution (e.g. 0 for no, 1 for yes, or -1 for no, 1 for yes, etc.). However if there is more than one output node, then typically (for classification problems) one node converges high, while the rest converge low (e.g. one node is ~1 and the rest ~0, if the sigmoid activation function is used). So a 4-class classification problem could have an output layer dimensionality of 4, and a pattern could be considered classified if one of those 4 nodes converged high and the rest converged low.

For this paper, the following notation is used when discussing MLP NNs:

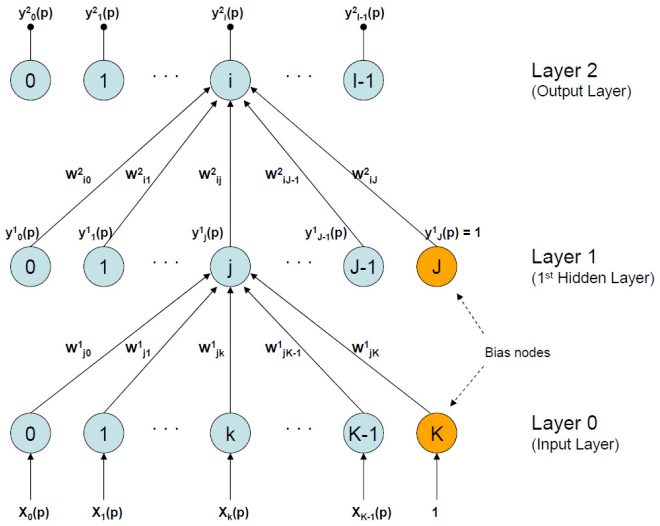

Figure 3: MLP architecture referenced throughout this paper. Note the existence of only one hidden layer, and bias nodes at the *end* of each layer.

First note that the input vector applied to the input layer is denoted by the vector $x(p)$, where p refers to a particular pattern index. So $x_0(p)$ is the first input of the pattern p, $x_0(p)$ is the second and so on till $x_{K-1}(p)$.

Next, note the placement of the bias nodes at the *end* of each layer. As explained later, this was only done for programmatic ease-of-use, since C arrays are 0-based. Mathematically, this change does not affect any equations. Traditional mathematical notation uses 1-based arrays of numbers and variables, however it was easier and more efficient to access the arrays as 0-based during implementation. Also note in this depiction that bias nodes always have an output value of 1 (as expected), and that the output layer has no bias node.

So K is still the dimensionality of the input, except now the input layer is accessed from *0* to *K-1*. J is the dimensionality of the hidden layer, accessed *0* to *J-1*, etc. Note that the patterns themselves are accessed 0-baseed as well, this is why $x_0(p)$ is loaded into node 0 of the input layer, etc.

Layer 0 is the Input layer, and the layers increase from there on up. The above notation shows a 3-layer (1 hidden layer network), though as noted below, any number of layers are supported by the implementation work of this paper.

Note the order of subscripts. w^1_{jk} in this case specifically refers to a weight emanating from *node k* in the input layer to *node j* in the hidden layer. The superscript of 1 denotes that the weight is feeding into a node in Layer 1, and is used as part of that nodes net input calculation.

The vector $y(p)$ refers to an output at a particular layer, thus $y^2_0(p)$ is the output of the first node in Layer 2 (or the output layer), and $y^1_j(p)$ is the output of the *node j* in Layer 1 (or the only hidden layer in this specific example model). Like the SLP neural network, we define an activation function per processing node (not the input layer), that defines how the net input

IV. BACK-PROPAGATION

When training an MLP NN, we are given a set of patterns (input/output pairs), and the ultimate goal is to generate the appropriate output (from the output layer) given the corresponding input of a particular pattern. For example, say we present the input vector of pattern p=5, which has an associated output of 10. Then the goal is to have our MLP NN generate an output in the output layer that corresponds to the output of 10 (irregardless of encoding scheme). If the MLP NN can do this successfully across all training data (i.e. *converge* in a sense), then it can likely be used to generalize over unseen data with the same dimensionality of input/output. As mentioned earlier, this constitutes the fact that the network has retained knowledge.

To achieve this, we need some form of measuring the actual error or success when a particular pattern (or group of patterns) is presented to the network. This is similar to the idea of a *fitness-function* in the field of *genetic algorithms*. We utilize some form of error function, of which there are many to choose from, and we change the interconnections (weights) of the network in such a way as to minimize this error function, often referred to as a *delta weight change*. This change hopefully minimizes the error, which means that the outputs produced by the network in the future, should be closer to the desired outputs. Thus after the learning process completes entirely, the final weight values represent the acquired knowledge attained by the network.

One very popular error function is the sum squared error across the output layer of the form:

$$E^p(w) = \frac{1}{2}\sum_{i=0}^{I-1}\left[d_i(p) - y_i(p)\right]^2$$

where p represents a particular pattern, d is the expected or desired output, and y, as explained earlier, is our actual produced output.

A. Gradient Descent (GD)

The *gradient descent* (GD) procedure is the most basic, standard minimization technique of the error function defined

by the back-propagation learning algorithm and used throughout the literature, which adjusts the weights of the network by an amount proportional to the negative gradient. It is also typically used in *stochastic-mode learning* (explained later in more detail), as can be seen by the error term being with respect to a particular pattern *p* and not global across an entire set of patterns. Here is the weight change equation using the standard GD procedure:

$$\Delta w = -\eta \nabla E^p(w)$$

where η represents the *learning rate* (a constant applied to the magnitude of the gradient vector).

Once we have an error function and a delta weight change procedure, we can solve formally for what the weight changes will be for each weight in the network. As will be shown, the changes for the weights connected to the output layer are simpler than lower layers of weights.

Consider a simple MLP feed-forward network with one hidden layer, as depicted above; we will now show the appropriate delta weight changes to weights w_{ij} and w_{jk}.

In an effort to be *concise* (and avoid unnecessary Microsoft Equation Editor pain) we defer to the partial derivative derivations found in [1] for a thorough, step-by-step explanation of how the following delta weight changes are derived. Instead we jump directly to the weight change equations themselves.

For weights converging into an output node, we have:

$$\Delta w_{ij} = \eta \left[d_i(p) - y_i(p) \right] g'\left(net_i(p)\right) y_j(p)$$

where *p* represents a particular pattern, *d* is the expected or desired output, *y* is the actual produced output, *g* is the activation function (note the derivative of it), and net_i refers to the net input into *node i*.

This tells us that the delta weight change to a weight converging into an output layer node is proportional to a *constant of proportionality* (learning rate times the derivative of the activation function of the net input on the node), times the error (expected minus actual output) that is committed at this node, times the output of the node from which the weight emanates.

For weights converging into a hidden layer node (from the input layer in this specific example), we have:

$$\Delta w_{jk} = \eta \delta_j x_k(p)$$

where δ is referred to as the *error term* (or *delta term*) of *node j* in the hidden layer, representing the constant of proportionality at *node j* times the sum of the errors of the output layer nodes, and x_k is of course the pattern input *k* presented to *node k*, since the node that this weight is emanating from, into the hidden layer, is in the input layer.

