Backpropagation Techniques for Tensor Operations: A Case Study in Tensor Contraction Layers

ANONYMOUS AUTHOR(S)**

The purpose of this work is to provide a comprehensive guide meant to help the reader understand and implement tensor operation backpropagation with a focus on manually deriving and coding complex tensor expressions. The approach begins with a ground-up introduction to fundamental concepts of backpropagation which helps build the reader's intuition. A detailed case study of a Tensor Contraction Layer is then explored, and the backpropagation process is demonstrated by using the backpropagation building blocks to derive abstract tensor expressions. The training of a Tensor Contraction Layer was programmed in C++ and can be accessed using GitHub. This discussion includes a step-by-step walk-through of the critical components to give the reader an in-depth understanding of how tensor equations can be programmed in practice. Practical tips for testing and debugging backpropagation implementations are present, with an emphasis on greedy layer-wise training, importance of tensor shape analysis, and tensor notation inspection.

Additional Key Words and Phrases: Educational/Learning, Machine Learning, Design Methods, Artifact or System

ACM Reference Format:

Anonymous Author(s). 2024. Backpropagation Techniques for Tensor Operations: A Case Study in Tensor Contraction Layers. In *Proceedings of CHI Conference on Human Factors in Computing Systems 2025 (CHI 2025).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Backpropagation [24] is a cornerstone of machine learning neural networks and software libraries such as PyTorch [22] and TensorFlow [1] will implement backpropagation without the programmer understanding how it works. This lack of understanding can make debugging neural networks difficult and experimentation becomes unfeasible. Section 2 will briefly explain the intuition behind backpropagation and how it can be used to decrease the loss of a neural network to achieve its overall goal. This section also shows how element-wise operations can be expressed more abstractly to make writing these formulas in software easier, as built-in functions can be utilized.

Next, we present a challenging example of how a Tensor Contraction Layer [4, 14, 15] can be backpropagated in a low level language (C++) with a corresponding repository that can be accessed and experimentally tinkered. Section 3 explains the concept of a Tensor Contraction Layer (TCL) and its purpose in neural networks. The forward and backward propagation equations are then presented from the original work. The goal of the given equations will be to both update the parameters used in the layer as well as to backpropagate loss further back into the network in order to improve additional parameters. The equations used to update the layer parameters are especially difficult at first glance, so a walk-through of that section of code is given to provide further understanding of how these expressions were converted into code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

Finally, in Section 4 we offer practical advice on how to backpropagate mathematical expressions by hand and how to approach programming them. Advice includes how to narrow down the location of a bug and potential causes of errors when programming these expressions.

2 Background

2.1 Backpropagation Fundamentals

Backpropagation [24] is a well-known technique to incrementally improve neural networks by slowly adjusting parameters toward their most ideal values in order to minimize a certain loss function [7, 10, 17, 23]. An example of backpropagation through an addition operation can be seen in Figure 1. The blue values on top of each connection represent the forward propagation values and the red values on the bottom of each connection represent the backward propagation values.

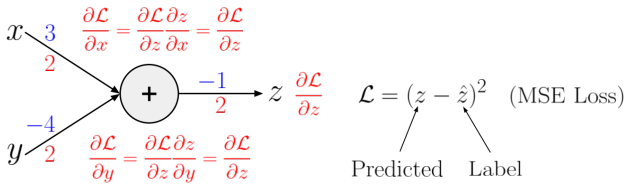

Fig. 1. Backpropagation behavior for addition gate. Loss values propagating backward are unaffected.

In this case, the partial derivative of loss with respect to z is equal to 2. This can be expressed mathematically as:

$$\frac{\partial \mathcal{L}}{\partial z} = 2$$

Where $\mathcal{L}$ represents the loss function the neural network is aiming to minimize. A positive partial derivative of loss with respect to a variable z indicates the value of z during forward propagation is too high and the network should adjust its parameters to decrease this value. This means that both values of x and y should be decreased, given that z is the sum of x and y. Thus, x and y are also backpropagated as positive values. More formally:

$$z = x + y \Rightarrow \frac{\partial \mathcal{L}}{\partial x}, \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial z} \quad \text{because} \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} = 1$$

by application of the chain rule [26].

The behavior of backpropagating multiplication is shown in Figure 2. The backpropagated value of the top connection is the product of the incoming backpropagation value multiplied by the forward propagation value of the bottom connection. The reverse is also true. This can be expressed mathematically as:

$$z = xy \Rightarrow \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial x} = \frac{\partial \mathcal{L}}{\partial z}y \tag{1}$$

A similar expression can be derived for the partial derivative with respect to y.

