EFFICIENT SPARSE NEURAL NETWORK TRAINING

by

KONSTANTIN SHVEDOV

A THESIS

Presented to the Department of Computer and Information Science
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2022

THESIS APPROVAL PAGE

Student: Konstantin Shvedov

Title: Efficient Sparse Neural Network Training

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Jee Choi	Chair
Hank Childs	Core Member

and

Krista Chronister	Vice Provost for Graduate Studies

Original approval signatures are on file with the University of Oregon Division of Graduate Studies.

Degree awarded June 2022

THESIS ABSTRACT

Konstantin Shvedov

Master of Science

Department of Computer and Information Science

June 2022

Title: Efficient Sparse Neural Network Training

Developments in neural networks have led to advanced models requiring large amounts of training time and resources. To reduce the environmental impact and to decrease the training times of models, acceleration techniques have been developed. One method is neural network pruning, which removes insignificant weights and preempts the generation of sparse models. This paper attempts to improve and explore a method of training sparse neural networks efficiently processing only non-zero values using optimized just-in-time kernels from the Libsxmm library while randomly pruning network layers at initialization. The algorithms explored within this paper show a proof of concept and the possibility of improving training time beyond what the highly optimized PyTorch library is currently capable of. Through the work in this paper algorithm's processing times are sped up over 100-fold. Further, this work provides additional evidence that advanced pruning algorithms and other improvements can significantly reduce training times and resources.

CURRICULUM VITAE

NAME OF AUTHOR: Konstantin Shvedov

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Washington State University, Pullman, WA, USA
Virginia Polytechnic Institute and State University, Blacksburg, VA, USA

DEGREES AWARDED:

Master of Science, Computer and Information Science, 2022, University of Oregon
Bachelor of Science, Computer Science, 2020, Washington State University

AREAS OF SPECIAL INTEREST:

Machine Learning
Artificial Intelligence

PROFESSIONAL EXPERIENCE:

Software Developer (developing new search algorithm over database with no standardized naming scheme), Hiline Engineering & Fabrication, 2021-Current
Capstone Project Leader, Washington State University, 2019-2020
Teachers Assistant, Advanced Data Structures, CPTS 223, Washington State University, 2019
Teachers Assistant, Intro to Computer Science, CPTS 121, Washington State University, 2018
Camp Instructor, White Wind, Volgograd, Russia, 2014-2018

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

Humans produce a massive amount of information through technology, and there is a growing need for understanding and processing this data. Although neural networks are resource-intensive machine learning (ML) models, some neural network architectures have become efficient enough to be used in handheld devices like smartwatches. However, the more significantly advanced neural networks are still incredibly difficult to train. For example, GPT-3, a 175 billion parameter model [4], requires 34 days of training on 1024 GPUs at a cost of $4.6 million, without including research and development pricing. On top of these expenses, the training of GPT-3 has a massive environmental impact. The model uses "several thousand petaflops/s-days of compute" [4] just during pretraining. The petaflops/s-days translate into massive electricity usage that not many companies, let alone individuals, have the money and equipment to develop and train in the attempts of creating an analogous network.

To reduce the costs and environmental impact (e.g., minimize electricity use) of training resource-intense neural networks, computer scientists constantly strive to find new ways to accelerate neural networks' training and make it more efficient. M6 [22] is a new, efficient ML model with 10 trillion parameters. By using one percent of GPT-3's energy cost after a training time of only ten days, M6 demonstrates that there is still room for improvements even with the most advanced models. One preprint even discusses cutting out the most important part of neural networks, backpropagation [2]. Within the paper, a method is proposed where gradients are computed based on directional derivatives. The updating approach during forward propagation presented by Baydin et al. [2] can reduce

training times by almost half, and the algorithm can be applied to almost all types of neural networks.

An alternative solution to finding new ways of processing neural networks is to reduce the amount of data being processed by the neural network. Sparsity has been a critical area of research in neuroscience research, and it is always present in the brain [27]. The activity of neurons is always sparse. On the other hand, sparsity in neural networks is not always present. To introduce sparsity, neural networks are pruned. Pruning is a method of compression where either weights are set to zero or entire nodes are removed. Setting weights to zero is a much easier process but require efficient sparse processing of the layer. Pruning can happen in different ways and patterns, also called granularity [24]. For example, whole filter levels or blocks may be pruned out. It is more typical for vectors or specific values to be pruned out randomly or based on predetermined algorithms that determine the least relevant values. By elevating the use of pruning post-training, a network can be compressed. Such an approach was used in a paper in the 1990s [19]. LeCun's work [19] was the first to spark an interest and start the field of neural network pruning. The field since then has progressed to significant extents and led to highly cited works like Han et al.'s 2015 paper which reduces the size of a neural network by first training a network, then pruning the less meaningful connections and finally retraining the model to fine-tune the remaining weight [17]. Although Han et al.'s approach reduces the size of the final network, it does not speed up the training. Molchanov et al. managed to also achieve smaller and more efficient neural networks by also pruning after training. Molchanov et al. managed to achieve a 40% FLOPS reduction while only removing 30% of the parameters [25].

Pruning is not only for reducing the neural network weight after training, as it can also be used to reduce training time. Pruning during training is less common due to the reduction of accuracy of the model. That said, pruning schedules are one way to tackle accuracy concerns. In this paper, the neural network is pruned during training. The PyTorch library is used to prune and set up the neural network models. During training, each pass through the network takes three main steps: forward propagation, backward propagation, and weight updates. The sparse neural network layers utilize Libxsmms highly optimized just in time (OPT-JIT) Kernels within each step which only process the non-zero values. Processing only non-zero values should reduce training times, but the sample implementation uses slow data access methods and redundant reinitializations. The goal is to optimize the algorithm that handles data transformations for the kernels. Within this paper, an average speedup of 96.8x times is achieved over the three steps of the training process. Even though this is still not enough to exceed the PyTorch dense layer training speed, the work done shows the library's potential. Through future improvements, the Libxsmm algorithm will be able to outperform the PyTorch library. In addition to improving the algorithm, a bug is fixed within the library code that calls the OPT-JIT kernels that caused invalid values to propagate through iterations and ultimately crash the training.

