Abstract

In recent years, neural networks have rapidly gained attention as powerful tools for learning representations. They are known to require notoriously protracted training times. This has led to interest in possible ways to accelerate computations during model training.

In this research, we propose a series of neural network architectures to implement computations efficiently. We harness Lin, Tegmark, and Rolnick's recent development of a "multiplication gate" – which multiplies floating point numbers using only additions and subtractions at the expense of four additional neurons per multiplication – to construct several no-multiplication architectures (NMAs). These NMAs are designed to expedite computations in both forward- and back-propagation for fully connected neural networks and convolutional neural networks (CNNs).

Broadly speaking, the non-use of explicit multiplication reduces the need for costly, complex GPU hardware and raises the possibility of running neural networks in real time on simple devices such as a smartphone. In particular, the reduction in computation time can be greatly magnified for CNNs, due to their convolutional layers. We expand upon these substantial benefits by deriving a generalized expression for the number of distinct multiplications that must be computed in a given convolutional layer.

1 Introduction

Neural networks have achieved vast improvements in areas ranging from speech recognition to translation and genomics [3]. However, these improvements are hindered by the computational cost of neural networks that results from the number of multiplications that networks must perform. As a result, neural networks have largely been unfeasible to run on mobile devices. By eliminating explicit multiplications, the applications of neural networks can potentially be greatly expanded and be used on simpler devices such as smartphones.

We propose a series of architectures (NMAs) that eliminate explicit multiplication by employing a "multiplication gate" that in effect replaces multiplication with addition and subtraction. Lin, Tegmark, and Rolnick prove that a multiplication gate can approximate any product arbitrarily well [3]. Our NMAs perform forward- and back-propagation for fully connected neural networks and convolutional neural networks (CNNs). After developing this architecture, we derive expressions for the number of distinct multiplications performed in forward- and back-propagation, which are necessary to evaluate the extent to which NMAs decrease computational cost.

2 Notation

2.1 Fully Connected Neural Networks

2.1.1 Forward Propagation

For an L-layer fully connected neural network (FCNN), we let U_ℓ denote the number of neurons in layer ℓ, $X^{(\ell)}$ the row vector whose components are neurons in layer ℓ, $X_i^{(\ell)}$ the ith neuron in layer ℓ, $w_{ab}^{(\ell)}$ the weight associated with the edge connecting $X_a^{(\ell)}$ to $X_b^{(\ell+1)}$, and let $W^{(\ell)}$ denote the U_ℓ by $U_{\ell+1}$ matrix of such weights for $1 <= \ell < L$.

2.1.2 Backpropagation

Consider the neural network as described in section 2.1.1. We summarize from [1] necessary results that are used to derive an expression for the error (δ) backpropagated to layer ℓ. First, let s denote the activation function, which by convention is the sigmoid function: $s(x) = \frac{1}{1+e^{-x}}$.

Next, given that $X^{(1)}$ represents neurons in layer one, we can represent neurons in subsequent layers as $X^{(\ell)} = s\left(X^{(\ell-1)}W^{(\ell)}\right)$ for $1 < \ell < L$. We define $D^{(\ell)}$ as the diagonal matrix of terms $X_i^{(\ell+1)}\left(1 - X_i^{(\ell+1)}\right)$ with $1 <= i <= U_{\ell+1}$. Then, we let T be the row vector whose components are the target values t_i and define e as the column vector whose components are $X_i^{(L)} - t_i$ for $1 <= i <= U_L$. Given these assignments, as shown in [1], we can define the error backpropagated to layer ℓ from subsequent layers in the network as

$$\delta^{(\ell)} = D^{\ell}W^{(\ell+1)} \cdots W^{(L-1)}D^{L-1}e. \tag{1}$$

A bit of algebra shows that each $\delta^{(\ell)}$ is a column vector of dimension $U_{\ell+1}$. From [1], we also have that the gradient of each weight $w_{ab}^{(\ell)}$ $(1 <= \ell < L)$ is $\frac{\partial E}{\partial w_{ab}^{(\ell)}} = X_a^{(\ell)}\delta_b^{(\ell)}$, with $1 <= a <= U_\ell$, and $1 <= b <= U_{\ell+1}$. This is used to adjust w_{ab}^{ℓ} to minimize error (α denotes the learning rate):

$$\Delta w_{ab}^{(\ell)} = -\alpha \frac{\partial E}{\partial w_{ab}^{(\ell)}} = -\alpha X_a^{(\ell)}\delta_b^{(\ell)}. \tag{2}$$

2.2 Convolutional Neural Networks

2.2.1 Forward Propagation

Now suppose we have a convolutional neural network (CNN) with $L - 1$ convolutional (conv.) layers. Let $X^{(\ell)}$ denote the input feature map to layer ℓ with height $H_x^{(\ell)}$, width $W_x^{(\ell)}$, and component values $X_{ij}^{(\ell)}$. We do not consider the depth of the input map, as operations carried out for one depth slice can be performed similarly for others. Let the kernel applied to $X^{(\ell)}$ be denoted $K^{(\ell)}$, with height and width f_ℓ and weights $k_{ab}^{(\ell)}$, and let $\otimes$ denote the convolution of $X^{(\ell)}$ with $K^{(\ell)}$. Let $Y^{(\ell)}$ denote the output feature map of layer ℓ with height $H_y^{(\ell)}$, width $W_y^{(\ell)}$, and component values $Y_{ij}^{(\ell)}$. We also let S denote the stride and P the amount of zero padding, which we assume in this report to be 1 and 0, respectively. From [2], we have that

$$H_y^{(\ell)} = \frac{H_x^{(\ell)} - f_\ell + 2P}{S} + 1 \quad \text{and} \quad W_y^{(\ell)} = \frac{W_x^{(\ell)} - f_\ell + 2P}{S} + 1.$$

