CS C267 Final Project

Pruning DNNs Using Weight Pruning and Deep Compression

Abel Yagubyan and Gurkaran Goindi

Abstract

In a number of applications of machine learning, especially those that involve having a very large number of features to correctly detect, such as image-recognition, Deep Neural Networks (DNNs) are increasingly useful to solve the problem. But as the tasks get more complex and the number of features increase, the size of DNNs becomes ever larger as they continue to increase in accuracy. Building on the work done on SensAI to decouple RNNs and parallelize their computation to increase efficiency for their use in parallel computing, we propose Weight Pruning as a means to reduce DNN model size by removing redundant weights. We intend to optimize this by customizing the pruning to the underlying hardware by leveraging SIMD units for hardware optimization. We also intend to leverage SIMD-aware Node Pruning, in addition to weight pruning, for optimizing our DNN. Weight pruning reduces DNN model size and the computation by removing redundant weights. Using SIMD-aware weight pruning for moderate parallelism hardware along with node pruning for high parallelism hardware, we will attempt to produce a pruning method that would be beneficial to the performance speedup of large DNN models along with its fraction of weights pruned. The implementation will then be compared with the original DNN model along with a pruned DNN model using Deep Compression as the main technique. Our work is largely inspired by the work done on Scalpel by Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahalke of the University of Michigan.

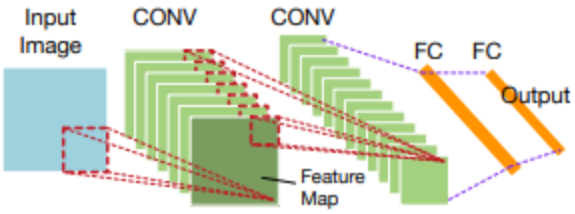
Introduction

A DNN consists of two major types of layers: fully-connected layers and convolutional layers. Fully-connected layers perform matrix-vector multiplication, whereas convolutional layers perform matrix-matrix multiplication. In large DNN models,

there is a lot of internal redundancy, which makes it possible to have a high accuracy, but often at the expense of unnecessary and redundant operations. The idea behind Weight Pruning is to reduce this internal redundancy and produce a leaner, slimmer model that is able to perform with a similar level of performance, but with less computational overhead. Our process of weight pruning involves measuring the importance of each weight and removing unnecessary or unimportant weights, thereby resulting in memory storage and computational reductions. We benchmark our process of weight pruning and SIMD-aware node pruning to Deep Compression. Scalpel was built by running this process on three levels of parallelism – low-level on microcontrollers, and moderate/high-level on CPUs and GPUs. Since we will be running our experiment on Cori, which as far as we know does not use any microcontrollers we can use, we will be running our experiments on the CPU and GPU hardware devices available to us.

Background

Since the fundamental object within Deep Neural Networks is the neuron, the neuron essentially combines the weighted inputs along with the bias values to get a finalized output using an activation function of choice. Particularly within DNNs, the models use a combination of Convolutional layers along with fully-connected layers to produce a multi-layer network that can be used for various applications (i.e. image classification, Regression-based problems, etc.). An example of a DNN structure is displayed in the image below, where the fully connected layers perform matrix-vector multiplication actions, whilst the convolutional layers perform matrix-matrix multiplication actions.



As displayed in the image above, the fully connected layers consist of input values that are connected to every neuron, whilst the convolutional layers consist of a stack of some 2-dimensional matrices that are used as feature maps. Just out of interest, after having pruned the weights and having turned the dense weight matrix into a sparse matrix, the output elements perform the following operation:

$$y_i = f\Big(\sum_{w_{ij}^{sparse} \neq 0, j \in [0, n-1]} w_{ij}^{sparse} x_j\Big)$$

After having implemented the weight pruning onto the DNN, the fully connected layers would need to perform sparse matrix-vector multiplication, whilst the convolutional layers would need to perform sparse matrix-matrix multiplication. The following image displays a good summary of how the approach of the weight pruning would modify the weight matrix of the model:

