Journal of Machine Learning Research 3 (2002) 16-30 Submitted 5/02; Published 12/02

Ideal Difference Based Back-propagation

Rick Ferreira PFERREI@G.CLEMSON.EDU
B19 Riggs Hall
433 Calhoun Dr.
Clemson, SC 29634, USA

Melissa Smith SMITHMC@CLEMSON.EDU
110 Riggs Hall
433 Calhoun Dr.
Clemson, SC 29634, USA

Abstract

There are two common problems when designing and using artificial neural networks. The first is the need for better performance. The second is the need to combat the increasing complexity with enhancements. In this paper we design a way to do both.

This is done in each iteration by calculating what weights would give the optimal answer for each input and output pair. The weights are then updated by the difference between the ideal weight and the current weights all of it times the learning rate.

We find that this method not only converges much faster for an image classification problem but it also is much simpler to understand and does not rely on using calculus or derivatives. However the method only works for a shallow or single layer neural network.

By using simple arithmetic, neural networks can be updated in a way that is both simpler and more efficient than back-propagation.

Keywords: Back-propagation, derivative-less, weight updates, optimization

1 Introduction

Artificial neural networks (ANNs) have been used extensively in many areas of research. ANNs are used in several applications such as medical imaging where they are used for detecting tumors and other diseases. They have also been used for other vision applications such as aiding self driving cars where object detection algorithms segment images into several patches that are then analyzed by ANNs for objects [1 YOLO]. Other areas also include speech recognition, machine translation, and decision making (such as reinforcement learning). Therefore enhancing the performance of ANNs is key to enabling the improvement of several different algorithms and real world systems.

There is always a demand for better performing algorithms. Especially when it comes to one as broadly used as artificial neural networks. There are overall improvements that can be applied to ANNs across broad ranges of applications. Some of these improvements are techniques such as optimizers and concepts such as momentum [2].

These techniques such as AdaGrad [3] and . Have shown to improve performance of ANNs. The good properties of these research methods is that they can be applied to several different type of ANN architectures. This is often done by only modifying a few simple options and the whole network is enhanced.

With all of these applications there is the desire to reduce the amount of iterations needed to achieve an acceptable level of accuracy. Reducing the number of iteration can potentially reduce the amount of time needed to train networks.

There is also the need to improve the total amount of accuracy possible. Sometimes this is worth while even if the cost to compute each individual iteration is more costly (whether that is time wise or compute wise). Such as when you can replicate the costly produced system cheaply across several devices once trained.

However with these techniques and desires, the common denominator is that they each often add layers of complexity to the ANN algorithms. This creates a greater amount of technical debt that new comers must overcome to be able to understand the concepts enough to apply the enhancements.

The question remains if techniques to modify ANNs generally can be developed that both improve performance and maintains or reduces the level of complexity.

With this question in mind, we seek a desirable algorithm that can pave the way to for simpler and more efficient ANN enhancements.

This paper presents a new weight optimization technique that can be applied to ANNs. It involves calculating what weights will transform the current input into the desired outputs using simple arithmetic for each input and output pair. We test this technique on one and multilayer ANNs.

This paper hopes to show that this method not only outperforms standard back-propagation for one layer ANNs but is also easier to understand, modify, and use than back-propagation.

This paper will be broken down into the following sections. In the Literature Review section will cover how forward and back propagation works. Paying specific attention to how the weights are updated along with how input gradients are back propagated. In the methodology section the paper will cover how weight are updated and input gradients are back propagated for the ideal difference based back-propagation. Although we will see that the input gradients did not work as intended they are still mentioned for completeness at an attempt to make this method work for multilayer neurons. There will then be a discussion section which will review the results and the what the consequences and trade offs are for this method. Finally there is a conclusion where closing remarks will be made.

Principle findings are that the proposed algorithm not only converges to lower loss values with less iterations but that it also approaches a lower asymptotic end loss than standard back-propagation. However this came at a little less than half reduction in speed.

2 Literature Review

The key component of our advancement involves updating an ANN that uses forward-propagation which we will briefly describe here and then go on to describing the background back-propagation algorithm.

2.1 Forward-propagation

A typical one layer neural network operates by having having each output be the summation of the products of each input with an individual weights for each input. This is shown in Fig. 1, Eq. 1, and Eq. 2 below.

