% LMS: A stochastic gradient algorithm inspired by neurobiology

Abhipray Sahoo
abhipray@stanford.edu

Jake Kaplan
jkapp@stanford.edu

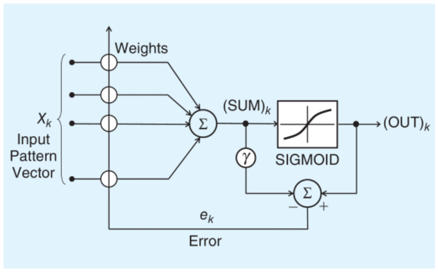
Stephane Remigereau
remigereau@stanford.edu

Abstract

A living neuron is connected to other neurons through small gaps called synapses. It adapts its connection strengths to pre-synaptic neurons in order to maintain a fixed average output firing rate. Neurobiology studies show that the mechanism of adaptation appears to scale the connection strengths multiplicatively. This inspired the % LMS algorithm which updates the strength of neural connections with a scaling factor to achieve homeostatic firing rate control. In this paper, we study its properties, performance and extensions through simulations and derivations.

1 Introduction & Motivation

Dr. Bernard Widrow recently proposed a linear model for a neuron, called Hebbian-LMS[1]. The model is pictured below:



The inputs are the firing rates of the pre-synaptic neurons. The output is the firing rate. Each weight represents the connection strength; it is physically the concentration of neuroreceptors at the synapse. The model is trained in an unsupervised manner to minimize the mean squared error between the linear combination of the inputs (SUM)$_k$ and its sigmoidal activation (OUT)$_k$.

The average error of the neuron output can be minimized by using stochastic gradient descent using the Widrow-Hoff LMS update rule:

$$w_{k+1} = w_k - \nabla_{w_k} J(w_k) \tag{1}$$
$$= w_k + \alpha \epsilon_k x_k \tag{2}$$
$$\epsilon_k = (y_k - w_k^T x_k) \tag{3}$$

w is the weight/parameter vector, α is the learning rate, y_k is the desired output, ϵ_k is the error at timestep k.

While working with Dr. Widrow on Hebbian learning rules, we discovered that the phenomenon of synaptic scaling and Hebbian learning[2] might imply a scaling of the weight vector. Particularly, a neuron only has access to the input x_j via the synapse and its associated weight w_j and hence only ever detects $w_j x_j$ for a synapse j. This led Dr. Widrow to propose the % LMS algorithm:

$$w_{k+1} = w_k + \alpha \epsilon_k x_k \circ w_k \tag{4}$$

$\circ$ is the Hadamard product or element-wise product.

It is called %-LMS because the weights are updated as a percentage of the previous weight values. Big weights adapt faster.

$$w_{k+1} = w_k + \alpha \epsilon_k x_k \circ w_k \tag{5}$$
$$= (1 + \alpha \epsilon_k x_k) \circ w_k \tag{6}$$

2 Convergence

2.1 Trajectory

We ran simulations of LMS and %LMS in a two dimensional least squares problem. We generated data using known weights and plotted the contours of the quadratic error bowl. The following figure demonstrates that the % LMS converges to the optimal solution like the LMS algorithm but follows a different trajectory.

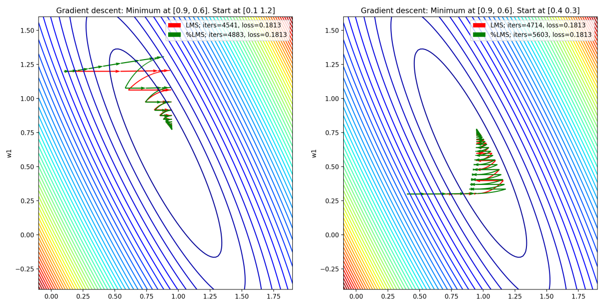


Figure 1: Trajectory taken by LMS vs %LMS from two different initial weight vectors.