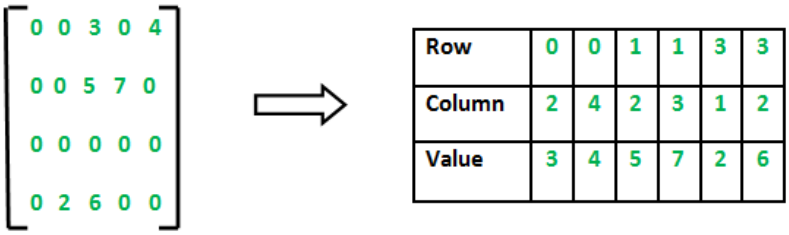
Dense Matrix

1	2	31	2	9	7	34	22	11	5
11	92	4	3	2	2	3	3	2	1
3	9	13	8	21	17	4	2	1	4
8	32	1	2	34	18	7	78	10	7
9	22	3	9	8	71	12	22	17	3
13	21	21	9	2	47	1	81	21	9
21	12	53	12	91	24	81	8	91	2
61	8	33	82	19	87	16	3	1	55
54	4	78	24	18	11	4	2	99	5
13	22	32	42	9	15	9	22	1	21

Sparse Matrix

1	.	3	.	9	.	3	.	.	.
11	.	4	.	.	.	.	.	2	1
.	.	1	.	.	.	4	.	1	.
8	.	.	.	3	1	.	.	.	.
.	.	.	9	.	.	1	.	17	.
13	21	.	9	2	47	1	81	21	9
.	.	.	.	.	.	.	.	.	.
.	.	.	.	19	8	16	.	.	55
54	4	.	.	.	11	.	.	.	.
.	.	2	.	.	.	.	22	.	21

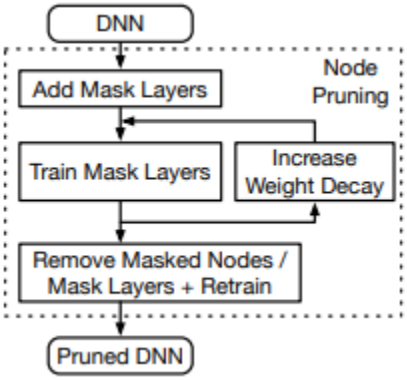
Lastly, we use the sparse matrix to store into a compressed sparse rows (CSR) format in the end:



Suggested Solution

Our implementation will use a customized pruning algorithm for different particular levels of parallelism within the architecture of the computer that we will run on. We particularly identified two different levels of parallelism within the architecture:

1. For high-level parallelism hardware such as GPUs, we equip the node pruning method to attempt to remove the redundancy in DNNs. Particularly for DNN layers, past research displays that the traditional weight pruning algorithms appear to decrease its performance in high-parallelism hardware mainly due to the fact that the matrix sparsity using these algorithms would essentially deteriorate the computational performance of the DNN layers. To accompany this particular issue, our implementation employs the node-pruning algorithm [3] that removes entire nodes rather than the weights directly (removing nodes in DNNs reduces the size of the layers, however it does not produce any sparsity issues like other traditional weight pruning algorithms). A great summary of the node pruning algorithm is illustrated below:



We initially add mask layers on every DNN layer, which will then be used to train these mask layers (and adjust the weight decay, if necessary). After having done so, we also enforce an L1 regularization constraint on the number of nodes that we have removed to avoid any outliers within our process. Lastly, we remove the masked nodes, the mask layers, and then retrain the entire network for a final process. Due to our limited knowledge within DNNs, we enforce the constraints and implement the equations of dropout ratio along with weight decay using the ones mentioned in the Scalpel implementation.

