In *International Journal of Neural Systems*, vol. **12**, no.3&4, pp. 303-318.

IMPROVED BACKPROPAGATION LEARNING IN NEURAL NETWORKS WITH WINDOWED MOMENTUM

Ernest Istook, Tony Martinez
butch@axon.cs.byu.edu, martinez@cs.byu.edu
Brigham Young University
Computer Science Department

Backpropagation, which is frequently used in Neural Network training, often takes a great deal of time to converge on an acceptable solution. Momentum is a standard technique that is used to speed up convergence and maintain generalization performance. In this paper we present the Windowed momentum algorithm, which increases speedup over standard momentum. Windowed momentum is designed to use a fixed width history of recent weight updates for each connection in a neural network. By using this additional information, Windowed momentum gives significant speed-up over a set of applications with same or improved accuracy. Windowed Momentum achieved an average speed-up of 32% in convergence time on 15 data sets, including a large OCR data set with over 500,000 samples. In addition to this speedup, we present the consequences of sample presentation order. We show that Windowed momentum is able to overcome these effects that can occur with poor presentation order and still maintain its speed-up advantages.

Keywords: backprop, neural networks, momentum, second-order, sample presentation order

In *International Journal of Neural Systems*, vol. **12**, no.3&4, pp. 303-318.

1. Introduction

Due to the time required to train a Neural Network, many researchers have devoted their efforts to developing speedup techniques [1 – 7]. Various efforts range from optimizations of current algorithms to development of original algorithms. One of the most commonly discussed extensions is momentum. [3, 8-14]

This paper presents the Windowed Momentum algorithm, with analysis of its benefits. Windowed Momentum is designed to overcome some of the problems associated with standard backprop training. Many algorithms use information from previous weight updates to determine how large an update can be made without diverging [1,13,15]. This typically uses some form of historical information about a particular weight's gradient. Unfortunately, some of these algorithms have other difficulties to deal with [3,10] as discussed in section 2.

Windowed Momentum relies on historical update information for each weight. This information is used to judge the worth of the individual updates resulting in better overall weight changes. Windowed Momentum achieved an average speed-up of 32% in convergence time on 15 data sets, including a large OCR data set with over 500,000 samples with same or improved accuracy. The computational complexity of Windowed Momentum is identical to that of Standard momentum with only a minor increase in memory requirements.

A background of standard neural network (NN) training with momentum and other speed-up techniques is discussed in Section 2. In Section 3, Windowed Momentum is analyzed. Section 4 describes the methodology of the experimentation. The results of

the experiments are presented in Section 5. Section 6 summarizes the efficacy of Windowed Momentum and presents further directions for research.

2. Background

With the Generalized Delta Rule (GDR), small updates are made to each weight such that the updates are proportional to the backpropagated error term at the node. The update rule for the GDR is

(1) $\Delta w_{ij}(t) = \eta \delta_j x_{ji}$

where i is the index of the source node, j is the index of the target node, η is the learning rate, δ_j is the backpropagated error term, and x is the value of the input into the weight. This update rule is effective, but slow in practice. For standard backprop the error term for output nodes is equal to $(t_j - o_j) * f'(net_j)$ where t_j is the target for output node j and net_j is the activation value at node j. With a sigmoid activation function, $f'(net_j) = o_j * (1 - o_j)$, where o_j is the actual value for output node j. This makes the final local gradient term

(1b) $\delta_j = (t_j - o_j) * o_j * (1 - o_j)$

The gradient for hidden nodes, where I is the input at that node, is calculated using

(1c) $\delta_h = \mathrm{I} o_h * (1 - o_h) * \sum_{k \in outputs} w_{kh} \delta_k$

In a binary classification problem typical values are $0 \leq t_j, o_j \leq 1$. Therefore, the maximum this local gradient term can take on is 0.1481. When combined with typical values for η [~0.2 … 0.5] and non-saturated weights, the change to weights each update is on the order of 10^{-2}. Although increasing the learning rate will result in larger updates, the likelihood of convergence decreases [3, 16]. Due to the gradient descent nature of backprop, many iterations are required for convergence and the individual steps must be small. An excessive learning rate can disrupt the gradient descent and possibly miss the local minimum; therefore a solution is not guaranteed.

2.1 Update Styles

When training a NN, weight updates can occur at several times. The two most common methods are called online and batch training. In batch training, all samples are presented and the sum of the error at each weight is used to perform one single update. In other words, the error is calculated for all samples before a single update is made such that the error is based on $\sum_j \delta_j$ - over all *j* samples. Conversely, online training updates weights after *each* sample is presented, thus providing *j* updates – one for each sample. This update style was used for the experiments presented here. By using batch training, fewer updates are made, but each update considers the entire set of samples. In online training, the current sample and the most recent samples (in the case of using momentum) are the only factors.

In addition to batch and online training, there is also semi-batch training. In semi-batch training, updates are made after every *k* samples. This gives more frequent updates than

batch mode and more accurate estimates of the true error gradient. Wilson and Martinez [17] discuss the efficacy of batch training when compared to online and semi-batch training. They conclude that batch training is ineffectual for large training sets and that online training is more efficient due to the increased frequency of updates.

Although frequent updates provide a speed up relative to batch mode, there are still other algorithms that can be used for improved performance. One of these is known as standard momentum.

2.2 Standard Momentum

When using momentum the update rule from equation (1) is modified to be

$$\Delta w_{ij}(t) = \eta \delta_j x_{ji} + \alpha \Delta w_{ij}(t\text{-}1) \quad (2)$$

where α is the momentum. Momentum in NNs behave similar to momentum in physics. In *Machine Learning* by Tom Mitchell [18, p. 100], momentum is described:

> "To see the effect of this momentum term, consider that the gradient descent search trajectory is analogous to that of a (momentumless) ball rolling down the error surface. The effect of α is to add momentum that tends to keep the ball rolling down the error surface."

In order to speed up training, many researchers augment each weight update based on the previous weight update. This effectively increases the learning rate [9]. Because many updates to a particular weight are in the same direction, adding momentum will typically result in a speedup of training time for many applications. This speedup has been shown to be several orders of magnitude in simpler applications with lower complexity decision boundaries [18].

Even though momentum is a well-known algorithm in the neural network community, there are certain criteria that have been considered when extending momentum. Jacobs [1] enumerates these criteria and analyzes the problems with basic backprop by giving several heuristics that should be used in neural network design:

- Each weight should have a local learning rate
- Each learning rate should be allowed to vary over time
- Consecutive updates to a weight should increase the weights' learning rate
- Alternating updates to a weight should decrease the weights' learning rate

Because Jacobs' Delta Bar Delta algorithm is criticized for being overly sensitive with respect to parameter selection [3] the approach taken with this research has sought to minimize the difficulty associated with parameter selection. This goal is placed at a higher priority when compared against the four principles just mentioned.

