Front Contribution instead of Back Propagation

Swaroop Mishra  
Arizona State University  
srmishr1@asu.edu

Anjana Arunkumar  
Arizona State University  
aarunku5@asu.edu

Abstract

Deep Learning’s outstanding track record across several domains has stemmed from the use of error backpropagation (BP). Several studies, however, have shown that it is impossible to execute BP in a real brain. Also, BP still serves as an important and unsolved bottleneck for memory usage and speed. We propose a simple, novel algorithm, the Front-Contribution algorithm, as a compact alternative to BP. The contributions of all weights with respect to the final layer weights are calculated before training commences and all the contributions are appended to weights of the final layer, i.e., the effective final layer weights are a non-linear function of themselves. Our algorithm then essentially collapses the network, precluding the necessity for weight updation of all weights not in the final layer. This reduction in parameters results in lower memory usage and higher training speed. We show that our algorithm produces the exact same output as BP, in contrast to several recently proposed algorithms approximating BP. Our preliminary experiments demonstrate the efficacy of the proposed algorithm. Our work provides a foundation to effectively utilize these presently under-explored “front contributions”, and serves to inspire the next generation of training algorithms.

1 Introduction and Related Work

Backpropagation of error (BP) [28] has been the best algorithm to train neural networks, and has driven deep learning to perform outstandingly across several domains [18]. However, it is not consistent with our findings about the brain [5,13]. In fact, it is not possible to execute BP in a real brain [2]. BP also suffers from several other problems, such as a vanishing/exploding gradient. Inspite of using careful initialization and architecture modifications [27,9]– for example, using RELU instead of sigmoid activations– quite effectively as workarounds, BP still is a key bottleneck for memory usage and speed. This may indicate that BP is a suboptimal algorithm and will be replaced.

Several different algorithms have been proposed to improve BP. However, those algorithms have tried to approximate BP and have not been able to work beyond toy datasets [2]; thus they cannot be applied in a real world setting. We survey three categories of BP literature– (i) better hardware implementation of BP [15,16,31,11,32,25], (ii) workarounds to approximate BP [33,7,10], and (iii) biologically inspired algorithms. Biologically inspired algorithms can further be segregated into four types: (i) Inspired from biological observations [29,7,26,17], these works try to approximate BP with the intention resolve its biological implausibility, (ii) Propagation of an alternative to error [19,21], (iii) Leveraging local errors, the power of single layer networks, and layer wise pre-training to approximate BP [24,23,3], (iv) Resolving the locking problem using decoupling [14,6,12,11,20] and its variants [27,8,22,4]. We were deeply motivated by (ii), (iii), and (iv) while coming up with the idea of ‘front contributions’– specifically, propagating something other than error, the idea of a single layer network, and decoupling, collectively inspire ‘front contributions’. The key distinction of our front contribution is that, it produces exact same output as BP unlike other approaches that have tried to approximate BP.

Beyond Backpropagation Workshop NeurIPS (2020), Vancouver, Canada.
2 Method

We know that a set of linear layers can be collapsed to a single layer network; however, non-linear activation functions have hitherto restricted the collapse of networks that they are applied in. Here, we justify that every fully connected multi-layer network can be collapsed to a single layer network, using the Front Contribution Algorithm, eliminating the requirement of backpropagation.

Formalization:

Let a neural network (NN) have $n$ layers, such that each layer of its weights are represented as $W_i : i \epsilon [1, n]$, where $W_1$ is applied to the input and $W_n$ is on the branch connected to the output node. Let the input of NN be $X$, intermediate layer outputs be $v_j : j \epsilon [1, n-1]$, and final output be $Y$. We can represent $Y$ as:

$$Y = f(X, W_1, W_2, \ldots, W_n)$$ (1)

Here, each of the weights are randomly initialized, and then updated using conventional backpropagation. Using the chain rule, we have:

$$\Delta W_1 = f(W_2, W_3, \ldots, W_n)$$ (2)
$$\Delta W_2 = f(W_3, W_4, \ldots, W_n)$$ (3)

From equation (2), we see that the input $X$ is multiplied with the weights, and then transformed by activations at various layers to produce the output $Y$. In other words, we can say that weight layers $W_1 \ldots W_n$ indirectly contribute towards deciding what $Y$ will be for a given $X$, in a hierarchical sequence starting from $W_1$ till $W_n$. The definition of ‘contribution’ varies depending on the application—for example, in the case of language models like BERT, contribution refers to attention.

Now, from equations (7,8) we see that in the training process, for the $i^{th}$ layer, $\Delta W$ is a function of all the $W_i$ layers, from the $(i+1)^{th}$ layer up till the $n^{th}$ layer. From equation (7) we see that $\Delta W_1$ is a function of $W_2 \ldots W_n$, but does not further depend on $W_1$, i.e., the value of $W_1$ at any iteration of training depends on the initial value of $W_1$—a random static value— and weights $W_2 \ldots W_n$. So, we can say that $W_1$ is not part of the system basis—the set of vectors that can be used to represent any system state, such as the use of $x, y, z$ bases to represent any state of a 3D system—as $W_2 \ldots W_n$ can represent the output value at any iteration. So, vector $W_1$ is not actually a necessary variable.

For example, by defining a 3D system in terms of $x, y, z, x+y+z$, we utilize an unnecessary variable, $x+y+z$. As $W_1$ is not an independent weight layer, we are therefore wasting GPU space by using it. However, if the weight layer $W_1$ is ignored, will the system still function as desired? The value of $v_1$ depends on weights in $W_1$, so if $W_1$ weights are not updated, $v_2$ will retain its old, incorrect value, consequently affecting $Y$ even if all other weights are updated correctly, as the network is connected as a hierarchy.