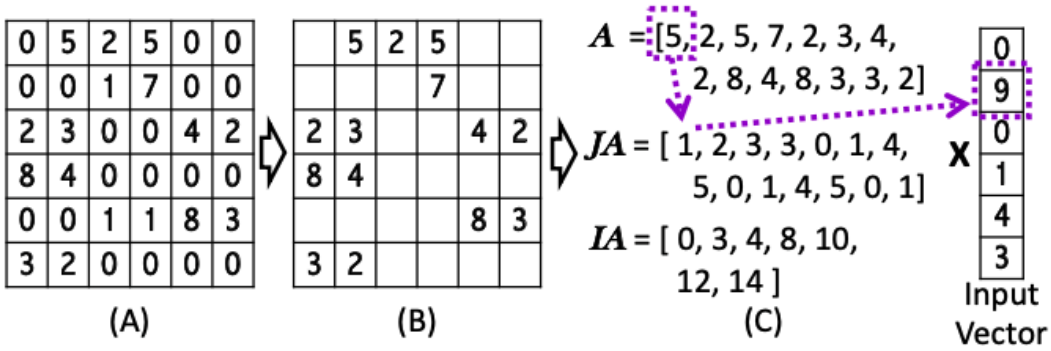
2. To accommodate for moderate-level parallelism hardware such as CPUs, we will need to introduce an idea that particularly is useful in low-level parallelism hardware such as microcontrollers, for which they are low-power processors that require a shallow pipeline, no cache, along with a very limited storage. Particularly for this hardware (microcontrollers, that is), a great implementation idea we found was SIMD-aware weight pruning. We begin the weight pruning method by initially grouping up all the weights by dividing them into subsets of aligned groups that contain the exact same size as the supported SIMD width of the machine. Moreover, after having grouped them together, we'll calculate the root-mean-square for every group and decide which one is not necessary by determining a threshold root-mean-square and pruning any of the weight groups that have a root-mean-square value below the required threshold. After having finished the aforementioned steps, the DNN model would then be retrained with the non-pruned weights until it performs better than the original accuracy of the non-pruned DNN model. Having introduced this idea, we can blend low-level parallelism pruning along with high-level parallelism pruning to take care of moderate-level parallelism hardware. As previously mentioned, since DNNs are broken into fully connected layers and convolutional layers (for which fully-connected layers perform matrix-vector multiplication with a batch size of 1, whilst as convolutional layers perform matrix-matrix multiplication with a batch

size of 1), we can use its properties to deduce that SIMD-aware weight pruning is a better option for fully connected layers, whilst as node pruning can be applied to convolutional layers. In general, we will initially apply node pruning to remove redundant nodes in convolutional layers, and then use SIMD-aware weight pruning to prune redundant weights from fully connected layers (SIMD-aware is particularly used in fully connected layers due to the fact that the memory footprint of the computation decreases (reduces number of indices), whilst as node pruning is used for convolutional layers to get rid of matrix sparsity, as previously mentioned).

Challenges

There are particularly two problems/challenges that we were able to identify in DNN models that we should be wary of and a third that was specific to our project:

1. Although the weight pruning approach decreases DNN model sizes and “MAC” (multiply-accumulate) operations, the approach could unfortunately worsen the DNN computation performance. This is because the sparse weight matrix has a high overhead in terms of too much extra data to record the sparse matrix format. The Compressed Sparse Rows (CSR) format uses three 1-D arrays to store and **m** x **n** matrix, as shown in the figure below.



(A) Dense weight matrix; (B) Sparse Weight Matrix (C) CSR format for sparse matrices