Another attempt to improve momentum is called Second-Order Momentum. The idea is that by using second order related information, a larger update can be made to the weights. This particular algorithm was shown to be ineffective in [10], but Magoulas and Vrahatis [19] have found success using second order error information in other algorithms to adapt local learning rates.

2.3 Momentum Performance

Although momentum speeds up performance, there are times when an improper selection of the momentum term causes a network to diverge [3, 16]. This means that instead of distinguishing one class from another, the network will always output a single selection. In one batch of experiments done by Leonard and Kramer [2], 24% of the trials did not converge to a solution in a reasonable amount of time.

Because of these problems, various efforts have focused on deriving additional forms of momentum. One such method is to give more weight to previously presented examples and less weight to the current example. By so doing, momentum is preserved when a sample attempts to update contrary to the most recent updates. Jacobs [1] efforts to face these problems resulted in:

(2b) $\Delta w_{ij}(t) = (1 - \alpha)\eta\delta_j x_{ji} + \alpha\Delta w_{ij}(t\text{-}1)$

This variant can give more influence to the historical updates based on the value of α. It should be noted that setting α to zero in equation (2b) results in equation (1). Unfortunately, the performance of equation (2b) in Jacob's work was comparable to the general performance from equation (2).

2.4 Sample Presentation in standard momentum

Momentum is a locally adaptive approach. Each weight *remembers* the most recent update. By doing this, each weight is able to update independent of other weights. This helps adhere to the principles previously mentioned by Jacobs [1]. Building on momentum can show improved performance, as seen in the SuperSAB algorithm by Tollenaere [6]. He integrates the standard form of momentum from (1) with a locally adaptive learning rate per node. His work, like that of Jacobs, uses a linear increase in local learning rates. Locally adaptive approaches, however, put a stronger emphasis on the most recent updates.

Because the presentation order is often randomized in machine learning algorithms, there will be circumstances where consecutive samples produce alternate directions of weight

updates. The problem with these approaches relates to presentation order. The worst-case scenario occurs if updates to any particular weight alternate directions. In those situations, only the most recent update will be important.

Many algorithms are susceptible to this problem. By using successive updates to determine changes in momentum, the presentation order becomes critical. All momentum-based algorithms suffer from this same shortcoming. In order to alleviate this problem, an algorithm should be robust to the sample presentation order.

2.5 Momentum Parameters

Neural networks have other parameters that can cause difficulty. In addition to having a momentum rate and learning rate, the network topology, activation function, and error functions must be chosen. The large number of these parameters can make network design difficult. Neural networks appear to be especially sensitive to the selection of learning rate and momentum. One specific property that Tollenaere mentions is that the higher a rate of momentum, the lower the learning rate should to be. This dependence between learning rate and momentum makes network initialization more complex.

Because of the potential of momentum, researchers have devoted their efforts to optimizing the effectiveness of momentum. Momentum can speed up learning, but there is still a large set of parameters that must be tuned for many momentum based equations. For example, the SuperSAB algorithm by Tollenaere [6] has various parameters to govern the increase and decrease of each local momentum rate. There are separate parameters for increasing and decreasing the momentum rate.

Although momentum can have a positive effect, introducing additional parameters can mitigate these improvements. By increasing the number and sensitivity of parameters, a learning task becomes prefaced by a phase of parameter optimization. This phase complicates the problem and can overcome the targeted speed gains.

3. Windowed Momentum

In this paper we propose the Windowed Momentum algorithm that is derived from basic momentum and locally adaptive algorithms. By augmenting the historical scope of the local weights, there is more freedom to use information obtained on previous updates. Windowed Momentum uses a fixed width window that captures more information than that used by standard momentum. By using more memory it is possible to overcome the problems of presentation order.

By using Windowed Momentum, one can establish how much of a history to use. This approaches batch training by allowing more samples to affect each update, but the regularity of updates is akin to online training. This gives the Windowed Momentum algorithm additional performance over batch training because more updates are made with fewer computations.

3.1 Principles of Windowed Momentum

The Windowed Momentum algorithm is a method that functions in a way similar to standard momentum. Momentum functions by remembering the most recent update to each weight and adding a fraction of that value to the next update. Windowed

momentum remembers the most recent *n* updates and uses that information in the current update for each weight. With standard momentum, the error from one previous update is partially applied. In the worst case, some consecutive samples will have opposite updates. This situation can disrupt the momentum that may have built up and it could take longer to train. Windowed momentum is able to look at a broader history to determine how much affect momentum should have.

In order to remove the problems related to presentation order, a sliding window is used which treats the most recent *k* updates as members of a set as opposed to elements in an ordered list. This eliminates the problems of alternating updates, but the computation still needs to be manageable. This is achieved with rolling averages. An accumulator is used to store the current sum of the sliding window. Then, as the window slides forward the oldest element is removed from that accumulator and the newest element can be added with a minimal number of operations. An example showing individual updates and their rolling averages is shown in Figure 1. This figure shows that averaging the previous 5 updates can smooth a chaotic sample presentation.

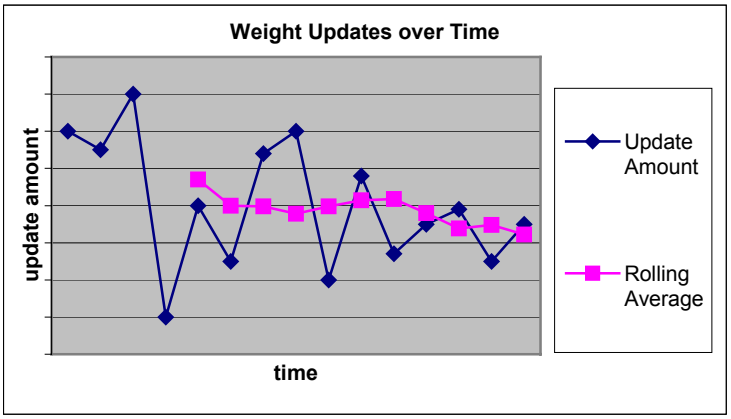

Figure 1 – Using a window size of 5 can smooth chaotic presentations

3.2 Analysis of Windowed Momentum

A performance analysis of the Windowed Momentum algorithm begins with the weight level. Assume w is a single weight and T is the set of all training points. Let Δw_e be the change in the weight that would come from training on $e \in T$. Batch mode operation would compute a Δw_T that considers all $e \in T$.