How do we compensate the non-updation of weights in $W_1$? Let a compensation weight $p$ be added to weights in $W_2$ such that $Y$ remains the same, i.e., $p$ compensates for the change in $v_2$ that normally happens with the updation of $W_1$. Here, $p$ must be a non-linear function of weights in $W_2$, as it compensates for an update that depends on $W_2 \ldots W_n$. $p$ can be derived by equating the $v_2$ value found post updation of $W_1$ in conventional backpropgration, and the $v_2$ value calculated without updating $W_1$ and replacing weights in $W_2$ as $W_2 + p$.

3 Finding an Expression for $p$:

Given a 2 layer neural network as shown in Figure, where inputs are $x_1, x_2$, and a single output $Y$ is produced, if $w_1, w_2, w_3, w_4 \epsilon W_1$ are not updated, then non-linear ‘compensation weights’ of $p, q$ must be added to $w_5, w_6 \epsilon W_2$ respectively in order to preserve $Y$ output updation. Here, we consider outputs at intermediate layer nodes after RELU activation to be $v_1, v_2$ (for example, $s_1 = w_1x_1 + w_2x_2$; $v_1 = RELU(s_1)$).
To derive values \( p, q \) of \( p, q \), consider that for the initial updation, we can say that:

\[
Y_{\text{backpropagation}} = Y_{\text{contribution factor}} \implies v_1 w_5 + v_2 w_6 = v_1' (w_5 + p) + v_2' (w_6 + q) \quad (4)
\]

where \( v_1' \) and \( v_2' \) are the constants-- their values don’t change, as \( w_1, w_2, w_3, w_4 \) don’t get updated.

During updation between the 0th and 1st iterations, \( dv_1' \) can have 3 possible values based on active and dead RELU before and after updation.

Consider a single input scheme, i.e., \([x_1, x_2]\), such that additional compensation to \( w_5 \) at each iteration is:

\[
\frac{dE}{dw_5} = p
\]

Using the chain rule, we can generalize this to calculate the total compensation weight \( P \) that must be added at the \( n \)th iteration. Here, \( p \) and \( P \) compensate for weights \( w_1, w_2 \). Similarly, \( q \) and \( Q \) can be defined to compensate for weights \( w_3, w_4 \). Let \( w_5 + p = r_5 \) and \( w_6 + q = r_6 \). \( r \) may be interpreted as a transformation on \( W_2 \) that takes care of the contribution factor from \( W_1 \). Hence, the general equation for updation after ‘n’ iterations is:

\[
r_5n = w_{5n} + \frac{w_{5n}^2 (x_1^2 + x_2^2)}{3(v_1 + \sum_{n=1}^{n} \frac{dE}{dY} w_{5n}(x_1^2 + x_2^2))} (A_{n-1} > 0 \ \&\ \& \ A_n > 0)
\]

\[
= w_{5n} + \frac{w_{5n}^2}{2n \frac{dE}{dY} (v_1 + \sum_{n=1}^{n} \frac{dE}{dY} w_{5n}(x_1^2 + x_2^2))} (A_{n-1} > 0 \ \&\ \& \ A_n < 0)
\]

where \( A_n = (w_{1c} + \sum_{n=1}^{n} \frac{dE}{dY} w_{5n} x_1) x_1 + (w_{2c} + \sum_{n=1}^{n} \frac{dE}{dY} w_{5n} x_2) x_2 \)

\[
r_6n = w_{6n} + \frac{w_{6n}^2 (x_1^2 + x_2^2)}{3(v_2 + \sum_{n=1}^{n} \frac{dE}{dY} w_{6n}(x_1^2 + x_2^2))} (B_{n-1} > 0 \ \&\ \& \ B_n > 0)
\]

\[
= w_{6n} + \frac{w_{6n}^2}{2n \frac{dE}{dY} (v_2 + \sum_{n=1}^{n} \frac{dE}{dY} w_{6n}(x_1^2 + x_2^2))} (B_{n-1} > 0 \ \&\ \& \ B_n < 0)
\]

where \( B_n = (w_{3c} + \sum_{n=1}^{n} \frac{dE}{dY} w_{6n} x_1) x_1 + (w_{4c} + \sum_{n=1}^{n} \frac{dE}{dY} w_{6n} x_2) x_2 \)

\[\text{Refer to Supplementary Material for the full derivation}\]
We proposed a simple, novel algorithm, the Front-Contribution algorithm, as a compact alternative to BP. Our algorithm has several advantages:

(i) Front Contribution is a one-time calculation, i.e., given a NN structure, with number of layers, activations, etc., the algorithm will output the non-linear weights of a corresponding collapsed network. While the expression for $W_n$ weights might be complex, the algorithm’s time complexity remains $O(1)$.

(ii) As the number of weights in a collapsed network is much lesser than in the original network, GPU space usage will be drastically improved. Deep Learning has significantly benefited from parallel computing, so by substantially improving parallel computing performance, we can potentially create deeper networks.

(iii) Due to the absence of back-propagation, we expect network training time to also be drastically reduced.

