Hardware Optimizations in Pursuit of Fast, Energy Efficient Deep Neural Networks

Charlie Caron, Oliver Newland, Nikhil Bhatia-Lin

Machine learning is increasing in popularity and usage, and with that rise comes the demand for better performance and power usage. Hardware optimizations are a potential option to improve performance in neural networks. In this paper, we examine three hardware optimizations that either decrease the amount of storage used by the neural network, directly lower the amount of power used by the system, or indirectly reduce power by lowering the number of operations performed by the neural network. These optimizations include lowering SRAM voltage, pruning non-useful weights, and quantizing weights. Our results find that by allowing for increased faults in the SRAM, prediction accuracy remains relatively unaffected. We find that about half of the weights can be pruned with negligible prediction accuracy losses. Finally, we find that weights can be quantized to 3 bits while losing less than 1% in prediction accuracy.

Introduction.

In the last decade, machine learning technology has improved drastically, allowing for increased applications in many different fields. Specifically, machine learning is becoming ever-present in image processing, natural language processing, and computer vision, to name only a few uses. Machine learning is especially good at creating predictive models and solving problems previously thought impossible, and will therefore continue to have significant value in the foreseeable future. With this increasing demand for efficient artificial intelligence technology comes the need for energy and performance optimizations. In particular, there is room for growth in the hardware space.

Neural networks, a specific tool used in many types of machine learning, are notably computationally intensive. They operate largely in parallel and require many matrix multiplies with matrices which can be as large as millions of rows. As a result, computer architects are taking up the challenge of creating systems more favorable to machine learning operations and are attempting to optimize neural networks to be more power-efficient.

In the course of this paper, several potential hardware optimizations are outlined that improve the performance and power output in systems using neural networks. Specifically, these operations include reducing SRAM voltage to an ideal fault tolerance level, pruning weights in the neural network that have a negligible effect on prediction accuracy, and quantizing bits in the weight matrix. These optimizations are inspired by *Deep Learning for Computer Architects* by Adolf et al, however our research both replicates and expands upon their work. The goal of our research was to identify the ideal point where these optimization techniques can be applied to save power without sacrificing prediction accuracy.

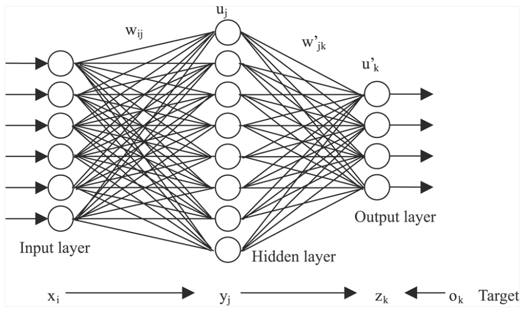

Fig. 1. Example Neural Network. Layers are connected by a series of weights that calculate a prediction.

Background.

Neural Networks Neural networks make predictions by taking a series of inputs, or features, that describe an image, an environment, or anything else that can be represented by a numerical vector, and transforming those inputs to a series of values, or neurons, that make up a layer. The output of the first layer then become the input of the next layer, and so on and so forth until a final layer consists of neurons that each represent a classification. The final neuron with the greatest value is the network's prediction for that specific input.

Typically, every input in a layer contributes to the value of every output, and by training the neural network we can discover how much each input should affect each output. How much they affect is represented by a weight, typically between -1 and 1, that signifies the connection between a specific input and a specific output value. A neuron's final value is the sum of each input multiplied by the weights that link the inputs and the neuron (generally this sum is then bounded by a softmax or sigmoid function). So to produce an output for a layer with 10 inputs and 100 outputs, there are a 1000 weights that must be loaded, 1000 multiplications, 100 summations, and 100 softmax-like calculations that must be run.

Neural networks range in size and complexity, but it is quite common for a network to have millions of weights across 2-3 or more layers. Their complexity allows them to generally outperform other machine learning techniques, but at the cost of using more memory and performing many more arithmetic operations.

Models train by comparing predicted results and desired results, and then back propagating small changes to weights throughout the network for 1000s of training inputs. This is even more computationally expensive, but our research is limited to the inference side of neural networks. Our optimizations are designed to be used on a system running a fully trained network, in that they generally remove unnecessary information. For an network in training, one cannot safely determine what information is necessary or not.