Also note that in this work, we assume that the architecture for any referenced CNN is such that each conv. layer is followed by the application of some nonlinearity σ. Thus, given that $X^{(1)}$ is the initial input map, we have that the input and output maps (X and Y, resp.) to each conv. layer are:

$$Y_{ij}^{(1)} = \sum_{a=1}^{f_1} \sum_{b=1}^{f_1} k_{ab}^{(1)} X_{(i+a-1)(j+b-1)}^{(1)}$$

$$X_{ij}^{(2)} = \sigma\left(Y_{ij}^{(1)}\right) \qquad Y_{ij}^{(2)} = \sum_{a=1}^{f_2} \sum_{b=1}^{f_2} k_{ab}^{(2)} X_{(i+a-1)(j+b-1)}^{(2)}$$

$$\vdots$$

$$X_{ij}^{(L-1)} = \sigma\left(Y_{ij}^{(L-2)}\right) \qquad Y_i j^{(L-1)} = \sum_{a=1}^{f_{L-1}} \sum_{b=1}^{f_{L-1}} k_{ab}^{(L-1)} X_{(i+a-1)(j+b-1)}^{(L-1)}$$

$$X_{ij}^{(L)} = \sigma\left(Y_{ij}^{(L-1)}\right).$$

For the above expressions, first note that since $S = 1$ and $P = 0$, we know that $H_x^{(\ell+1)} \leq H_x^{(\ell)}$ for $\forall \ell$. Second, as a bit of algebraic manipulation shows that $i + a - 1 = H_x^{(L-1)}$ and $j + b - 1 = W_x^{(L-1)}$ for $i = H_x^{(\ell)} = H_y^{(\ell-1)}$, $j = W_x^{(\ell)} = W_y^{(\ell-1)}$, and $a = b = f_{\ell-1}$, we know that

$$X_{(H_x^{(L)}+f_{L-1}-1)(W_x^{(L)}+f_{L-1}-1)}^{(\ell-1)} = X_{H_x^{(\ell-1)} W_x^{(\ell-1)(\ell-1)}}. \tag{3}$$

2.2.2 Backpropagation

Now we wish to backpropagate the error with respect to each kernel weight for the CNN described in section 2.2.1. Consider some layer ℓ of the CNN. We compute the gradient of each weight in the kernel $K^{(\ell)}$ as follows:

$$\frac{\partial E}{\partial k_{ab}^{(\ell-1)}} = \sum_{i=1}^{H_x^{(\ell)}} \sum_{j=1}^{W_x^{(\ell)}} \frac{\partial E}{\partial X_{ij}^{(\ell)}} \cdot \sigma'\left(Y_{ij}^{(\ell-1)}\right) \cdot X_{(i+a-1)(j+b-1)}^{(\ell-1)} \tag{4}$$

Note that $H_x^{(\ell)} = H_y^{(\ell-1)}$ and $W_x^{(\ell)} = W_y^{(\ell-1)}$ (based on (3)). Now as in section 2.1.2, we use this and the learning rate α to adjust the weights of $K^{(\ell)}$ and minimize error.

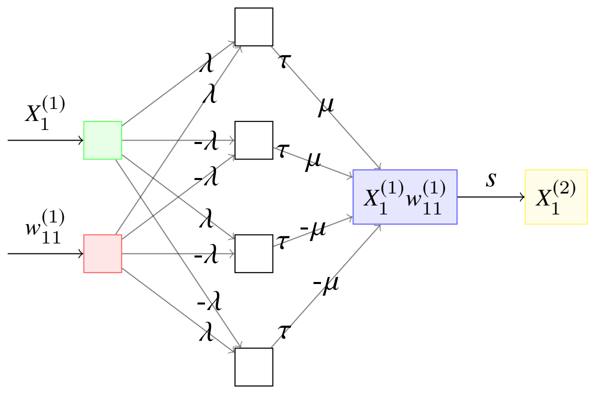


Figure 1: NMA for forward propagation through FCNN with $L = 2$ and $U_1 = U_2 = 1$.

3 Preliminaries

In this section, we recall a theorem from [3] upon which our work is based:

Theorem 1 (Lin, Tegmark, and Rolnick)**.** *Let $\boldsymbol{f}$ be a neural network of the form $\boldsymbol{f} = \mathbf{A}_2\sigma\mathbf{A}_1$, where σ acts elementwise by applying some smooth non-linear function σ to each element. Let the input layer, hidden layer and output layer have sizes 2, 4 and 1, respectively. Then $\boldsymbol{f}$ can approximate a multiplication gate arbitrarily well.*

To avoid confusion, we use τ to denote the nonlinear function referenced in the theorem. Since the approximation of some product uv becomes "arbitrarily accurate as $\lambda \to 0$ [3], we use this MG to compute the product of two given values efficiently, accurately, and without explicit use of the multiplication operation. This MG is diagrammed in Figure 2 of [3].

4 Results, Part 1

In this section we propose and discuss four neural network architectures that are based upon the MG. The purpose of these four architectures is to efficiently implement forward- and back-propagation for both fully connected neural networks and CNNs. As our results are concerned primarily with expediting multiplications, bias terms need not be considered. As an analogy for the feasibility of removing explicit multiplications to increase the efficiency of artificial neural networks, we can look to the human brain. Neurons do not explicitly have the ability to perform the multiplication operation on synaptic signals. However, it is possible for the brain to execute complex mental arithmetic efficiently and in real time by regulating the amount of neurotransmitter released into synapses – in effect, by enhancing ("adding") or supressing ("subtracting") signals. This hints at the possibility of implementing artificial neural networks that rely on only addition and subtraction operations that are more efficient than networks that explicitly use multiplication.

4.1 Fully Connected Neural Networks

4.1.1 Forward Propagation

We first introduce a no-multiplication architecture (NMA) for forward propagation through an FCNN with $L = 2$ layers and then extend the result for $L >= 2$. We begin with the simplest case, in which $L = 2$ and $U_1 = U_2 = 1$. There is only one weight matrix containing a single weight, and only one product, $X_1^{(1)} w_{11}^{(1)}$, must be computed. We apply the MG to this computation in Figure 1, which is nearly identical to the original MG. In layer one of Figure 1, one neuron represents each factor. In layer two, we have four intermediate neurons for each pair of factors we wish to multiply. In layer three, we have a single neuron whose value is the desired product. In layer four, we apply some activation function, if included in the original neural network. The weights between layers one and two and layers two and three are identical to those between the input and hidden layers and the hidden and output layers of the original MG, resp.