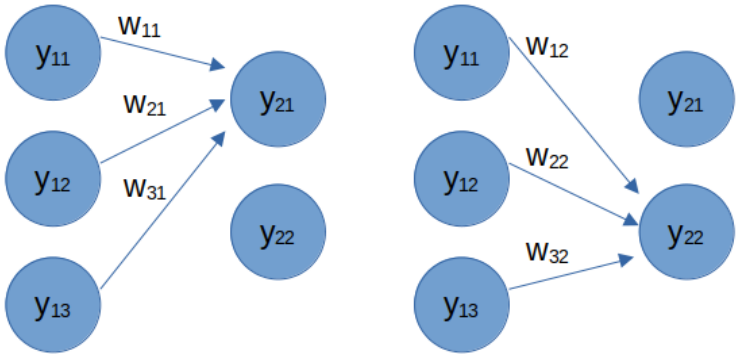

Fig. 1 ANN inputs , weights and outputs

$$y_{21} = w_{11} y_{11} + w_{21} y_{12} + w_{31} y_{13} + b_1 \quad \text{Eq. 1}$$

$$y_{22} = w_{12} y_{11} + w_{22} y_{12} + w_{32} y_{13} + b_2 \quad \text{Eq. 2}$$

This process is repeated until you have a multi layer ANN as shown in Fig. 2, where y_1 is the initial inputs and y_2, y_3 are hidden layers with y_4 being the outputs.

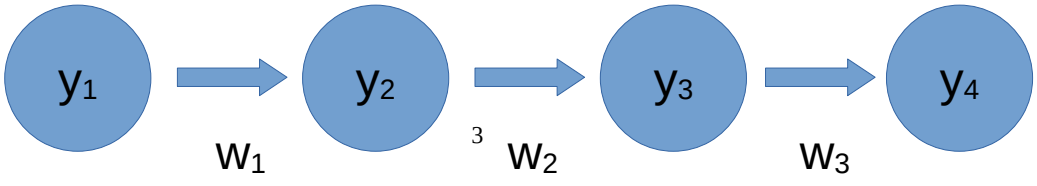

Fig. 2 Multilayer ANN

2.2 Back-propagation

The mechanism behind regular back-propagation will now be presented here.

2.2.1 Weight updates

Standard back-propagation works by updating each weight in proportion to how much it effects the loss. This is done by modifying the weight by the derivative of the loss with respect to the weight being updated. The loss is the difference between the correct output values and the output of the neural network. Fig. 3 shows how the loss is propagated backwards to update each weight, where this time y_c is the correct output.

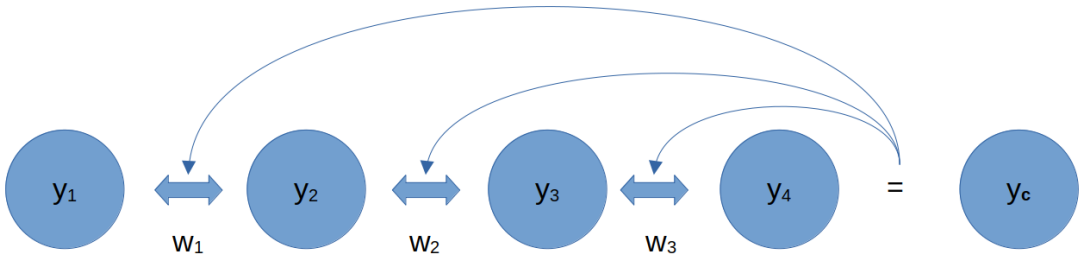

Fig. 3 Loss being back-propagated through ANN

The details of standard back-propagation will now be shown. For simplicity we will represent each layer of weights as a single variable w. This w can be interpreted as each layer having only one weight w or the w can represent a matrix of the weight connections. So for the ANN shown in Fig. 3 we have the following three equations shown in Eq. 3, Eq. 4, and Eq. 5.

$$y_4 = w_3 y_3 \quad \text{Eq. 3}$$

$$y_3 = w_2 y_2 \quad \text{Eq. 4}$$

$$y_2 = w_1 y_1 \quad \text{Eq. 5}$$

The key is to back-propagate the end loss as shown in Eq. 6. The difference between y_c and y_4 is squared and multiplied by a half so that when later the derivative is taken with respect to y_4 the equation is simplified to Eq. 7.