(iv) The post-training non-linear weights (represented in terms of their layers, $W_i$) – consider $W_n = W_n + k_1 W_n^3 + k_2 W_n^5 + ...$ – can potentially represent the features that the network has learned. Later terms in the preceding expression should represent simpler features (like edges), while initial terms should represent more complex features. However, $W_n$ will give us an idea of the aggregate features that are important for NN. We define aggregate features as the final set of features that NN cares about, after attending to various features in a hierarchical manner.

(v) The modified network can always be expanded to a conventional network depending on our requirements – for example, we might want to see activations of attention neurons, and can therefore re-expand the network up till those neurons, without having to expand it fully.

(vi) This derivation can be modified and extended to other types of networks, such as CNNs, RNNs and transformers.

\[ W_{n-1} = W_{n} + k_1 W_{n}^3 + k_2 W_{n}^5 + ... \]

Refer to Supplementary Materials: Experiments
References

[1] P. Baldi, P. Sadowski, and Z. Lu. Learning in the machine: Random backpropagation and the learning channel. *arXiv preprint arXiv:1612.02734*, 2016.

[2] S. Bartunov, A. Santoro, B. Richards, L. Marris, G. E. Hinton, and T. Lillicrap. Assessing the scalability of biologically-motivated deep learning algorithms and architectures. In *Advances in Neural Information Processing Systems*, pages 9368–9378, 2018.

[3] E. Belilovsky, M. Eickenberg, and E. Oyallon. Greedy layerwise learning can scale to imagenet. *arXiv preprint arXiv:1812.11446*, 2018.

[4] A. Choromanska, B. Cowen, S. Kumaravel, R. Luss, M. Rigotti, I. Rish, B. Kingsbury, P. Di-Achille, V. Gurev, R. Tejwani, et al. Beyond backprop: Online alternating minimization with auxiliary variables. *arXiv preprint arXiv:1806.09077*, 2018.

[5] F. Crick. The recent excitement about neural networks. *Nature*, 337(6203):129–132, 1989.

[6] W. M. Czarnecki, G. Swirszcz, M. Jaderberg, S. Osindero, O. Vinyals, and K. Kavukcuoglu. Understanding synthetic gradients and decoupled neural interfaces. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 904–912. JMLR. org, 2017.

[7] M. Ernoult, J. Grollier, D. Querlioz, Y. Bengio, and B. Scellier. Updates of equilibrium prop match gradients of backprop through time in an rnn with static input. In *Advances in Neural Information Processing Systems*, pages 7079–7089, 2019.

[8] S. Flennerhag, H. Yin, J. Keane, and M. Elliot. Breaking the activation function bottleneck through adaptive parameterization. In *Advances in Neural Information Processing Systems*, pages 7739–7750, 2018.

[9] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[10] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse. The reversible residual network: Backpropagation without storing activations. In *Advances in neural information processing systems*, pages 2214–2224, 2017.

[11] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves. Memory-efficient backpropagation through time. In *Advances in Neural Information Processing Systems*, pages 4125–4133, 2016.

[12] Z. Huo, B. Gu, Q. Yang, and H. Huang. Decoupled parallel backpropagation with convergence guarantee. *arXiv preprint arXiv:1804.10574*, 2018.

[13] E. M. Izhikevich and G. M. Edelman. Large-scale model of mammalian thalamocortical systems. *Proceedings of the national academy of sciences*, 105(9):3593–3598, 2008.

[14] M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, and K. Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1627–1635. JMLR. org, 2017.

[15] R. Kumar, M. Purohit, Z. Svitkina, E. Vee, and J. Wang. Efficient rematerialization for deep networks. In *Advances in Neural Information Processing Systems*, pages 15146–15155, 2019.

[16] M. Kusumoto, T. Inoue, G. Watanabe, T. Akiba, and M. Koyama. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. In *Advances in Neural Information Processing Systems*, pages 1161–1170, 2019.

[17] B. J. Lansdell, P. Prakash, and K. P. Kording. Learning to solve the credit assignment problem. *arXiv preprint arXiv:1906.00889*, 2019.

[18] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.
5 Supplemental Material

We know that a set of linear layers can be collapsed to a single layer network; however, non-linear activation functions have hitherto restricted the collapse of networks that they are applied in. Here, we justify that every fully connected multi-layer network can be collapsed to a single layer network, using the Front Contribution Algorithm, eliminating the requirement of backpropagation.

Analogy:

Consider an assembly line, where 10 people ($P_1 - P_{10}$) work sequentially to produce a toy car from plastic. $P_1$ cuts raw plastic (input) into different sizes, and gives it to $P_2$. $P_2$ smooths the plastic and gives it to $P_3$ (intermediate output), and so on until $P_{10}$ produces the toy car (final output). Initially, all the people do the task with no prior knowledge, so they perform random actions, aimed at producing the final output (car). $P_{10}$ sees the output produced, compares it to the desired toy car,
and rectifies his actions based on the input he receives from $P_9$, to make the output closer and closer to the toy car. $P_9$, $P_8$, etc. do the same— they look at the original error that they receive, find out how much they contribute to the error, and then try to rectify their actions by small amounts (not knowing exactly how much to rectify) to produce a perfect toy car. Therefore, every individual finds their respective individual actions to be performed and collectively produce a toy car. They can use this approach to build many other things, such as houses, fountains, etc., until they succeed as often as possible. Is this however an efficient method of learning to build anything?