The algorithms within this paper are only implemented and used on Intel CPUs, but similar approaches can be used for GPUs. There is also a possibility of reducing training times by half if backpropagation can be eliminated by using the methodology mentioned in the paper "Gradients without Backpropagation" [2]. We hope that our work promotes research and more uses of the technologies covered.

CHAPTER II

RELATED WORKS

The technique of introducing sparsity to neural networks has been around since their creation in the 1940s. There are four main techniques that have been explored and used in multiple papers:

1. Pruning before training or at initialization
2. Pruning during training
3. Pruning after training
4. Pruning utilizing the architecture of a neural network

Each of the pruning methodologies can also vary based on different pruning granularity [24]. One-Cycle Pruning [13] is a new technique that binds pruning during training with neural network architecture search. The method utilizes non-random pruning to introduce sparsity into neural networks during training and has considerable potential.

Granularity, within pruning, has two main categories. The first is structured pruning, where either complete blocks of weights, vectors or kernels are removed. Structured pruning can also remove specific weights based on metrics to heuristically preserve the most significant values [13]. The methods of leaving the most significant weights aim at finding "The Lottery Ticket" from The Lottery Ticket Hypothesis (LTH) [9]. LTH theorizes that in a randomly initialized dense neural network, there is a subnetwork that can be independently trained to match the accuracy of the full network, and it will do so in, at worst, the same number of iterations as it takes to train the full network [9]. The second category is

unstructured pruning, which is when the pruning happens randomly in an attempt to remove weights without the intention of keeping any specific structure [16]. Both categories can produce sparse weight matrices, which can be utilized to reduce computational costs and increase the speed, just like in the preprint paper. Both granularities can be and are used for the compressions of a neural network.

Several methods of pruning before training or at initialization have been proposed. One of these methodologies was proposed alongside the LTH by Frankle and Carbin [9] within the second paper on the topic by Frankle et al.[10]. Unfortunately, this method is highly expensive as the "Lottery Ticket" is challenging to find within a neural network.

Pruning after training is one of the more common methodologies. It is a less difficult process where all network weights are known after training. The post-training pruning is done to reduce costs during the use of the network. For example, Molchanov et al. managed to achieve a 40% FLOPS reduction while only removing 30% of the parameters [25]. Within the scope of pruning during training, structured pruning is the go-to method to avoid losing connections within a neural network that could be part of the LTH. One proposed technique is pruning on a fixed interval schedule. Feature relevance scores guide the pruning process. It is tested within a paper [1] and produces positive results displaying drops in processing times while achieving less than 1% drop in accuracy with significant model compression over CIFAR-10 [14], CIFAR-100 [15], and ImageNet [6] datasets.

A recent paper by Hubens et al. [13] also uses feature relevance scores in their One-Cycle pruning technique. Hubens et al. approach scheduling with a math formula that gradually increases the pruning done at each step and reduces pruning rates when approaching high sparsity. The method achieves higher accuracy under

80%, 90% and 95% sparsity than One-Shot, Iterative and Automatic Gradual pruning on the CIFAR-10 [14], CIFAR-100 [15] and Caltech-101 [8] datasets using the ResNet-18 [11] network.

While pruning remains the most significant element that has to be optimized in the future to achieve high accuracy, there are alternative methods of handling sparse data. As an alternative to the kernels used within this paper, Lewis et al. [20] developed a technique where data is efficiently routed to "expert" kernels that had a fraction of the model's parameters and were specialized working with those parameters. Another method utilizes exponentially smoothed gradients (Momentum)[7]. The momentum is used to redistribute pruned weights across layers using the mean momentum magnitude of each layer. Dettmers and Zettlemoyers methodology manages to achieve up to a 5.61x faster training than dense layers.

Future works with techniques from this paper should include testing on GPUs and with deep neural networks. Marcin Pietroń, Dominik Żurek explore such techniques in their paper [26].

CHAPTER III

SPARSE JIT KERNELS

Training a sparse neural network involves three kernel-related steps. The first step is forward propagation. The backwards pass within neural networks consists of the two remaining steps: backwards propagation and weight updates. The sparse OPT-JIT kernels from the Libxsmm library are the basis of the algorithms that are optimized within this paper. The kernels are optimized for sparse matrices and multi-core CPUs. The algorithms for the three steps are written in C++ using PyTorch structures (tensors) but have a connecting layer of Python. Python allows for easy data access, model building and training. In the future, when the algorithms are fully optimized, they may be added to the PyTorch library.

Within a dense neural network, all matrices are stored in either a row-major or column-major format. Each steps algorithm transforms the data into appropriate formats for each kernel before transforming it back. Once internal improvements are complete, interlayer communication optimization should be explored since the removal of extra transformations of data will lead to a significant speedup of the algorithms.

The forward propagation algorithm focuses on transforming data for the kernel. It accepts the batch input vectors I (size $M \times N$) that are multiplied with the weight matrix W (size $N \times A$), producing the output vectors O (size $M \times A$). Within the forward kernel, all parts are in the column-major format. The forward kernel executes the following:

$$I \times W = O \tag{3.1}$$

The backward propagation algorithm uses a kernel that multiplies dO a column-major differential of the output matrix with W^T, a row-major transposed weight matrix. The output is dI a column-major differential of the input matrix. The differential matrices have the same sizes as the original matrices.

$$dO \times W^T = dI \tag{3.2}$$

The update algorithm uses a kernel that multiplies I^T a row-major transposed input matrix with dO a column-major differential of the output matrix, and the output is dW a column-major differential of the weight matrix.

$$I^T \times dO = dW \tag{3.3}$$

In neural networks with sparse inputs there are multiple ways to store sparse weight matrices in order to decrease the size and increase processing speeds during forward and backward passes. The Libxsmm library uses Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats, which are complementary to each other in the way that row-major and column-major formats are to each other. The CSR and CSC store the data using three one-dimensional arrays: *values, rowptr/colptr*, and *colidx/rowidx*. The values array contains the values of all non-zero elements from the sparse matrix. The *rowptr/colptr* arrays contain the indices of the first element in each *row/column*. The *colidx/rowidx* arrays contain the column/row index of each non-zero element.