Now we similarly construct an NMA for the case of $L = 2$ and $U_1, U_2 > 1$ by applying the MG once for each of the $U_1 U_2$ products that we must compute in multiplying row vector $X^{(1)}$ (dimension U_1) by matrix $W^{(1)}$ (dimensions U_1 by U_2). Since the weights between layers implementing multiplication are identical to those of Figure 1, we do not diagram MG-related weights in architectures for the remainder of this report. In layer one of the NMA, one neuron represents each component of $X^{(1)}$ and $W^{(1)}$. In layer two, we have four intermediate neurons for each product, for a total of $4U_1 U_2$ neurons. In layer three, one neuron represents each product $X_i^{(1)} w_{ab}^{(1)}$, computed using the four intermediate neurons associated with that pair of factors. In layer four, we sum the products and if necessary, apply the activation function, giving neurons representing $X^{(2)}$ in layer five. This NMA can be extended to an NMA for $L >= 2$ (Figure 2). To do this, we extend layer five with neurons representing the components of $W^{(2)}$. Layers five through

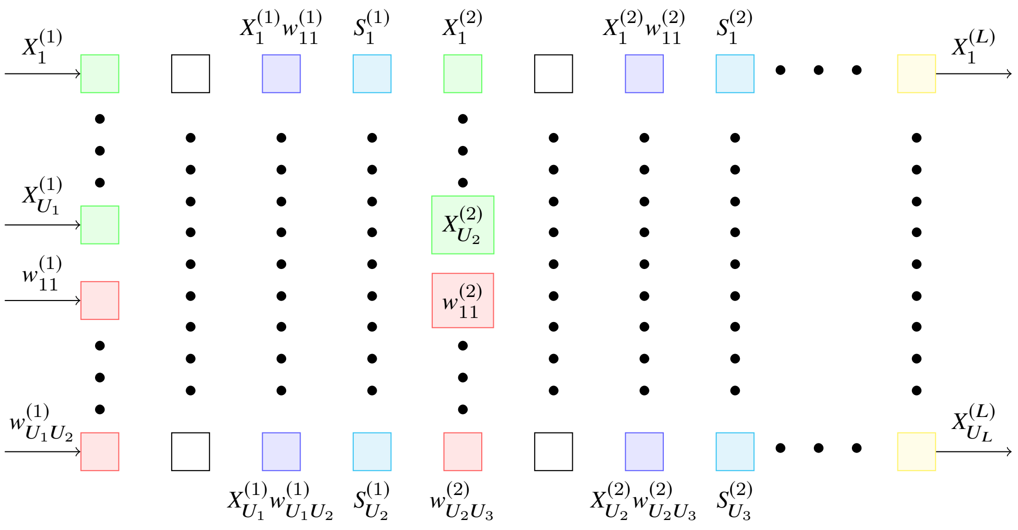


Figure 2: NMA for forward propagation through an FCNN with $L >= 2$

nine represent the computation of $X^{(2)}W^{(2)} = X^{(3)}$ in a similar manner, and this structure repeats for subsequent $X^{(\ell)}$. As each layer after $\ell = 2$ of the original neural network requires the addition of four layers in the NMA, this yields a total of $5 + 4(L-2) = 4L - 3$ layers in the NMA.

4.1.2 Backpropagation

Now we turn to an NMA (Figure 3) that implements backpropagation for the same case ($L >= 2$). This NMA is essentially a continuation of Figure 2 and implements computations for the error backpropagated to each layer (see (1)). We first compute $\delta^{(L-1)} = D^{(L-1)}e$. Layer one contains neurons representing $X^{(L)}$. Layer two contains neurons representing e and the nonzero components of $D^{(L-1)}$ (the zero-valued components can be ignored). Layer three contains four intermediate neurons for each of the U_L products $\left(D_{11}^{(L-1)}e_1, D_{22}^{(L-1)}e_2, \ldots, D_{U_LU_L}^{(L-1)}e_{U_L}\right)$. Layer four contains neurons representing these products, the components of $\delta^{(L-1)}$. Now $\delta^{(L-1)}$ can be used to adjust the weights in $W^{(L-1)}$ (by (2)). We extend the NMA to compute $\delta^{(L-2)} = D^{(L-2)}W^{(L-1)}\delta^{(L-1)}$. We first extend layer four with $U_{L-1}U_L$ additional neurons representing the weights in $W^{(L-1)}$. Layer five contains four intermediate neurons for each of the $U_{L-1}U_L^2$ products in the computation of $W^{(L-1)}\delta^{(L-1)}$. Layer six contains $U_{L-1}U_L^2$ neurons representing these products. Layer seven first contains U_{L-1} neurons that each represents a sum of products totaling to one component of product $W^{(L-1)}\delta^{(L-1)}$. We use these sums to compute $D^{(L-2)}\left(W^{(L-1)}\delta^{(L-1)}\right) = \delta^{(L-2)}$ by extending

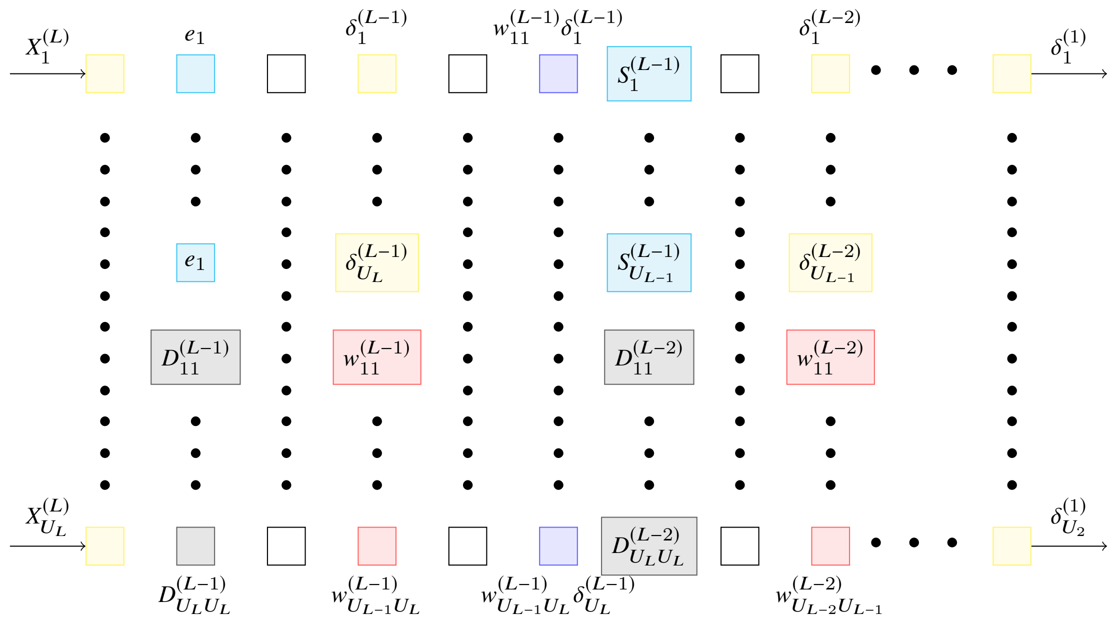


Figure 3: NMA for backpropagation through FCNN with $L >= 2$.

layer seven with U_{L-1} neurons representing the nonzero elements of D^{L-2}. Layer eight contains four intermediate neurons for each of product in the computation. Layer nine contains neurons representing the U_{L-1} components of $\delta^{(L-2)}$. Again, we can now adjust the weights in matrix $W^{(L-2)}$. We then extend the NMA with with five additional layers for the computation of each $\delta^{(\ell)}$, for a total of $4 + 5(L-2) = 5L - 6$ layers. Thus, we can appropriately adjust the weights of the original network without explicit multiplications.