Consider a person, $P_{11}$, who observes that $P_2$ is merely doing a fine-tuned version of $P_1$’s work by getting plastic to a desired physical state, and can therefore reliably do both $P_1$’s and their own work as they both are new to building. If $P_1$ is eliminated, and $P_2$ is asked to both cut and smooth plastic, by hiring one less person and reducing cost of transfer between $P_1$ and $P_2$, the budget allows for extra training time for $P_2$. $P_2$ also doesn’t need to necessarily take more time to complete their work, if given a tool that both cuts and smooths the cut edges of plastic simultaneously— i.e., the output from $P_2$ becomes the sum of rectified effort from $P_1$ and $P_2$. This is a one time cost, and $P_{11}$ can develop the required tool by finding the ‘contribution’ of $P_1$’s actions to $P_2$, i.e., how much $P_1$’s work attends to $P_2$. Similarly, by observing the contributions of workers down the assembly line, $P_{11}$ is able to eliminate all but the last worker, and provide him with a simple tool that can be used to perform all the work. This results in a drastic decrease in budget and increase in building speed. It also allows more time for process and output analysis, to understand how different steps and features contribute towards a specific task and overall output— the process becomes more transparent, akin to process explainability.

This can be extended towards other concepts such as center of mass in physics as well. We aim to calculate a single-layer network with non-linear weights (calculated with Forward Contribution) whose performance is tantamount to a conventional multi-layer network that uses backpropagation. This is not an approximation (mimic) network, but an equivalent network to BP.

**Formalization:**

Let a neural network (NN) have $n$ layers, such that each layer of its weights are represented as $W_i : i \in [1, n]$, where $W_1$ is applied to the input and $W_n$ is on the branch connected to the output node. Let the input of NN be $X$, intermediate layer outputs be $v_j : j \in [1, n-1]$, and final output be $Y$. We can represent $Y$ as:

$$Y = f(X, W_1, W_2,...,W_n)$$  \hspace{1cm} (6)

Here, each of the weights are randomly initialized, and then updated using conventional back propagation. Using the chain rule, we have:

$$\Delta W_1 = f(W_2, W_3...W_n)$$  \hspace{1cm} (7)
$$\Delta W_2 = f(W_3, W_4...W_n)$$  \hspace{1cm} (8)

From equation (6), we see that the input $X$ is multiplied with the weights, and then transformed by activations at various layers to produce the output $Y$. In other words, we can say that weight layers $W_1...W_n$ indirectly contribute towards deciding what $Y$ will be for a given $X$, in a hierarchical sequence starting from $W_1$ till $W_n$. The definition of ‘contribution’ varies depending on the application— for example, in the case of language models like BERT, contribution refers to attention.

Now, from equations (7) and (8) we see that in the training process, for the $i^{th}$ layer, $\Delta W$ is a function of all the $W_i$ layers, from the $(i+1)^{th}$ layer up till the $n^{th}$ layer. From equation (7) we see that $\Delta W_1$ is a function of $W_2...W_n$, but does not further depend on $W_1$, i.e., the value of $W_1$ at any iteration of training depends on the initial value of $W_1$— a random static value— and weights $W_2...W_n$. So, we can say that $W_1$ is not part of the system basis— the set of vectors that can be used to represent any system state, such as the use of $x$, $y$, $z$ bases to represent any state of a 3D system— as $W_2...W_n$ can represent the output value at any iteration. So, vector $W_1$ is not actually a necessary variable.

Since $W_1$ is not part of the basis, it is actually not a necessary variable; for example, by defining a 3D system in terms of $x$, $y$, $z$, $x + y + z$, we utilize an unnecessary variable, $x + y + z$. As $W_1$ is not an independent weight layer, we are therefore wasting GPU space by using it. However, if the weight layer $W_1$ is ignored, will the system still function as desired? The value of $v_1$ depends on weights in $W_1$, so if $W_1$ weights are not updated, $v_2$ will retain its old, incorrect value, consequently affecting $Y$ even if all other weights are updated correctly, as the network is connected as a hierarchy.
Figure 2: Neural network structure for (a) Backpropagation, (b) Forward Contribution. Here, layer $W_1$ contains weights $w_1, w_2, w_3, w_4$ and $W_2$ contains weights $w_5, w_6$.

How do we compensate the non-updation of weights in $W_1$? Let a compensation weight $p$ be added to weights in $W_2$ such that $Y$ remains the same, i.e., $p$ compensates for the change in $v_2$ that normally happens with the updation of $W_1$. Here, $p$ must be a non-linear function of weights in $W_2$, as it compensates for an update that depends on $W_2...W_n$. $p$ can be derived by equating the $v_2$ value found post updation of $W_1$ in conventional backpropgation, and the $v_2$ value calculated without updating $W_1$ and replacing weights in $W_2$ as $W_2 + p$.

6 Finding an Expression for $p$

Let $X$ be the input vector: $[x_1, x_2], [x_1^\#, x_2^\#]$. Let $Y_2$ be the vector of gold labels: $[y_g, y_g^\#, y_g^\*, y_g^\#]$. Let $w_1, w_2, w_3, w_4$ be the weights of the first layer ($W_1$). Let $w_5, w_6$ be the weights of the last layer ($W_2$). We define the state of the intermediate nodes $v_1$ and $v_2$ for a given input as:

\[
s_1 = w_1 \cdot x_1 + w_2 \cdot x_2, s_2 = w_3 \cdot x_1 + w_4 \cdot x_2
\]

The activation function for intermediate nodes is RELU, i.e.:

\[
v_1 = RELU(s_1), v_2 = RELU(s_2)
\]

where

\[
RELU(s) = \begin{cases} 
  s & \text{if } s > 0 \\
  0 & \text{if } s < 0
\end{cases}
\]