So this tells us that the delta weight change to a weight converging into a hidden layer node is composed of all the error contributions of the nodes in the higher layer (in this case output layer) weighted appropriately by each corresponding weight, times the output of the node from which this weight emanates, in this case the input layer.

B. Gradient Descent w/ Momentum (GDM)

GD with momentum (GDM) is similar to GD, except that an additional term (from the previous weight change) is included in the current weight change, multiplied by a constant called the *momentum* (α). The momentum term typically has a value between the interval of [0, 1].

Basically this is one way to achieve individual learning rates per weight, implicitly. What this accomplishes is either a smoother convergence (when α is large), as opposed to the dramatic oscillations that can occur with standard GD technique, or faster convergence (when α is small). While smoothed oscillations are generally good because the convergence is more stable and the network will generalize with less variance, it can also force the network into local minimum traps because the convergence is additively applying the previous weight change, forcing the weight to deviate less.

GDM is typically applied in *batch-mode learning* (explained in detail later).

$$\Delta \mathbf{w}(t) = -\eta \frac{\partial E}{\partial \mathbf{w}(t)} + \alpha \Delta \mathbf{w}(t-1) = -\eta \sum_{s=0}^{t} \alpha^{t-s} \frac{\partial E}{\partial \mathbf{w}(s)}$$

Figure 4: GDM delta weight change. Equation borrowed from *Georgiopoulos* et al. [1]

C. Resilient Propagation (RPROP) [3]

The contribution behind RPROP is the fact that the gradient value itself is not used, but rather the sign of the gradient only is used. The authors achieve this by defining an adaptive *update-value* (Δ_{ij}) per individual weight in the network, which represents a scalar magnitude applied during the weight update change. The update-value is modified by a constant amount based upon the gradient direction *only*, not the gradient value, and modified when the sign changes. This signifies that the last weight update was too large and that it "hopped-over" its local minimum during convergence. Thus if the last weight update overshot its local minimum, the update-value is decreased (by a magnitude of η^-) to help prevent it from over-shooting the next time, and if the gradient does not change signs, then the update-value is increased (by a magnitude of η^+) to help it converge faster.

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ * \Delta_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^- * \Delta_{ij}^{(t-1)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)} & , \text{ else} \end{cases}$$

$$\text{where } 0 < \eta^- < 1 < \eta^+$$

Figure 5: Shows the update-value being set depending upon the change in sign of the gradient of the error function. Image borrowed from RPROP paper [3].

Once the update weight is set, it is applied to the delta weight change as follows: if the gradient has a positive sign (increasing error), then the update-value is subtracted from the current weight, if negative, then it is added.

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ +\Delta_{ij}^{(t)} & , \text{ if } \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ 0 & , \text{ else} \end{cases}$$

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} + \Delta w_{ij}^{(t)}$$

Figure 6: Shows the delta weight change. Image borrowed from RPROP paper [3].

One of the major benefits of this solution is that no learning rate (η) or momentum rate (α) needs to be discovered (and consequently, passed to the model), like with GD and GDM, respectively. The constant values for η^- and η^+ were found empirically by the authors to be 0.5 and 1.2, respectively, and in practice they work quite well over a number of problems. Additionally, a constant magnitude of 0.1 is applied to the initial update-weight values.

Note that the authors indicate that RPROP should be applied during *batch-mode learning* (explained in detail later):

> "The update-values and the weights are changed every time the whole pattern set has been presented once to the network (learning by epoch)." [3]

D. BP Algorithm

The actual steps of the back-propagation learning algorithm are as follows:

(Step 1): Initialize the weights. Suggestions on how to do this properly are discussed in the next section. Set pattern selection index p equal to 0 (in 0-based array referenced systems like the one we have implemented).

(Step 2): Present the current pattern p's input vector. Note this is the start of the training algorithm loop, as of course multiple patterns will be presented to the network.

(Step 3): Propagate the pattern forward through the network, by calculating the output of all hidden and output layer nodes. Recall that an output calculation involves using the *adder* and *activation* functions per node.

(Step 4): Check the output of the network. If the produced output is equal to the corresponding pattern output, then go to Step 7, otherwise proceed. Note this is an equality test, not a convergence test.

(Step 5): Calculate the error (or delta) terms of the output and hidden layer nodes. This is the actual back-propagation step, as you start at the output and work backward using the error term functions. Recall that the error of an output node is the difference between actual and expected output, whereas a hidden layer node incorporates the errors of all nodes in the above layer that it is connected to times the weight values that connect them, summed up. Both of these delta terms are multiplied by the derivative of the activation function at that particular node as well. From Geogiopoulos et al. [1], we have the following (below), except ($0 <= i <= I-1$) and ($0 <= j <= J-1$) using our predefined model. Note that bias nodes do not calculate error terms, this is why they are not incorporated into this equation or the ranges for i and j.

$$\delta_i^2(p) = g'(\text{net}_i^2(p))\,[d_i^2(p) - y_i^2(p)]$$

$$\delta_j^1(p) = g'(\text{net}_j^1(p)) \sum_{i=1}^{I} w_{ij}^2\, \delta_i^2(p)$$

(Step 6): Calculate and apply the delta weight changes, using the weight change functions mentioned previously. This is where the minimization functions such as (GD, GDM and RPROP) are utilized.

(Step 7): Finally we either loop back to Step 2 with p equal to the next selected pattern, or if we have exhausted our patterns, then we check for *convergence* (or *stopping criteria*). If satisfied, then we quit, otherwise set $p = 0$ again and loop back to Step 2, which now constitutes another *epoch* of learning.

V. LEARNING CONSIDERATIONS

Since there is no guarantee that back-propagation will even converge, let alone converge to an optimal solution and do so quickly, there are a number of extra conditions to take in consideration while learning. Most of these have been investigated thoroughly over the years and so general empirical observations have been made. All of the observations below were found during the implementation of this paper as well.

A. Pattern Order

The order in which patterns are presented is not as important as in FAM networks, however it should be noted that the pattern order itself presented in such a way so that patterns belonging to the same class do not follow one another very often. In other words, one should not present a list of patterns ordered by group (an entire class of patterns followed by patterns of a second class, etc.). The reason for this is two-fold, (1) networks will limit their focus to learning a single pattern class, meaning they will learn one class really well, and (2) networks learn the fastest from the most unexpected samples [2]. The former is actually detrimental to the network as the last class of patterns presented, especially if there are many patterns in that class, will encourage the network to learn that specific class only for the most part. The previous learning will be degraded as the network encounters many examples of patterns of the final class, encouraging it to converge to a local minimum that generalizes well for that class only.

There are various strategies on how to adjust the pattern

order in order to encourage the network to not only learn all patterns, but also encourage it to converge to a global minimum error. The simplest solution is to just shuffle (or randomize) the pattern each epoch, though there are many more complex schemes. *LeCun*, et al. propose another solution called an *emphasizing scheme*, which suggests measuring the error produced after presenting a particular pattern and if high (higher than when other patterns are presented), then presenting that particular pattern more often. The reason for this is that high error indicates the network has not learned that particular pattern well enough yet, as so it represents a lot of potential knowledge the network could gain if presented more often. From *LeCun* et al. [2], we have:

Choose Examples with Maximum Information Content

1. Shuffle the training set so that successive training examples never (rarely) belong to the same class.
2. Present input examples that produce a large error more frequently than examples that produce a small error.