2.2 Backpropagation Abstraction

These simple building blocks for backpropagation serve as the foundation for constructing more complex expressions which can be represented by abstract mathematical definitions. An example of this is the inner product of a matrix and a vector, which can describe a fully connected layer in a neural network. Given vector $\mathbf{x}$ and matrix $\mathbf{W}$, the inner

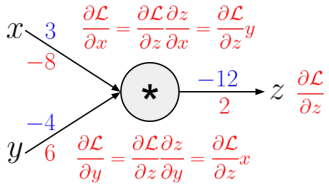

Fig. 2. Backpropagation behavior for multiplication gate. Loss values propagating backward are multiplied by the forward propagation values of opposite connection.

product between these two variables can be expressed as shown in the following equation.

$$y_i = \sum_j W_{ij} x_j \tag{2}$$

To determine the loss with respect to the input vector $\mathbf{x}$ and weight matrix $\mathbf{W}$, it is necessary to find the partial derivatives of the output with respect to these inputs. The partial derivative of output element y_i with respect to an individual input element of x_j can be expressed with the following expression by application of Equation 1:

$$\frac{\partial y_i}{\partial x_j} = W_{ij} \tag{3}$$

Given the partial derivative of loss with respect to $\mathbf{y}$, the partial derivative of loss with respect to the input vector $\mathbf{x}$ can be found by application of the chain rule:

$$\frac{\partial \mathcal{L}}{\partial x_j} = \sum_i \frac{\partial \mathcal{L}}{\partial y_i} \frac{\partial y_i}{\partial x_j} = \sum_i \frac{\partial \mathcal{L}}{\partial y_i} W_{ij} \tag{4}$$

which evaluates to the dot product with respect to the loss propagating backwards and the weight matrix used in the forward propagation:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \mathbf{W}^T \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \tag{5}$$

Using similar logic to find the partial derivative with respect to the weight matrix W is not as straightforward. A direct application of taking the partial derivative would result in the following expression:

$$\frac{\partial y_i}{\partial W_{ij}} = x_j \tag{6}$$

The index i is present in both the indices for $\mathbf{y}$ and $\mathbf{W}$ on the left-hand side. This is mathematically ambiguous and can imply an unintended summation on the left hand side. Instead of being undefined, it is preferable to express that the partial derivative is 0 when the first index of $\mathbf{W}$ is not equal to the index of $\mathbf{y}$. To properly express the partial derivative, multiply the right-hand side of Equation 2 with the identity matrix, leaving the value of the left-hand side unchanged:

$$y_i = \sum_k \sum_j \mathbf{I}_{ik} W_{kj} x_j \tag{7}$$

The identity matrix $\mathbf{I}_{ik}$ is traditionally 1 if $i = k$, and 0 if otherwise. Taking the partial derivative with the elements of $\mathbf{W}$ now gives:

$$\frac{\partial y_i}{\partial W_{kj}} = \mathbf{I}_{ik} x_j \tag{8}$$

And finally, solving for the partial derivative of loss with respect to $\mathbf{W}$ results in:

$$\frac{\partial \mathcal{L}}{\partial W_{kj}} = \sum_i \frac{\partial \mathcal{L}}{\partial y_i} \frac{\partial y_i}{\partial W_{kj}} = \frac{\partial \mathcal{L}}{\partial y_i} \mathbf{I}_{ik} x_j = \frac{\partial \mathcal{L}}{\partial y_k} x_j \tag{9}$$

thus:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \otimes \mathbf{x} \tag{10}$$

In this example, ignoring the ambiguity in Equation 6 would have resulted in the same answer, but that is not always the case for more complicated expressions. The loss with respect to the input vector and input matrix facilitates further backpropagation to preceding layers of the machine learning model, enabling refinement of the parameters to ensure a reduction in future loss.

2.3 Tensor Folding

Tensor folding [11] is used in both the forward and backward propagation of the Tensor Contraction Layer case study shown in the next section. Tensor n-mode folding is a method of representing a tensor of shape $(I_1, I_2, ..., I_N)$ as a matrix of shape $(I_n, I_1 \times ... \times I_{n-1} \times I_{n+1} \times ... \times I_N)$. As described by [13], the element $(i_1, i_2, ..., i_N)$ of the original tensor will be mapped to element (i_n, j) of the n-mode folded matrix where j is given by Equation 11.

$$j = \sum_{\substack{k=1 \\ k \neq n}}^{N} \left[i_k \times \prod_{\substack{m=1 \\ m \neq n}}^{k-1} I_m \right] \tag{11}$$

This can be more clearly seen with an example. Suppose an order 3 tensor of shape (3,2,2) where slices k = 1, 2 are given:

$$\mathcal{X}(i,j,1) = \begin{bmatrix} 0 & 3 \\ 1 & 4 \\ 2 & 5 \end{bmatrix} \quad \mathcal{X}(i,j,2) = \begin{bmatrix} 6 & 9 \\ 7 & 10 \\ 8 & 11 \end{bmatrix}$$

Folding with respect to the first dimension (0-mode fold) would result in the matrix shown below.

$$X_{(0)} = \begin{bmatrix} 0 & 3 & 6 & 9 \\ 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \end{bmatrix}$$

Folding with respect to the second dimension (1-mode fold) would result in the matrix shown below.

$$X_{(1)} = \begin{bmatrix} 0 & 1 & 2 & 6 & 7 & 8 \\ 3 & 4 & 5 & 9 & 10 & 11 \end{bmatrix}$$

Folding with respect to the third dimension (2-mode fold) would result in the matrix shown below.

$$X_{(2)} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 \end{bmatrix}$$

To further compress a matrix, vectorization can be used to stack the elements of a matrix column by column. The vectorization of matrix $\mathbf{A}$ will be denoted by vec($\mathbf{A}$).