Let $\eta$ be the learning rate and $Y$ be the output vector: $[y, y^\#, y^\*, y^\#]$, where

\[
y = w_5 \cdot v_1 + w_6 \cdot v_2
\]

Let the error $E$ be defined as:

\[
E = \frac{1}{2}(y - y_g)^2, \text{ i.e., } \frac{dE}{dY} = y - y_g
\]

The following repeats successively for each epoch, for a single given input:

\[
w_5' = w_5 - \eta \frac{dE}{dY} v_1, w_6' = w_6 - \eta \frac{dE}{dY} v_2
\]
$\frac{dE}{dw_1}$ can have 2 possible values based on whether the RELU is dead or not, as can $\frac{dE}{dw_2}$, $\frac{dE}{dw_3}$, $\frac{dE}{dw_4}$:

$$
\begin{align*}
\frac{dE}{dw_1} &= 0 \quad \text{(or)} \\
\frac{dE}{dw_2} &= \frac{dE}{dY}.w_5.x_1(s_1 > 0) \\
\frac{dE}{dw_3} &= \frac{dE}{dY}.w_5.x_2(s_1 > 0) \\
\frac{dE}{dw_4} &= \frac{dE}{dY}.w_6.x_1(s_2 > 0) \\
\frac{dE}{dw_5} &= \frac{dE}{dY}.w_6.x_2(s_2 > 0)
\end{align*}
$$

Based on this, we update the weights of the first layer as:

$$
\begin{align*}
w'_1 &= w_1 - \eta \frac{dE}{dY} w'_5 x_1 (s_1 > 0) \quad \text{(or)} \quad w'_1 = w_1 (s_1 < 0) \\
w'_2 &= w_2 - \eta \frac{dE}{dY} w'_5 x_2 (s_1 > 0) \quad \text{(or)} \quad w'_2 = w_2 (s_1 < 0) \\
w'_3 &= w_3 - \eta \frac{dE}{dY} w'_6 x_1 (s_2 > 0) \quad \text{(or)} \quad w'_3 = w_3 (s_2 < 0) \\
w'_4 &= w_4 - \eta \frac{dE}{dY} w'_6 x_2 (s_2 > 0) \quad \text{(or)} \quad w'_4 = w_4 (s_2 < 0)
\end{align*}
$$

$$
\begin{align*}
&= s'_1 = w'_1 \cdot x_1 + w'_2 \cdot x_2, \\
&= w'_3 \cdot x_1 + w'_4 \cdot x_2, \\
&= v'_1 = \text{RELU}(s'_1), \\
&= v'_2 = \text{RELU}(s'_2), \\
&= y = w'_5 \cdot v'_1 + w'_6 \cdot v'_2
\end{align*}
$$

where

$$
\frac{dE}{dY} = y - y_g = (w_5 \cdot v_1 + w_6 \cdot v_2) - y_g
$$

From equations 14, 15:

$$
\begin{align*}
s'_1 &= (w_1 - \eta \frac{dE}{dY} w'_5 x_1) x_1 + (w_2 - \eta \frac{dE}{dY} w'_5 x_2) x_2 \\
&= w_1 \cdot x_1 + w_2 \cdot x_2 - \eta \frac{dE}{dY} w'_5 (x_1^2 + x_2^2) \\
&= s_1 - \eta \frac{dE}{dY} w'_5 (x_1^2 + x_2^2)
\end{align*}
$$

Parallely,

$$
\begin{align*}
s'_2 &= s_2 - \eta \frac{dE}{dY} w'_6 (x_1^2 + x_2^2)
\end{align*}
$$

If the next input is $[x_1^\circ, x_2^\circ]$ then for the next epoch, we have:

$$
\begin{align*}
s'_1^\circ &= (w_1 - \eta \frac{dE}{dY} w'_5 x_1^\circ) x_1^\circ + (w_2 - \eta \frac{dE}{dY} w'_5 x_2^\circ) x_2^\circ \\
&= w_1 \cdot x_1^\circ + w_2 \cdot x_2^\circ - \eta \frac{dE}{dY} w'_5 (x_1^\circ x_1^\circ + x_2^\circ x_2^\circ)
\end{align*}
$$
\[ s_2' = (w_3 - \eta \frac{dE}{dY} w_0^* x_1^*) x_1^* + (w_4 - \eta \frac{dE}{dY} w_0^* x_2^*) x_2^* \]
\[ = w_3 \cdot x_1^* + w_4 \cdot x_2^* - \eta \frac{dE}{dY} w_0^* (x_1 x_1^* + x_2 x_2^*) \]

Using \( s_1^* = w_1 \cdot x_1^* + w_2 \cdot x_2^* \), \( s_2^* = w_3 \cdot x_1^* + w_4 \cdot x_2^* \), and equation [17], we can simplify equation [18] as:

\[ s_1' = s_1^* + \frac{s_1^* - s_1}{x_1^2 + x_2^2} (x_1 x_1^* + x_2 x_2^*) \]  \hspace{1cm} (19)
\[ s_2' = s_2^* + \frac{s_2^* - s_2}{x_1^2 + x_2^2} (x_1 x_1^* + x_2 x_2^*) \]