2.2 Eigenvalue spread of quadratic cost function

The convergence speed of stochastic gradient descent depends on the eigenvalue spread of the autocorrelation matrix R:

$$\chi(R) = \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}} \tag{7}$$

When the eigenvalues are unequal, the contour plot of the quadratic surface of the cost function is elliptical. Convergence is fastest along the short axis of the ellipse and slowest along the long axis of the ellipse. When there is an eigenvalue spread, the initial weight vector has an effect on convergence speed. The following table shows how % LMS performs comparatively to LMS algorithm for different eigenvalue spreads but same initial weight vector. It was generated by constructing an input design matrix with autocorrelation matrix with a range of different eigenvalue spreads and recording number of iterations until convergence.

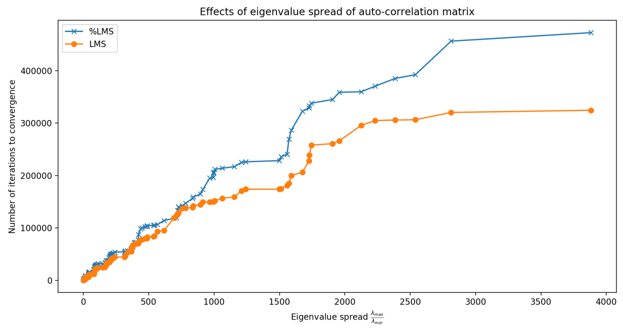


Figure 2: Number of iterations to convergence vs how non-uniform the quadratic bowl is.

We observed that for both the last two simulations, we defined convergence to mean the iteration number when the L2 norm of the difference in weight vector reduces to a value below a pre-defined ϵ: $|\theta_{k+1} - \theta_k|_2 < \epsilon$. This convergence criteria gives the LMS an unfair advantage as we will see in the next section.

2.3 Cost function

In this section, we derive the cost function that leads to the % LMS update rule. The general optimization problem for learning can be decomposed into two parts:

$$\min L = \min \left[d(w_{k+1}, w_k) + \alpha J(w_k, x_k, y_k)\right]$$

$d(w_{k+1}, w_k)$ is the distance between the updated weights and the current weights; it represents memory or inertia i.e how big can an update to our current vector be. $J(w_k, x_k, y_k)$ is the loss function measuring how far our current prediction of the linear model $w_k^T x$ is from the desired output y_k. α controls the degree to which we want to focus on memory, i.e small updates vs minimizing loss at the current iteration.

We can solve for w_{k+1} by setting the gradient of L to zero.

$$\nabla_{w_{k+1}} L = \nabla_{w_{k+1}} d(w_{k+1}, w_k) + \alpha \nabla_{w_{k+1}} J(w_{k+1}, x_k, y_k) = 0 \tag{8}$$

Assuming each iteration makes small changes to w, we can approximate

$$\nabla_{w_{k+1}} J(w_{k+1}, x_k, y_k) \sim \nabla_{w_k} J(w_k, x_k, y_k)$$

In this problem, we focus on the mean squared error loss

$$J(w_k, x_k, y_k) = \text{MSE} = ||y_k - w_k^T x||_2^2 \tag{9}$$

$$\nabla_{w_k} J(w_k, x_k, y_k) = -2(y_k - w_k^T x)x_k \tag{10}$$

For LMS, we observe that $d(w_{k+1}, w_k) = ||w_{k+1} - w_k||_2^2$. To see this, we take its gradient and plug it into equation 8,

$$2(w_{k+1} - w_k) - \alpha 2(y_k - w_k^T x)x_k = 0 \tag{11}$$

$$w_{k+1} - w_k = \alpha(y_k - w_k^T x)x_k \tag{12}$$

$$w_{k+1} = w_k + \alpha(y_k - w_k^T x)x_k \tag{13}$$

This is the familiar LMS equation.