$$L=\frac{1}{2}(y_c-y_4)^2$$ Eq. 6

$$\frac{dL}{dy_4}=y_c-y_4$$ Eq. 7

To find how much the weight in the last layer (w_3) should change by we find the derivative of the loss with respect to the weight change. This should tell us how much the weight effects the end loss so that we can decide how much to change the weight by. This value is derived in Eq. 8.

$$\Delta w_3=\frac{dL}{dw_3}=\frac{dL}{dy_4}\cdot\frac{dy_4}{dw_3}=(y_c-y_4)\,y_3$$ Eq. 8

The same process is repeated for the previous layers as shown in Eq. 9 and Eq. 10.

$$\Delta w_2=\frac{dL}{dw_2}=\frac{dL}{dy_4}\cdot\frac{dy_4}{dy_3}\cdot\frac{dy_3}{dw_2}=(y_c-y_4)\,w_3\,y_2$$ Eq. 9

$$\Delta w_1=\frac{dL}{dw_3}=\frac{dL}{dy_4}\cdot\frac{dy_4}{dy_3}\cdot\frac{dy_3}{dy_2}\cdot\frac{dy_2}{dw_1}=(y_c-y_4)\,w_3\,w_2\,y_1$$ Eq. 10

2.2.2 Matrix equations

The equations for the matrix version of back propagation is shown in Eq. 11, Eq. 12, and Eq. 13. These can be plugged into Eq. 8 and Eq. 9. The matrix equations for Eq. 10 is left out for brevity but it follows the same chain of reasoning.

$$\begin{bmatrix} y_{41} \\ y_{42} \end{bmatrix}=\begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix}\begin{bmatrix} y_{31} \\ y_{32} \\ y_{33} \end{bmatrix}+\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$ Eq. 11

$$\frac{dy_4}{dw_3}=\begin{bmatrix} y_{31} & y_{31} \\ y_{32} & y_{32} \\ y_{33} & y_{33} \end{bmatrix} \qquad \frac{dy_3}{dw_2}=\begin{bmatrix} y_{21} & y_{21} & y_{21} \\ y_{22} & y_{22} & y_{22} \\ y_{23} & y_{23} & y_{23} \\ y_{24} & y_{24} & y_{24} \end{bmatrix}$$ Eq. 12

$$\frac{dy_4}{dy_3}=\begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{22} \end{bmatrix}^T \qquad \frac{dL}{dy_4}=\begin{bmatrix} y_{c1}-y_{41} \\ y_{c2}-y_{42} \end{bmatrix} \qquad \text{Eq. 13}$$

3 Methodology

This paper focuses on the method of weight updates by using simple arithmetic to solve for the weights given the inputs and the expected outputs.

Traditionally the method of updating weights depends on back-propagation, which involves the use of derivatives from calculus. Instead of trying to change each individual weight in relationship to how it effects the loss or error amount this paper tries to calculate what weights would directly produce the output given the input pair by solving what the weights should be based on the output and inputs.

In this paper we test out this new method to see if it is both easier to implement, easier to understand, works, and if it can even output form derivative based back-propagation.

3.1.1 Weight updates

The method presented here is called the ideal difference based back-propagation or ideal difference weight updating.

The details of ideal difference based back-propagation will now be shown. For simplicity we will represent each layer of weights as a single variable w again. This w can be interpreted as each layer having only one weight w or the w can represent a matrix of the weight connections. So for the ANN shown in Fig. 4 we have the following three equations shown in Eq. 3, Eq. 4, and Eq. 5.