2.4 Element-wise Formulas of Abstract Tensor Expressions

Abstract tensor expressions can be broken up into element-wise formulas which allow for deeper analysis. Examples of common tensor products are shown below to reference in further sections. The inner (dot) product [5] between two matrices, abstractly written as $\mathbf{A} \cdot \mathbf{B}$, expressed using individual elements evaluates to:

$$C_{i,j} = \sum_k A_{i,k} B_{k,j} \tag{12}$$

The outer product [5] between two matrices, abstractly written as $\mathbf{A} \otimes \mathbf{B}$, expressed using individual elements evaluates to:

$$C_{i,j,k,l} = A_{i,j} B_{k,l} \tag{13}$$

The Hadamard (element-wise) product [6] between two matrices, abstractly written as $\mathbf{A} \circ \mathbf{B}$, expressed using individual elements evaluates to:

$$C_{i,j} = A_{i,j} B_{i,j} \tag{14}$$

The Kronecker product [8] between two matrices, abstractly written as $\mathbf{A} \otimes \mathbf{B}$, expressed using individual elements evaluates to:

$$C_{i,j} = A_{i \div I, j \div J} B_{i\%I, j\%J} \tag{15}$$

where $\mathbf{B}$ has shape (I, J).

The Kronecker product is pictorially defined as:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{11} \circ B & A_{12} \circ B \\ A_{21} \circ B & A_{22} \circ B \end{bmatrix} \tag{16}$$

Note that the symbol for outer product and Kronecker product are identical. In proceeding sections of this paper, we will strictly use $\otimes$ to denote the Kronecker Product. Two matrices adjacent to each other, such as AB, denote an inner product.

2.5 Einstein Notation

Einstein notation [25], also known as the Einstein summation convention, is a concise and powerful notation used primarily in physics and tensor calculus to simplify expressions involving sums over indices. When an index variable appears twice in a single term, it implies summation over all possible values of that index. Using Einstein notation, Equation 12 could be written as:

$$C_{i,j} = A_{i,k} B_{k,j} \tag{17}$$

where there is a implicit summation over index k. This notation may be used in future sections to simply expressions.

3 Case Study: Tensor Decomposition

To demonstrate the process of backpropagating complex tensor expressions, the example of the Tensor Contraction Layer proposed by Giuseppe et al. in their work titled "Compression and Interpretability of Deep Neural Networks via Tucker Tensor Layer: From First Principles to Tensor Valued Back-Propagation" [4] can be used. While this layer is relatively straightforward during the forward pass, backpropagation requires evaluating and coding of a range of mathematical expressions. To demonstrate these backpropagation equations, a Convolutional Neural Network with a Tensor Contraction Layer was programmed in C++ to illustrate how the backpropagation can be implemented using a low

level language to perform the mathematical calculations by hand. All libraries used in this implementation are either built-in or open source. Eigen [9] is an open-source linear algebra library created by Mozilla which is used to represent matrices and vectors throughout the program. The code can be found on GitHub at *github.com/jitanonymous/TCL_CPP*, and it is advised that the reader look through it to get a stronger understanding of how tensor operations are backpropagated. There is step by step documentation to aid the readers ability to relate the program to the mathematical expressions shown in this section. The tensor contraction layer described in this section was implemented inside of a Convolutional Neural Network trained to classify MNIST [19] handwritten digits. This simple example thus includes code on how to backpropagate linear, convolution, MaxPooling, and ReLU activation layers for those interested [3, 18, 21].

3.1 Context of Tensor Contraction Layer

In its proper context, this layer is used in a Convolutional Neural Network [3, 16] before the fully connected layer at the end of the model to reduce the number of required parameters. A diagram of the Tensor Contraction Layer can be seen in Figure 3. This layer takes an order 3 input tensor and performs the n-mode product [12] with learned matrices to output a core tensor that represents the features of the image while using less space. The mathematical expression to represent the forward propagation in this layer is shown in Equation 18.

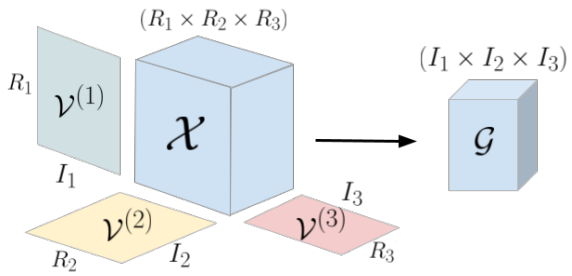

Fig. 3. Visualization of Tensor Contraction Layer (TCL). Third order input tensor is multiplied by 3 matrices using the n-mode product. Depending on the size of dimensions of the matrices, this can be used to decrease the size of the input tensor to give a core tensor.