For % LMS, we observe that $d(w_{k+1}, w_k) = \sum_{j=1}^{d} \frac{(w_{k+1,j} - w_{k,j})^2}{w_{k,j}}$ where j indexes into the individual vector elements. To see this, we can take the gradient of an individual vector element and plug it into equation 8.

$$\nabla_{w_{k+1}} d(w_{k+1}, w_k) = \nabla_{w_{k+1}} \sum_{j=1}^{d} \frac{(w_{k+1,j} - w_{k,j})^2}{w_{k,j}} = \left(\frac{2(w_{k+1,1} - w_{k,1})}{w_{k,1}}, \cdots, \frac{2(w_{k+1,n} - w_{k,n})}{w_{k,n}} \right) \tag{14}$$

$$\frac{2(w_{k+1,j} - w_{k,j})}{w_{k,j}} - \alpha 2(y_k - w_k^T x)x_k = 0 \tag{15}$$

$$\frac{w_{k+1,j}}{w_{k,j}} - 1 = \alpha(y_k - w_k^T x)x_k \tag{16}$$

$$w_{k+1,j} = (1 + \alpha(y_k - w_k^T x)x_k) * w_{k,j} \tag{17}$$

Thus to summarize this section, LMS minimizes the Euclidean distance whereas % LMS minimizes the relative change in weights.

3 Generalized % LMS

3.1 Extended Algorithm

We observe that the original % LMS algorithm works only with positive weight values. When weights turn negative, they tend to continue moving in the negative direction and when a weight turns zero, it stops changing. This works fine in nature, since neurons operate with non-negative quantities. However, we are interested in applying this algorithm to models with negative weights. To extend this algorithm, we can make the following modification:

$$w_{k+1} = w_k + \alpha \epsilon_k x_k \text{sign}(w_k) \circ w_k \tag{18}$$

$$= (1 + \alpha \epsilon_k x_k \text{sign}(w_k)) \circ w_k \tag{19}$$

$$\text{sign}(w_k) = \begin{cases} 1 & w_k \geq 0 \\ -1 & w_k < 0 \end{cases} \tag{20}$$

When a weight tries to move from negative to positive value or vice-versa, it will stop at 0 before being able to move past. In order to fix this, we can add random noise when the weight value is in between some neighborhood ε of 0.

$$w_{k+1} = w_k + \alpha \epsilon_k x_k \text{sign} \circ w_k + g(x) \tag{21}$$

$$= (1 + \alpha \epsilon_k x_k \text{sign}(w_k)) \circ (w_k) + g(x) \tag{22}$$

$$g(x) = \begin{cases} z \sim \mathcal{N}(0, \varepsilon^2) & -\varepsilon^2 < x < \varepsilon^2 \\ 0 & \text{else} \end{cases} \tag{23}$$

After training, we can collapse any weights within ε of 0 to 0.

3.2 Random Noise on the Convergence of % LMS

Note that the random noise in general % LMS can cause problems when any component of the optimal weight is within ε of 0. Indeed, in this case, the noise would work to push the weight past the optimal value in an effort to allow it to cross 0.

Setting ε too small, in an attempt to circumvent this issue, has its own problems. Since the difference in weights between iterations in % LMS is scaled by the weight itself, allowing any component of the weight vector to get too close to 0 could result in that component's change being small enough to pass for convergence.

Hence, it would seem unfavorable to set ε either much larger or much smaller than the convergence criterion. However, recalling that the gradient is small when the weight vector is near the optimal weights, and large otherwise, it would appear that the gradient possesses the exact properties we would like from ε. Specifically, we would like ε to only be large if the optimal weight in the component near 0 is not itself near 0. Thus, the noise might perturb the weight past 0 to bring it closer to the optimal weight when this is a large value, while it would not be large enough to have this effect if the optimal weight is also near 0.

This motivates the use of the gradient to set ε, the variance of the random noise, in each iteration. The below graphs depict the results of 2 sets of trials comparing LMS to % LMS with constant and gradient noise.

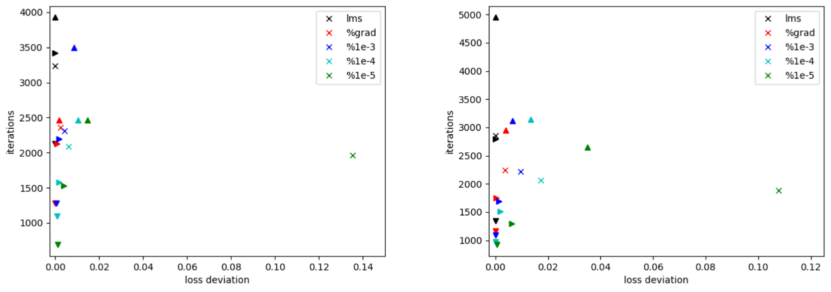


Figure 3: 15 trial pilot study (left) and 100 trial experiment (right) comparing deviation in loss (from LMS benchmark) against convergence rate of % LMS with values of $\varepsilon^2 \in \{\nabla J, 10^{-3}, 10^{-4}, 10^{-5}\}$ to standard LMS.