For Windowed Momentum, a set $S \subseteq T$ is used to approximate the gradient at weight w. The set S is comprised of the most recent samples that have been trained. In traditional batch mode training, the update to w is

$$(4)\ \Delta w_T = \sum_x^T \frac{\Delta w_x}{|T|}.$$

Windowed Momentum computes

$$(5)\ a = \sum_x^S \frac{\Delta w_x}{|S|}$$

which gives an estimate to the traditional Batch mode gradient. At each presentation of a sample t, Δw_e is compared to a. If a and Δw_e are in the same direction, either both positive or both negative, then the weight is updated $w = w + \Delta w_e$. Otherwise the Δw_e is considered spurious and w remains unchanged. By using an approximation to the true gradient, the likelihood of updating a particular weight contrary to the true gradient is reduced.

In order to maintain efficiency, the set S is comprised of the most recent samples trained. After a sample t is presented and the average Δw_e is computed, the oldest element of S is

removed and t is added to S. The benefit of Windowed momentum is realized because an update can be made with every sample presented and any updates made are more likely to be in the direction of the true gradient. Additionally, training occurs much quicker because each presentation of a sample is likely to update several weights. Instead of waiting until the end of an epoch, Windowed Momentum updates weights for each presented sample. By approximating the true gradient and not postponing weight updates, Windowed momentum is able to converge on a solution considerably faster than batch mode training. This is shown empirically in section 3.3.2. In the extreme case, $S = T$, the information being used would be similar to batch training. The primary difference is that more updates will be made than with batch training.

3.3 Windowed Momentum Formula

To create windowed momentum, equation (1) is altered to

$$(6) \qquad \Delta w_{ij}(t) = \eta \delta_j x_{ji} + f(\eta \delta_j x_{ji}, \Delta w_{ij}(t-1), \Delta w_{ij}(t-2), \ldots, \Delta w_{ij}(t-k)).$$

Equation (6) makes use of a function f creating a family of Windowed Momentum functions. This function determines how to use the history of weight updates for each particular weight. There are k+1 arguments to the function f, the first is the current update and the remainder are the k previous updates where k is the window size for the windowed momentum algorithm.

In order to minimize the sensitivity of parameters, the f function used here has no adjustable parameters aside from window size. An analysis of the window size is discussed in section 4.1.3. However, the goal is to obtain an algorithm that will still do

reasonably well with sub-optimal parameters. To optimize learning speed simple formulae are favored.

The following f function is tested:

$$(7) \qquad f(z, \delta_1, \ldots, \delta_k) = \begin{cases} -z & if \quad z * \sum \delta_i < 0 \\ 0 & if \quad z * \sum \delta_i \geq 0 \end{cases}$$

Equation (7) ignores any updates that are in the opposite direction from the sum of the most recent k updates. Updates that do not conform to the recent gradient are ignored.

3.4 Computational Complexity of Standard Momentum

When training a multi-layer perceptron, there are several factors that determine the computational complexity. We will assume that any multi-layer network can be computed with a single hidden layer network [20 – 22], thus we only need consider the single hidden layer scenario. It is assumed that the network is fully connected.

For this discussion, the following terms are used:

i = number of input nodes
n = number of hidden nodes
o = number of output nodes

The dominating factor in training is the number of weights. For the fully connected case, there are $n * i$ weights from the inputs to the hidden nodes, and $n * o$ weights from the hidden nodes to the output nodes. This gives a total of $n * (i + o)$ weights. Because the number of inputs and outputs for a given problem are generally fixed, the only variable term is n.

In terms of complexity, the feed forward phase of multi-layer perceptrons is fixed. This leaves the back propagation of error phase. The computation of error at the output nodes uses the same backpropagation error term as mentioned in equation (1b), which is repeated here for convenience.

(1b) $\delta_j = (t_j - o_j) * o_j * (1 - o_j)$

This error term is O(1) for each output node and that there are $o*n$ output weights, which yields O($o*n$) for the output layer. After computing the error at the output nodes, the error at each hidden node is computed. Using the same assumptions as for equation (1b)

(3) $\delta_h = o_h * (1 - o_h) * \sum_{k \in outputs} w_{kh} \delta_k$

where w_{kh} represents the weight from node h in the hidden layer to node k in the output layer. Note that this computation is O(o) for each hidden node and that there are $n*i$ weights, yielding O($i*o*n$) for the entire hidden layer.

The total order of complexity is O($i*o*n + n*o$) or O($n * o * (i+1)$) for training a single epoch. When using standard momentum, this order of complexity is unchanged but fewer epochs are required for convergence than without momentum. Standard momentum merely requires additional storage for each weight. Storage requirements are directly proportional to the number of weights. With $n * (i + o)$ weights, there are $2 * n * (i + o)$ values to store.

3.5 Computational Complexity of Windowed Momentum

Windowed Momentum has the same time complexity as Standard momentum. In order to compute the complexity of Windowed momentum, the complexity of Standard momentum is used as a staring point. As stated in the section 3.4, the order of complexity for training is O($n * o * (i+1)$) per epoch. When training with Windowed momentum, each weight update is conditional on the previous k updates – where k is the window size used. This raises the complexity for training to O($n * o * (i+1) * k$) per epoch.

In order to efficiently compute the weight updates with Windowed momentum, there are additional storage requirements. At each weight, an additional k - 1 values are needed. This raises the memory requirements from $2 * n * (i + o)$ numbers with standard momentum to $k * n * (i + o)$ numbers for Windowed momentum.

There are additional optimizations that can improve the computation speed in exchange for additional memory requirements. Since the most recent k weight changes for each weight are already stored it is simple to iterate and compute the sum. However, instead of iteratively computing the direction of the previous k weight changes, an accumulator can be used to store the sum. Since the most recent k weight changes must be stored already, it is a trivial task to subtract the oldest weight change from the accumulator and add the newest weight change to the accumulator. By doing this, the computational complexity of Windowed momentum becomes the same as standard momentum. The cost for this is adding an accumulator for each weight, thereby increasing the required storage space to $(k + 1) * n * (i + o)$ numbers.

3.6 Combination Momentum

Without addition of time complexity and a minor increase in memory required, one could use Windowed momentum in conjunction with Standard momentum. We call this merging of algorithms Combination momentum. Combination momentum starts with Windowed Momentum, which is defined in equation (6) from section 3.3:

$$(6) \qquad \Delta w_{ij}(t) = \eta\delta_j x_{ji} + f(\eta\delta_j x_{ji}, \Delta w_{ij}(t-1), \Delta w_{ij}(t-2), \ldots, \Delta w_{ij}(t-k)).$$

For simplicity, let

$$(8) \qquad f_t = f(\eta\delta_j x_{ji}, \Delta w_{ij}(t-1), \Delta w_{ij}(t-2), \ldots, \Delta w_{ij}(t-k)).$$

Then combine equation (6) with (8), resulting in

$$(9) \qquad \Delta w_{ij}(t) = \eta\delta_j x_{ji} + f_t.$$

Next incorporate the momentum parameter α and the momentum term:

$$(10) \qquad \Delta w_{ij}(t) = \begin{cases} \eta\delta_j x_{ji} + \alpha\Delta w_{ij}(t) & \text{if} \quad f_t = 0 \\ 0 \quad \text{if} \quad f_t \neq 0 \end{cases}$$

Note that the update becomes a standard momentum based update or no update at all.