Here, subscripts denote the folding of the tensor along the given dimension. N in this case would equal 2.

$$\mathcal{G}_{(n)} = \mathbf{V}^{(n)} \mathcal{X}_{(n)} \left(\mathbf{V}^{(N+1)} \otimes \cdots \otimes \mathbf{V}^{(n+1)} \otimes \mathbf{V}^{(n-1)} \otimes \cdots \otimes \mathbf{V}^{(1)} \right)^T = \mathbf{V}^{(n)} \mathcal{X}_{(n)} \left(\bigotimes_{i=N+1, i \neq n}^{1} \mathbf{V}^{(i)} \right)^T \tag{18}$$

3.2 Mathematical Expressions

During forward propagation, the input to the layer is tensor $\mathcal{X}$, and output is tensor $\mathcal{G}$. Thus, during backpropagation, the partial derivative of loss with respect to tensor $\mathcal{G}$, will be propagated backward to compute the partial derivative of loss with respect to the input tensor, $\mathcal{X}$, and the partial derivative of loss with respect to the parameter matrices, $\mathbf{V}^{(n)}$. This process can be summarized by:

$$\frac{\partial \mathcal{L}}{\partial \mathcal{G}} \Rightarrow \frac{\partial \mathcal{L}}{\partial \mathcal{X}}, \frac{\partial \mathcal{L}}{\partial \mathbf{V}^{(n)}}$$

The backpropagation equations can be extrapolated from [4] and the necessary equations for this layer are shown in the following equations. Equation 19 relates the partial derivative of loss with respect to the core tensor to the partial derivative of loss with respect to the input tensor. This can be used to backpropagate the loss further down the network

to earlier stages.

$$\frac{\partial \mathcal{G}_{(N)}}{\partial \mathcal{X}_{(N)}} = \left(\mathbf{V}^{(N)} \otimes \cdots \otimes \mathbf{V}^{(1)}\right) \otimes \mathbf{V}^{(N+1)} \tag{19}$$

Equations 20, 22, 21 can be used to relate the partial derivative of loss with respect to the core to the partial derivative of loss with respect to the matrices involved in the contraction operation. This can be used to update the parameters which help the model learn dimensional features about the input and perform more accurate classifications.

$$\frac{\partial \mathcal{G}_{(N)}}{\partial \mathbf{V}^{(n)}} = \mathcal{P}\left(\frac{\partial \mathcal{G}_{(n)}}{\partial \mathbf{V}^{(n)}}\right) = \mathbf{P}_n \frac{\partial \mathcal{G}_{(n)}}{\partial \mathbf{V}^{(n)}} \tag{20}$$

where $\mathbf{P}_n$ is a permutation matrix defined by:

$$\text{vec}(\mathcal{G}_{(n)}) = \mathbf{P}_n \text{vec}(\mathcal{G}_{(N)}) \tag{21}$$

and the partial derivative being permutated is defined by:

$$\frac{\partial \mathcal{G}_{(n)}}{\partial \mathbf{V}^{(n)}} = \left(\mathbf{V}^{(N+1)} \otimes \cdots \otimes \mathbf{V}^{(n+1)} \otimes \mathbf{V}^{(n-1)} \otimes \cdots \otimes \mathbf{V}^{(1)}\right) \mathcal{X}_n^T \otimes \mathbf{I}_k = \left(\bigotimes_{\substack{i=N+1 \\ i \neq n}}^{1} \mathbf{V}^{(i)}\right) \mathcal{X}_n^T \otimes \mathbf{I}_k \tag{22}$$

where k is the length of the nth mode of $\mathcal{G}$, and $\mathbf{I}_k$ represents an identity matrix with shape (k, k). The property described by Equation 20 follows from Equation 26, and a more detailed proof can be found in [4]. The reason permutation is used in this instance is because in the code the backpropagation of loss with respect to the core tensor $\mathcal{G}$ is represented by the next layer as the flattening of the N-mode folded tensor. Thus, equations will use this as a starting point to solve for other backpropagation values.

3.3 Backpropagation Formula Intuition

The backpropagation Equations 19 & 22 can be derived using the simple building blocks described in the background material. First, break up the expression into its element-wise components. Second, solve for the partial derivative with respect to a given component. Third, convert into an abstract tensor expression which can be more easily handled by tensor libraries.

Forward propagation of the Tensor Contraction Layer, as described in Figure 3, uses Equation 18 with parameters $n = N = 2$, which gives the following equation:

$$\mathcal{G}_{(2)} = \mathbf{V}^{(3)} \mathcal{X}_{(2)} \left(\mathbf{V}^{(2)} \otimes \mathbf{V}^{(1)}\right)^T \tag{23}$$

The reason these equations are written in terms of $\mathcal{G}$ folded along mode 2 is because that form is easy to construct from a vector representation which is used in code.