Let:
\[ s_1^* = w_1 \cdot x_1^* + w_2 \cdot x_2^* \], \( s_1^# = w_1 \cdot x_1^# + w_2 \cdot x_2^# \)
\[ s_2^* = w_3 \cdot x_1^* + w_4 \cdot x_2^* \], \( s_2^# = w_3 \cdot x_1^# + w_4 \cdot x_2^# \]

Using the above with equation [19] for the next 2 epochs with inputs \([x_1^*, x_2^*]\) and \([x_1^#, x_2^#]\), we get:

\[ s_1'' = s_1^* + \frac{s_1^* - s_1}{x_1^2 + x_2^2} (x_1 x_1^* + x_2 x_2^* ) + \frac{s_1'^* - s_1'}{x_1^2 + x_2^2} (x_1 x_1^* + x_2 x_2^*) \]  \hspace{1cm} (20)
\[ s_1'^# = s_1^# + \frac{s_1^* - s_1}{x_1^2 + x_2^2} (x_1 x_1^* + x_2 x_2^* ) + \frac{s_1'^# - s_1'}{x_1^2 + x_2^2} (x_1 x_1^* + x_2 x_2^*) \]
\[ s_2'' = s_2^* + \frac{s_2^* - s_2}{x_1^2 + x_2^2} (x_1 x_1^* + x_2 x_2^* ) + \frac{s_2'^* - s_2'}{x_1^2 + x_2^2} (x_1 x_1^* + x_2 x_2^*) \]
\[ s_2'^# = s_2^# + \frac{s_2^* - s_2}{x_1^2 + x_2^2} (x_1 x_1^* + x_2 x_2^* ) + \frac{s_2'^# - s_2'}{x_1^2 + x_2^2} (x_1 x_1^* + x_2 x_2^*) \]

If \( w_1, w_2, w_3, w_4 \) are not updated, then non-linear 'compensation weights' of \( p, q \) must be added to \( w_5, w_6 \) respectively in order to preserve \( Y \) output updation. To derive values of \( p, q \), consider that for the initial updation, using equations [20][21][22][23], we can say that:

\[ Y_{\text{backpropagation}} = Y_{\text{contribution factor}} \implies v_1 \cdot w_5 + v_2 \cdot w_6 = v_1^* \cdot (w_5 + p) + v_2^* \cdot (w_6 + q) \]  \hspace{1cm} (21)

where \( v_1^* \) and \( v_2^* \) are the constants—their values don’t change, as \( w_1, w_2, w_3, w_4 \) don’t get updated. So, for the first updation, we get:

\[ v_1^* \cdot w_5' + v_2^* \cdot w_6' = v_1^* \cdot (w_5' + p) + v_2^* \cdot (w_6' + q) \]
\[ \implies (v_1^* + dv_1^*).w_5' + (v_2^* + dv_2^*).w_6' = v_1^*.(w_5' + p) + v_2^*.(w_6' + q) \]
where \( w_5', w_6' \) are updated as per equation [13]. Now, compensating \( v_1 \) and \( v_2 \) separately, we have:

\[ (v_1^* + dv_1^*).w_5' = v_1^*.(w_5' + p) \implies dv_1^*.w_5' = v_1^* \cdot p \]  \hspace{1cm} (22)

Similarly,
\[ (v_2^* + dv_2^*).w_6' = v_2^*.(w_6' + q) \implies dv_2^*.w_6' = v_2^* \cdot q \]
During updation between the 0th and 1st iterations, \( dv_1' \) can have 3 possible values: (i) RELU is dead at the 0th iteration and stays dead after the 1st update, (ii) RELU is active at the 0th iteration and stays active after the 1st update, and (iii) RELU is dead at the 0th iteration and becomes active after the 1st update:

\[
dv_1' = 0
\]  

\[
dv_1' = dv_1'.x_1 + dv_2'.x_2 \quad ((s_1 \geq 0) \& \& (s_1' \geq 0))
\]

\[
dv_1' = -v_1 = -s_1 \quad ((s_1 \geq 0) \& \& (s_1' < 0))
\]

where

\[
s_1 = w_1.x_1 + w_2.x_2 \\
s_1' = s_1(s_1 < 0)
\]

i.e.,

\[
s_1' = (w_1'.x_1 + w_2'.x_2)(s_1 \geq 0)
\]  

\[
\Rightarrow s_1' = (w_1 - \eta \frac{dE}{dw_1}).x_1 + (w_2 - \eta \frac{dE}{dw_2}).x_2 \\
\Rightarrow s_1' = (w_1 - \eta \frac{dE}{dY}.w_5'.x_1(s_1 \geq 0)).x_1 + (w_2 - \eta \frac{dE}{dY}.w_5'.x_2(s_1 \geq 0)).x_2 \\
\Rightarrow s_1' = s_1 - \eta \frac{dE}{dY}.w_5'(x_1^2 + x_2^2)(s_1 \geq 0)
\]

Using equation 24 in 23,

\[
dv_1' = -\eta \frac{dE}{dY}.w_5'(x_1^2 + x_2^2)(((w_1.x_1 + w_2.x_2) \geq 0) \& \& \\
((w_1 - \eta \frac{dE}{dY}.w_5'.x_1).x_1 + (w_2 - \eta \frac{dE}{dY}.w_5'.x_2).x_2) \geq 0))
\]

\[
dv_1' = -(w_1.x_1 + w_2.x_2)(((w_1.x_1 + w_2.x_2) \geq 0) \& \& \\
((w_1 - \eta \frac{dE}{dY}.w_5'.x_1).x_1 + (w_2 - \eta \frac{dE}{dY}.w_5'.x_2).x_2) < 0))
\]