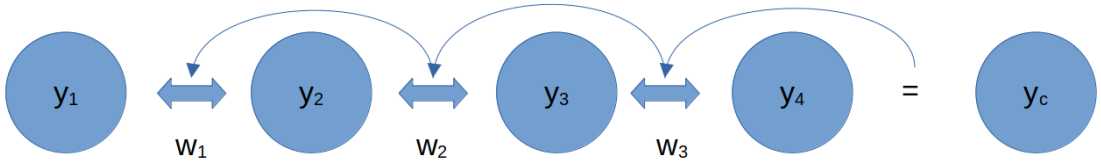

Fig. 4 Loss being back-propagated through ANN

Eq. 14, Eq. 15, and Eq. 16 below shows equations where for an input y_1 there is a correct weight w_{1c} that produces the correct output y_{2c}. This process is repeated for the next layer where the input for the next layer is the correct output of the previous layer.

$$y_c=w_{3c}y_{3c} \qquad \text{Eq. 14}$$

$$y_{3c} = w_{2c} y_{2c}$$ Eq. 15

$$y_{2c} = w_{1c} y_1$$ Eq. 16

Eq. 17, Eq. 18, and Eq. 19 each solve for the change needed to update weights w to be equivalent to the correct weight w_c.

$$\Delta w_3 = w_{3c} - w_3 = \frac{y_c}{y_3} - \frac{y_4}{y_3} = \frac{y_c - y_4}{y_3}$$ Eq. 17

$$\Delta w_2 = w_{2c} - w_2 = \frac{y_{3c}}{y_2} - \frac{y_3}{y_2} = \frac{\frac{y_c}{w_{3c}} - y_2}{y_2}$$ Eq. 18

$$\Delta w_1 = w_{1c} - w_1 = \frac{y_{2c}}{y_1} - \frac{y_2}{y_1} = \frac{\frac{y_c}{w_{3c} w_{2c}} - y_1}{y_1}$$ Eq. 19

3.1.2 Matrix equations

The equations for the matrix version of out new back propagation method is shown in Eq. 11. Eq. 11 shows the matrix version of the weight matrix. Eq. 20 shows the equations for the correct outputs $y_{c\_}$, correct outputs of the 2nd layer y_{3_c}, along with the regular weights $w_{\_}$.

$$\begin{bmatrix} y_{c1} \\ y_{c2} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix} \begin{bmatrix} y_{31c} \\ y_{32c} \\ y_{33c} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$ Eq. 20

Eq. 21 is the matrix version of Eq. 17 which is the new method for updating the weights. Note that the weights shown here are the ones for w_3. An important note is that in the experiment the element wise inverse of the $y_{3\_}$ matrix is used and then any gradient explosion values are set to zero.

$$\begin{bmatrix} \Delta w_{11} & \Delta w_{21} & \Delta w_{31} \\ \Delta w_{12} & \Delta w_{22} & \Delta w_{32} \end{bmatrix} = \left(\begin{bmatrix} y_{c1} \\ y_{c2} \end{bmatrix} - \begin{bmatrix} y_{41} \\ y_{42} \end{bmatrix} \right) \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \div \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} y_{31} & y_{32} & y_{33} \end{bmatrix}$$ Eq. 21

The process is then repeated for previous layers by calculating the output of the previous layer by solving for y_{3_c}. There are two ways of calculating these values. The first method shown in Eq. 22 is to use the newly updated weights. The problem with this is that the weights need to be updated during each layer. The second method shown in Eq. 23 is to simply use the current weights. In the experiment the Pseudo inverse is used instead of the inverse since the matrix isn't a square matrix. For our experiments we used the current weights since it is easier to implement.

Eq. 22 and Eq. 23 both solve for y_{3_c} in Eq. 20.

$$\begin{bmatrix} y_{31c} \\ y_{32c} \\ y_{33c} \end{bmatrix} = \begin{bmatrix} w_{11c} & w_{21c} & w_{31c} \\ w_{12c} & w_{22c} & w_{22c} \end{bmatrix}^{-1} \begin{bmatrix} y_{c1} \\ y_{c2} \end{bmatrix} - \begin{bmatrix} w_{11c} & w_{21c} & w_{31c} \\ w_{12c} & w_{22c} & w_{32c} \end{bmatrix}^{-1} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad \text{Eq. 22}$$

$$\begin{bmatrix} y_{31c} \\ y_{32c} \\ y_{33c} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{22} \end{bmatrix}^{-1} \begin{bmatrix} y_{c1} \\ y_{c2} \end{bmatrix} - \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix}^{-1} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad \text{Eq. 23}$$

The same method is then applied to the layer before it in Eq. 24. Note that the weights represented here are the w_2 weights.