Both $\mathbf{V}^{(3)}$ and $(\mathbf{V}^{(2)} \otimes \mathbf{V}^{(1)})^T$ are matrices. For simplicity, they will be referred to as $\mathcal{P}$ and $\mathcal{Q}$ respectively. Rewriting the forward propagation equation using indices to denote individual elements gives:

$$\mathcal{G}_{(2)i,j} = \mathcal{P}_{i,k} \mathcal{X}_{(2)k,l} \mathcal{Q}_{l,j} \tag{24}$$

The range of each index is given by:

$$0 \leq i < I_3 \quad 0 \leq k < R_3 \quad 0 \leq l < R_1 \cdot R_2 \quad 0 \leq j < I_1 \cdot I_2$$

Note that adjacent matrices share a common index to denote matrix multiplication. To get the partial derivative of $\mathcal{G}$ with respect to $\mathcal{X}$ as described in Equation 19, the result will be equal to the values multiplied by $\mathcal{X}$:

$$\frac{\partial \mathcal{G}_{(2)i,j}}{\partial \mathcal{X}_{(2)k,l}} = \mathcal{P}_{i,k}\mathcal{Q}_{l,j} \tag{25}$$

At this point, a 4-dimensional tensor which represents the partial derivative is obtained. The following goal is to construct an abstract mathematical formula which does not deal directly with individual elements. The outer product of two matrices is a valid answer, but it is much more convenient to work with an output matrix rather than a 4-dimensional object. To remedy this, the authors of [4] utilized the following property:

$$\frac{\partial \mathcal{Y}}{\partial \mathcal{X}} = \frac{\partial \text{vec}(\mathcal{Y})}{\partial \text{vec}(\mathcal{X})} \tag{26}$$

where $\mathcal{Y}$ is a function of $\mathcal{X}$. Proof can be found in [4, 20]. Vectors only have single indices so the new expression will have a single index to identify the element of $\mathcal{G}$ and a single index to identify the element of $\mathcal{X}$ involved in the operation:

$$\frac{\partial \text{vec}(\mathcal{G}_{(2)})_s}{\partial \text{vec}(\mathcal{X}_{(2)})_r} = \mathcal{P}(s\%I_3, r\%R_3)\mathcal{Q}(r \div R_3, s \div I_3) \tag{27}$$

where new indices are defined as:

$$s = i + j \cdot I_3 \qquad r = k + l \cdot R_3 \tag{28}$$

thus:

$$i = s\%I_3 \qquad j = s \div I_3 \qquad k = r\%R_3 \qquad l = r \div R_3 \tag{29}$$

The right-hand side of Equation 27 can be written in abstract form as the Kronecker product of the two matrices:

$$\mathcal{Q}^T \otimes \mathcal{P} \tag{30}$$

Substituting the definition of these matrices gives the formula shown in Equation 19. Note that a transpose is necessary to swap the order of the indices of $\mathcal{Q}$ in order to align with the definition of the Kronecker product. Performing the same process to find the partial derivative with respect to $\mathbf{V}^{(3)}$ is problematic, as the same ambiguity explained in Section 2 is faced with the case of backpropagating a linear layer. Simply using the same indices as in Equation 24 to perform the derivative results in the following expression:

$$\frac{\partial \mathcal{G}_{(2)i,j}}{\partial \mathcal{P}_{i,k}} = \mathcal{X}_{(2)k,l}\mathcal{Q}_{l,j} \tag{31}$$

The left-hand side of this equation uses the index i in both the numerator and the denominator, suggesting an unintended summation. This also prevents defining expressions where the first index of $\mathcal{P}$ is unequal to the first index of $\mathcal{G}_{(2)}$. This was not an issue when considering the derivative with respect to $\mathcal{X}$, as seen in Equation 24, because there are no common indices between $\mathcal{G}$ and $\mathcal{X}$. We can recreate this for the partial derivative with respect to $\mathcal{P}$ by simply multiplying both sides of Equation 24 with the identity matrix:

$$\mathcal{G}_{(2)i,j} = I_{i,m}\mathcal{P}_{m,k}\mathcal{X}_{(2)k,l}\mathcal{Q}_{l,j} \tag{32}$$

This does not change the value of the equation because multiplication by the identity matrix leaves no impact on the matrix expression. Defining Φ as the matrix multiplication between $\mathcal{X}_{(n)}$ and $\mathcal{Q}$, the right-hand side can be simplified to:

$$\mathcal{G}_{(2)i,j} = I_{i,m}\mathcal{P}_{m,k}\Phi_{k,j} \tag{33}$$

As demonstrated, this equation shares the same form as Equation 24. Converting the equation into an abstract expression yields Equation 22. Folding the tensors $\mathcal{X}$ and $\mathcal{G}$ along different directions can be used to give partial derivatives with respect to the other matrices $\mathbf{V}^{(1)}$ and $\mathbf{V}^{(2)}$. These examples illustrate how tensor expressions can be broken up into their element-wise components for easier analysis. Given that backpropagation relies on these partial derivatives, neural networks involving these tensor expressions can be effectively backpropagated by applying basic building blocks shown in Figures 1 and 2.

3.4 Code Section Walk-through

This subsection discusses the code portion of computing Equations 20, 21, and 22, which are necessary to update the matrices used in contraction operation in an effort to minimize the loss. Lines referenced in this section can be found in the file `main.cpp` in the GitHub repository.

Solving for $\mathbf{P}_n$ directly using Equation 21 would be very difficult to calculate and it would also require an extreme amount of memory. For a core tensor with shape $(10 \times 10 \times 20)$ which is a relatively small case, the matrix would be of shape (2000×2000) and therefore would contain 4,000,000 elements. This is impractical to implement and has poor time complexity. Thus, an alternative option can be used which directly captures what the permutation matrix is doing. This step is simply mapping rows of one matrix to rows of another matrix. This operation can be represented by a vector where the index is the starting row and the value the index of the destination row. For instance, `permutation_vector[i]=j` would denote mapping row i of the original matrix to row j of the final permuted matrix.