B. Stochastic vs. Batch Learning

If node connection weights are updated after each pattern input presentation, this is known as *stochastic*, *continuous* or *online learning*, whereas if they are adjusted after a cumulative presentation of all the input patterns, then this is known as *batch learning*. Though batch learning and its associated equations correspond to the true gradient procedure applied to the cumulative error function

$$E(w) = \sum_{p=1}^{PT} E^p(w)$$ [1],

stochastic learning is widely used in practice. This is because it tends to converge much faster, since not all patterns are needed to train the network.

Consider a set of 1000 training patterns, many of them may be very similar, meaning high redundancy. In fact there may be only 100 relatively unique patterns within that set of 1000. And so by applying all 1000 patterns and updating the weights only once, the network will have learned as much as if it just applied 100 of the unique patterns, adjusting as it goes. Thus the network is waiting unnecessarily long to apply a change of weighting.

There is no founded mathematical explanation as to why stochastic learning works as well as it does, since the error function keeps changing between patterns, however in practice stochastic learning performs quite well. It often finds better solutions, or at least this was found in practice in this paper, and is stated in various papers [2].

Batch learning has its own advantages though, one being that many acceleration techniques, such as RPROP discussed in this paper, are only typically only available in batch learning. Batch learning also represents the true gradient change, as mentioned above.

Of course there are also *hybrid* modes that merge the two learning modes, say applying a weight change after so many (but not all) patterns are presented. In practice though, these are not used as often.

C. Weight Initialization

The initial weight values can dramatically affect the rate of learning. This is because when using a sigmoid function, very small or very large values will produce a very small gradient value, and so when weight are adjusted, they will be adjusted minimally. Weights should be chosen randomly initially, but the values should be closer to the linear part of the sigmoid function (e.g. between [-0.5, 0.5], versus between [-1, 1]), so that learning begins quickly. Not doing so is what is known as the *premature saturation problem*.

One solution suggested by [*Russo* 1991] takes into account the *fan-in* (i.e. the number of weights converging into a node) to any particular node when initializing those weights. They suggest the weights be uniformly distributed over the range

$$\left[\frac{-\alpha_i}{F_i}, \frac{\alpha_i}{F_i}\right]$$

where α_i is an appropriate constant and F_i is the fan-in [1].

D. Number of Hidden Layers / Hidden Nodes

It has been proven that an MLP NN with just one hidden layer of nodes and a non-linear activation function, can be used to satisfy any function of practical interest [1]. However in practice, this is not always the case. Sometimes authors will build multi-layered structures to achieve better performance. Note however that models with higher numbers of layers will take longer to train in both computing power and general information flow throughout the network. Thus a typical scheme is to setup a model with only a single layer, and only add additional layers if performance is not satisfactory. One could also generate an automatic looping scheme that tries one layer and tests, then another layer and tests, etc.

As for the number of hidden nodes, it has been shown by [Baum, Haussler 1989] that a good estimation for a single hidden layer network is:

$$J = \frac{PT}{10(K + I)}$$

where PT represents the number of patterns, K the dimensionality of the input, and I the dimensionality of the output. Another analytical way of looking at it is that the number of interconnections (J(K+I)), should be roughly equal to $\frac{1}{10}$ the total number of patterns [1]. This was proposed over research with binary networks, however many have found it to be a good estimator of nodes. Specifically for this paper, this equation was used, and the number of nodes was rounded up when fractional. Of course not all network ad-hear to this generalization.

A more elegant approach, which can be automated, starts with an empty hidden layer, and continues a looping process of adding a single hidden node, testing the error, adding another, testing the error, etc., until a satisfactory overall residual error is achieved. As explained later, a combination of both procedures mentioned above were utilized during experimentation.

E. Learning Rate

If there is one parameter that has been studied over and over, it

would be the learning rate η. There have been many techniques throughout the literature that propose changes to η during learning at various points, or at least the existent of local η per weight. Many techniques involve adjusting η, increasing it when far from its minimum, and decreasing it as it gets closer or even surpasses its minimum (oscillation). We have seen that the RPROP solution ignores the gradient and *only* modifies η.

F. Stopping Criteria

There are many ways to define stopping criteria, however two common ones include a maximum number of epochs to run, which can change dynamically depending on the results of batch run of patterns, and the average sum of squared errors. Another strategy involves testing the Euclidean norm of the gradient vector of the total error function, and stopping once a sufficiently small threshold is reached. Yet another strategy involves testing the rate of change in the total error function from one epoch to another, if not much is changing, then there may be little point to continuing learning. Still another strategy involves testing generalization by presenting patterns and testing the percentage of correct classifications, though of course this can be expensive if performed too often. Some strategies involve using combining multiple tests, or using certain ones for certain types of problems.

One algorithm proposed by Gallant [9], though used in conjunction with his *Pocket Algorithm* for SLP networks, proposes changing the max number of epochs dynamically based upon if the pocket-weights were changed after some proportion of epochs. Although this algorithm wasn't proposed for MLP networks specifically, it further demonstrates the amount of stopping criteria variations found in the literature.

G. Other Considerations

Additional considerations to keep in mind when training an MLP NN include, but are not limited to, choosing appropriate target convergence values, not necessarily just using the activation function's asymptotes, normalizing pattern inputs, including transforming them if necessary, etc.

VI. IMPLEMENTATION

The implementation of this project involved writing a program to create MLP NNs, train them, and then test them against some unseen dataset to see how well they generalized. In this case, it was done by writing a program that accepted various flags to put the program in one of the aforementioned states, as well as accepted the appropriate parameters. The program was first tested with a dataset representing the XOR problem, and then compared to hand calculations, to verify that it worked properly. A debugger was used to step through each line of the learning processes to ensure that the net inputs, error terms, delta weight changes, etc. were being generated properly. After the implementation was thoroughly tested against the XOR problem, with each variation of BP learning discussed in this paper, GD, GDM and RPROP, the program was tested on some of the provided classification problems, also used to gage its performance.

Various features discussed earlier in this paper were implemented, such as randomizing the pattern selection order, stochastic versus batch mode learning (the program can literally do GD learning in a batch mode), as well as other features, including any number of hidden layers (within reason), any number of hidden nodes (used for all hidden layers), additive training, stopping criteria based upon reaching a specified global error threshold, generating and storing a pseudo-random-number (PRN) seed, etc.

The implemented program also has the ability to generate an *N-M-N Encoder Problem* [4] dataset, given the specified parameters, as well as an XOR model with a simply flag passed as an argument.

A. Program Details

```
1038 void calculateOutputs()
1039 {
1040   uint i,j,ni;
1041
1042   // for all layers execept input layer
1043   for (j=1; j<g_model.numLayers; j++)
1044   {
1045     ni = nodesPerLayer(j);
1046
1047     // bias nodes not involved in these calculations (output
1048     if (j < O_LAYER) ni--;
1049
1050     // for all nodes in the current layer (except bias)
1051     for (i=0; i<ni; i++)
1052       g_model.nodes[j][i].output = g(netInput(j,i));
1053   }
1054 }
```

Figure 7: A small program snippet showing forward propagation throughout the network, after the pattern inputs have been presented to the input layer.