To maintain the consistency of the formats through the forward and backward passes during training, the forward kernel multiplies column-major input I with the W matrix stored in a CSC format (sparse), producing a row-major

output matrix O. The backward kernel multiplies the output dO, a column-major differential, with the transposed CSR weight matrix W^T (sparse). This produces a column-major differential of input I^T. The update kernel multiplies a row-major input matrix I^T a column-major differential of the output matrix dO and produces the differential weight matrix dW in CSC format (sparse).

The Libxsmm library uses common optimization techniques for the algorithm and for the kernels, such as loop unrolling. On top of the common techniques, the kernels use Just-in-Time techniques (OPT-JIT) to hard code indices for CSC and CSR formats and save addresses. This is done to reduce overhead. The hard-coding of these values is justified by the unchanging sparsity pattern over 100 to 1000 iterations.

Within each of the three steps of the training process, the Libxsmm library transforms the full size, pruned matrices into compressed formats. The compressed data is then split into processing blocks that are passed to kernels that calculate the results in parallel. The final step of each algorithm is to transform the results back into full size matrices, before passing them back. Each of the kernels that are used at each step, were initially optimized through C before applying just-in-time techniques.

3.1 Forward Kernel

To achieve high performance, the forward kernel must process $I \times W = O$, where I is in CSC format while the other two matrices are dense. The multiply and add operations are fully unrolled within the kernel to avoid instruction latency. The sparse multiplication that is completed is a sparse matrix by matrix multiplication (SpMM) and can be bound by computation. To tackle the possibility of computational bottlenecks, the multiply and add operations are implemented

as Single Instruction/Multiple Data (SIMD) operations to increase instruction-level parallelism (ILP). Cache locality is improved through the blocking of the matrices; specifically, they are blocked along the rows of the input matrix and the columns of the weight matrix. The executions of the blocks are then parallelized by assigning them to different threads. To further reduce execution times of the kernel, a temporary vector is used to accumulate results and only writes the vector back after the final results are calculated.

Algorithm 1: Forward Kernel

Input : Sparse matrix W stored in CSC format (size N x A), Input Matrix $I[M][N]$
Output: Output Matrix $O[M][N]$

```
1  // parallel
2  for k = 0 to A by AB do
3      for m = 0 to M by MB do
4          for ab = 0 to AB by 1 do
5              aa = a + ab
6              colstart = W.colptr[aa]
7              colend = W.colptr[aa + 1]
8              // Initialize a vector of length MB to zero
9              O_vector ← 0.0
10             for i = colstart to colend do
11                 j = W.values[i]
12                 k = W.colidx[i]
13                 // loop unrolling and SIMD
14                 O_vector[m]+ = I[m][k] * j
15             end
16             O[aa][m] = O_vector[m]
17         end
18     end
19 end
```

In order to leverage just-in-time code generation to handle sparse data, finite element method (FEM) simulations are used from other works [3], [12]. The underlying idea is that sparsity patterns of the FEM operator are hardcoded into

the instruction stream. The last piece of the puzzle is to transform the data into a five-dimensional set. Each three-dimensional block is handled separately by a kernel that is created specifically for its patterns. The five-dimensional data storage is required to avoid instructional cache overflow. Each of the kernels is scheduled to different cores. To improve the kernel even more, the result matrix columns are treated as scratchpads, which allows the implementation of completely unstructured access and higher levels of parallelization.

3.2 Backward Kernel

Algorithm 2: Backward Kernel

Input : Sparse matrix W^T stored in CSR format (size A x N), Differential of Output Matrix $dO[M][A]$, Index translation from CSR to CSC $idxmap[y]$, W^T's $rowptr$, W^T's $colidx$

Output: Differential of Input Matrix $dI[M][A]$

```
1  // parallel
2  for n = 0 to N by NB do
3      for m = 0 to M by MB do
4          for nb = 0 to NB by 1 do
5              nn = n + nb
6              rowstart = rowptr[nn]
7              rowend = rowptr[nn + 1]
8              // Initialize a vector of length MB to zero
9              dI_vector ← 0.0
10             // loop unrolling and SIMD
11             for i ← rowstart to rowend do
12                 j = W.values[idxmap[i]]
13                 k = colidx[i]
14                 dI_vector[m]+ = dO[m][k] * j
15             end
16             dI[nn][m] ← dI_vector
17         end
18     end
19 end
```

The backwards kernel that is used during the backwards pass executes $dO \times W^T = dI$, where W^T is a sparce matrix in CSR format, while the other two are

dense matrices. In comparison to the forward kernel, due to the format used to store the sparse matrix, the backward kernel needs a separate approach to avoid the either loading W^T multiple times or read and write to each location in dI.

Due to the sparsity patterns being the same over many iterations, an *idxmap* is created that stores the index translations from CSR to CSC of non-zero values and stores them in an array. The *rowptr* and *colidx* arrays are also precomputed. With these edits, the W^T matrix can be treated as a CSC matrix allowing the optimizations used in the forward kernel to be applied to the backward kernel. The index lookup is done before executing the SpMM computations. The OPT-JIT kernel for the backwards propagation is identical to the one in forwards propagation.