$$\begin{bmatrix} \Delta w_{11} & \Delta w_{21} & \Delta w_{31} & \Delta w_{41} \\ \Delta w_{12} & \Delta w_{22} & \Delta w_{32} & \Delta w_{42} \\ \Delta w_{13} & \Delta w_{23} & \Delta w_{33} & \Delta w_{43} \end{bmatrix} = \left(\begin{bmatrix} y_{31c} \\ y_{32c} \\ y_{33c} \end{bmatrix} - \begin{bmatrix} y_{31} \\ y_{32} \\ y_{33} \end{bmatrix} \right) \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \div \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} y_{21} & y_{22} & y_{23} & y_{24} \end{bmatrix}$$

Eq. 24

Sometimes the bias also needs to be back-propagated. In Eq. 24 this method tries to devise a way of calculating this.

$$\Delta b_3 = b_{3c} - b_3 = y_c - w_c y_3 - (y_4 - w_c y_3) = y_c - y_4 \quad \text{Eq. 24}$$

Again the inputs may also need to be back-propagated. Eq. 25 shows the condensed equation where as Eq. 26 shows the matrix equivalent.

$$\Delta y_3 = y_{3c} - y_3 = \frac{y_c}{w_3} - \frac{y_4}{w_3} = \frac{y_c - y_4}{w_3} \quad \text{Eq. 25}$$

$$\Delta y_3 = \begin{matrix} y_{31c} - y_{31} \\ y_{32c} - y_{32} \\ y_{33c} - y_{33} \end{matrix} = \begin{matrix} 1/2\left[\frac{y_{c1} - y_{41}}{w_{11}} + \frac{y_{c2} - y_{42}}{w_{12}}\right] \\ 1/2\left[\frac{y_{c1} - y_{41}}{w_{21}} + \frac{y_{c2} - y_{42}}{w_{22}}\right] \\ 1/2\left[\frac{y_{c1} - y_{41}}{w_{31}} + \frac{y_{c2} - y_{42}}{w_{32}}\right] \end{matrix} = 1/2\left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \div \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{22} \end{bmatrix}^T\right)\begin{bmatrix} y_{c1} - y_{41} \\ y_{c2} - y_{42} \end{bmatrix}$$

Eq. 26

3.1.3 Experimental setup

The experiment was run on a Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz with GeForce GTX 1650 Mobile / Max-Q GPU.

This experiment used PyTorch since it is a popular python ANN framework and has some easily modifiable modules for running a regular and modified ANNs.

Popular and easily documented framework. Makes reusability, also common and standardize test database

The ANNs where tested on the MNIST dataset.

This dataset consists of gray scale images of numbers from 0 to 9 where each image has a number label corresponding to the images numerical value. Example of the data set is shown in Fig. 5.

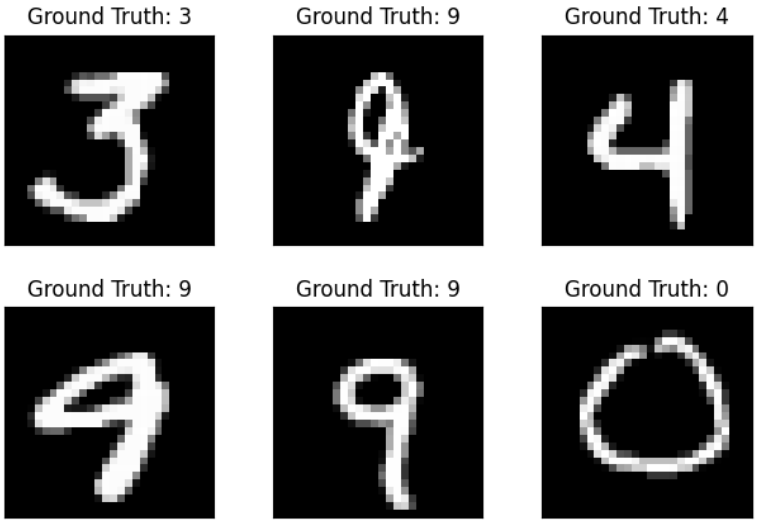

Fig. 5 MNIST data set example

Training vs Testing loss, time to execute total algorithm [TIMER]

The training dataset was composed of batches of 64 images which was repeated for a total of 3 epochs. The ANNs are tested at the beginning and end of each epoch on a batch of 1000 images that where not included in the in the training dataset. The ANNs are also timed for how long it takes to train and execute the tests (excluding the first test since no training was done before the first test).