4. Experimental Methods

[Note: An on-line appendix found at http://axon.cs.byu.edu/~butch/wm_appendix01.ps. contains additional information relating to the experiments discussed in this paper. The full tables of results pertaining to the UCI data sets are found first, followed by the results of varying window size for the UCI data sets. Next is a reproduction of the letter

distribution found in Hahn [1994]. Finally, the full results pertaining to the character recognition experiments are listed.]

There were two major rounds of experimentation. The preliminary round was designed to explore the properties of Windowed Momentum using relatively small data sets. During this round of experiments we tested the effects of window size and learning rate and the effects of sample presentation order. The final round of experiments used the results of round one and applied them to a large real-world data set. In all cases a standard sigmoidal activation function was used.

4.1 Preliminary Experiments

In this round of experimentation we performed three main trials. Windowed Momentum was tested in conjunction with varying learning rates and window sizes.

Unless otherwise noted, each experiment was run 20 times with different random seeds and the results were averaged to produce the final statistics. All stopping criteria were based purely on the results from the training data and each test was run until either of two stopping criteria was met. If the sum-squared error did not decrease on the training set for 50 epochs then the results were computed and the test halted. Additionally, if the sum-squared error decreased below a certain threshold the results were computed and the test halted. This threshold was arbitrarily fixed at 4 for the duration of the experiments. The error and accuracy values reported come solely from test set performance.

A comparison was not made to batch or semi-batch training because the comparison would not be accurate. Batch training has been shown to be less effective [17] and the use of semi-batch would perform *k* times slower with a *k* sample batch size.

Separate experiments were conducted with varying learning rates and varying window sizes to determine general performance of Windowed Momentum as compared to Combination Momentum and Standard Momentum. The number of epochs required to converge and the accuracy were both measured.

4.1.1 Data Sets

The following data sets were used for the experiments in this round. Each data set was separated into a training set and a test set with 70% of the data used for training. All data sets listed come from the UC Irvine Machine Learning Database.

abalone – 4,177 instances with 7 numeric attributes and one nominal attribute. The goal is to predict the age of an abalone from physical measurements.

balance – 625 instances with 8 numeric attributes. The output class represents the direction the scale will tip towards.

cmc – 1,473 instances with 5 numeric attributes and 4 nominal attributes. The inputs represent socio-economic characteristics of married women and the goal is to determine the contraceptive method currently used.

derm – 366 instances with 34 numeric attributes. This task is to determine the presence of various skin diseases based on personal and family history information.

digit – 5,620 instances with 64 numeric attributes. Standard recognition of hand-written digits.

ecoli – 336 instances with 7 numeric attributes. The goal is to determine the localization site of various proteins based on the input attributes.

glass – 214 instances with 9 numeric attributes. Various measurements of the contents of glass samples are used to determine the usage of the glass sample.

haberman – 306 instances with 3 numeric attributes. The output classes are whether or not a patient survived 5 years after breast cancer surgery.

iris – 150 instances with 4 numeric attributes. This classic machine learning data set classifies the species of various iris plants based on physical measurements. This data set was only used in the sample presentation order experiments.

ionosphere – 351 instances with 34 numeric attributes. This data set classifies the presence of free electrons in the ionosphere.

pendigits – 10,992 instances with 16 numeric attributes. This is another set for handwritten digit recognition.

pima – 768 instances with 8 numeric attributes. The predictive class in this data set is whether or not the tested individual has diabetes.

spam – 4,601 instances with 57 numeric attributes. The goal is to distinguish unsolicited emails from normal emails based on word frequencies.

wine – 178 instances with 13 numeric attributes. The attributes give various parts of the chemical composition of the wine and the task is to determine the wines' origins.

yeast – 1,484 instances with 8 numeric attributes. The task is to predict the localization site of various proteins based on the composition of the wines.

Further experiments tested Windowed Momentum on a large data set consisting of samples of handwritten characters. This OCR data consists of over 500,000 samples of characters and numbers that have been hand classified. The letter distribution followed that of the English language. The results of the preliminary experiments were used to determine learning rate, window size and momentum settings. A separate NN was trained for each desired output class using approximately 80% of the data for training and the remaining 20% as test data. Each sample was partitioned into an 8 x 8 grid for inputs. Unless otherwise mentioned a window size of 100 was used.

The number of nodes used for each Data set is listed below.

Data Set	Hidden Nodes Used
Abalone	11
Balance	12
Cmc	14
Derm	51
Digit	96
Ecoli	11
Glass	13
Haberman	5
Iris	6
Ionosphere	51
Pendigits	24
Pima	12
Spam	86
Wine	20
Yeast	12

4.2 Sample Presentation Order

When training a NN, the standard approach has been to randomly shuffle the training samples before presentation. However, different approaches to sample presentation order can affect the training speed. This set of experiments was designed to determine the effects of presentation order.

The presentation order used was either shuffled or alternated. Shuffled data is synonymous to random presentation order within an epoch. Alternated data trains alternately on positive and negative examples. Both presentation orders used the same random seed so that all data was arranged in identical orderings. Every sample in the training set was trained on with shuffled data, but with alternated data samples of the class last trained on are skipped. For example, when a positive sample was presented

immediately after another positive sample it was skipped. It is worth noting that the number of samples trained on per epoch using shuffled presentation order is a constant while the number of samples trained on per epoch varied using alternated presentation order. The number of samples used in a single epoch was approximately 40% less than that used in shuffled presentation.

The momentum rule used was Standard, Combination or no momentum. Standard momentum is the same as mentioned in Equation (2) in section 2.2. Combination momentum is the conjunction of both Standard momentum and Windowed momentum as shown in equation (10) in section 3.5. Combination Momentum was chosen in lieu of Windowed Momentum because Combination Momentum empirically performed better than Windowed Momentum for low learning rates.

The remainder of parameters in each NN was identical. In order to easily determine the results a low learning rate (0.05) was used. A subset of the UCI data was chosen for this experiment.

5 Experimental Results

This section contains the results and basic analysis of the experiments described in Section 4.

The preliminary experiments tested accuracy and convergence speed of Standard Momentum, Combination Momentum and Windowed Momentum. First the results from the UC Irvine data sets are presented followed by the OCR data results. Finally the results related to sample presentation order are presented.