Specifically, we explore their application to convolutional neural networks (CNNs), a class of deep neural networks (DNNs), or networks with many large layers and specialized layers. CNNs' convolutional layers make them adept at classifying images. In short, rather than a test image being inputted as a list of pixel values, the image is broken down into overlapping sections, and descriptive features are extracted from those sections and pooled together. These features are then fed into a large neural network to produce classifications. Because the important details in a picture are typically not as granular as single pixels, preprocessing by chunks leads to far more accurate results. CNNs are needed for everything from text transcription to automated car sensors, so accuracy and speed are both highly valued. This makes them a good candidate for hardware optimizations. Our work is specific to the normal neuron layers that are common to all CNNs, rather than the varied and complex preprocessing layers.

Static Random Access Memory (SRAM)

SRAM is the type of memory that is typically used in a CPU's caches. Unlike dynamic random access memory, used in a computer's memory, it does not need to be refreshed as long as it is powered. Still, when running on a lower than optimal voltage, SRAM is vulnerable to bit faults. Depending on how low the voltage is scaled, SRAM cells will fail at a predictable rate. The lower the voltage, the higher the probability of a bit error. Voltage scaling can be used to save energy, but only if the applications is able to withstand faults in its data. Our work with SRAM revolves around investigating how many faults a neural network can withstand.

Even with millions of weights, a reasonably sized neural network can fit comfortably in a typical L3 cache. For a single use piece of hardware, say a license plate reading sensor, those weights might only ever refresh as often as the cache is flushed. In fact, the hardware would likely be specifically designed to hold the entire network in its cache. Thus, any faults that occur at a certain voltage will persist within the cache for as long as that data is not reloaded from main memory. Our research explores how often the SRAM data needs to be refreshed in order to prevent the performance degradation caused by accumulating bit faults. When a bit fault occurs, one of several mitigation strategies can be employed. We are concerned with three: doing nothing and allowing the bit to flip, assigning the faulted bit the same value as the data word's sign bit, and assigning the faulted bit zero, which is our own, neural network specific solution. Other notable strategies include zeroing out the whole word, but that strategy yielded poor performance (Reagen et al.) and was not experimented on.

Floating Point Format

The weights in a neural network are generally stored as floats that range from -1 to 1, but most often quite close to 0. In order to understand how bit fault mitigation strategies might affect a word and how quantizing a float might affect its precision, a close look at floating point format is needed.

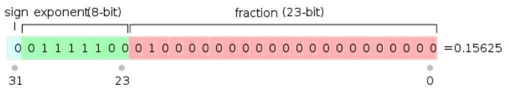

Fig. 2. Floating point machine representation. Three sections include the 1-bit sign, 8-bit exponent, and 23-bit significand (mantissa).

A 32 bit float can be broken down into 3 parts. The first is the leading bit, which determines the sign of the decimal. Next is 8 bits that determine the exponent of the decimal. That exponent is determined by taking the integer value of those 8 bits (between 0 and 255), and subtracting 127, the difference being the power of 2 of the float value. Thus, 00000001 is 2^{-126}, while 11111110 is 2^{127}. 11111111 signifies NaN, while 00000000 signifies a signed 0. The remaining 23 bits make up a fraction-based significand with an implied leading bit of 1.

This format gives floats a few interesting behaviors when we change particular bits. First, if we zero out all of the significand, we still keep the implied 1. This means that the number still has approximately the same value as before, as long as the exponent bits are intact. For our applications, 0.0000001 is not that different than 0.000000123892. Similarly, flipping bits in the significand has way less effect on the magnitude of a decimal than flipping exponent bits. Second, if one takes a float less than zero and changes one of the exponent bits, it can only become positive if a 0 is changed to a 1. If a 1 is changed to a 0, then it will become more negative, moving the value of the float closer to 0.

These properties allow us to do significant damage to a given weight represented in float format, without eroding most of its significance. This allows us to maintain accuracy in our neural networks while performing unsafe optimizations.

Related Work. Our primary source of inspiration is the work of Brandon Reagan et al. in their report *Deep Learning for Computer Architects*, in which Reagan et al. provide an overview of recurrent and convolutional neural networks, detail the distribution of operations most commonly found in cutting edge benchmarks, and present their Minerva hardware and software simulation tool. From Deep Speech to AlexNet to Deep-Q Reinforcement learning, they reported that multiplication and other arithmetic operations constitute from 35-89% of neural network workload, making these operations prime targets for optimization. They go on to note that simply throwing GPUs at the problem is a finite solution, in that there is not much more we can do to improve that technique. Instead, they recommend "unsafe" optimizations that throw away a small (often negligible) amount of information and accuracy for a large boost in inference time and power consumption. Those optimizations are quantization, in which we restrict our values to a less granular set of numbers, pruning, which eliminates multiplications with weights at 0 or close to 0, and fault rate adjusting, in which the voltage running through the SRAM is reduced at the cost of more frequent faults. In our work we replicate the work of Adolf et al. and expand. Specifically, we implement a previously unincluded bit masking method, analyze the effect quantization has on prediction accuracy, and find the exact threshold for pruning weights that still yields a favorable accuracy.