Regular neural network, One layer neural network, two layer neural network.

The paper presents three ANNs that where trained and tested on. The first one is the control ANN which is a regular one layer 784x10 ANN. The second ANN uses a custom linear layer which implements the new method shown in this paper. The third ANN uses two custom linear layers which use the new method shown in this paper. The independent variable is the number of samples seen while the dependent variable is the ANN and the loss or the time to execute. Separation of training and test data for the sake of measuring over-fitting and generalization. MNIST was used since it is very popular and easy to implement and replicate.

For the experiment we graph training and testing loss over batch iterations where the loss is the same one shown in Eq. 6. The training loss is the loss after each training iteration while the testing loss is only tested at the beginning and after each epochs for 3 epochs.

The data was analyzed for lower losses for both training and testing data. Specifically for faster convergence to the asymptotic loss value. This and speed of convergence measured by both number of iterations and time run.

These metrics where sought since they are what was expected that there would be an improvement in less iterations. Another reason for these measurements is to also see if the processing time would be extra.

4 Results

time blue
31.730149030685425
time red
50.588770389556885
time green
295.94398760795593

This paper will now present th our methods with a the control ANN so that we can see what the trade-offs are.

This paper will now present the the results. They are shown for three of the ANNs tested. Although more ANNs where tested the results shown will later be explained to be enough to get the necessary points across about the new method.

Fig. 6 shows the training loss for the three different types of ANNs. Fig. 7 shows the testing loss for the three different types of ANNs. Table 1 shows the time it takes to train and test the ANNs.

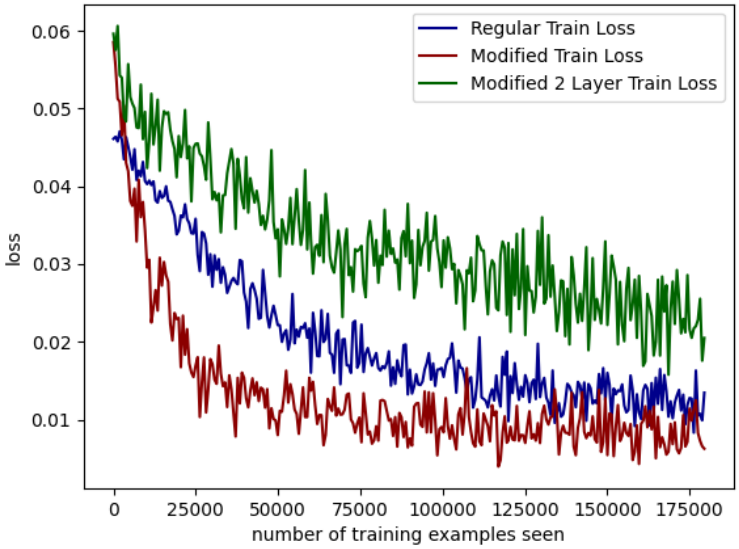

Fig. 6 Training Loss

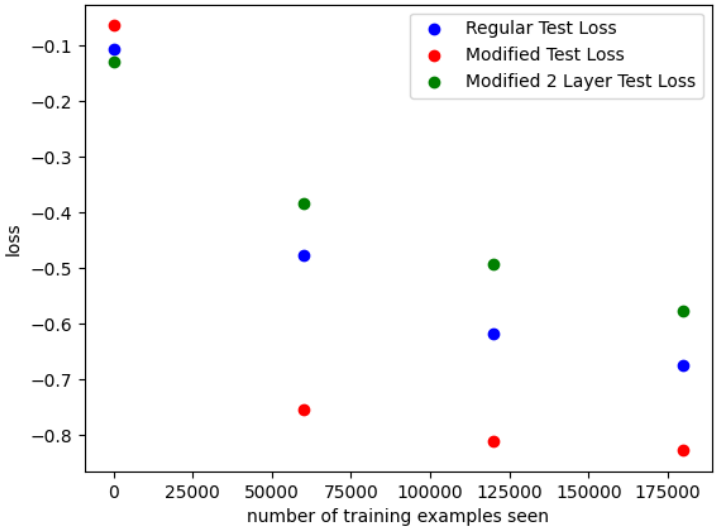

Fig. 7 Test Loss

Table 1 Time to train and test the ANNs

ANN	Regular ANN	New Method 1 Layer	New Method 2 Layer
Time to train and test for 187,500 iterations	31.73 s	50.59 s	295.94 s
Loss reduced per sec for 62,500 iterations	0.5/(30/3) = 0.05	0.75/(50/3) = 0.045	0.4/(300/3) = 0.004
Time to train and test for 562,500 iterations	424.73 s	758.08 s	2,277.53 s
Loss reduced per sec until -0.7 loss	0.7/425 = 0.0016	0.7/(50/3) = 0.042	About same as Regular ANN