Equation 21 requires that the vectorized representation of the core tensor G folded along the mode N can be mapped to tensor G folded along mode n. Lines 37-44 of `main.cpp` discover this mapping. First, a dummy vector is created where the value of the vector at a given index is the same index. This vector is then folded separately with respect to all three dimensions to generate matrices P_{0-2}:

```
VectorXd dummy = VectorXd::Zero(2000);
for(int i=0; i<2000; i++){dummy(i)=i;}
MatrixXd P_0=foldDim0(dummy,10,10,20);
MatrixXd P_1=foldDim1(dummy,10,10,20);
MatrixXd P_2=foldDim2(dummy,10,10,20);
```

The function titled `generate_permutation_vector` takes two vectors as arguments and is now called on the flattened pairs $(\text{vec}(P_0), \text{vec}(P_2))$, $(\text{vec}(P_1), \text{vec}(P_2))$, and $(\text{vec}(P_2), \text{vec}(P_2))$, the last of which returns the original dummy vector because no transformation is being made.

Now that the permutation policy is known, the next step is to compute the expression shown in Equation 22. In the case of n = 0,1,2, the expression can be simplified to the following formulas:

$$\frac{\partial \mathcal{G}_{(0)}}{\partial \mathbf{V}^{(1)}} = \left(\mathbf{V}^{(3)} \otimes \mathbf{V}^{(2)}\right) \mathcal{X}_0^T \otimes \mathbf{I}_{10} \tag{34}$$

$$\frac{\partial \mathcal{G}_{(1)}}{\partial \mathbf{V}^{(2)}} = \left(\mathbf{V}^{(3)} \otimes \mathbf{V}^{(1)}\right) \mathcal{X}_1^T \otimes \mathbf{I}_{10} \tag{35}$$

$$\frac{\partial \mathcal{G}_{(2)}}{\partial \mathbf{V}^{(3)}} = \left(\mathbf{V}^{(2)} \otimes \mathbf{V}^{(1)}\right) \mathcal{X}_2^T \otimes \mathbf{I}_{20} \tag{36}$$

these are the expressions shown in lines 167-169 of the source code, which are also permutated using the function `permutate` which has the appropriate permutation vector as the first argument and the matrix to be permutated as the second argument. The rows are then rearranged according to the rule `permutation_vector[i]=j`.

Giving the resulting values which are an application of Equation 20:

$$\frac{\partial \mathcal{G}_{(i)}}{\partial \mathbf{V}^{(i)}} \Rightarrow \texttt{permutate()} \Rightarrow \frac{\partial \mathcal{G}_{(2)}}{\partial \mathbf{V}^{(i)}}$$

The derivative of the parameter matrices with respect to loss can be derived by performing the dot product with the loss with respect to G folded by the second mode shown below. This operation starts with the vectorized form of partial derivative with respect to the core folded along mode 2. This can be easily reshaped after the operation.

$$\frac{\partial \mathcal{L}}{\partial \text{vec}(\mathbf{V}^{(i)})} = \frac{\partial \text{vec}(\mathcal{G}_{(2)})}{\partial \text{vec}(\mathbf{V}^{(i)})} \frac{\partial \mathcal{L}}{\partial \text{vec}(\mathcal{G}_{(2)})} \tag{37}$$

This is shown in lines 172-174 of the source code.

```
dot(DGDU3.transpose(), del_G);
dot(DGDU2.transpose(), del_G);
dot(DGDU1.transpose(), del_G);
```

Note that in the code the result of the dot product must also be `.reshaped()` to have the same dimensions of the corresponding parameter matrix.

$$\frac{\partial \mathcal{L}}{\partial \text{vec}(\mathbf{V}^{(i)})} \Rightarrow \texttt{.reshaped()} \Rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{V}^{(i)}}$$

Other components of the program are relatively straight forward compared to finding the partial derivative of loss with respect to the parameter matrices. The code is thoroughly commented to understand individual components and complicated functions are broken up into simple functions to aid in readability.

4 Practical Tips

This section includes advice on how to test whether backpropagation written by hand is working correctly or if there is a bug that needs finding. If there is a bug, this advice will help in narrowing down the location of this bug in an efficient manner.

4.1 Greedy Layer-Wise Training

It is a common practice to test the validity of a machine learning model by first over-fitting on a single test case. This can be taken one step farther by over-fitting a single layer to a single test case. Testing a layer individually [2] can be done by first forward propagating the layers preceding the test layer and next repeatedly forward and backpropagating the test layer and those after it. A visualization of this can be seen in Figure 4, in which $\mathcal{X}_i$ symbolizes the input to layer i, $\mathcal{P}_i$ symbolizes the parameters used in layer i, and Φ represents an operation being tested.