5.1 Learning Rate

In order to determine the effectiveness of Windowed and Combination Momentum relative to Standard Momentum, we tested the UCI data sets at a variety of learning rates ranging from 0.05 to 1.0 at increments of 0.05. The momentum parameter was held constant at 0.4. This value was determined based on initial experimentation. It was shown to be enough momentum to recognize its effect without increasing the risk of divergence. For each learning rate and data set combination, 20 separate test runs were conducted. The topology of the NN was a single hidden layer that was fully connected to all input and output nodes. The size of the hidden layer was arbitrarily set at 1.5 times the number of inputs.

Figures 2 and 3 show summary information that was obtained by averaging the performance over all UCI data sets and all test runs. In Figure 2 it is shown that the quickest convergence consistently came from Combination Momentum. At the lowest tested learning rate of 0.05, Standard Momentum took approximately 370 epochs to converge while Combination Momentum took about 205 epochs and Windowed Momentum took around 295 epochs. As the learning rate increased, all three algorithms tended to converge in approximately the same number of epochs with Windowed and Combination momentum taking slightly fewer epochs.

Although speed to converge is important, one cannot ignore the accuracy of the tested NNs. The Combination Momentum algorithm performed poorest when subjected to a high learning rate, but it had the greatest performance for low learning rates. Overall, Windowed and Standard momentum produced equivalent accuracy. The point at which

the highest accuracy for Windowed and Standard Momentum was obtained corresponded to a learning rate of 0.2 and at this point Windowed Momentum took about 16% fewer epochs. The window size experiments further improve on this speed-up and used 0.2 as the learning rate because the best accuracy obtained using Standard Momentum was at this point.

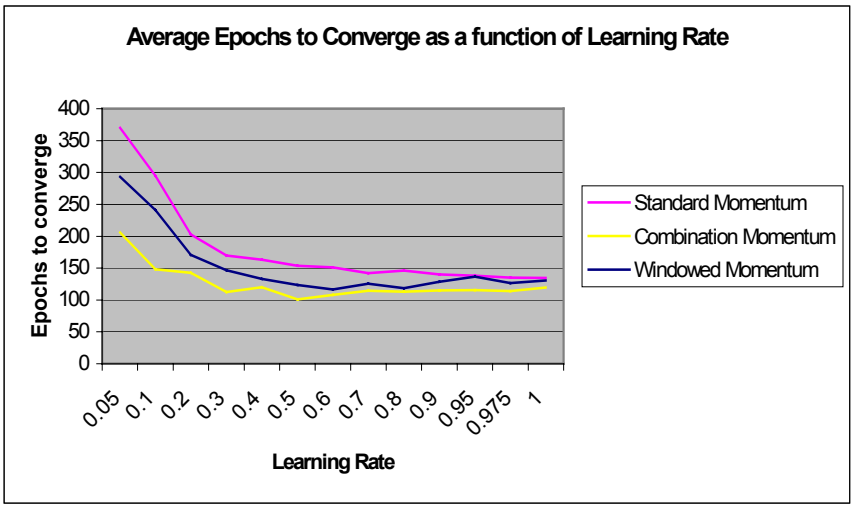

Figure 2 – Comparison of convergence time on the UCI data sets

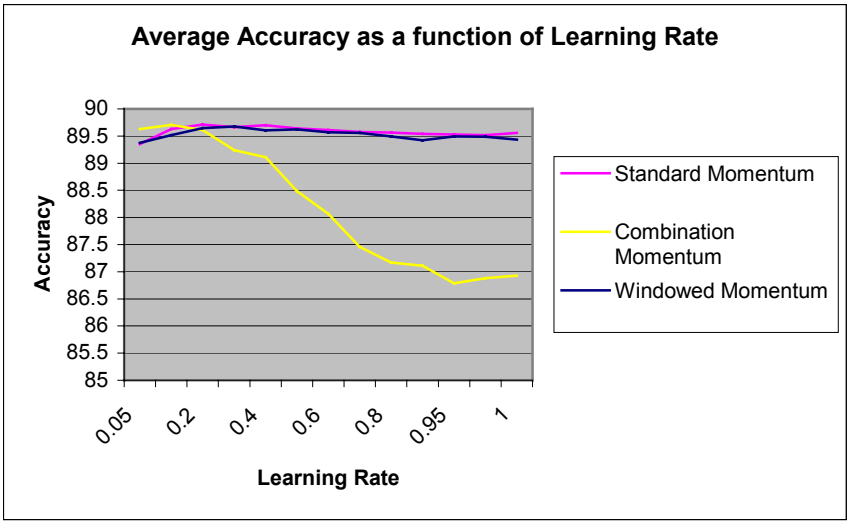

Figure 3 – Comparison of accuracy on the UCI data sets

After analyzing the results, it becomes apparent that Combination Momentum is overly aggressive when the learning rate gets large. This occurs because the individual weight updates get too large too quickly. By having such large updates, the ability to perform a successful gradient descent is lost. Windowed Momentum shows the best balance of speed and accuracy.

5.2 Window Size

The next experiment was designed to observe the effects of varying window size of Windowed Momentum. The window sizes used ranged from 5 to 50 at increments of 5 and from 60 to 150 in increments of 10. The learning rate was held constant at 0.2. Again we used the UCI data sets and the network topology was identical to that listed in 4.1.2.

Figures 4 and 5 show the behavior of convergence speed and accuracy as a function of window size. These results are averaged over all test runs in the same manner described in section 5.1.

By increasing the window size the time to converge decreased by an additional 16% (from 170 down to 142) as shown in figure 5. Also note in figure 4 that accuracy increases as window size increases. These experiments confirm that a larger window size improves training speed. When comparing the results of Standard Momentum with the same learning rate of 0.2 we are able to decrease time to converge from 202 down to 142. This shows that Standard Momentum required approximately 31% more epochs on average to converge.

When looking at the behavior relative to window size it becomes apparent that the accuracy improves when the window size increases. This occurs because the weight change is considering more data points at each update. This allows the NN to make smarter updates that allow for more accurate recognition. The effect of varying the window size on a large data set is shown in the next section.