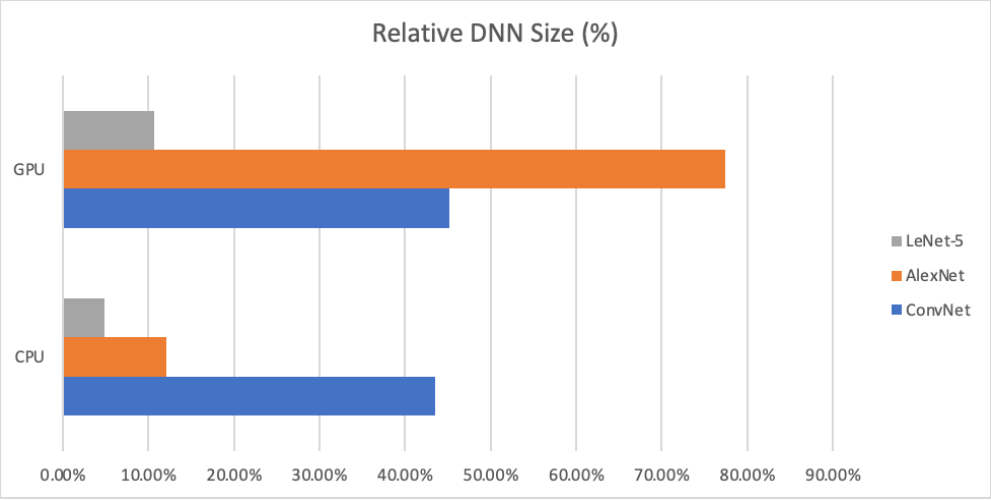
2. The other major challenge was that weight pruning can adversely affect DNN computation performance (primarily due to matrix sparsity in high-level parallelism hardware).
3. The third challenge was that Scalpel's work involved 3 levels of hardware – low-level parallelism with microcontrollers, moderate level parallelism with CPUs, and high level parallelism with GPUs. Since we don't have access to any microcontrollers Cori might have, we had to adapt our DNN parallelism technique to work with just CPUs and GPUs (hence, primarily working with moderate and high level parallelism hardware implementations, respectively).

Experiments

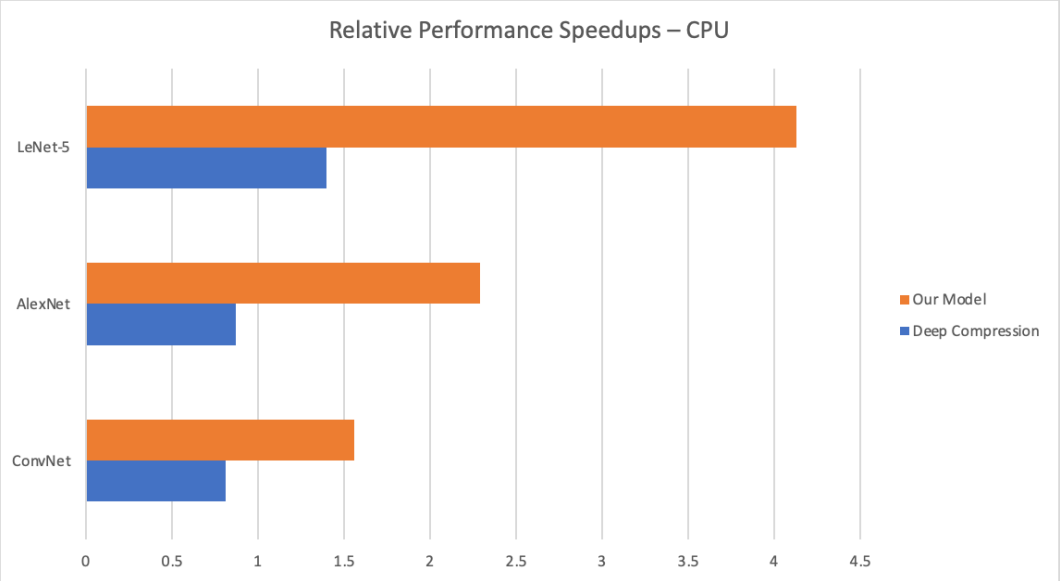
To evaluate the usefulness of the mentioned implementation, we ran our model with the LeNet-5, ConvNet, and AlexNet DNNs on CPU and GPUs, which was then compared to the Deep compression implementation (as it is a great compression implementation) particularly relative to its performance and memory saveup.