The two experiments depicted in Figure 3 above consisted of 15 and 100 respective trials of Poisson regression on distinct Gaussian randomized data sets fit to a Poisson distribution with Gaussian randomized optimal weight vector. Each trial involved fitting weight vectors using 5 different LMS models with full batch gradient descent. This consisted of a standard LMS model and 4 % LMS models, one using the gradient as the value of ε^2 and the other three using $1e^{-3}$, $1e^{-4}$, and $1e^{-5}$ respectively. The criterion for convergence was $1e^{-4}$ and learning rate was 0.25.

In Figure 3, the x axis depicts the deviation from the loss of the optimal solution in the standard LMS algorithm (hence for standard LMS, this value is always 0). The y axis depicts the number of iterations to convergence. The $\times$ symbols mark the mean of each algorithm over all trials, while the $\triangleright$, $\triangle$, and $\triangledown$ symbols mark the median and upper/lower quartiles respectively. The results in the pilot study indicate that % LMS actually outperforms standard LMS in rate of convergence, though at the cost of increased loss. This showed potential that the % LMS algorithm could indeed be worth further study, hence the more complete follow-up experiment.

The results of the full study show that gradient variance % LMS gives the best ratio of improvement in convergence rate to increase in loss of the 4 % LMS models compared. In contrast, the 3 constant variance % LMS models provided marginal gains in convergence rate with respect to the gradient variance model, while taking on much more significant increases in deviation from the loss of the standard LMS model. In particular, the fact that the average loss deviation is larger than that of the median indicates a right skewed distribution in which the constant variance models show improvement in more than half of the trials, but pose higher probabilities of outliers with much greater deviations in loss.

4 Generalized % LMS in other models and problems

To understand the behavior of % LMS we apply this rule to classification tasks with different models and training process. We analyze results in terms of number of iterations needed to convergence, the loss for the weights obtained, etc.

4.1 Logistic Regression

We apply the %LMS update rule to a logistic regression classification problem. We used a dataset of two dimensional points as in figure 4. A logistic regression model is trained to maximize the log-likelihood of the data. In this setup, we modify the %LMS update rule to perform gradient ascent to maximize the log-likelihood.

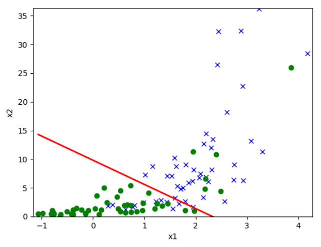


Figure 4: Dataset used for logistic regression classification and optimal line maximizing log-likelihood of data

Note that in this case, the generalized % rule necessary because the weights can be negative. When we use the % rule without taking the sign into account, we cannot update the initial sign of each weight component and thus obtain a sub-optimal solution (with around 69% accuracy, compared to 0.83% accuracy with the optimal solution.

By running the experiment multiple times, we see that the randomness associated with the noise added when the weight is close to 0 can have a big impact on the solution. Indeed, if the variance is too small, it can take a long time to find a solution. On the other side, if the variance is too large, we might find some solutions that are sub-optimal because the delta between the new and the old weight might be less than the defined epsilon. Thus, in practice, we use the formula seen before where the variance depends on the gradient of the loss function directly, so that we can find the optimal solution in all cases.