4.2 Convolutional Neural Networks

4.2.1 Forward Propagation

We begin with the general case, a CNN with $L - 1$ conv. layers ($L >= 3$) and a distinct kernel $K^{(\ell)}$ for each layer. We first construct an NMA for the subcase in which all components of an input map $X^{(\ell)}$ are distinct and all weights in a kernel $K^{(\ell)}$ are distinct (Figure 4). Layer one contains $H_x^{(1)}W_x^{(1)}$ neurons representing the components of the input map $X^{(1)}$ and f_1^2 neurons representing the weights in $K^{(1)}$. Layer two contains four intermediate neurons for each product $X_{ij}^{(1)}k_{ab}^{(1)}$. (There are $f_1^2 H_y W_y$ products, as each of the f_1^2 kernel weights is multiplied by $H_y^{(1)}W_y^{(1)}$ components of $X^{(1)}$.) Layer three contains neurons representing these $f_1^2 H_y W_y$ products. We then compute specific

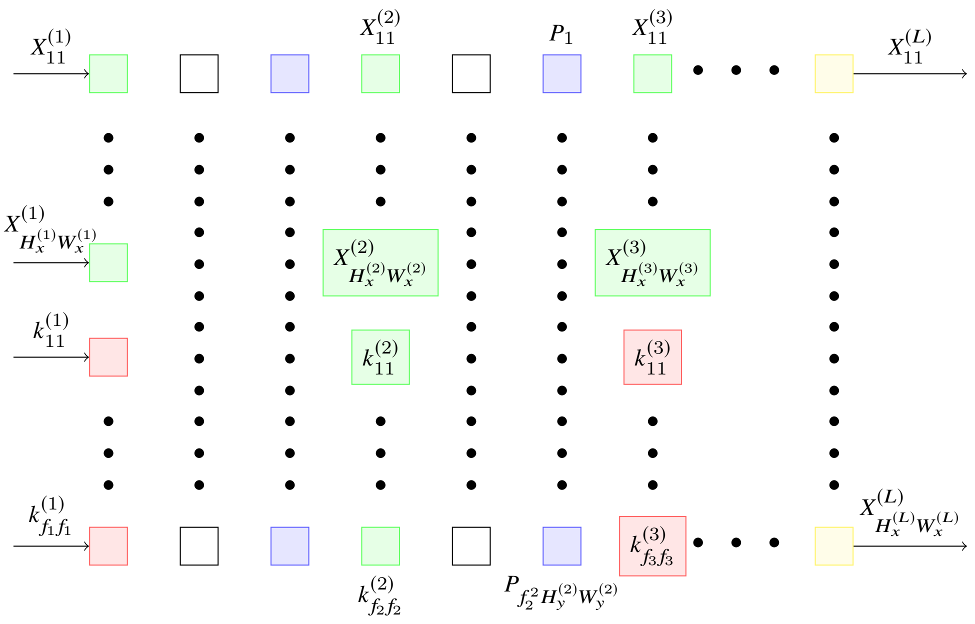


Figure 4: NMA for forward propagation through a CNN with $L-1$ conv. layers and distinct input values.

sums of these products, which yields the values of the output map $Y^{(1)}$ $(Y_{11}^{(1)}, Y_{12}^{(1)}, \ldots, Y_{H_y^{(1)}W_y^{(1)}}^{(1)})$. We also apply the nonlinearity σ at this point, so the neurons in layer four that represent $Y^{(1)}$ also represent $X^{(2)}$. If $L = 1$, then the NMA ends here. Otherwise, we proceed to compute $X^{(2)} \otimes K^{(2)}$. To apply the MG, the $k_{ab}^{(2)}$ must be represented as neurons in the same layer as the $X_{ij}^{(2)}$. Thus, we add f_2^2 neurons representing $K^{(2)}$ to layer four. Layer five contains four intermediate neurons for each of the $f_2^{(2)}H_y^{(2)}W_y^{(2)}$ products in $X^{(2)} \otimes K^{(2)}$. Layer six contains neurons representing these products. Layer seven consists of $H_y^{(2)}W_y^{(2)}$ neurons representing sums of these products. Similar to above, as σ is applied at this point, the layer seven neurons also correspond to the values of $X^{(3)}$. The values for subsequent $X^{(\ell)}$ can be computed by extending layer seven and the NMA itself in the same manner as detailed for $X^{(2)}$. The final output of the NMA is $X^{(L)}$.

We now consider the subcase in which not all components of some $X^{(\ell)}$ are distinct while the weights of $K^{(\ell)}$ are. Here we describe an NMA for a CNN with only one conv. layer ($L = 1$), as it is possible to extend this basic NMA to implement computations for multiple conv. layers. We do not provide a diagram for this NMA, as it differs from Figure 4 only in the number of neurons per layer. Again, layer one contains neurons representing the components of $X^{(1)}$ and the weights of $K^{(1)}$. However, it is insensible to include multiple neurons that represent the same input value, so we can eliminate neurons that are essentially "duplicate copies." Then layer two contains four neurons for each distinct product $X_{ij}^{(1)} k_{ab}^{(1)}$. The number of distinct products is less than $f_1^2 H_y W_y$ and can vary, as discussed in section 5. Thus, the number of neurons in layers two and three decreases. The number of neurons in layer four, however, remains unchanged because the number of components in $Y^{(1)}$ remains unchanged. Thus, the NMA is complete if $L = 1$ and can otherwise be extended.