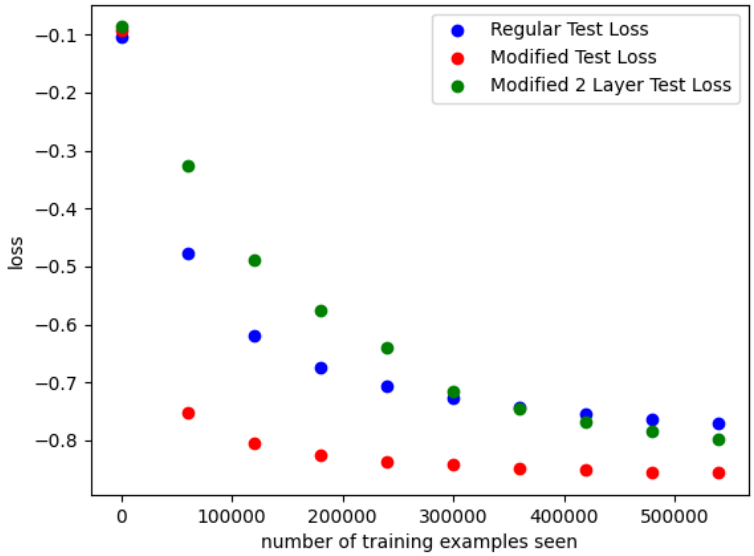

Fig. 8

The trends for each ANN is the same for Fig. 6 and Fig. 7. The one layer of the new or modified method has a significantly lower loss to the second lost loss. The regular method performs in the middle for the middle amount of iterations. The two layer new method performs the worst for the middle amount of iterations. Fig. 8 shows the test loss for more iterations. In this instance the modified 1 layer reaches close to it's limit quickly and the fastest, while the regular one layer performers worse than 1 layer modified but better than the 2 layer modified in the middle until eventually being out done by the 2 layer modified after several iterations.

In Table 1 the regular method has the best time, meanwhile the new method with 1 layer performs a little less than twice as long and the new method with 2 layers performs almost ten times as long. Also in Table 1 is shown the loss per second for 62,500 iterations. The new method with 1 layer performs about slightly worse than the regular ANN while the new method with 2

layers performs 10 times worse. The 1 layer modified algorithm reached a 0.7 loss 25.5 times faster than regular ANN algorithm and the 2 layer modified algorithm.

Notice that not only does the new one layer method converge on less iterations, but it's asymptotic value is also significantly lower.

A positive note is that the algorithm not only works, but has better performance on amount learned per iteration.

The negative results are that the new method takes almost twice as long with one layer and does not scale well with more layers. Not only does the performance get worse than the regular method but it is ten times slower.

Method for adding non linearity between layers not working at all.

An interesting observation is that the new method works for any multilayer neural network without any non-linearity between layers due to it working for one layer. This is true due to several linear layers being equivalent to one equivalent linear layer property. Due to matrix multiplication of matrices multiplying together to form one matrix as shown in Eq. 27 where the matrix between y_3 and y_2 (w_{32}) can be multiplied with the matrix between y_2 and y_1 (w_{21}) to form one matrix w_{31}. However this method will not work if there are non-linearity between layers. Which is why the correct output value of each layer was calculated so that non-linearities could be added between them.

$$y_3 = w_{32}\, y_2 = w_{32}\, w_{21}\, y_1 = w_{31}\, y_1 \qquad \text{Eq. 27}$$

Overall the experiments show that there is an improvement over loss reduction per iteration at the cost of slower training per iteration. The surprise result is that the method does not scale for several layers with non-linearities between them.

5 Discussion

Prior studies have prioritized the importance of performance in terms of speed and lowering loss values. Often at the cost of added complexity.