3.3 Update Kernel

Algorithm 3: Update Kernel

Input : Sparse matrix dW stored in CSC format (size N x A), Differential of Output Matrix $dO[M][A]$, Input Matrix $I^T[N][M]$, Column Index $kidx[y]$

Output: Differential of sparse weight Matrix dW

```
Initialize all points in dW.values to zero
for m = 0 to M by MB do
    // parallel
    for i = 0 to y by YB do
        n_x = dW.cidx[i + x] (x=1,...,YB)
        a_x = kidx[i + x] (x=1,...,YB)
        // Initialize YB vectors to zero
        dW_x ← 0.0(x=1,...,YB)
        // loop unrolling and SIMD
        for mb = 0 to MB by 1 do
            mm = m + mb
            dW_x+ = I^T[n_x][mm] * dO[mm][a_x]
        end
        dW.values[i+x] += sum(dW_x) (x=1,...,YB)
    end
end
```

The update kernel that is used during the backward pass executes $I^T \times dO = dW$ where dW is a sparse matrix in a CSC format of size N x A while the other two are dense matrices. The update kernel is only required to produce the non-zero values of the output matrix when multiplying the input matrix. Just as in previous kernels the multiplication and addition operations are fully unrolled and implemented as SIMD operations to increase instruction-level parallelism. But under the circumstances of when A is small, the algorithm may not be able to provide the high levels of parallelism that the kernel relies on. In order to circumvent this issue, each non-zero values position index needs to be known, and the matrix is stored as Coordinate list format (COO) instead. Since the row indices are known, column indices are calculated. The values are then grouped and executed in parallel on different threads using the unrolled SIMD operations mentioned earlier.

Overall, the OPT-JIT kernel has not changed from the update kernel. The sparsity pattern is still hardwired into the kernel. The inner product is computed as a real inner product, not as an outer product how it was in the forward kernel. This is due to the chosen memory layout.

3.4 Bug in Update Kernel

During testing of neural networks, a problem was found that during training of a small three-layer neural network. The crashing of the training process was happening at random times and random epochs. To find the problem, seeded randomizers were used to reproduce the problem. The problem persisted with high levels of randomness. In order to find the error, multiple techniques of debugging were used. The Libxsmm library algorithms were dissected in order to find the error.

The bug location was found within the C++ library code and related to how memory is allocated to the PyTorch tensors that are transferred in from Python. The `empty()` function is used, which allocates memory without setting any of the values to a default value. The solution was to use `zeros()` function, which set all allocated memory to zero values. Without this change, the values cause propagation and exponential growth of values during training. The change also brings a 19x increase in allocation times even though both allocation time values are below 0.3 milliseconds. The change is insignificant in the current state of the algorithm and processing times, but in the future will be addressed through the allocation of space during model construction.

The most likely reasoning for the implementation that was causing problems during testing for this paper is that the library sample was only tested with a one-layer neural network, and then the work was dropped. The library sample on git currently does not have the implementation and code that is used within the preprint paper. The design of the neural network tested within the scope of this paper is a 3-layer neural network. The three layers need to have zeroed out values to be able to operate within a model properly. All extra layers used in the model are not intended for sparse processing and use the incorrect values due to the library's sample implementation.

3.5 Algorithm Improvements

The improvement of the algorithms and the discovery of future improvements within the code connecting PyTorch to the kernels is the primary goal of this paper, as they show the proof of concept needed to continue with this work. The code has the potential to exceed the speed of PyTorch executions,

especially when the neural networks become extremely large. There are currently four problematic areas within the algorithms:

1. PyTorch C++ API
2. Access to data in PyTorch tensors
3. Structure and data reinitialization
4. Backward propagation

Firstly, the PyTorch C++ API, which is used everywhere, is slow due to it imitating its python counterparts to ease the use of the API. Secondly, the access to all the data within the PyTorch tensors is a bottleneck. Thirdly, within the current implementation, many structures are reinitialized at each epoch, which minimizes the reuse of structures. As the fourth and final point, the backwards propagation is split into two sections that do not communicate with each other.

The initial implementation of the Libxsmm algorithm that this paper is working with uses the PyTorch API as an easy way to implement and test solutions for their kernels. The kernels are optimized and outperform many other libraries like Intel MKL SpMM. On the other hand, the use of the PyTorch API dramatically reduces efficiency. The API is aimed at smoothing the boundaries between C++ and Python but is not intended to be called hundreds of times. This is the case in the Libxsmm library during the transformation of matrices from dense to sparse and back after the execution of the kernels. In order to counter the relatively slow translations of the API to access data, an alternative is used. Specifically, the data is accessed through flat pointers, even in the case of 5-dimensional data. The flat pointers allow for quick, direct access to the values

without the need for convoluted API calls. Furthermore, a few transformations like reshape and permute from the PyTorch API are used multiple times within each of the three algorithm steps. These algorithms are easily translatable into C++ and could improve the algorithm significantly once all other bottlenecks are handled.

The current version of the algorithm recreates all structures required by the model layer on each call. The reinitialization of different structures, especially kernels, is quite slow and is done at every epoch. As mentioned in the kernel descriptions, the sparsity does not change for 100 to 1000 epochs currently and therefore, the reinitialization of structures is redundant. Since each layer is kept as an object, saving the structure and reinitializing them at specific epochs is a change that can be implemented without having significant changes within the algorithm of Libxsmm.

Within the implementation initially taken from the Libxsmm library, the backwards pass is split into a backwards function and an update function, which are called directly one after another. Considering the preprint, this could be explained that each of the kernels described earlier had to be handled separately to remove any kind of overlap. Each kernel and the code around it had to be optimized independently. The combination of the sparse backwards function and the sparse update function would lead to significant improvements. The unity of the two functions would remove redundant variables and structure reinitializations within the same epoch. The unity of the two functions would also lead to fewer structures being saved for each epoch. The algorithm within the Libxsmm function has other possible improvements that could be found after the four problems described above are addressed. If the four problems are resolved, the sparse layers

will outperform the PyTorch library layers. In the future, sparse layer interactions can be explored to avoid full matrix operations.