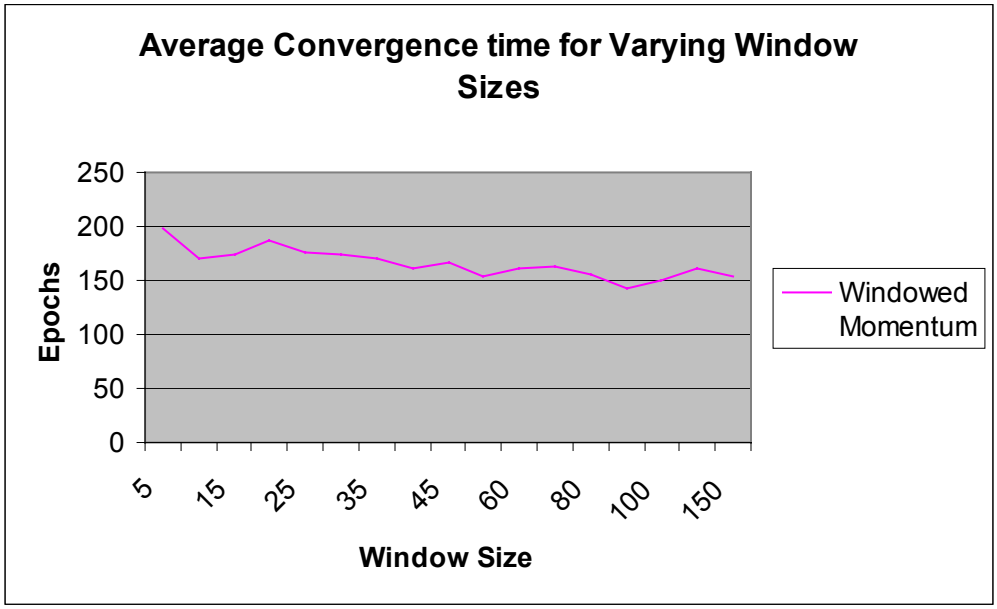

Figure 4 – Accuracy on UCI data sets with varying window size

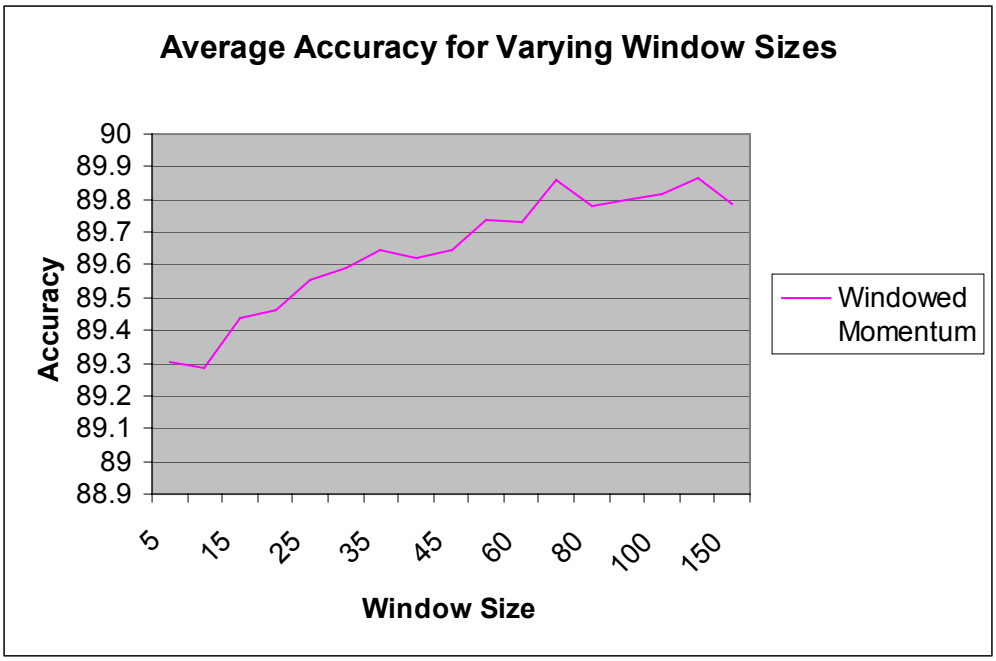

Figure 5 – Convergence time for UCI data sets with varying window size

5.3 OCR Experiments

An initial experiment was conducted to determine the required time to train since run-times are so long. A NN was trained with the letter 'a' as the positive class. The performance on the training set at each epoch was recorded. After approximately 45 epochs there was little to no improvement on the test set. Based on these results we report only the accuracy of the other letters for the first 50 epochs. The letters were still trained on until completion so that the convergence results could be reported.

Due to the large time requirements for training, a subset of letters was selected. This selection was made based on letter frequencies in the English language. There are many sources of letter distributions so we arbitrarily selected one that was based on occurrences in Dickens' A Tale of Two Cities [23]. This letter distribution has been used in

cryptographic research [24]. We then selected every other letter from that list so that a variety of common and rare letters could be trained. This resulted in half of the 26 English letters being trained. Letters with distinct upper and lower case variants were trained on both versions.

Further experiments were conducted to determine convergence speed. Due to the extreme time requirements on such a large data set each letter variant was only tested twice. The parameter settings were identical to those of the 50-epoch limited experiment.

Finally the window size was varied from 25 to 300 in increments of 25. This was done on a single letter with 3 random runs for each value of window size.

Over all the letters trained, Standard Momentum took 1409 epochs to converge on average. Windowed Momentum required 32% less epochs and Combination Momentum required 52% less epochs. Table 1 shows the average convergence time for all the letters.

Table 1 – Total Epochs to Converge on OCR data (based on training set)

	Combination	Windowed	Standard
a	1184	946	1440
A	448	613	1508
c	591	1147	1657
C	993	1442	1470
e	1114	877	1344
E	384	1065	1639
g	197	1695	2204
G	672	597	608
h	894	671	1526
H	543	1074	1230
j	786	1167	1467
J	582	928	1947
I	610	1895	2730
L	607	1060	1492
M	717	1211	1360
n	1364	1206	1162
N	448	967	1844
P	526	678	1245
q	298	466	444
Q	467	303	455
r	1412	1181	1694
R	470	564	1003
V	293	331	952
Average	678.26	960.17	1409.60

The next experiment is the accuracy test. Over all the letters trained, there seemed to be two types of results. On 59% of the letters the Windowed momentum algorithm clearly achieved better results. One example of this behavior is shown in figure 6. The remaining 41% appeared to have no significant differences in performance. The performance of Combination Momentum gave consistently higher error over all letters. The average error over all letters is shown in figure 7. The full set of individual letter graphs is available in the online appendix.

General performance of windowed momentum varies from comparable to better depending on the particular letter. Windowed Momentum never did consistently worse than Standard Momentum for any of the letters tested.