This paper had the main goal of seeing if a conceptually simpler method of updating weights would work. The secondary goal was to see if there was any performance benefits to the new method.

The main findings was that the new algorithm learns more per iteration than the one layer ANN at the cost of longer processing time. It was also found that the asymptotic value was also lower for the modified one layer. The algorithm got worse as the new method was extended to two layers by using calculated correct outputs of each layer.

The results show that this simpler method works for ANNs without non-linearities.

This paper shows yet another method that can beat the most popular weight optimization algorithm. Showing that there is still room for improvement.

The unexpected outcomes where that the two layer version of the network performed worse. This shows that the understanding of what the correct values at each previous layer is not accurate.

Although there is a trade off for increased learning versus higher time per iteration the new algorithm still learns enough per iteration to make it learn about the same rate when calculating loss per second for the first set of iterations. This might easily be fixed by some slight improvements by optimizing the code but is a good result for a first attempt. However the data showed that this algorithm can also be up to 25.5 times or more faster for reaching lower losses than the regular 1 layer ANN.

The new algorithm has been shown to be a good alternative ANN. Especially when it involves teaching others ANN updating techniques without the use of derivatives from calculus.

Close attention must be paid to understand the importance of the improvements of this new method. It should be understood that although the time to complete each iteration is more for the modified layers, the overall time to reach certain losses can be less or equal for the 1 layer modified case versus the 1 layer unmodified case. It is also important to understand that using two modified layer was worse than using one.

What this work shows is that the procedure to solving for the correct weight values for each iteration can be used as a learning algorithm for ANNs.

6 Conclusion

The aims of this paper was to make a weight updating algorithm that was both conceptually easier to understand and had better performance than the typical ANN back-propagation techniques.

The main finds where that the algorithm works best for one layer as opposed to several layers. The algorithm takes longer per iteration but can reach lower losses at an overall faster time.

What this paper implies is that there is another technique for optimizing neural networks weights and for learning that is not being utilized that involves the use of solving a system of equations.

What this paper can contribute is to better learning and teaching of ANN algorithms by reducing the technical debt required to jump in and learn and use the techniques. Someone who understands this simpler model might still be able to jump in using regular ANNs by at least understanding and easier notion of what an ANN should be trying to solve.

However there are limitations. The theory shown in this paper does not scale well to several layers. This means that there is still room for improvement.

Future work involves making this algorithm work better by making adding more layers work better. The experiments have shown the the logic of the weight update is a valid and works but that the logic behind the correct previous outputs is off or not taking certain aspects into account is not as good as it could be.

It is recommended that the user of this paper not see the previous layers correct output values to be less of an absolute truth as more as a guide into a potential way of going about in calculating these values.

7 Acknowledgements

This work is supported by Clemson University. Part of the support came from the Graduate Assistance in Areas of National Need (GAANN) for sponsoring the author of this paper and putting the importance of education into the for front of their mind.

References

[1] Minsky, M. L., & Papert, S. A. (1969). *Perceptrons*. Cambridge, MA: MIT Press.

[2] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In CVPR

[3] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).

[4] Sutskever, Ilya, et al. "On the importance of initialization and momentum in deep learning." *International conference on machine learning*. PMLR, 2013.

[5] Tijmen Tieleman, et al. RMSprop: Divide the gradient by a running average of it’s recent magnitude. COURSERA: Neural networks for machine learning, 4(2):26–31, 2012.

[6] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research (JMLR), 12:2121–2159, 2011.

[7] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

[8] D. E. Rumelhart et al., “Learning representations by back-propagating errors,” Cogn. Model., vol. 5, no. 3, 1988, Art. no. 1.

[9] M.A. Rakha, “On the Moore-Penrose generalized inverse matrix,” Applied Mathematics and Computation, 158, pp. 185-200, 2004.

[10] (Gradient Clipping)
I. Goodfellow, Y. Bengio, and A. Courville. Deep Learning (2016), MIT Press

[11] L. Deng. The MNIST database of handwritten digit images for machine learning rese

[12] (Convolution Neural Networks)
Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio. Object recognition with gradient-based learning. In Shape, contour and grouping in computer vision, pages 319–345. 1999.

[13] (ReLu)
Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, 36:193–202, 1980