The MLP program was written purely in C, mainly for efficiency and ease of access of array elements of nodes and weights. It was compiled w/ *gcc* in a *Cygwin* environment; thus it should compile on a UNIX machine with little to no changes made to the program. It was debugged with *gdb*. At one point is was also compiled with Microsoft Visual Studio to preliminarily verify cross-platform support and test with a more mature debugger.

The program supported three primary operating modes, *create*, *train*, and *run*, used to: create an MLP NN model, train an existing model over some dataset, and test that model against a dataset, respectively. Each time a model is created it is saved to a file, which can then be referenced when training, testing, or simply to view the model. Each model has a PRN seed associated with it, saved with the model file.

As far as node connectivity of the MLP NN models generated by this implementation, only standard feed-forward networks were supported; weights could not hop layers or go from higher to lower index, nor could they be connected to any other nodes within the same layer. Recurrent connections were also not supported.

Bias nodes were added to the input and hidden layers (as expected), except at the *end of each layer* (the last node in that layer), as opposed to the first node, as indicated in most NN

literature. This was done purely for ease of programming, so that the last node in those particular layers could be ignored during times when the input layer is not being taken into consideration. Recall that in a standard feed-forward network, weights from a lower layer do not connect to the bias node in the immediately higher layer. So this must be kept in mind when performing the learning or even generating the output. This seemed more intuitive then skipping the first node of a layer, although of course mathematically there is no difference.

The average sum of squared errors could be used as convergence criteria, as well as a specified maximum number of epochs. This was useful if the model never quite converged all the way, but typically converged around a specific error, say 5%. In this case the error calculated could be used to stop the training once the error hit a specified threshold. This was preferable to just letting the training end after a max number of epochs, as the last epoch may not have produced the "best" weights (or at least the weights that contributed to the lowest error during training). Directly from *Georgiopoulos* et al., we have:

> "An example of a stopping criterion is the one mentioned in the step by step description of the back-propagation algorithm [i.e., the average sum of the squared errors $\frac{1}{PT}\sum_{p=1}^{PT}E^{p}(w)$] to be smaller than a threshold. Another example of a stopping criterion is to stop training when the *maximum number of epochs allowed* is reached." [1]

A pseudo-random-number (PRN) was stored in each model created. It can also be passed in manually via parameter. This was done to support the duplication of results. Recall that this implementation randomly swaps the pattern selection order at each training epoch, as well as randomly initializes the weights between the range of [-0.5, 0.5]. Thus, by specifying the same PRN when creating a model, the exact same initial weights and pattern selection order can be attained again.

Currently the weight initialization takes place at the start of training, however, it could happen once upon model creation and therefore be tied to the PRN seed. This would be an interesting way to test the exact same weight initializations against different training patterns. During testing, the initial weight configuration can affect not just learning speed, but the actual classification performance of the network, since it may converge to different local minimum when training.

Pattern selection order was randomized during *Step 2* of the BP learning process *each epoch*. Of course this should really only affect stochastic learning, as batch will some up the gradient for all patterns anyway, regardless of order. However patterns were still shuffled each epoch, since the program was so efficient. From *Georgiopoulos* et al., we have:

> "The order in which the input/output training pairs are presented to the network should be randomized from one epoch to the next." [1]

Note that the following pattern selection order suggested by *LeCun* et al. [2], though possibly more effective, was not implemented. They write:

> "Present input examples that produce a large error more frequently than examples that produce a small error." [2]

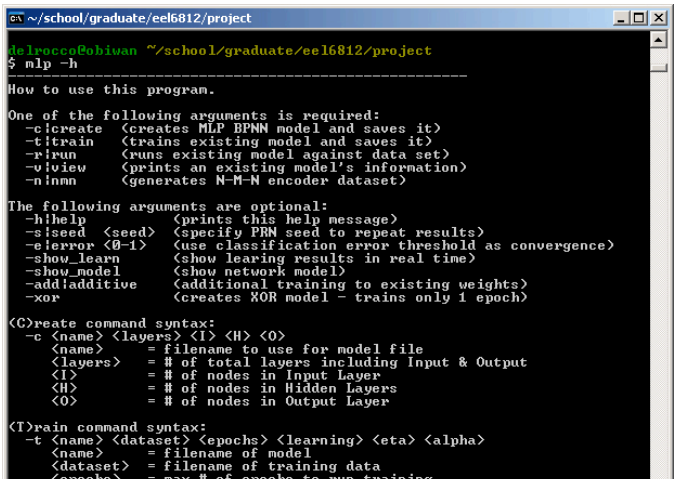

Figure 8: Part of the MLP program help menu.

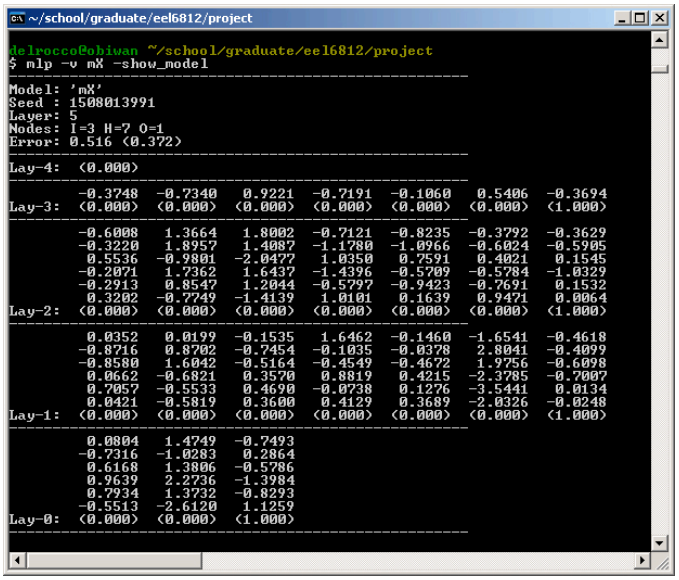

Figure 9: Viewing a network which has an input dimensionality of 2 (+ bias), 3 hidden layers, 6 hidden nodes (+ bias), and a single output node. The errors shown represent the final error after training completed and the best error (in parenthesis) ever achieved by this model. Per layer, the parenthesized numbers represent node output values, and the stacked numbers represent weights emanating from each node to all nodes in the immediately higher layer (except bias).

VII. XOR TEST

To achieve the XOR test, an example was taken from *Georgiopoulos* et al. [1]. The initial weights and network configuration (meaning the same number of nodes, layers and initial weights were used, as well as the same learning η of 1.0). The only difference being that networks in this paper store bias nodes at the end of layers. However, this does not affect the performance or output of the network in any way; it is strictly a programming optimization.