CHAPTER IV

NEURAL NETWORK AND SPARSIFICATION

The Libxsmm sparse library code used in this paper is not as flexible as its dense python counterpart. One of the most significant constraints is that the input size has to be the same size as the output, which brings in a couple of issues, especially when timing. To overcome the issue within the parameters of the algorithm's acceleration, the data used for testing, drawn from the IMDb dataset [23], was created to accommodate specific input parameters. The sparsification used for testing is also very limited and had to stay at relatively high percentages due to the library's constraints.

4.1 Neural Network Testing Methodology

The neural network used for all testing has only three initially layers. The layers are either dense linear PyTorch or the sparse Libxsmm layers. Each layer is identical in size to the other two but, in some cases, sparsified using a different random seed. The Modified National Institute of Standards and Technology (MNIST) dataset [18] is used for its relatively large feature size (784) while still producing helpful training results. Loss is used over the accuracy value because the optimization of the network and testing of different models is not the target. Therefore, the accuracy is irrelevant until the analysis of the effects of sparsification in future works. The IMDb dataset [23] is used in conjunction with MNIST during final testing due to the high malleability of data through an embedding and a pooling layer. Embedding layers are mainly used for Natural Language Processing as an alternative to one-hot encoding and help reduce dimensionality. The embedding process turns each word into a fixed-length array of real values instead of 0's and 1's. First, each word in the dataset is one-hot encoded into

numbers that represent words. This paper uses a set that has already completed that step. Then each input is padded with zeros in order to achieve the same length. Next, each value in each input array is transformed and is embedded and turned into an array. We now have a two-dimensional array of values. The Libxsmm sparse layers only accept one-dimensional input. An average pooling layer is used to downsize average values into a single dimension. The Global Average Pooling averages each index with the same index of every array within a single input. With these two layers, any size of input can be created, which is precisely what is needed for timing the different layers and speedups.

To find the bottlenecks within the Libxsmm algorithm, the IMDb dataset [23] was used with a single layer model. The one-layer allowed for a more straightforward testing process and smaller wait times. Initially, each of the three algorithms were divided into four sections. The sections were grouped together based on there functionalities like initialization, dense to sparse matrices translations, kernel creation and execution. Unfortunately, the values produced from the timing of these sections provided minimal information and were all more or less the same. Each algorithm was then divided into ten or more sections which, each had individual times. This brought a clear understanding of the bottlenecks since some sections took 0.03 seconds to execute while one would take 7 seconds to execute. The largest values section were then looked at and dissected in the attempt of finding a possible improvement.

4.2 Future Work of Sparsification

The specific pruning technique focused on and partially used in this paper is unstructured gradual pruning. The pruning of matrices within neural networks is common, especially between layers. Pruning strengthens some connections and

allows for a better adjusted neural network. The pruning done in this paper is not intended to optimize accuracy but aims to decrease the training time while upholding similar loss values when compared to dense PyTorch layers. The testing complete within the paper only executes unstructured pruning on the layers right after creation. The pruning used is anywhere between 80 and 95 percent as this is the best working range for the Libxsmm library.

The Kernels used for each of the three steps are optimized to be used with sparse data. There is a problem with pruning right from the start, as it may prevent the neural network from learning critical values that are needed further down in testing. To avoid accuracy loss through pruning in the early stages of training, alternative methods have been developed to solve the issue. There are three types of pruning that are often used within neural networks during training. The first type is One-Shot pruning. One-Shot pruning is one of the first pruning methods adopted, where redundant weights are pruned within one step [21]. The second type is iterative pruning which was first used and tested in 1997 by Castellano et al. [5]. Within the iterative pruning method, the pruning happens based on a criterion and happens at specific intervals. After each pruning event, weights are adjusted in order to lessen the impact on the network's performance. The third type that is still being developed and tested is Automated Gradual Pruning [28]. The pruning quite often starts after a few full iterations and then follows a logarithmic curve leading up to a specific sparsity level. AGP pruning relies on effective scheduling. New methods of pruning are being developed constantly. For example, a new pruning method was developed in April of 2022, One-Cycle Pruning (OCP) [13]. Based on a mathematical equation, OCP is scheduled and starts from the second training step. OCP holds a low sparsity

while slowly pruning within the first 20% of training, then raises the sparsity levels relatively quickly before slowing down the pruning closer towards the middle of training. The schedule can be adjusted to fit different data. Overall One-Cycle Pruning has the best performance out of all previously mentioned methods [13]. The only other pruning method that comes close is AGP.

In the case of our neural network, pruning scheduling is critical to be addressed in future developments of the project. The idea is to switch to the Libxsmm library layers after a specific point within training. The swap would happen as soon as the improved Libxsmm algorithms would start outperforming the PyTorch dense layers, approximated to be within the range of 50-70 percent sparsity. Within the scope of the design of the Libxsmm kernels described in 3.1-3.3 it may seem currently, that iterative pruning may be the only methodology to follow. This is a false assumption. The Libxsmm speed and efficiency depend on reusing the same sparsity patterns over 100-1000 iterations. When considering how pruning happens within any schedule that is not One-Shot pruning, any additional pruning after the initial pruning step is only the addition of ignored values to the previous set of values (replaced by zero's). Suppose the pruning of the layers is done using One-Cycle pruning, but the sparsity patterns are update after 100-1000 iterations. In that case, there is a possibility of not only keeping the high levels of accuracy while pruning but to also decrease training times through the use of optimized JIT kernels from Libxsmm. Scheduling the pruning patterns updates on an iterative schedule to match the sparsity patterns at certain iterations generated by OCP should then be combined with swapping between dense PyTorch layers and sparse Libxsmm layers. This complicated approach should reduce the training time

while keeping high accuracy levels and using the kernels from Libxsmm to their full potential.