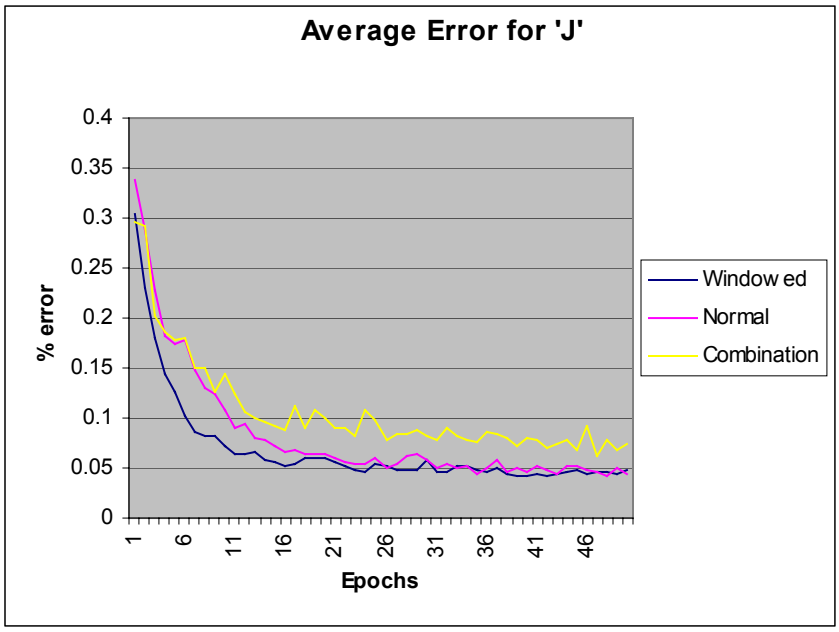

Figure 6 – Classification accuracy on OCR data for a single letter based on test set results

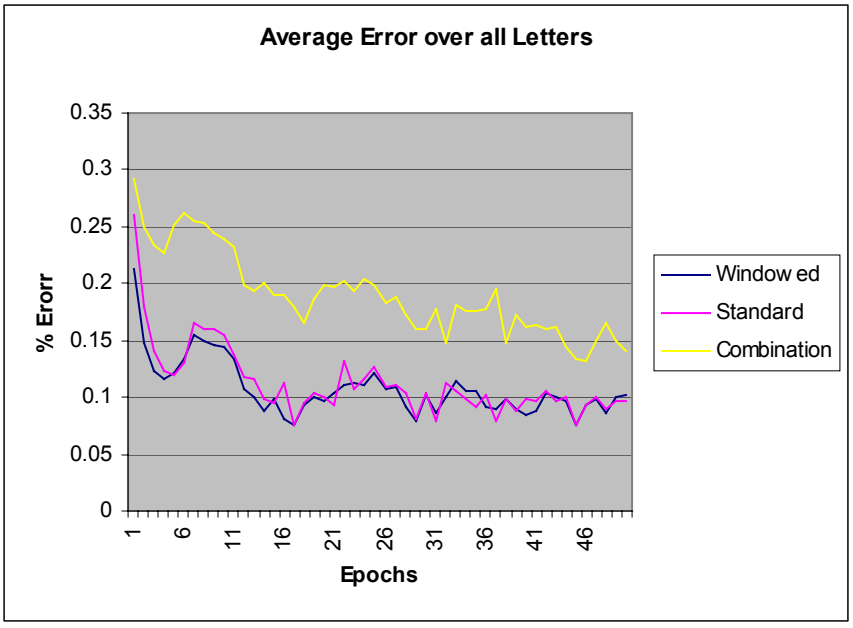

Figure 7 – Classification accuracy on OCR data averaged over all letters based on test set results

In order to determine how varying window sizes affected the time to converge, the letter ‘e’ was trained with window sizes ranging from 25 to 300 inclusive. Each window size was started with three separate random seeds. The results are shown in Figure 8.

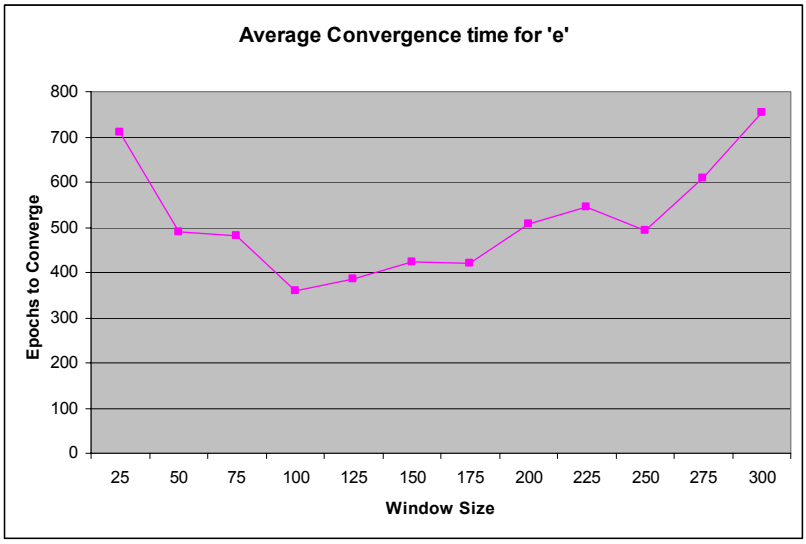

Figure 8 – Convergence time on OCR data with varying window size

The minimum convergence speed occurs with a window size of 100. This closely correlates with the optimum window size of 90 for the UCI data sets. An increased window size tends to lengthen the convergence time due to the gradient information becoming out-of-date. Because the information used in a window size of 200 was computed over the previous 200 samples presented, it becomes increasingly less reliable.

5.4 Sample Presentation Order

For this experiment the derm, iris, and ionosphere datasets were arbitrarily selected. The results are found in the table below. The results for the Ionosphere data set with the Shuffled presentation order (italicized in the results) diverged in a few of the test runs. In these cases, there was a single sample that was being misclassified. The epoch at which all other points were correctly classified was used for determining convergence time for the Ionosphere dataset.

Table 2 – Average epochs to converge with varying presentation order

	Alternated			Shuffled		
	derm	iris	ionosphere	derm	iris	ionosphere
Windowed M.	57.3	16.1	133.2	56.2	18.2	*128.2*
Combination M.	48.1	13.7	122.7	49.1	9.4	*120.8*
Standard M.	68	111.4	669.2	87.8	288.1	*1218.4*
No Momentum	89.5	200.5	1267.4	113.1	577.7	*3712.8*

To explain the difference in convergence times we must consider the behavior associated with training the NN. Assume a positive sample is presented to a single output NN and is misclassified. During training, this will result in the output node being updated to produce a more positive value. The net effect of training any sample is to bias the NN in the direction of that sample. Training on several consecutive samples from a single class tends to over bias the output nodes against the other output classes. This hinders the training of the NN and results in a longer training time when compared to Alternated presentation.

For the Standard and No momentum cases, Alternated presentation order took between 21% and 66% fewer epochs to converge and never took longer then Shuffled presentation order. However, it is interesting to note that the Windowed and Combination momentum algorithms converged quicker than the Standard or No momentum cases. Additionally, note the similarity in convergence times for Alternated and Shuffled presentation order within the Combination and Windowed momentum tests. The small difference in the number of epochs for Windowed Momentum to converge shows that the algorithm is able to overcome the affects of sample presentation order.