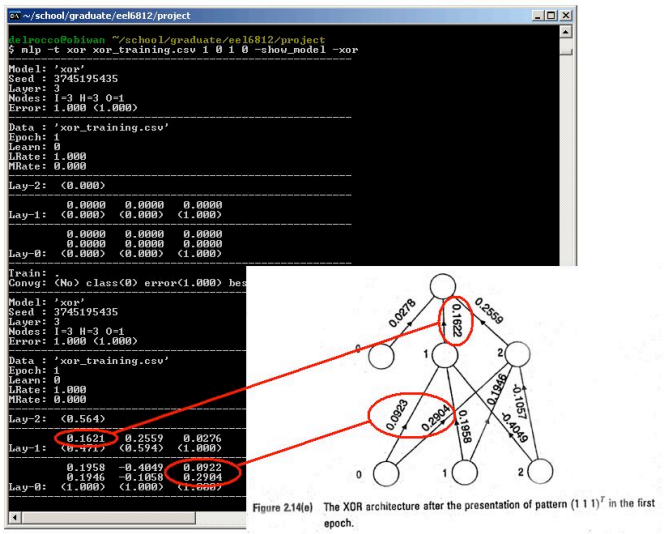

Figure 10: XOR example compared to example in book [1].

In an effort to be concise, the screenshot alone is shown, as opposed to writing out all the trivial equations by hand, which would be tedious and not add value to this paper.

Above we see that after the first epoch (after presenting all four unique patterns of the XOR problem with 2 inputs), the network learns the same weights. Note there is some very minor rounding difference between the calculations in the book versus the calculation on the CPU.

VIII. CLASSIFICATION PROBLEMS

After the program was thoroughly tested against the XOR problem, with each variation of BP learning discussed in this paper, GD, GDM and RPROP, experiments were run against provided classification datasets, as well as the *10-5-10 Encoder Problem*, discussed at great length in [4]. One of the provided classification datasets is a derivative of the very famous *Iris Plant Database*, provided by *Fisher* [7], called the *modified Iris dataset*.

For each problem below, and for each learning method, a set of learning parameters was chosen after some experimentation and preliminary testing against a validation data set. For the RPROP learning method, the standard parameters for learning rate were chosen, but various numbers of hidden layers and nodes were tried. For GD and GDM learning methods, everything was up for experimentation, learning rate, momentum rate, hidden layers, and hidden nodes. Of course maximum number of learning epochs and convergence error criteria were experimented with as well. Once a "good" set of parameters was discovered, they were used to generate the results, discussed in the next Section.

A. 10-5-10 Encoder Problem

The N-M-N encoder problem refers to a binary input pattern problem, setup as a 3 layer (1 hidden layer) network, with N input nodes, N output nodes, and M hidden nodes (not including biases). The idea is to be able to generalize the output layer to be identical to the input layer. Typically only one of the pattern inputs is toggled on (set to 1), while the other inputs are toggled off (set to 0). Though other adaptations of the problem exist, including one that uses all combinations possible with the N pattern inputs (i.e. more than one pattern input can be toggled on).

Note that M is typically smaller than N to establish a bottle-neck for the information. This forces the inputs to be encoded and then decoded through the hidden layer, hence the problem name *encoder*. From *Fahlman* [4], we have the description of the classic encoder problem as:

> "This network is presented with N distinct input patterns, each of which has only one of the input units turned on (set to 1.0); the other input bits are turned off (set to 0.0). The task is to duplicate the input pattern in the output units. Since all information must flow through the hidden units, the network must develop a unique encoding for each of the N patterns in the M hidden units and a set of connection weights that, working together, perform the encoding and decoding operations." [4]

Note that *Fahlman* also provides a complement encoder problem, where only one input per pattern is toggled off (set to 0), with the rest toggled on. This requires more information to be encoded / decoded, and thus in general requires more learning.

B. G4C (25%)

This problem represents 4 Gaussian clouds of data points, one in each quadrant of a 2D Cartesian coordinate system. And the linear boundaries between the sets are the X and Y-axes. Thus the dimensionality of input is 2, and the dimensionality of output is 4. The 2D points (pattern inputs) range from [0, 1] for each input, therefore the center of the coordinate system is located at (0.5, 0.5). The idea is to classify all 4 possible clouds of points.

Note that there is a 25% inherent minimum classification error associated with the particular dataset with which we were provided.

C. G6C (15%)

This problem represents 6 Gaussian clouds of data points on a 2D Cartesian coordinate system, each one starting near the center of an imaginary clock and moving outward toward the following hour markers: 1, 3, 5, 7, 9, and 11. Thus, the linear boundaries between the sets are the lines along the following hours of an imaginary clock: 2, 4, 6, 8, 10, and 12. The center of the imaginary clock is located at (0.5, 0.5). The idea is to classify all 6 possible clouds of points.

Note that there is a 15% inherent minimum classification error associated with the particular dataset with which we were provided.

D. Circle-in-Square (0%)

This problem also represents data points on a 2D Cartesian coordinate system, except the 2 classifications of points are either inside of an imaginary circle or outside of it. And again the 2D points (pattern inputs) range from [0, 1] for each input, hence the name *circle in square*. The decision boundary here is the edge of the imaginary circle itself, centered at (0.5, 0.5), with an approximate radius of 0.4. So the goal is to classify the 2 sets points, inside and outside of the circle.

Note that there is no classification error associated with the particular dataset with which we were provided, so ideally the problem could be classified perfectly.

E. Modified Iris Problem

This problem is based on the famous Iris dataset as provided by *R.A. Fisher* [7], one of the most well-known databases to be found in pattern recognition literature. The dataset initially contained 3 classifications of points in a 4D space (input dimensionality of 4); the three classifications representing the flowers: *Iris Setosa*, *Iris Versicolor*, and *Iris Virginica*. However the modified Iris problem, removes Iris Setosa points as they are clearly linearly separable from the other two, and the other two are *not* linearly separable from each other. Thus, the goal of this problem is to determine which of the two classes the remaining points belong to, either Iris Versicolor or Iris Virginica.

Note the original 4 input dimensions are sepal length, sepal width, petal length, and petal width. However, for the purposes of this specific classification problem, only petal length and width are used.

IX. EXPERIMENTATION & RESULTS

Below are the results of running the MLP BP implementation program after the "relative best" parameters were chosen for the various learning methods, for each problem provided. Note that for the 10-5-10 encoder problem, the results represent learning convergence speed (in epochs), mean and standard deviation of such, and the number of models that converged, whereas the results for the other problems represent the number of correct classifications given the test data set – indicating how well the model was able to generalize over unseen data.

For each learning method (GD, GDM, and RPROP), and for each problem below, 25 models with different initial weights were created and trained, and then tested. Thus 75 models were executed from creation to test, for all 5 classification problems – for a total of 375 total models tests.

To find "optimal" parameters, many of them were just tested randomly. For hidden node count, often the Baum, Haussler estimation was used first and tested, and then the number of hidden nodes would be increased or decreased until it was "obvious" what the optimal number was. For hidden layers, at first several hidden layers were tried, but eventually a single hidden layer was used for all but one of the classification problems. In most cases, adding another hidden layer actually hurt the performance of the network.

In bar-charts below, the order of BP variation results are GD (blue), followed by GDM (green), followed by RPROP (pink).