CHAPTER V

TESTING AND RESULTS

Testing was accomplished on two systems. After finding the bug and fixing the issues, the testing of the neural network was conducted on Windows 11 within a Windows Subsystem for Linux on an Intel i9-11900k CPU with sixteen 3.504GHz cores and 64 GB of available RAM. Unfortunately, the system was deemed unstable for speed testing purposes due to processes in the background that could have potentially affected the resources and given uneven results. Therefore, the timings were carried out on an isolated system on Azure servers with Intel Xeon Platinum 8272CL with eight 2.594GHz cores and 16GB of available RAM.

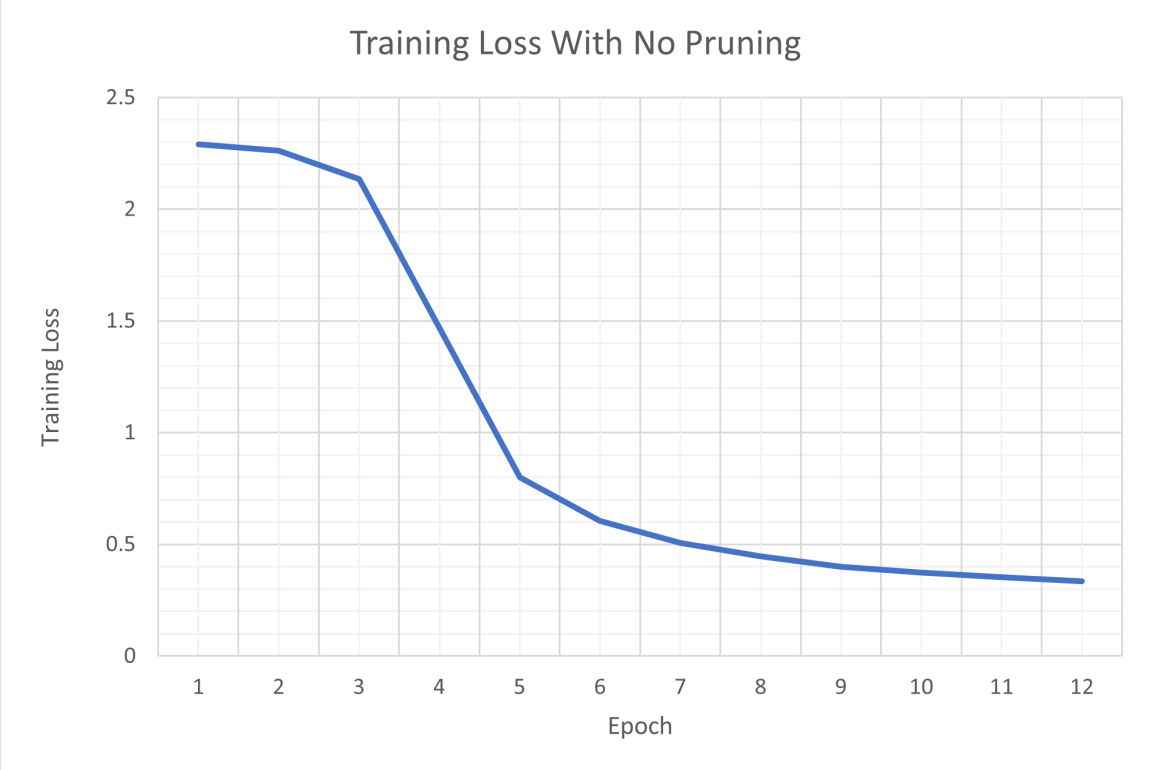

Figure 1. Testing of built neural network before pruning tests

The work done within the parameters of this paper is a continuation of the existing work within the Libxsmm library. As the first step, as described in 3.4,

the bug was found, and testing was done to ensure that the library was stable. Real data was used as the data set for training the neural network for testing purposes. This was done as a matter of progression from the artificially generated dataset used prior within the library. The MNIST data set [18] was used as the basis for testing of the stability of the Libxsmm library versus PyTorch. Each training input image was flattened (784 parameters) and sent through a linear layer with an output of 256. The linear layer was used as a means of preparation for the Libxsmm library as it currently only handles the same size input and output. Three more linear layers follow. Each of the three layers is either a Libxsmm sparse layer or a PyTorch linear layer. Each of these three layers are also followed by a ReLU activation function. The final layer is the PyTorch linear layer used for prediction with an output of size ten since the MNIST dataset has 10 possible outcome classes. The neural network is updated based on Cross-Entropy Loss, and Stochastic Gradient Descent (SGD) is used as the optimizer. A run of a non-pruned network was completed in order to get results to show the neural network worked. The results of the run can be seen through the plotting of the loss value within Figure 1.

In order to test if the Libxsmm library could run after the bug was found, a pruning value of 80% was introduced to each of the three middle layers. The neural network was then run with PyTorch pruned linear layers and afterwards ran with sparse Libxsmm layers. As seen in Figure 2, both layer types had relatively similar results staying within 0.0011 of each other. The goal of the first tests was not to achieve high accuracy due to the simplistic nature of the neural network. The goal was to achieve proof that the Libxsmm library can perform in a similar fashion to PyTorch while both were pruned.