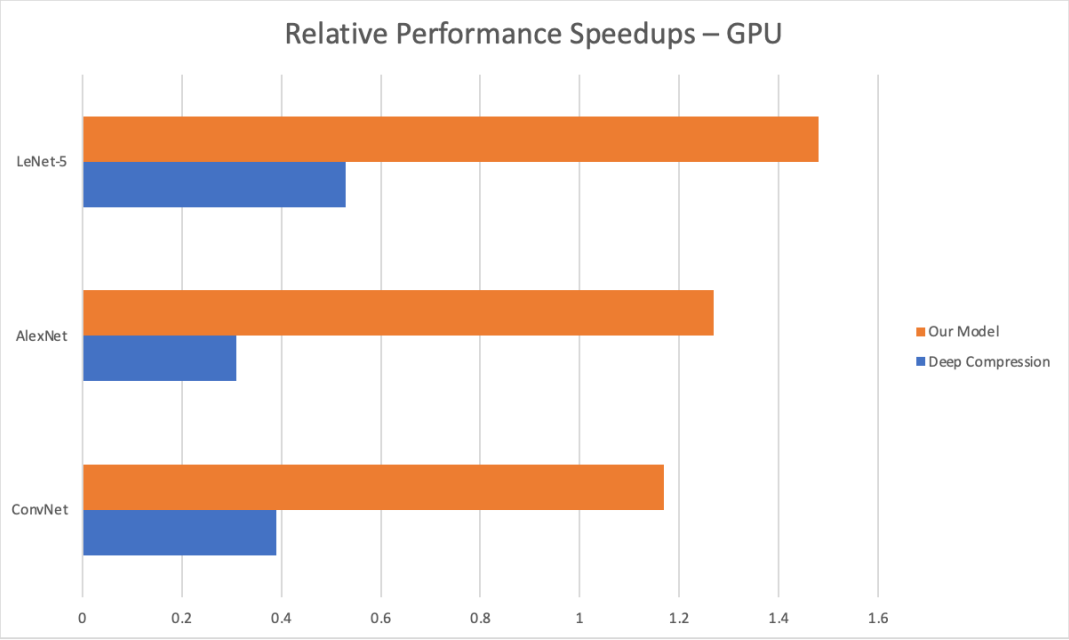
Results

We ran our model with the LeNet-5, ConvNet, and AlexNet DNNs on CPU and GPUs.



As visible in the relative DNN size plot above, our implementation saves a significant chunk of memory from the original DNN models (i.e. AlexNet is ~78% of its original size after having run through node pruning in the GPU), pretty nice results!





Lastly, as seen in the plots above, the relative performance speedups of our model vs Deep compression in the GPU and CPU is significantly better, which is a great outcome for us!

Reproducibility

The code was produced primarily by using python files (and the torch package) to initialize the models, prune the models (node and SIMD-aware), and finally run the models to reproduce the statistics of the fraction of the weights pruned along with the time taken to execute the model on the aforementioned well-known and defined models, found in the “Experiments” section. To recreate our outputs, please visit the following Github repo and run the source code for each model:

- https://github.com/Abelo9996/CSC267_Project

Conclusion

In conclusion, we attempted to implement DNN pruning for different hardware platforms based on their parallelism. Particularly within our implementation, we implemented a combination of node pruning and SIMD-aware weight pruning to introduce a pruning method that attempts to speedup (and save memory) DNNs particularly with moderate-level parallelism. On the other hand, we also implemented node pruning to speedup (and save memory) DNNs particularly with high-level parallelism. After having implemented these algorithms upon the LeNet-5, ConvNet, and AlexNet DNNs on CPU and GPUs, we found that the implementation performs very well in comparison to Deep compression (and saves a significant chunk of memory of DNNs via weight pruning and node pruning).

Citations

1. Han, Song, Huizi Mao, and William J. Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." *arXiv preprint arXiv:1510.00149* (2015).
2. J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), 2017, pp. 548-560, doi: 10.1145/3079856.3080215.

3. T. He, Y. Fan, Y. Qian, T. Tan and K. Yu, "Reshaping deep neural network for fast decoding by node-pruning," 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2014, pp. 245-249, doi: 10.1109/ICASSP.2014.6853595.
4. Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.
5. R. LiKamWa, Y. Hou, Y. Gao, M. Polansky and L. Zhong, "RedEye: Analog ConvNet Image Sensor Architecture for Continuous Mobile Vision," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016, pp. 255-266, doi: 10.1109/ISCA.2016.31.