A. 10-5-10 Encoder Problem

For the encoder/decoder problem, which by definition only has one hidden layer, the following parameters were used:

GD had a learning rate of 1.0.

GDM had a learning rate of 0.05, and momentum of 0.9.

RPROP used the paper defined update-values.

The results below are averaged over 25 runs, each with different initial weights, as explained above.

	GD	GDM	RPROP
# converged	25	25	25
mean epochs	7	176.44	25.68
std. dev.	1.55	42.73	2.95

Note that stochastic GD still performs very well on this type of problem. In the RPROP paper, the results were compared against other batch-learning algorithms, and the "standard BP" shown compared in the paper was actually GDM. The conclusions in the original paper made RPROP look great given its convergence speed compared to GDM, however they did not show a comparison with stochastic GD, as we do here. Note that the mean number of epochs and standard deviation shown here for GDM, are very similar to those shown in the RPROP paper for the "BP" technique.

Below is a table showing the 25 runs using stochastic GD minimization technique. The pseudo-random-number seeds are shown so that results can be duplicated.

id	epochs	classed	converg	seed
1	6	500	1	4100281208
2	8	500	1	4133187208
3	7	500	1	4152984208
4	4	500	1	4168609208
5	6	500	1	4178906208
6	6	500	1	4189047208
7	6	500	1	4200172208
8	7	500	1	4211875208
9	10	500	1	4223281208
10	7	500	1	4235031208

11	7	500	1	4244547208
12	8	500	1	4253594208
13	9	500	1	4270047208
14	9	500	1	4280562208
15	5	500	1	4294109208
16	9	500	1	8329912
17	10	500	1	17501912
18	7	500	1	29391912
19	6	500	1	39798912
20	6	500	1	48610912
21	6	500	1	59016912
22	6	500	1	67548912
23	5	500	1	76735912
24	8	500	1	86063912
25	7	500	1	96719912

B. G4C

For the 4-Gaussian cloud problem, recall that there is an inherent 25% error in the dataset. Still the MLP did slightly worse than expected, probably due the parameters used during learning, most notably the increased number of hidden layers.

The following parameters were used:

GD used 2 hidden layers, 9 hidden nodes, and a learning rate of 0.85. It was trained over maximum number of 2000 epochs, though many converged before reaching that upper-bound.

GDM used the same number of hidden layers and nodes as GD, but a learning rate of 0.04, and momentum of 0.75. It was also trained over 2000 epochs.

RPROP used only 1 hidden layer and only 2 hidden nodes, though training was performed over a much greater amount of epochs, 12000 in this case. However, because the network was so small, the training was extremely fast.

The results below are averaged over 25 runs, each with different initial weights.

	GD	GDM	RPROP
classifications	3324.80	3145.16	3000.84
std. dev.	108.07	52.57	132.77

For an arbitrary run shown below in a scatter plot, note that the patterns that could not be classified (shown in gray), are the patterns within the center of the 4 Gaussian clouds, which makes sense, since they are along the boundary region. However, the amount of them is likely due to non-optimal parameters used. Similarly the misclassified patterns (shown in yellow) are near this boundary region as well, but located within one of the clouds. Of course some of the misclassifications are due to the inherent classification error of this data set as well. In any case, the further patterns are from the boundary region, the more likely they are to be classified.

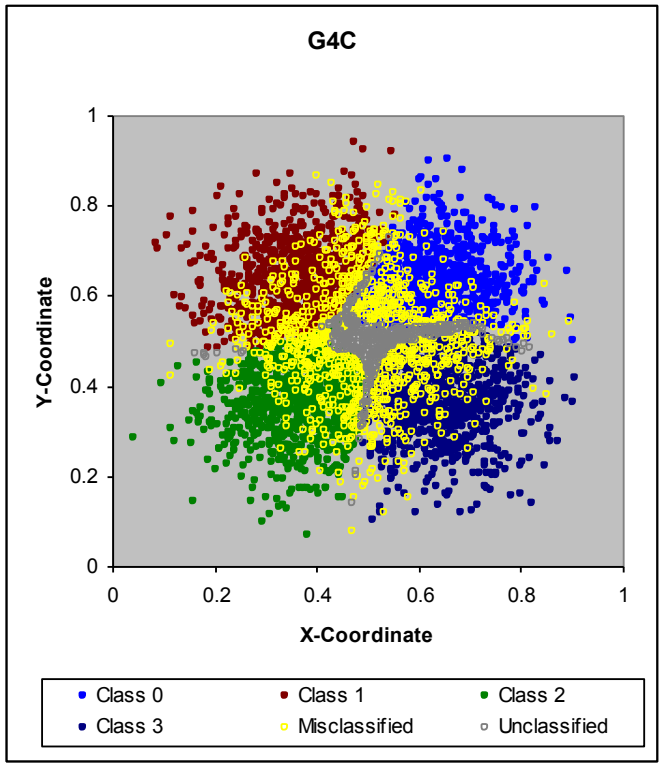

Figure 11: Scatter-plot of G4C problem. Note the location of misclassified and unclassified patterns.

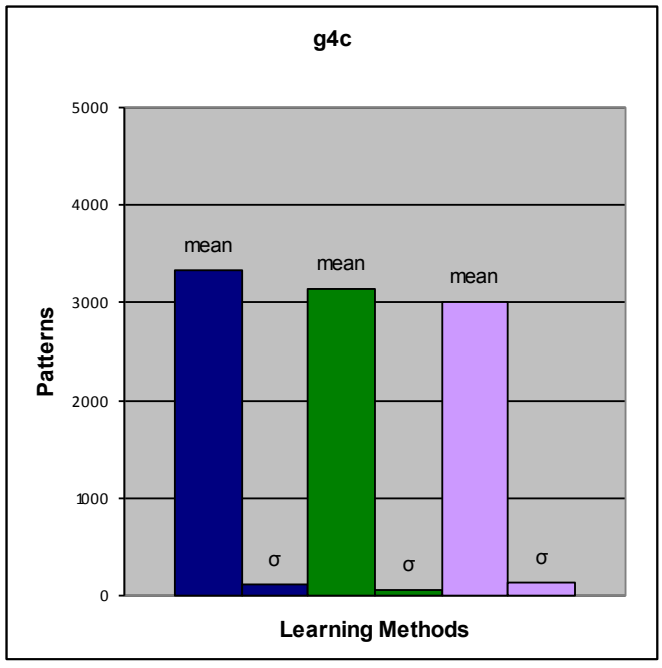

Figure 12: G4C classification performance using GD (blue), GDM (green), and RPROP (pink).

C. G6C

For the 6-Gaussian cloud problem, recall that there is a smaller 15% inherent error in the dataset. Here the MLP networks seemed to perform a little better, possibly because of the use of only one hidden layer, or maybe because better parameters

were chosen.

Each of the variations used a network with only 1 hidden layer, and the following parameters were used:

GD used 7 hidden nodes, and a learning rate of 0.85. It was also trained over 2000 epochs.

GDM used 7 hidden nodes, a learning rate of 0.04, and momentum of 0.75. Training for this variation took a max of 10000 epochs, many of which never converged.

RPROP used 16 hidden nodes, trained over 12000 epochs. Either the variation is not well-suited for this problem, or more likely, a good set of parameters was not discovered.