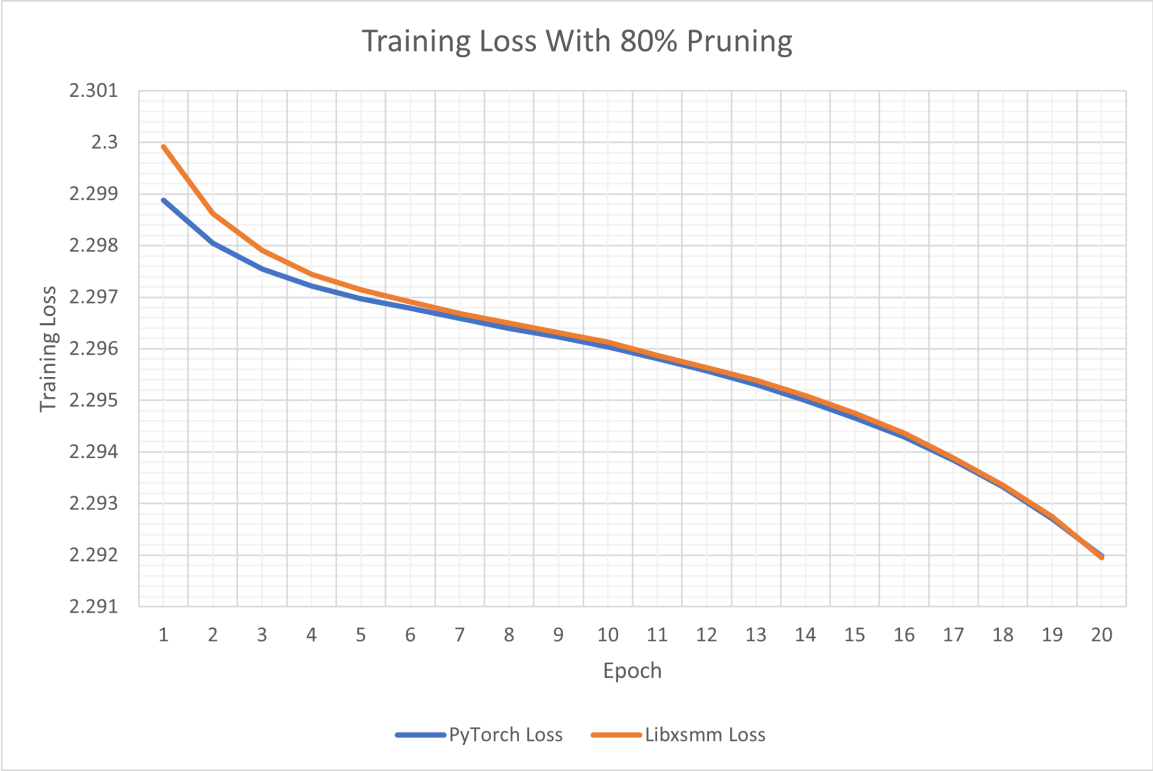

Figure 2. Comparison of loss value of 20 epochs on 80% pruned layers of PyTorch and Libxsmm

Many different tests were run to find the bug and increase the speed of the algorithm. In order to have a proper testing platform for timing the three steps of a single pass through, multiple data sets were generated from the Review IMDB dataset [23]. The 32k word vocabulary preprocessed set was taken as the starting point. The dataset was then sent through an embedding and pooling layer (a separate neural network which did nothing else). It was set up so that it could produce any size output, from a number of parameters to a number of labelled inputs and then saved in a compressed NumPy format. The compressed format allowed for fast loading speeds during testing. The training set consisted of 1024 values within the speed tests, but only 256 were processed. Three sizes of parameters were used in the final stages of timing: 2048, 4096 and 8192, with 95%

Table 1. Improved sections and average improvement over five runs with 8192 parameters

Algorithm Step	Section	Average Original	Average Improved	Speedup
Forward Propagation	3	8.8879556	0.2492186	35.66329158
	6	290.689211	1.0675274	272.3014051
	14	10.1442202	0.0054376	1865.569406
	Total Time	4.7637284	0.0045542	1046.007729
Backward Propagation	3	155.6714024	0.1494578	1041.574293
	6	10.0698044	0.0072006	1398.467405
	17	171.8117866	1.4448802	118.9107489
	Total Time	171.8117866	1.4732346	116.6221501
Update	3	8.7762556	0.2540864	34.54043821
	4	4.6152412	0.0051104	903.1076237
	7	284.6178568	1.089135	261.3246813
	17	16.3454872	0.1275142	128.1856232
	Total Time	316.5985916	3.638552	87.01224872

of the values being pruned. The neural network was reduced to just one processing layer, which was timed internally for Libxsmm cases and one output layer. The layer had an exact input size of the number of parameters presented within the data. In order to determine the areas of focus for improvement, each of the three sections, forward propagation, backwards propagation, and the weight updates, were split up into initially four sections that correlated with the allocation of data, the transformation of data, kernel creation and kernel execution. Unfortunately, this methodology didn't bring enough insight. As a result, the forward pass was divided into 15 sections, and the backward pass and update were divided into 17 sections. Each bottleneck was treated one step at a time from this point onwards.

As seen in Table 1, a total of 10 sections were addressed. The treated section were all using PyTorch C++ API to access or interact with data in tensors. The API that was used originally is intended for an easy way of interacting with C++ code using similar formatting to Python but was not optimized. In order to avoid the API, the structures were directly addressed through flat pointers.

Table 2. Average time taken to execute a single pass over five runs

	Execution Time (s)		
Number of Parameters	**PyTorch**	**Original**	**Improved**
2048	0.05687	57.76065	0.53287
4096	0.19420	149.1451566	1.9170886
8192	0.61506	799.42606	7.66747

In two cases, the flat pointers addressed five-dimensional structures (section 6 of the forward pass and section 7 of the update). The next step is to address kernel creation, as it is currently the largest bottleneck. Kernels are created at every call of the algorithm, and the retainment of kernels is crucial to the speedup of the algorithm.

Table 3. Comparison of speed over different implementations

	x Times SpeedUp		
Number of Parameters	**PyTorch over Original**	**PyTorch over Improved**	**Improved over Original**
2048	1015.6407	9.3698	108.3953
4096	767.9916	9.8716	77.7977
8192	1299.7570	12.4663	104.2620
Average Speedup	1027.7964	10.5692	96.8184

Table 2, shows the time it took for each of the algorithms to execute with 95% sparsity. From Table 3, it can be seen that even though an x96.8 times speed up was achieved, the improved algorithm is ten times slower than PyTorch. The algorithm can outperform PyTorch, but it will require addressing the issue of reusing structure with each pass. Currently, all structures are recreated. Avoiding the reinitializations will reduce processing speed considerably.