Using equation 22 and 26, we get:

\[
p = \frac{dv_1'.w_5'}{v_1}
\]  

\[
= -\eta \frac{dE}{dY}.w_5'(x_1^2 + x_2^2)(((w_1.x_1 + w_2.x_2) \geq 0) \& \& \\
((w_1 - \eta \frac{dE}{dY}.w_5'.x_1).x_1 + (w_2 - \eta \frac{dE}{dY}.w_5'.x_2).x_2) \geq 0))
\]

\[
= -\eta \frac{dE}{dY}.w_5'(x_1^2 + x_2^2)(((w_1.x_1 + w_2.x_2) \geq 0) \& \& \\
((w_1 - \eta \frac{dE}{dY}.w_5'.x_1).x_1 + (w_2 - \eta \frac{dE}{dY}.w_5'.x_2).x_2) \geq 0))
\]

\[
p = \frac{2v_1'.w_5'}{v_1}
\]  

\[
= -(w_1.x_1 + w_2.x_2)(((w_1.x_1 + w_2.x_2) \geq 0) \& \& \\
((w_1 - \eta \frac{dE}{dY}.w_5'.x_1).x_1 + (w_2 - \eta \frac{dE}{dY}.w_5'.x_2).x_2) \geq 0))
\]

\[
= -(w_1.(w_1.x_1 + w_2.x_2) \geq 0) \& \& \\
((w_1 - \eta \frac{dE}{dY}.w_5'.x_1).x_1 + (w_2 - \eta \frac{dE}{dY}.w_5'.x_2).x_2) \geq 0))
\]

The above represents the amount of weight \( p \) to be added for the given iteration. Consider only a single input scheme, i.e., \([x_1,x_2]\), such that additional compensation to \( w_5' \) at each iteration is: 

11
\[ \frac{dE}{dw_5^\alpha} = p \]

Then using the chain rule, we can generalize this to calculate the total compensation weight \( P \) that must be added at the \( n \)th iteration as follows:

\[
Additional \quad -\eta \frac{dE}{dY} \cdot v_i^1 \cdot \frac{dP}{dw_5} = p
\]

\[
\Rightarrow \frac{dP}{dw_5} = \frac{-p}{\eta \frac{dE}{dY} \cdot v_i^1}
\]

\[
\Rightarrow P = \int \frac{-p}{\eta \frac{dE}{dY} \cdot v_i^1} \, dw_5
\]

We can find \( P \) in terms of \( w_5^* \) using equation 18, 20 as:

\[
\begin{align*}
&= \int \left[ \eta \frac{dE}{dY} \cdot v_i^1 \right] \, dw_5' \\
&= w_5^* \left( \left( (w_1, x_1 + w_2, x_2) \geq 0 \right) \& \& \left( (w_1 - \eta \frac{dE}{dY} \cdot v_i^1, x_1 + (w_2 - \eta \frac{dE}{dY} \cdot v_i^1, x_2) \geq 0 \right) \right) \, dw_5'
\end{align*}
\]

Using equation 19, 20 as:

\[
\begin{align*}
&= \frac{w_5^* \left( \left( (w_1, x_1 + w_2, x_2) \geq 0 \right) \& \& \left( (w_1 - \eta \frac{dE}{dY} \cdot v_i^1, x_1 + (w_2 - \eta \frac{dE}{dY} \cdot v_i^1, x_2) \geq 0 \right) \right) \, dw_5'}{2 \eta \frac{dE}{dY} \cdot v_i^1}
\end{align*}
\]

Here, \( p \) and \( P \) compensate for weights \( w_1, w_2 \). Similarly, \( q \) and \( Q \) can be defined to compensate for weights \( w_3, w_4 \). Let \( w_5 + p \) be \( r_5 \) and \( w_6 + q \) be \( r_6 \). \( r \) may be interpreted as a transformation on \( w_{layer2} \) that takes care of the contribution factor from \( w_{layer1} \). Hence, the general equation for updation after ‘\( n \)’ iterations is:

\[
r_{5n} = w_{5n} + \frac{w_{5n}^3 (x_1^2 + x_2^2)}{3(v_{1c} + \sum_1^n \frac{dE}{dY} \cdot w_{5n} (x_1^2 + x_2^2)^2)} (A_{n-1} > 0 \& \& A_n > 0)
\]

\[
= w_{5n} + \frac{w_{5n}^2}{2 \eta \frac{dE}{dY} \cdot (v_{1c} + \sum_1^n \frac{dE}{dY} \cdot w_{5n} (x_1^2 + x_2^2))} (A_{n-1} > 0 \& \& A_n < 0)
\]

where \( A_n = (v_{1c} + \sum_1^n \frac{dE}{dY} \cdot w_{5n} x_1) x_1 + (v_{2c} + \sum_1^n \frac{dE}{dY} \cdot w_{5n} x_2) x_2 \)

\[
r_{6n} = w_{6n} + \frac{w_{6n}^3 (x_1^2 + x_2^2)}{3(v_{2c} + \sum_1^n \frac{dE}{dY} \cdot w_{6n} (x_1^2 + x_2^2)^2)} (B_{n-1} > 0 \& \& B_n > 0)
\]

\[
= w_{6n} + \frac{w_{6n}^2}{2 \eta \frac{dE}{dY} \cdot (v_{2c} + \sum_1^n \frac{dE}{dY} \cdot w_{6n} (x_1^2 + x_2^2))} (B_{n-1} > 0 \& \& B_n < 0)
\]

12
where $B_n = (w_{3c} + \sum_{i=1}^{n} \frac{dE}{dx_i} w_{6n}x_1)x_1 + (w_{4c} + \sum_{i=1}^{n} \frac{dE}{dx_i} w_{6n}x_2)x_2$

Here subscript $n$ represents value at the $n^{th}$ iteration, and subscript $c$ represents the constant value, i.e., the initial value which does not change across iterations.