4.2.2 Backpropagation

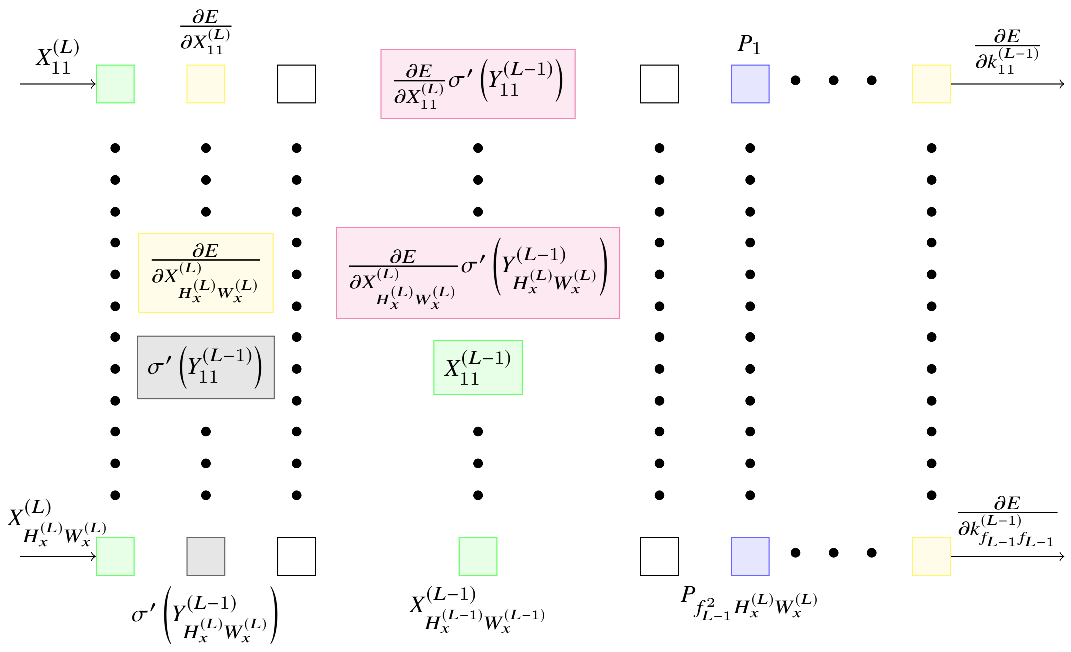
We now implement computations for the gradient of each weight $k_{ab}^{(\ell)}$ in an NMA (Figure 5) that is essentially an extension of Figure 4. We begin with $\ell = L - 1$. While (4) entails computing the product of three values, our MG applies only for multiplication between two values. Thus, we split the product into an expression of the form $(ab)c$:

$$\frac{\partial E}{\partial k_{ab}^{(L-1)}} = \sum_{i=1}^{H_x^{(L)}} \sum_{j=1}^{W_x^{(L)}} \left(\frac{\partial E}{\partial X_{ij}^{(L)}} \cdot \sigma' \left(Y_{ij}^{(L-1)} \right) \right) \cdot X_{(i+a-1)(j+b-1)}^{(L-1)}. \tag{5}$$

We call the computation of ab step one and the computation of $(ab)c$ step two. A bit of algebraic manipulation reveals that for step one, we must compute $H_x^{(L)} W_x^{(L)}$ products:

$$\frac{\partial E}{\partial X_{11}^{(L)}} \sigma' \left(Y_{11}^{(L-1)} \right), \frac{\partial E}{\partial X_{12}^{(L)}} \sigma' \left(Y_{12}^{(L-1)} \right), \ldots, \frac{\partial E}{\partial X_{H_x^{(L)} W_x^{(L)}}^{(L)}} \sigma' \left(Y_{11}^{(L-1)} \right)$$

and that for step two, we must compute the multiplication of each of the above with f_{L-1}^2 components of $X_{ij}^{(L-1)}$, which amounts to $f_{L-1}^2 H_x^{(L)} W_x^{(L)}$ products. Layer one of the NMA contains neurons representing the components of $X^{(L)}$. In layer two, we first compute $\frac{\partial E}{\partial X_{ij}^{(L)}}$ by evaluating some error function to $X^{(L)}$ and representing the results as neurons. We then evaluate the derivative of the nonlinearity σ at each $Y_{ij}^{(L-1)}$ (whose values are previously stored during forward propagation) and represent the resulting $\sigma' \left(Y_{ij}^{(L-1)} \right)$ as $H_x^{(L)} W_x^{(L)}$ additional neurons in layer two. Layer three contains four intermediate neurons for each product. Layer four contains neurons representing the $H_x^{(L)} W_x^{(L)}$ products, and step one is complete. We proceed with step two by representing the components of $X^{(L-1)}$ as additional neurons in layer four. Layer five contains four intermediate neurons for each of the $f_{L-1}^2 H_x^{(L)} W_x^{(L)}$ products in step two. Layer six contains one neuron per



$X_{11}^{(L)}$
$\frac{\partial E}{\partial X_{11}^{(L)}}$
$\frac{\partial E}{\partial X_{11}^{(L)}}\sigma'\left(Y_{11}^{(L-1)}\right)$
P_1
$\frac{\partial E}{\partial k_{11}^{(L-1)}}$
$\frac{\partial E}{\partial X^{(L)}_{H_x^{(L)}W_x^{(L)}}}$
$\frac{\partial E}{\partial X^{(L)}_{H_x^{(L)}W_x^{(L)}}}\sigma'\left(Y^{(L-1)}_{H_x^{(L)}W_x^{(L)}}\right)$
$\sigma'\left(Y_{11}^{(L-1)}\right)$
$X_{11}^{(L-1)}$
$X^{(L)}_{H_x^{(L)}W_x^{(L)}}$
$\frac{\partial E}{\partial k^{(L-1)}_{f_{L-1}f_{L-1}}}$
$\sigma'\left(Y^{(L-1)}_{H_x^{(L)}W_x^{(L)}}\right)$
$X^{(L-1)}_{H_x^{(L-1)}W_x^{(L-1)}}$
$P_{f_{L-1}^2 H_x^{(L)}W_x^{(L)}}$

Figure 5: NMA for backpropagation through a CNN with $L-1$ conv. layers.

product. Finally, layer seven contains one neuron for each gradient $\frac{\partial E}{\partial k_{ab}^{(L-1)}}$, computed as the sum of $H_x^{(L)}W_x^{(L)}$ specific products from layer six. We can use these gradients to adjust the kernel weights (by (4)). Unlike in section 4.1.2, we do not extend this NMA to compute the gradients for weights in previous kernels. Instead, for each $K^{(\ell)}$ we create a separate NMA similar to that in Figure 5.

5 Results, Part 2

We now present our findings with regard to the number of distinct multiplications that must be computed for several cases. In the following proofs, for an input map (*image*) X with dimensions H_x and W_x, we refer to each distinct component of X as a *pixel value*. For an image X convolved with kernel K, we denote the number of repeated multiplications by R and denote the dimensions of the output map as H_y and W_y. We denote the dimensions of kernel K as $f \times f$.