In the scenario described in Figure 4, we assume that layers past the testing layer are fully operational, but layers up to and including the testing layer have unknown validity. Layers before the testing layer should complete a single forward propagation to generate the input to the testing layer. Next, backpropagation shall be used to update the parameters used in this layer and the input to this layer. This should be done one at a time, first testing if the parameters update correctly, then testing if the input updates correctly. If the model does not over-fit on this single test case, this

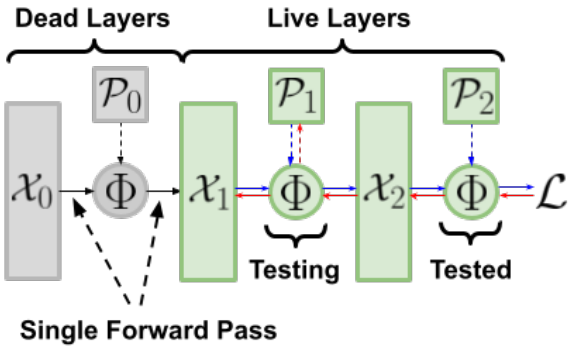

Fig. 4. Visualization of greedy layer-wise training of a neural network. Each layer is trained one by one and the layers after the testing layer are assumed to be operational.

layer must be the source of the problem because the loss is not backpropagating past this test layer. The bug can be narrowed down to either the connection between inputs or the connection between parameters. If the layer is working properly and the model over-fits, this process can be applied to the layer before it until the entire network is confirmed to not experience any bugs. Note that some layers may not use any parameters, ReLU is an example of this, so this process would be simplified to updating the input to cause improvement. On the pseudocode-level, the layers shown in Figure 4 can be written as:

```
X1 = layer1(X0,P0)
loop:
    X2 = L2_forward(X1,P1)
    out = L3_forward(X2,P2)
    //compute loss
    X2_prime, P2_prime = L3_back(loss)
    X1_prime, P1_prime = L2_back(X2_prime)
update(X1,X1_prime)
update(P1,P1_prime)
```

If layer 3 needed to be tested instead of layer 2, the line `X2 = L2_forward(X1,P1)` would be removed from the loop and placed before it, the line `X1_prime, P1_prime = L2_back(X2_prime)` would be removed, and the parameters being updated would change.

4.2 Focus on Shapes

When deriving and programming with tensor expressions, it is of utmost importance to pay attention to the shape of the tensors used in the expression to ensure that the operations are logical. Some examples of relationships between input and output tensor shapes are shown below:

Inner Product:

$$(I_1, I_2) \cdot (I_2, I_3) \Rightarrow (I_1, I_3) \tag{38}$$

Outer Product:

$$(I_1, I_2) \otimes (I_3, I_4) \Rightarrow (I_1, I_2, I_3, I_4) \tag{39}$$

Hadamard Product (Element-wise Product):

$$(I_1, I_2) \circ (I_1, I_2) \Rightarrow (I_1, I_2) \tag{40}$$

Kronecker Product:

$$(I_1, I_2) \otimes (I_3, I_4) \Rightarrow (I_1 \times I_3, I_2 \times I_4) \tag{41}$$

Operations between tensors of incorrect shape will most likely cause an error by the compiler which can be used to narrow down the location to a line in the code. The shapes of the tensors used in the operation that cause the error can be printed and analyzed to see if they are what is expected. Analyzing the shapes of tensors can also provide a better intuition about which dimensions represent what. An example of this is analyzing the shape of the backpropagation Equation 19, with the dimension sizes used in Figure 3. Equation 19 can be simplified to:

$$\frac{\partial \mathcal{G}_{(2)}}{\partial \mathcal{X}_{(2)}} = \left(\mathbf{V}^{(2)} \otimes \mathbf{V}^{(1)}\right) \otimes \mathbf{V}^{(3)} \tag{42}$$

where the shapes of the tensors are:

$$\mathcal{X} \in (R_1, R_2, R_3) \quad \mathcal{G} \in (I_1, I_2, I_3)$$
$$\mathbf{V}^{(1)} \in (R_1, I_1) \quad \mathbf{V}^{(2)} \in (R_2, I_2) \quad \mathbf{V}^{(3)} \in (R_3, I_3)$$

The relationship between the input size and output size of the Kronecker Product is described in Equation 41. Using this relationship, the right hand side of Equation 42 should be equal to a matrix with the following dimensions:

$$(R_1 \times R_2 \times R_3, \quad I_1 \times I_2 \times I_3)$$

This information indicates that the first index of this matrix is pointing to an element of $\mathcal{X}$ and the second index is pointing to an element of $\mathcal{G}$, where the value at these indices is the partial derivative of $\mathcal{G}$ with respect to $\mathcal{X}$. This alludes that a matrix multiplication by the left side of this expression would be utilizing a relationship with the $\mathcal{X}$ entries to make a connection with the $\mathcal{G}$ entries.

4.3 Tensor Notation Inspection

If a backpropagation bug is narrowed down to a single layer and nothing seems incorrect by analyzing the code, it may be time to perform further inspection on the mathematical notation used in the source equations. There may be a difference in interpretation compared to the library being used. A common source of disagreement is how tensors should be represented when they are transformed through the use of folding or flattening. Although it is expected that a function which flattens matrices to vectors does so by stacking rows, it may stack columns instead. Tensor folding can also be a source of confusion as some sources use nontraditional index ordering, although these cases are rare and should be explicitly mentioned in the source. When in doubt, it is always wise to perform small test cases on the tensor operations used in the expressions to be tested. Using matrices of size $(2, 2)$ should be sufficient to test the Kronecker Product function of a given library and to make the comparison to the results calculated by hand.