CHAPTER VI

CONCLUSION

The work completed in this paper in the direction of accelerating the Libxsmm sparse layer algorithms to outperform the PyTorch dense layer implementations has delivered a x96.8 times decrease in time to the original code while maintaining similar values of loss to the PyTorch dense layers. With the current increase in speed of the algorithms, there is a lot of potential for improvement. There are a multitude of approaches that can be used in the attempts of improving the code to be able to outperform PyTorch. The choice should be made through finding the bottlenecks through the current timing scheme. With the current state of the algorithm, the next step for improvement is to develop a method of retaining structures and kernels from one epoch to another. The bug that was initially the biggest concern in the project has been found and corrected.

The improvements to the Libxsmm code, unfortunately, still are not close enough to PyTorch values to say definitively how much of a speed up future development to the code could have. Assuming the Libxsmm algorithm can outperform PyTorch, the possibilities and potential of using the sparse algorithm within different pruning schedules are very promising. Different strategies such as iterative pruning, Automated Gradual Pruning and One-Cycle pruning should be tested. The kernel implementations are essential to consider since they are designed to reuse the same sparsity patterns over 100-1000 iterations. A possible solution is to have separate schedules for pruning and for updating the kernels. This solution entails pruning a network, possibly with One-Cycle pruning, and having the kernels updated to catch up with the sparsity levels. Another area to explore is the swap

between dense and sparse layers at a certain level of sparsity. The swapping of layer implementations could considerably impact performance since dense layers can outperform substantially any kind of sparse layer with sparsification densities of below 50%. Research must be done into the comparison of the performances depending on sparsity if a swapping approach is to be used at any point.

Future work centers around the improvement of the Libxsmm algorithm using the optimized kernels to the point where it outperforms the PyTorch dense layers for seventy percent sparsity and above. Once these improvements are complete and all other possible improvements that were mentioned within 3.5. are attempted, pruning scheduling can start. Overall, the work done is very promising.

REFERENCES CITED

[1] Aketi, S. A., Roy, S., Raghunathan, A., and Roy, K. Gradual channel pruning while training using feature relevance scores for convolutional neural networks. *IEEE Access 8* (2020), 171924–171932.

[2] Baydin, A. G., Pearlmutter, B. A., Syme, D., Wood, F., and Torr, P. Gradients without backpropagation, 2022.

[3] Breuer, A., Heinecke, A., and Cui, Y. Edge: Extreme scale fused seismic simulations with the discontinuous galerkin method. In *International Supercomputing Conference* (2017), Springer, pp. 41–60.

[4] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems 33* (2020), 1877–1901.

[5] Castellano, G., Fanelli, A., and Pelillo, M. An iterative pruning algorithm for feedforward neural networks. *IEEE Transactions on Neural Networks 8*, 3 (1997), 519–531.

[6] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition* (2009), pp. 248–255.

[7] Dettmers, T., and Zettlemoyer, L. Sparse networks from scratch: Faster training without losing performance, 2019.

[8] Fei-Fei, L., Fergus, R., and Perona, P. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer Vision and Pattern Recognition Workshop* (2004).

[9] Frankle, J., and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks.

[10] Frankle, J., Dziugaite, G. K., Roy, D. M., and Carbin, M. Stabilizing the lottery ticket hypothesis, 2019.

[11] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015.

[12] Heinecke, A., Breuer, A., and Cui, Y. Tensor-optimized hardware accelerates fused discontinuous galerkin simulations. *Parallel Computing 89* (2019), 102550.

[13] Hubens, N., Mancas, M., Gosselin, B., Preda, M., and Zaharia, T. One-cycle pruning: Pruning convnets under a tight training budget, 2021.

[14] Krizhevsky, A., Nair, V., and Hinton, G. Cifar-10 (canadian institute for advanced research).

[15] Krizhevsky, A., Nair, V., and Hinton, G. Cifar-100 (canadian institute for advanced research).

[16] Laurent, C., Ballas, C., George, T., Ballas, N., and Vincent, P. Revisiting loss modelling for unstructured pruning, 2020.

[17] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature 521*, 7553 (2015), 436–444.

[18] LeCun, Y., Cortes, C., and Burges, C. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist 2* (2010).

[19] LeCun, Y., Denker, J., and Solla, S. Optimal brain damage. In *Advances in Neural Information Processing Systems* (1989), D. Touretzky, Ed., vol. 2, Morgan-Kaufmann.

[20] Lewis, M., Bhosale, S., Dettmers, T., Goyal, N., and Zettlemoyer, L. Base layers: Simplifying training of large, sparse models. In *Proceedings of the 38th International Conference on Machine Learning* (18–24 Jul 2021), M. Meila and T. Zhang, Eds., vol. 139 of *Proceedings of Machine Learning Research*, PMLR, pp. 6265–6274.

[21] Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets, 2016.

[22] Lin, J., Men, R., Yang, A., Zhou, C., Ding, M., Zhang, Y., Wang, P., Wang, A., Jiang, L., Jia, X., et al. M6: A chinese multimodal pretrainer. *arXiv preprint arXiv:2103.00823* (2021).

[23] Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies* (Portland, Oregon, USA, June 2011), Association for Computational Linguistics, pp. 142–150.

[24] Mao, H., Han, S., Pool, J., Li, W., Liu, X., Wang, Y., and Dally, W. J. Exploring the granularity of sparsity in convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (July 2017).

[25] Molchanov, P., Mallya, A., Tyree, S., Frosio, I., and Kautz, J. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2019).

[26] Pietroń, M., and Żurek, D. Speedup deep learning models on gpu by taking advantage of efficient unstructured pruning and bit-width reduction, 2021.

[27] Quian Quiroga, R., and Kreiman, G. Measuring sparseness in the brain: comment on bowers (2009).

[28] Zhu, M., and Gupta, S. To prune, or not to prune: exploring the efficacy of pruning for model compression, 2017.