5.1 Forward Propagation

Theorem 2. *The number of distinct multiplications when an image X is convolved with kernel K with distinct values is*

$$f^2 H_y W_y - R, \text{ with } R = \sum_{n \in N} \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i P_x$$

where N is the set of distinct pixel values in X, t_n is the number of pixels with value n, T_i is the set of i-tuples of pixels with value n, and P_x is the number of kernel weights that are multiplied by all of the elements of tuple $x \in T_i$ in convolution.

Proof. The number of multiplications performed in a conv. layer, assuming that all values in the kernel and the image are distinct, is $f^2 H_y W_y$. We wish to find the number of multiplications that are repeated when there are repeated values in the image. Define the *field* of a kernel unit K_{ab} to be the set of pixels in image X by which K_{ab} is multiplied in convolution: all X_{ij} such that $a \leq i \leq a + H_x - f$ and $b \leq j \leq b + W_x + f$. For a pixel value n, let $u_{n,ab}$ be the number of pixels with value n in the field of K_{ab}. For all of the pixels with value n in the field of K_{ab}, the product $n \times K_{ab}$ is identical, so there are $u_{n,ab} - 1$ repeated multiplications for the field of every K_{ab}. We observe that

$$\begin{aligned} u - 1 = -1 + u + (1-1)^u &= -1 + u + \left[\binom{u}{0} - \binom{u}{1} + \binom{u}{2} - \ldots + (-1)^u \binom{u}{u} \right] \\ &= \binom{u}{2} - \ldots + (-1)^u \binom{u}{u} = \sum_{i=2}^{u} (-1)^i \binom{u}{i} \end{aligned}$$

The number of repeated multiplications R is the sum of $u_{n,ab} - 1$ for every field. If N is the set of all distinct pixel values in the image and F is the set of all fields, then

$$R = \sum_{n \in N} \sum_{x \in F} (u_{n,x} - 1) = \sum_{n \in N} \sum_{x \in F} \sum_{i=2}^{u_{n,x}} (-1)^i \binom{u_{n,x}}{i}.$$

For every field x and pixel value n, $u_{n,x} \leq t_n$. Thus, if $u_{n,x} = t_n$,

$$\sum_{i=2}^{u_{n,x}} (-1)^i \binom{u_{n,x}}{i} = \sum_{i=2}^{t_n} (-1)^i \binom{u_{n,x}}{i}.$$

If $u_{n,x} < t_n$,

$$\sum_{i=2}^{u_{n,x}} (-1)^i \binom{u_{n,x}}{i} = \sum_{i=2}^{u_{n,x}} (-1)^i \binom{u_{n,x}}{i} + \sum_{i=u_{n,x}+1}^{t_n} (-1)^i \binom{u_{n,x}}{i} = \sum_{i=2}^{t_n} (-1)^i \binom{u_{n,x}}{i}.$$

Thus, R can be rewritten as

$$R = \sum_{n \in N} \sum_{i=2}^{t_n} \sum_{x \in F} (-1)^i \binom{u_{n,x}}{i}$$

We now use Lemma 1, which is proven separately from this theorem. For all $n \in N$ and $2 \leq i \leq U_n$,

$$\sum_{x \in F} \binom{u_{n,x}}{i} = \sum_{y \in T_i} P_y$$

where T_i is the set of i-tuples of pixels with value n and P_y is the number of kernels that are multiplied by all of the elements of the i-tuple y. Thus, we can rewrite R as

$$R = \sum_{n \in N} \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i P_x$$

The number of of distinct multiplications is the number of multiplications when there are no repeated pixel values minus the number of repeated multiplications where pixel values are repeated: $f^2 H_y W_y - R$. $\square$

Lemma 1. *For all $n \in N$ and $2 \leq i \leq U_n$: $\sum_{x \in F} \binom{u_{n,x}}{i} = \sum_{y \in T_i} P_y$.*
T_i is the set of i-tuples of pixels with value n and P_y is the number of kernels that are multiplied by all of the elements of the i-tuple y.

Proof. We define Q_{yx} as the number of times tuple y appears in field x. Since the positions of pixels are unique, tuple y appears at most once: $Q_{yx} \in \{0,1\}$. We note that $P_y = \sum_{x\in F} Q_{yx}$.

$$\begin{aligned}\sum_{y\in T_i} P_y &= \sum_{y\in T_i}\sum_{x\in F} Q_{yx} = \sum_{x\in F}\sum_{y\in T_i} Q_{yx} = \sum_{x\in F}\left(\sum_{y\in T_i\cap x} Q_{yx} + \sum_{y\in T_i\setminus x} Q_{yx}\right)\\ &= \sum_{x\in F}\sum_{y\in T_i\cap x} 1 = \sum_{x\in F} |T_i \cap x| = \sum_{x\in F}\binom{u_{n,x}}{i}\end{aligned}$$

□

We now derive an approximation of the number of distinct products for another case.

Theorem 3. *The the number of distinct multiplications to compute in the convolution of an image X whose pixel values are all distinct with a kernel K containing repeated weight values is*

$$f^2 H_x W_x - R, \text{ with } R = \sum_{n\in N}\sum_{i=2}^{t_n}\sum_{x\in T_i}(-1)^i P_x$$

where P_x is the number of pixels that lie in fields of all of the kernel weights in a tuple $x \in T_i$. Additionally, if $H_x \geq 2f-1$ and $W_x \geq 2f-1$, then

$$\sum_{n\in N} H_y W_y (t_n - 1) \leq R \leq \sum_{n\in N}(H_y W_y (t_n-1) + (f-1)^2(t_n-1))$$

Proof. If all kernel weights are distinct, then the number of distinct products is $f^2 H_x W_x$. We wish to count the number of products that are repeated in convolution when there are repeated values in the kernel. The expression R can be derived with a similar proof to the proof for repeated pixel values. We now wish to find an expression for P_x, the number of pixels in the intersection of the fields of all of the kernels. The field of every kernel weight K_{ab} is all X_{ij} such that

$$a \leq i \leq a + H_x - f = a + H_y - 1 \quad \text{and} \quad b \leq j \leq b + W_x - f = b + W_y - 1.$$

For a tuple x of kernel weights K_{ab}, let $i_{overlap}$ represent a value of i within all of the fields of the weights in x, and let $j_{overlap}$ be a value of j within all of the fields. Thus,

$$\max a \leq i_{overlap} \leq \min a + H_y - 1 \quad \text{and} \quad \max b \leq j_{overlap} \leq \min b + W_y - 1$$