In the condition $s_1 \geq 0$ and $s_1' \geq 0$, using equations \[13\] \[21\] \[18\], for initial input $[x_1, x_2]$ we have:

\[
p = -\eta \frac{dE}{dy} w^{\prime 2}_5 (x_1^2 + x_2^2) v_1^c, \quad q = -\eta \frac{dE}{dy} w^{\prime 2}_6 (x_1^2 + x_2^2) v_2^c
\]

\[
y' = (w'_5 + p)v_1^c + (w'_6 + p)v_2^c, \quad \frac{dE'}{dy} = y' - y_g
\]

\[
w'_5 = w'_5 - \eta \frac{dE'}{dy} v_1^c = w'_5 - \eta(v_1^c w'_5 + v_1^c p + v_2^c w'_6 + v_2^c q - y_g)(v_1^c - w'_5)
\]

\[
w'_6 = w'_6 - \eta \frac{dE'}{dy} v_1^c = w'_6 - \eta(v_1^c w'_6 + v_1^c p + v_2^c w'_6 + v_2^c q - y_g)(v_2^c - w'_6)
\]

For next input $[x_1', x_2']$ we have:

\[
v_1' = (w'_5 + p + p')v_1^c \quad \Rightarrow \quad -\eta \frac{dE}{dy} w^{\prime 2}_5 (x_1 x_1' + x_2 x_2') = p' w_1 x_1' + w_2 x_2' + p(x_1 x_1' + x_2 x_2')
\]

\[
p' = p - \frac{\eta}{v_1^c} w^{\prime 2}_5 (x_1 x_1' + x_2 x_2')
\]

Parallely, $q' = q - \frac{\eta}{v_2^c} w^{\prime 2}_6 (x_1 x_1' + x_2 x_2')$ \hspace{1cm} (31)

From the above, we see that for multiple inputs, the entire input sequence must be stored to calculate the $P$ and $Q$ values. Therefore, in the cases of updating either (i) $w_5, w_6$ by adding compensatory weights $p, q$, or (ii) updating the compensatory weights $p, q$ for a fixed $w_5, w_6$, we only use one input.

To handle multiple inputs, we update $s_1, s_2$ as follows:

Initially:

\[
s_1 = w_1 x_1 + w_2 x_2, \quad s_2 = w_3 x_1 + w_4 x_2
\]

\[
S_1 = w_1 x_1 - w_2 x_2, \quad S_2 = w_3 x_1 - w_4 x_2
\]

Updated as:

\[
s_1 = \frac{s_1' + s_1'}{2} x_1' + \frac{s_2' - s_2'}{2} x_2', \quad s_2 = \frac{s_2' + s_2'}{2} x_1' + \frac{s_2' - s_2'}{2} x_2'
\]

\[
S_1 = \frac{s_1' + s_1'}{2} x_1' - \frac{s_1' - s_1'}{2} x_2', \quad S_2 = \frac{s_2' + s_2'}{2} x_1' - \frac{s_2' - s_2'}{2} x_2'
\] \hspace{1cm} (32)

Above, we have shown that in the context of \[8\] $(W_2 + p)$ (where $W_2$ is the second weight layer comprising of $w_5, w_6$) can contribute (attend for text) equivalent to updated $W_1$, and $W_2, p$ is the contribution/weight attention factor of $W_1$ to $W_2$, i.e., the amount by which $W_1$ contributes/attends to $W_2$. In this case, $W_2$ weights are now non-linear based on equation \[30\] $W_2$ weights can take the form of $W_2 + W_2^2$. By introducing non-linear weights, we can therefore reduce the total number of weights by 1 set (i.e., the number of weights corresponding to a particular layer). By showing that NN of n layers can be collapsed to n-1 layers, we can further use the method of induction to theoretically prove that NN can be collapsed from $(n-1) -> (n-2) -> ... -> 1$ layer, such that the final network has just non-linear weights for layer $W_n$.

This method of $W_n$ calculation is the Front Contribution Algorithm, as instead of propagating error backwards, the network propagates contribution forward to collapse the network.

Experiments:
The results of the three techniques we use for Forward Contribution, over the implementation of the XOR task are displayed in Figure 3.

The order of error is $< 10^{-15}$, proving that Forward Contribution is a true equivalent of backpropagation.

![Figure 3](image)

Figure 3: Here in all 3 figures, x-axis represents the training iteration, and the y-axis represents the error between output produced through backpropagation and the output produced by forward contribution. Top: On updation of weights $w_5, w_6$ as $w_5 + p, w_6 + q$. Middle: On updation of values of $p, q$ without updation of $w_5, w_6$. Bottom: Updation of states $v_1, v_2$ without weight updation.

**Backpropagation vs. Front Contribution Illustrations:** We illustrate the process flows of Back Propagation and Front Contribution in Figure 4.
Figure 4: On the left hand side, from top->bottom are the steps for Back-Propagation; the top right image has the final summary of Back-Propagation. The remaining images on the right show how Front-Contributions functions (top->bottom).