5 Conclusion

Tensor operation backpropagation is an important aspect of neural network optimization, especially when it comes to the manual derivation and coding of complex tensor expressions. Achieving a deep understand of backpropagation's foundational concepts is crucial for effectively implementing these operations in real-world applications. The Tensor

Contraction Layer (TCL) demonstrates the process of backpropagation and exemplifies how the low-level language implementation can be powerful and serve as a tool for instructional purposes. Bugs present in these types of programs can be found and fixed using simple methodologies which can accelerate development of these systems. Adopting these methodologies fosters deeper insights and proficiencies in machine learning practices and can pave the way for more advanced innovations in the field.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Y. Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, and U. Montreal. 2007. Greedy layer-wise training of deep networks.

[3] Laurent Boué. 2018. Deep learning for pedestrians: backpropagation in CNNs. arXiv:1811.11987 [cs.LG] https://arxiv.org/abs/1811.11987

[4] Giuseppe G. Calvi, Ahmad Moniri, Mahmoud Mahfouz, Qibin Zhao, and Danilo P. Mandic. 2020. Compression and Interpretability of Deep Neural Networks via Tucker Tensor Layer: From First Principles to Tensor Valued Back-Propagation. arXiv:1903.06133 [cs.LG] https://arxiv.org/abs/1903.06133

[5] Gene H. Golub and Charles F. Van Loan. 2013. *Matrix Computations* (4 ed.). Johns Hopkins University Press, Baltimore, MD.

[6] Jacques Hadamard. 1893. Sur les matrices à éléments donnés. *Bulletin de la Société Mathématique de France* 21 (1893), 27–30.

[7] Geoffrey E. Hinton, Alex Krizhevsky, and Ilya Sutskever. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (2017), 84–90. https://dl.acm.org/doi/10.1145/3065386

[8] Roger A. Horn and Charles R. Johnson. 1985. *Matrix Analysis*. Cambridge University Press, Cambridge, UK.

[9] Benoît Jacob, Gael Guennebaud, and Bruno Levy. 2010. Eigen: A C++ Template Library for Linear Algebra. In *Proceedings of the 2010 IEEE International Conference on Computer Vision*. IEEE. https://eigen.tuxfamily.org

[10] D.P. Kingma and J.B. Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980* (2014). https://arxiv.org/abs/1412.6980

[11] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. In *SIAM Review*, Vol. 51. Society for Industrial and Applied Mathematics, 455–500. https://doi.org/10.1137/07070111X

[12] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (2009), 455–500. https://doi.org/10.1137/07070111X

[13] Jean Kossaifi. 2020. Understanding Tensor Unfolding and Tensor Products. https://jeankossaifi.com/blog/unfolding.html Accessed: 2024-08-29.

[14] Jean Kossaifi, Aran Khanna, Zachary C. Lipton, Tommaso Furlanello, and Anima Anandkumar. 2017. Tensor Contraction Layers for Parsimonious Deep Nets. arXiv:1706.00439 [cs.LG] https://arxiv.org/abs/1706.00439

[15] Jean Kossaifi, Zachary C. Lipton, Arinbjorn Kolbeinsson, Aran Khanna, Tommaso Furlanello, and Anima Anandkumar. 2020. Tensor Regression Networks. arXiv:1707.08308 [cs.LG] https://arxiv.org/abs/1707.08308

[16] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. In *Neural Computation*, Vol. 1. MIT Press, 541–551.

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. In *Proceedings of the IEEE*, Vol. 86. IEEE, 2278–2324. https://ieeexplore.ieee.org/document/726791

[18] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, Vol. 86. IEEE, 2278–2324. https://doi.org/10.1109/5.726791

[19] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. 1998. The MNIST Database of Handwritten Digits. In *Proceedings of the IEEE*, Vol. 86. IEEE, 2278–2324. http://yann.lecun.com/exdb/mnist/

[20] Jan R. Magnus and Heinz Neudecker. 1988. *Matrix Differential Calculus with Applications in Statistics and Econometrics* (revised edition ed.). John Wiley & Sons, Inc., New York.

[21] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (2010), 807–814. http://www.cs.toronto.edu/~hinton/absps/reluICML.pdf

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., 8024–8035. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[23] B.T. Polyak. 1964. Some Methods of Speeding Up the Convergence of Iterative Methods. In *USSR Computational Mathematics and Mathematical Physics*, Vol. 4. Elsevier, 1–17.
[24] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536.
[25] John Smith and Jane Doe. 2019. Einstein Summation Convention and Its Applications in Physics. *J. Math. Phys.* 60, 3 (2019), 033101. https://doi.org/10.1063/1.5091234
[26] James Stewart. 2015. *Calculus: Early Transcendentals* (8 ed.). Cengage Learning, Boston, MA.

Received xxx; revised xxx; accepted xxx