Note that the fields overlap if and only if $\max a - \min a \leq H_y - 1$ or $\max b - \min b \leq W_y - 1$. If $H_x \geq 2f - 1$ and $W_x \geq 2f - 1$, then for any tuple of kernel weights, $\max a - \min a \leq f - 1 \leq H_x - f = H_y - 1$ and $\max b - \min b \leq f - 1 \leq W_x - f = W_y - 1$. Let $Rng(a) = \max a - \min a$ and $Rng(b) = \max b - \min b$. The number of values of i in the overlapping region is $H_y - Rng(a)$, and the number of values of j in the overlap is $W_y - Rng(b)$. Thus, the number of pixels in the overlapping region is

$$P_x = (H_y - Rng(a))(W_y - Rng(b)), \quad \text{with} \quad Rng(a) \leq H_y - 1,\ Rng(b) \leq W_y - 1$$

Substituting this expression for P_x into the expression for R, we obtain:

$$\begin{aligned} R &= \sum_{n \in N} \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i (H_y - Rng(a))((W_y - Rng(b)) \\ &= \sum_{n \in N} \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i (H_y W_y - H_y Rng(b) - W_y Rng(a) + Rng(a)Rng(b)) \end{aligned}$$

We consider four separate summations. For a given kernel weight value n,

$$r_1 = \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i H_y W_y \qquad r_3 = \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i Rng(a)$$

$$r_2 = \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i Rng(b) \qquad r_4 = \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i Rng(a)Rng(b)$$

Note that $R = \sum_{n \in N} (r_1 - H_y r_2 - W_y r_3 + r_4)$.

$$r_1 = H_y W_y \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i = H_y W_y \sum_{i=2}^{t_n} \binom{t_n}{i} = H_y W_y (t_n - 1)$$

In considering r_2, we label the b coordinates of kernel weights with the same value: $b_1 \leq b_2 \leq$

$\ldots \leq b_k \leq \ldots \leq b_{t_n}$. Additionally,

$$r_2 = \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i \max b - \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i \min b$$

We consider the summation involving $\max b$. For each b_k, we find number of tuples for which b_k is the maximum and substitute into the summation. Every b_k has $k-1$ other coordinates less than or equal to it, and the number of i-tuples with maximum b_k is the number of ways to choose $i-1$ coordinates from the $k-1$ coordinates less than or equal to b_k. Thus, the coefficient on b_k in the first summation is $\binom{k-1}{1} - \binom{k-1}{2} + \ldots + (-1)^{t_n}\binom{k-1}{t_n-1}$. Since $k-1 \leq t_n - 1$, the coefficient simplifies to

$$\binom{k-1}{1} - \binom{k-1}{2} + \ldots + (-1)^k \binom{k-1}{k-1} = 1 - (1-1)^{k-1} = 1$$

Now we consider the summation involving $\min b$. For each b_k, there are $t_n - k$ other coordinates greater than or equal to b_k, so the coefficient on b_k in the second summation is:

$$\begin{aligned}\binom{t_n-k}{1} - \binom{t_n-k}{2} + \ldots + (-1)^{t_n}\binom{t_n-k}{t_n} &= \binom{t_n-k}{1} - \ldots + (-1)^{t_n-k+1}\binom{t_n-k}{t_n-k} \\ &= 1 - (1-1)^{t_n-k} = 1\end{aligned}$$

Thus, since for every b_k the coefficients from the first and summation are both 1, when the summations are subtracted, every b_k cancels out: $r_2 = \sum_{k=1}^{t_n} b_k - \sum_{k=1}^{t_n} b_k = 0$. Thus, $r_2 = 0$, and by a similar argument, $r_3 = 0$. r_4 is dependent on the arrangement of repeated values within the kernel. However, we can derive a range of possible of r_4. Since $0 \leq Rng(a) \leq f-1$ and $0 \leq Rng(b) \leq f-1$,

$$\begin{aligned}0 &\leq \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i Rng(a) Rng(b) \\ &\leq \sum_{i=2}^{t_n} \sum_{x \in T_i} (-1)^i (f-1)^2 = (f-1)^2 \sum_{i=2}^{t_n} \binom{t_n}{i} \\ &= (f-1)^2 (t_n - 1 + (1-1)^{t_n}) = (f-1)^2 (t_n - 1)\end{aligned}$$

Substituting r_1, r_2, r_3, and r_4 back into the expression for R, we obtain:

$$\sum_{n\in N} H_y W_y(t_n - 1) \leq R \leq \sum_{n\in N}(H_y W_y(t_n - 1) + (f-1)^2(t_n - 1))$$

$\square$

5.2 Backpropagation

We find the number of distinct multiplications in calculating the partial derivative given by (4). Let V_L be the number of distinct outputs in layer $X^{(\ell)}$. We first multiply $\frac{\partial E}{\partial X_{ij}^{(\ell)}} \cdot \sigma'\left(Y_{ij}^{(\ell-1)}\right)$. If we define $E = \frac{1}{2}(t_{ij} - x_{ij}^{(\ell)})^2$, then $\frac{\partial E}{\partial X_{ij}^{(\ell)}} = x_{ij}^{(\ell)} - t_{ij}$. Thus, $\frac{\partial E}{\partial X_{ij}^{(\ell)}} = \frac{\partial E}{\partial X_{kl}^{(\ell)}}$ if and only if $x_{ij}^{(\ell)} = x_{kl}^{(\ell)}$. Since σ is one-to-one, if $x_{ij}^{(\ell)} = x_{kl}^{(\ell)}$, then $y_{ij}^{(\ell-1)} = y_{kl}^{(\ell-1)}$, so $\sigma'\left(Y_{ij}^{(\ell-1)}\right) = \sigma'\left(Y_{kl}^{(\ell-1)}\right)$. Thus, the number of distinct products is equal to the number of distinct values of $x_{ij}^{(\ell)}$; that is, V_L. We then multiply each value of $\frac{\partial E}{\partial X_{ij}^{(\ell)}} \cdot \sigma'\left(Y_{ij}^{(\ell-1)}\right)$ by $X_{(i+a-1)(j+b-1)}^{(\ell-1)}$. This step can be thought of as a convolution with kernel K such that $K_{ij} = \frac{\partial E}{\partial X_{ij}^{(\ell)}} \cdot \sigma'\left(Y_{ij}^{(\ell-1)}\right)$. and the image $X^{(\ell-1)}$, resulting in a matrix of partial derivatives A with dimensions $f^{(\ell-1)} \times f^{(\ell-1)}$ such that $A_{ij} = \frac{\partial E}{\partial k_{ij}^{(L-1)}}$. Thus, $A = X^{(\ell-1)} \otimes K$ and

$$A = \begin{pmatrix} \frac{\partial E}{\partial k_{11}^{(L-1)}} & \cdots & \frac{\partial E}{\partial k_{1f^{(\ell-1)}}^{(L-1)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial k_{f^{(\ell-1)}1}^{(L-1)}} & \cdots & \frac{\partial E}{\partial k_{f^{(\ell-1)}f^{(\ell-1)}}^{(L-1)}} \end{pmatrix}$$

Note that values of K are identical if and only if the corresponding values of $X_{ij}^{(\ell)}$ are the same. Thus, the number of distinct multiplications, assuming no overlap between the values in K and the values in $X^{(\ell-1)}$ and the values in $X^{(\ell)}$, is equal to the number of distinct multiplications performed in the convolution $X^{(\ell-1)} \otimes X^{(\ell)}$.