6 Conclusion

Windowed Momentum achieved an average speed-up of 32% in convergence time on 15 data sets, including a large OCR data set with over 500,000 samples. Windowed Momentum is also able to overcome the effects that can occur with poor presentation order and still maintain its speed-up advantages. Accuracy on all data sets was same or improved over Standard momentum. This algorithm gives several new directions for research and can be used anywhere that Standard momentum is used.

6.1 Future Research

Future work will examine additional *f* functions. One additional *f* function was examined during the course of the original experimentation but was unsuccessful. Further optimizations can determine appropriate constants to improve this new *f* function. For this equation, let n equal the percent of δ_i in the same direction as z, and let θ be a scaling parameter:

$$(11) \qquad f(z, \delta_1, \ldots, \delta_k) = \begin{cases} 2 * zn & if \quad n < 0.5 \\ z * (1 + (2n - 1)(\theta - 1)) & if \quad n > 0.5 \end{cases}$$

The affect of this particular *f* function is to diminish the effect of the update when less than half of the recent updates are in the same direction. Conversely, the update is increased proportional to the percentage of historical updates that are in the same direction. If all updates from the window are in the same direction then the update will be multiplied by θ. If all updates from the window are in the opposite direction from the update being considered then there will be no change applied to the weights.

In addition to the investigation of Windowed Momentum the behavior related to sample presentation order merits further analysis. Larger and more complex data sets or data sets with large (10+) numbers of classes may require a different style of algorithm to improve performance.

One alternative *f* function that was tested used a history that was diminished over time. An update considered from *n* time steps ago was decayed by 0.95^n. This produced poor results but a full examination and optimization phase was not performed.

Finally, the Windowed Momentum algorithm can be altered to use the historical information for the weight updates. Instead of comparing the average from the previous *k* updates to the current update, the average can be used in place of the current update.

7. Bibliography

1. Jacobs, Robert A., "Increased Rates of Convergence Through Learning Rate Adaption", Neural Networks, Vol. 1, pp 295-307, 1988.

2. Leonard, J. and Kramer, M. A.: Improvement of the Backpropagation Algorithm for Training Neural Networks, Computers Chem. Engng., Volume 14, No. 3, pp 337-341, 1990.

3. Minai, A. A., and Williams, R. D., Acceleration of Back-Propagation Through Learning Rate and Momentum Adaptation, in *International Joint Conference on Neural Networks*, IEEE, pp 676-679, 1990.

4. Schiffmann, W., Joost, M., and Werner, R., "Comparison of Optimized Backprop Algorithms", Artificial Nerual Networks. European Symposium, D-Facto Publications, Brussels, Belgium, 1993.

5. Silva, Fernando M., & Almeida, Luis B.: "Speeding up Backpropagation", Advanced Neural Computers, Eckmiller R. (Editor), page 151-158, 1990.

6. Tollenaere, Tom, "SuperSAB: Fast Adaptive Backpropagation with Good Scaling Properties", Neural Networks, Vol. 3, pp 561-573, 1990.

7. Wilamowski, Bogdan W., Chen, Yixin, and Malinowski, Aleksander, "Efficient Algorithm for Training Neural Networks with one Hidden Layer", *Proceedings on the International Conference on Neural Networks*, San Diego, CA, 1997.

8. Ampazis, N., and Perantonis, S. J., "Levenberg-Marquardt Algorithm with Adaptive Momentum for the Effiecient Training of Feedforward Networks, in *International Joint Conference on Neural Networks*, IEEE, 2000.

9. Hagiwara, M., Theoretical Derivation of Momentum Term in Back-Propagation, in *International Joint Conference on Neural Networks*, IEEE, pp682-686, 1992.

10. Pearlmutter, Barak A., *Gradient descent: Second-order momentum and saturating error*. In *Advances in Neural Information Processing Systems 4*, pp 887-894. Morgan Kaufmann, 1992.

11. Rattray, M., and Saad, D., The Dynamics of Matrix Momentum, in *Proceedings of the Eighth International Conference on Neural Networks*, New York, 2 vols, pp 183-188, 1998.

12. Scarpetta, S., Rattray, M., and Saad, D., Natural Gradient Matrix Momentum, in *Proceedings of the Ninth International Conference on Neural Networks*, The Institution of Electrical Engineers, London, pp 43-48, 1999.

13. Swanston, D. J., Bishop, J. M., and Mitchell, R. J., "Simple Adaptive momentum: new algorithm for training multiplayer perceptrons", Electronics Letters, Vol. 30, pp 1498-1500, 1994.

14. Wiegerinck, W., Komoda, A., and Heskes., T., *Stochastic dynamics of learning with momentum in neural networks*. Journal of Physics A, 27:4425--4437, 1994.

15. Schraudolph, Nicol N., "Fast Second-Order Gradient Descent via $O(n)$ Curvature Matrix-Vector Products", Neural Computation 2000.

16. Qui, G., Varley, M. R., and Terrell, T. J., Accelerated Training of Backpropagation Networks by Using Adaptive Momentum Step, *IEE Electronics Letters*, Vol. 28, No. 4, pp 377-379, 1992.

17. Wilson, D. Randall, and Tony R. Martinez, The Inefficiency of Batch Training for Large Training Sets, In *Proceedings of the International Joint Conference on Neural Networks (IJCNN2000)*, Vol. II, pp. 113-117, July 2000.

18. Mitchell, Tom M., Machine Learning, McGraw-Hill, Boston, MA, 1997. p 119

19. Magoulas, G.D., Androulakis, G. S., and Vrahatis, M.N., Improving the Convergence of the Backpropagation Algorithm Using Learning Rate Adaptation Methods., in *Neural Computation,* Vol. 11, pp 1769-1796, 1999, MIT Press.

20. Cybenko, G., Approximation by Superpositions of a sigmoidal function, *Mathematical Control Signals Systems*, Vol. 2, pp 303-314, 1989.

21. Hornik, K., Stinchcomber, M., and White, H., Multilayer feedforward networks are universal approximators, *Neural Networks,* Vol. 2, pp 359 – 366, 1989.

22. Funahashi, K.-I., On the approximate realization of continuous mappings by neural networks, *Neural Networks,*Vol. 2, pp 183-192, 1989.

23. Hahn, Karl, "Frequency of Letters", English Letter Usage Statistics using as a sample, "A Tale of Two Cities" by Charles Dickens, Usenet Sci.Crypt, 4 Aug 1994. (statistics archived at http://www.arachnaut.org/archive/freq.html)

24. Nichols, Randall K., *Classical Cryptography Course*, Aegean Park Press Laeguna Hills, CA, 1996.