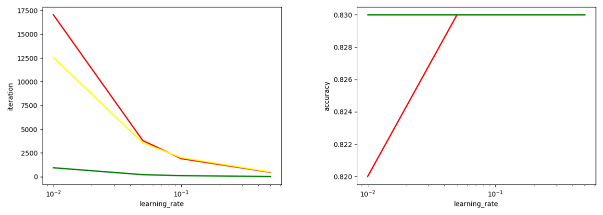
Detailed results obtained for a learning rate of 0.01:

	Gradient Descent	% Gradient Descent	Newton
Number of Iterations	13442	17513	941
Accuracy	0.83	0.82	0.83

Table 1: Algorithm comparison

For this learning rate, we find almost the optimal solution with %LMS as with Newton and %LMS. Following these observations, we wanted the change the learning rate to see how both the number of iterations and accuracy would evolve to all three of these algorithms. From the figure below, we can see that with higher learning rates, %LMS performs better and has the optimal accuracy of the Newton and LMS algorithms. The number of iterations drops significantly, which makes %LMS a comparably useful algorithm to use. More specifically, with learning rates higher than 0.1, the algorithm converged with the same number of iterations and same accuracy as LMS.

Figure 5: Normal Classification vs % Classification on Dataset 1



4.2 Neural Network

Here we are interested in the comparative behavior of %LMS in a non-convex optimization problem. We modified the backpropagation algorithm for training neural networks to update each weight as a scaling factor like the generalized %LMS. We then applied this to a classification task on the MNIST handwritten digit dataset[3]. The network has a single hidden layer with 150 logistic units and softmax output layer. It is trained for 20 epochs with batch size 20, learning rate 0.1, with cross-entropy loss. Training is done on 50K samples and testing on 10K samples. The regular backprop algorithm (LMS) and the %LMS variation are compared in the following table:

	LMS	%LMS
Training loss	0.10528	0.14121
Test loss	0.3178	0.3279
Accuracy	92.21%	91.02%

Keeping the same conditions as before but varying the learning rates, we can see in figure 6 that the learning rate at which performance drops is different for the two algorithms. This indicates that for comparable performance, %LMS requires a higher learning rate than LMS.

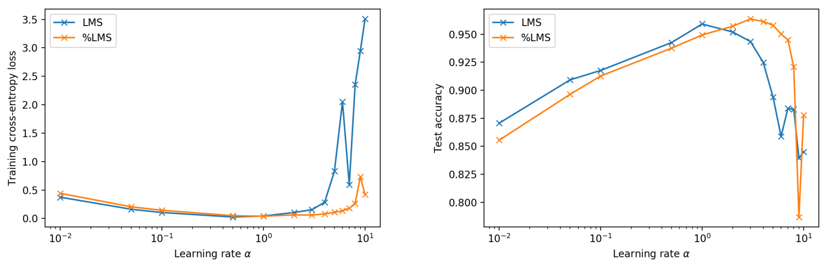


Figure 6: Experiment comparing the training loss and test accuracy of LMS vs %LMS for different values of learning rate.

5 Conclusion

We demonstrated through simulations that %LMS converges to the optimal solution to a convex optimization problem. We derived the cost function that % LMS is minimizing; it is minimizing the relative change in weights as part of the cost function. We then generalized the algorithm to work with negative weights and applied it to two classification problems with logistic regression and neural networks. Its performance appears to be worse than regular LMS/backprop in terms of convergence speed and learning curve for the same learning rate. %LMS can achieve better performance To understand this better, next steps are to do analytic derivations of the learning curve and factors that influence it. We will also evaluate the %LMS algorithm as part of the Hebbian-LMS neuron model. Other extensions of this research could include an analysis of how the performance of the %LMS algorithm compares in classifying data from other distributions than the Gaussian and Poisson distributions examined in this paper.

Contributions

Abhipray Sahoo: sections 1, 2, 3.1, 4.2, 5 | Jake Kaplan: sections 1, 3.2, 4, 5 | Stephane Remigereau: sections 1, 4.1

Code

All code used for this project can be found here: https://github.com/Abhipray/percent-lms

Acknowledgments

We would like to thank Dr. Bernard Widrow for sharing his ideas and his guidance.

References

[1] Bernard Widrow, Youngsik Kim, and Dookun Park. The hebbian-lms learning algorithm. *ieee ComputatioNal iNtelligeNCe magaziNe*, 10(4):37–53, 2015.

[2] Gina G Turrigiano and Sacha B Nelson. Hebb and homeostasis in neuronal plasticity. *Current opinion in neurobiology*, 10(3):358–364, 2000.

[3] Yann LeCun. The mnist database of handwritten digits, 1998.