Theorem 4. *The number of distinct products that must be computed for backpropagation is*

$$f^2 H_x^{(\ell-1)} W_x^{(\ell-1)} - R \quad \textit{with} \quad R = \sum_{n\in N}\sum_{i=2}^{t_n}\sum_{x\in T_i}(-1)^i P_x.$$

When there are repeated pixel values in $X^{(\ell-1)}$ but not in $X^{(\ell)}$, N represents the set of distinct pixel values in $X^{(\ell-1)}$, t_n represents the number of pixels with value n in $X^{(\ell-1)}$, T_i represents the set of all i-tuples of pixels of value n, and P_x represents the number of pixels of $X^{(\ell)}$ that cover all pixels in the tuple $x \in T_i$. When there are repeated pixel values in $X^{(\ell)}$ but not in $X^{(\ell-1)}$, N represents the set of distinct pixel values in $X^{(\ell)}$, t_n represents the number of pixels with value n in $X^{(\ell)}$, T_i represents the set of all i-tuples of pixels of value n, and P_x represents the number of pixels that lie on the fields of all of the pixels in the tuple $x \in T_i$. The field of a pixel $X^{(\ell)}_{ab}$ is all $X^{(\ell-1)}_{ij}$ such that $a \le i \le a + H^{(\ell-1)}_x - H^{(\ell)}_x = a + f - 1$ and $b \le j \le b + W^{(\ell-1)}_x - W^{(\ell)}_x = b + f - 1$.

6 Discussion

The implications of our results are significant in comparison to those of existing approaches. First, since multiplication of floating point numbers comprises a majority of training time [6], our NMAs are able to accelerate model training. These benefits are magnified when we turn to CNNs. Second, the elimination of explicit multiplications reduces the need for costly GPU hardware. Considering these, our NMAs raise the possibility of training neural networks in real time, using only additions and subtractions, on both CPUs and simpler devices. We note that our approach can be used to expedite computations in existing architectures such as AlexNet [2].

Limited research on the simplification of computations during network training has been published. One recent approach is the elimination of explicit multiplications through binarization of weights during forward propagation and conversion of "multiplications into bit-shifts" [4] during backpropagation. The drawback of this work is that it is predicated on efficient stochastic random number generation. Another approach distills multiplications into simple binary shifts by quantizing weight values to only integer powers of two. However, any benefits of doing so are eclipsed by both the loss of certainty with regard to the convergence of model training and the often poor performance of resulting models [4]. Another approach relies on a Boolean network architecture that reduces computation time for model testing and produces models of acceptable performance levels. However, the impacts of this strategy are much smaller in scope, as it does not serve to lessen training time [4]. Other approaches try to decrease training complexity. One example restricts learning rates and gradients to powers of two to the effect of eliminating explicit multiplications [4].

From these approaches, it is evident that model performance is limited by any reduction in the scope of network values (such as weight values, learning rates, gradients). As our approach implements an alternate architecture for model training and does not rely on such reductions, and has the potential benefit of increasing training efficiency without compromising model performance.

7 Conclusions

In this research we explore and answer two key questions. First, is it possible to expedite computations during neural network training without explicit multiplication operations? Second, for a given CNN, how many distinct multiplications must be computed for a single convolutional layer (given that the component values of the input map, but not of the kernel, might be repeated)? We answer the first question by devising several NMAs that serve as alternate architectures for model training. Specifically, we implement NMAs for both forward- and back-propagation for fully connected and convolutional neural networks. We answer the second question by deriving an expression for the number of distinct multiplications. With this result, we show for CNNs that our approach allows for a potentially substantial reduction in computation time. This conclusion is validated by findings from [6]. Ultimately, our approach opens the door for decreased training time and increased computation efficiency.

With more time, we could numerically evaluate the use of NMAs to implement examples of fully connected networks and CNNs and observe the extent to which training time is reduced. We could also proceed with our goals for future work. One such goal entails investigating the number of distinct multiplications that must be computed for CNN given that component values of both input maps and kernels exhibit repeated patterns. A second goal entails expanding upon this area of research by using Taylor expansions to derive an expression for the Softmax function that can be implemented as an architecture analogous to the MG architecture for multiplication [2, 3].

References

[1] Rojas, Raul. “The Backpropagation Algorithm.” *Neural Networks: A Systematic Introduction.* Berlin: Springer-Verlag, 1996. 151–171.
https://page.mi.fu-berlin.de/rojas/neural/neuron.pdf

[2] “Convolutional Neural Networks (CNNs/ConvNets).” *CS231n: Convolutional Neural Networks for Visual Recognition.* Stanford University, 2017.
https://cs231n.github.io/convolutional-networks/

[3] Lin, Henry W., Max Tegmark, and David Rolnick. “Why does deep and cheap learning work so well?” *arXiv preprint arXiv:1608.08225v4 [cond-mat.dis-nn],* 2017.
https://arxiv.org/pdf/1608.08225v4.pdf

[4] Lin, Zhouhan, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. “Neural Networks with Few Multiplications.” *arXiv preprint arXiv:1510.03009v3 [cs.LG],* 2015.
https://arxiv.org/pdf/1510.03009.pdf

[5] Marchesi, Michele, Gianni Orlandi, Francesco Piazza, and Aurelio Uncini. “Fast neural networks without multipliers.” *IEEE Transactions on Neural Networks,* 4(1):53–62, 1993.
https://www.academia.edu/32961416/Fast_neural_networks_without_multipliers

[6] Vanhoucke, Vincent, Andrew Senior, and Mark Z. Mao. “Improving the speed of neural networks on CPUs.” *Deep Learning and Unsupervised Feature Learning NIPS Workshop,* 2011.
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37631.pdf

[7] Simard, Patrice Y. and Hans Peter Graf. “Backpropagation without Multiplication.” *Advances in Neural Information and Processing Systems*, pp.232–239, 1994.
https://papers.nips.cc/paper/833-backpropagation-without-multiplication.pdf