	GD	GDM	RPROP
classifications	3929.72	3892.28	3547.88
std. dev.	57.99	29.76	394.38

Note in the scatter plot that very few patterns were left unclassified, so the network as able to better learn the relationship between input/output pattern pairs. Also note that the misclassified patterns again lie along the separable boundaries. Some of these are due to the inherent classification error, while a few % others are due to lack of generalization performance in the network.

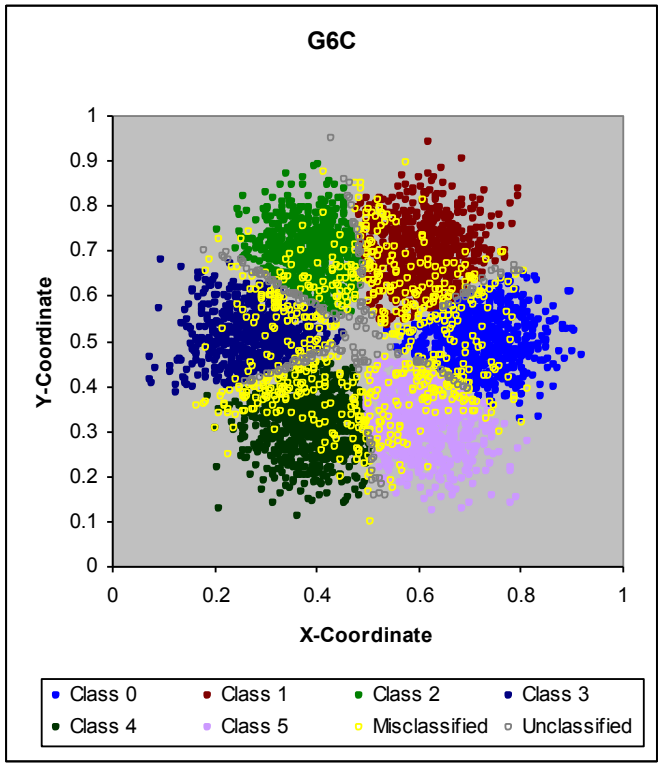

Figure 13: Scatter-plot of G6C classification problem. Note relatively good performance given inherent 15% error. Note boundary region results.

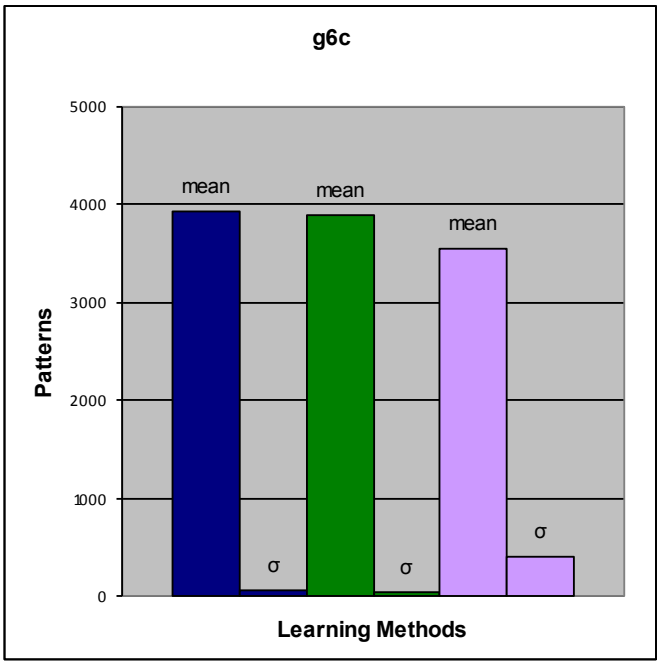

Figure 14: G6C classification performance using GD (blue), GDM (green), and RPROP (pink).

D. Circle-in-Square

For the Circle-in-Square classification problem (CinS), there is no inherent error. Each of the variations used a network with only 1 hidden layer, and GD and GDM used 16 hidden nodes in that layer, whereas RPROP needed only 3. The learning parameters are as follows:

GD had a learning rate of 0.65.

GDM had a learning rate of 0.06, and momentum of 0.5.

RPROP used the paper defined update-values.

As for the number of max epochs specified during training, GD needed only 3000, whereas GDM used 5000, and RPROP needed close to 10,000.

The results below are averaged over 25 runs, each with different initial weights.

	GD	GDM	RPROP
classifications	4841.20	4726.88	4127.28
std. dev.	18.30	125.65	168.15

One of the better runs using GD is shown in the scatter plot below. Note that overall, very few points were misclassified or unclassified, which makes sense given no inherent classification error. Also note that the very few misclassifications that did occur show up tightly on the boundary line (the edge of the circle). For the most part, the classification success rate was very high using GD for this problem, whereas RPROP performed the worst, although not horribly.

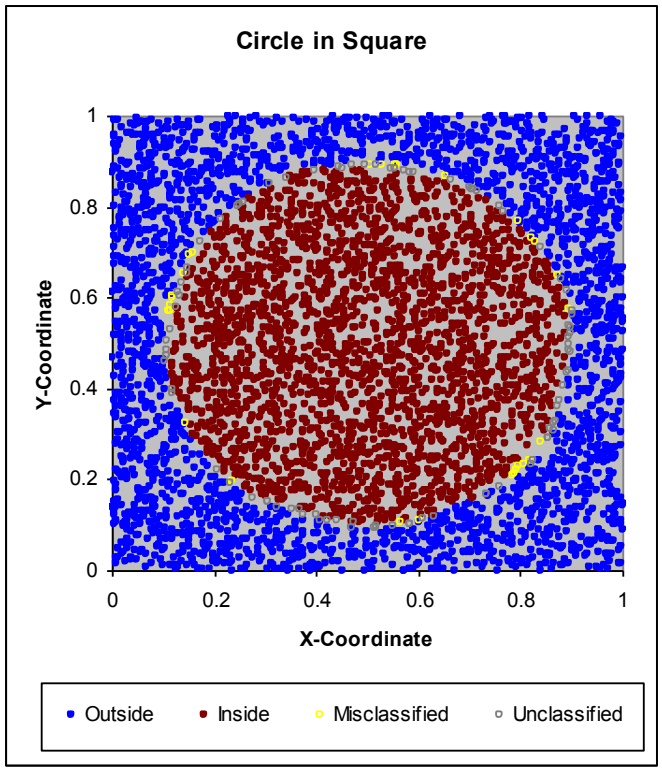

Figure 15: Scatter-plot of CinS classification problem w/ no inherent error.

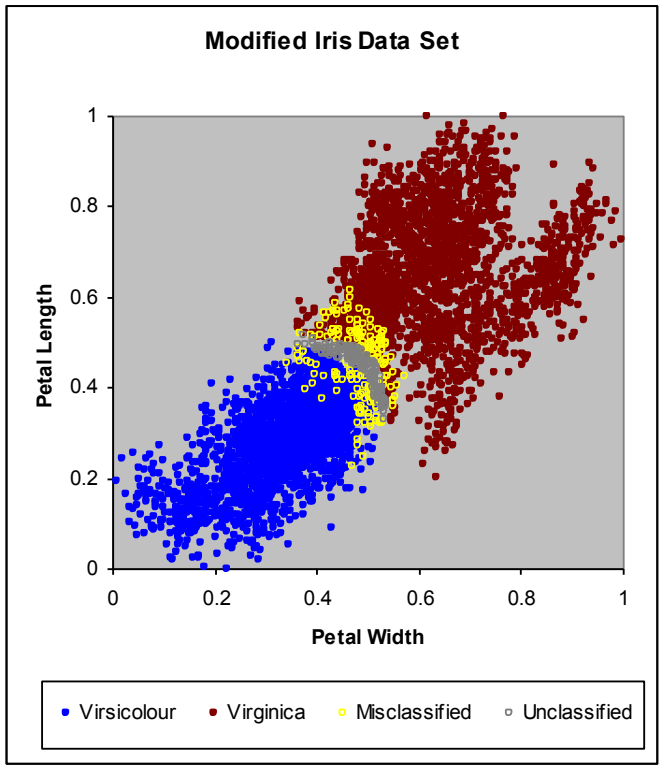

Figure 16: Scatter-plot of *Modified Iris* problem after classification. Note the misclassified and unclassified patterns near the separation boundary. Note the type of Iris discovered as result of the inputs.

E. Modified Iris

For the Modified Iris classification problem, each of the variations used a network with only 1 hidden layer, and GD and GDM used 4 hidden nodes in that layer, whereas RPROP needed only 2. The learning parameters are as follows:

GD had a learning rate of 0.75.

GDM had a learning rate of 0.06, and momentum of 0.85.

RPROP used the paper defined update-values.

As for the number of max epochs specified during training, all three techniques converged, albeit sub-optimally, under 5000 epochs.

The results below are averaged over 25 runs, each with different initial weights.

	GD	GDM	RPROP
classifications	4443.80	4449.64	4423.84
std. dev.	23.18	11.04	37.72

As shown in the results below, it appears that shorter petal width and length indicates an Iris Versicolor, whereas longer petal width and length indicates an Iris Virginica. In reality this appears to be the case.

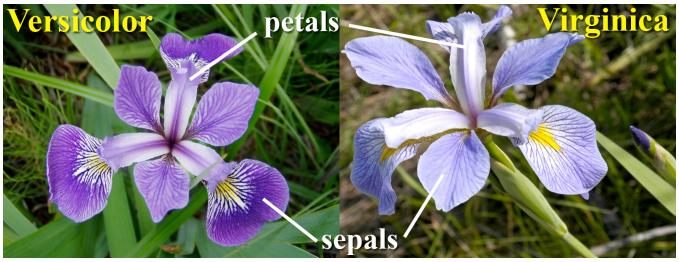

Figure 17: Iris Versicolor and Virginica compared. Note the length and width of the petals. Sepal widths and lengths are not considered in the *Modified Iris* problem.

X. CONCLUSION

Neural networks model neurobiology with nodes and weights representing neurons and synapses. A multi-layer perceptron neural network in particular is a type of NN that, in addition to an input and output layer of nodes, supports any number of hidden layers in between, each with some arbitrary number of nodes (though theoretically any problem of practical interest can be solved using just one hidden layer).

For a relatively long period of time, MLP NNs were not used as there was no practical way to train them. This all changed with the emergence of the back-propagation algorithm. Then, for years afterward, MLP NNs and the BP training algorithm were studied exhaustively, resulting in many variations of minimization functions, some of which are

presented in this paper, as well as optimization to the back-propagation learning algorithm itself.

GD is the default minimization function defined by the BP algorithm. GDM adds a momentum factor that can either speed up convergence or smooth out oscillations during convergence, and is typically applied during batch-mode learning, representing the true mathematical gradient of the error function. RPROP was proposed by *Riedmiller* et al. [3] which does not use the gradient magnitude at all during delta weight changes, but simply the sign of it, and then applies a per-weight scalar value to help with increasing convergence speed.

The research in this paper involved the implementation of an MLP NN program responsible for creating, training and testing NNs, as well as the experimentation of various error minimization functions integrated into the back-propagation learning algorithm.

The program was tested on the simple, but important, XOR classification problem, as well as four different 2D classification problems and one binary encoder/decoder problem. The results are relatively positive and show that all of the variations can be used to train an MLP network successfully, so that it can generalize over unseen data with good results.

Stochastic GD seems to perform the best, but definitely suffers from oscillations when converging. If the network never converges properly and simply reaches the maximum number of training epochs, then it is very possible that the very last value for the network weights is at the very least sub-optimal, but also not even equal to the "best" set of weights encountered throughout the learning process.

GDM can be used to smooth out these oscillations by setting the momentum value relatively high, which additively applies the delta weight changes. While this does seem to improve the learning in such a way as to provide a steady, smooth convergence, this variation has the tendency to force the network to converge to sub-optimal local minimum, since the delta weight change cannot hop out of these minimum as easily anymore.

RPROP seems to have the advantage of fast convergence over GDM, not necessarily stochastic GD, and can be used with hidden layers of much smaller number of nodes for some reason. It is not quite clear why this is the case, and it might just be a result of the parameter tuning used in this paper. For instance, if a greater number of hidden nodes were used, then likely training would have been much faster. But instead a small number of hidden nodes was used (based upon empirical testing of network performance), and so training time (in terms of the number of epochs required to converge) was greatly lengthened.

At any rate, this paper demonstrates that all three variations can be used to train the network successfully to provided relatively good generalization performance.

REFERENCES

[1] CHRISTODOULOU, C. and GERGIOPOULOS, M. *Applications of Neural Networks in Electromagnetics*, Artech House, 2001.

[2] LECUN, Y., BOTTOU, L., ORR, G.B., and MULLER, K.-R. *Efficient BackProp*, Neural Networks: Tricks of the Trade, Lecture Notes in Computer Science, Volume 1524, pp. 9-50, 1998.

[3] RIEDMILLER, M., and BRAUN, H. *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm*, IEEE, 1993.

[4] FAHLMAN, S. *An Empirical Study of Learning Speed in Back-Propagation Networks*, CMU-CS-88-162, September 1988.

[5] LECUN, Y., BOSER, B., DENKER, J.S., HENDERSON D., HOWARD, R.E., HUBBARD, W., JACKEL, L.D. *Handwritten digit recognition with a backpropagation network*, Advances in Neural Information Processing Systems, Volume 2, Morgan Kaufmann, 1990.

[6] TOU, J., TUBBS, S., and GONZALEZ, R. *Pattern Recognition Principles*, Addison Wesley Publishing Company, 1978.

[7] FISHER, R.A. *Iris Data Set*, UCI Machine Learning Repository, http://archive.ics.uci.edu/ml/datasets/Iris, originally published 1936.

[8] GERGIOPOULOS, M. *EEL6812: Introduction to Neural Networks*, Lecture Notes, University of Central Florida, Spring 2011.

[9] GALLANT, STEPHEN. *Perceptron-Based Learning Algorithms*, IEEE Transactions on Neural Networks, Volume 1, Number 2, June 1990.