Research on quantization in neural networks has been conducted in the past, specifically, an outline of how to implement a simple value quantization in DNNs is provided by Philipp Gysel et al. in their paper Hardware-Oriented Approximation of Convolutional Neural Networks. They describe Ristretto, a model approximation framework that compresses existing CNNs by turning their 32 bit floating point values into 8 bit fixed point values, at only a 2.3% accuracy loss. The project is open source and was maintained until a few months ago. Ristretto is built on top of Caffe, so it is not set up to integrate with the other optimizations experimented over. However, it provided an excellent guide for how to implement such approximations in our own work.

Han et al also conducted research on quantization and weight pruning in neural networks. The authors address the issue of very large neural networks, which consume too much storage and energy on a device. This makes them, for example, too unwieldy to operate on most mobile devices. Their solution is Deep Compression, a three-part optimization, including pruning, quantization and weight sharing, Huffman coding. In the pruning step of the compression, the neural network trains and then weights below a certain threshold (these weights have very little effect on the final outcome of the neural network) are removed. The network is then retrained and compressed into a compressed sparse row to reduce its size further. In the weight sharing and quantization step of the process, weights are clustered by similarity and then reused for each similar connection within the same layer. In their example, 16 weights can be reduced to four weights by clustering and then quantizing down to a much smaller number of bits. This step both reduces the number of weights and the size of each weight. The overall effect of this compression is a 35x to 49x decrease in storage size.

Methodology. MNIST Convolutional Neural Network For our experiments, we created a convolutional neural network using PyTorch. This neural network featured two convolutional layers, a drop-out layer, and two non-convolutional layers with 3,136,000 and 10,000 weights. The hyperparameters and framework of the network are borrowed from adventuresinmachinelearning.com(**?**). The model was trained on the MNIST dataset, which is a collection of handwritten numbers between 0 and 9. The MNIST dataset is ideal for testing for a couple of reasons – it is not so complicated that tests take days to run, nor is it so simple that our classification task is trivially easy. PyTorch allows us to examine the weights of our pretrained network and make the changes to their values as we wanted, depending on the experiment. Additionally, it is readily available for public use and should make our work easily replicable.

Since our neural network will be testing on the MNIST dataset, which attempts to differentiate between each of the 10 digits, a neural network that is not operating with any level of correctness will have a prediction accuracy of about 10% (the equivalent of predicting randomly). Our trained neural network has a prediction accuracy of about 99.01% when tested on 10,000 images. We take 99.01% as our benchmark for maximum performance, and compare our experimental results to that high watermark.

SRAM Faults Rather than experimenting on real hardware and enduring real SRAM faults, we simulate the effect of low voltage by inserting bit faults at a certain probability every time a weight is used. Over the course of 10,000 test images, the faults accumulate and the network performance degrades as the weights stray further and further from their trained values. We aimed to be able to identify how many images could be processed at a desirable accuracy before the data needed to be refreshed for each different fault rate. One could then find the fault rate for their specific voltage constraint and SRAM hardware and refresh their cache accordingly.

We simulated three different fault outcomes. The first is simply flipping the bit, which is what happens if a real SRAM fault is left unchecked. The second is the most effective strategy described in (Reagan et al), which changes the compromised bit to the value of the sign bit. The third is our proposed solution, which simply changes the bit to 0, no matter the sign bit or the previous value of the bit. For each strategy, we experiment across a variety of fault rates, from an absolute rate of 100% to a rate of 10^{-6}.

It should be noted that our simulation was limited to changing the first 31 bits, as for some reason our network broke when we allowed the sign bit to change. Further work is needed here, but the limitation applied across all experiments, so we believe our results are still valid. From here our analysis assumes that the sign bit cannot change for simplicity.

Pruning To test the effect of pruning on prediction accuracy, we used the MNIST dataset with the same convolutional neural network as before. The theory behind pruning weights is to remove any weights that equal zero, because they mathematically will no effect on the prediction result. Going even further, we can prune weights that are close to zero, because they will have a very small, if any, effect on the final result. In order to test how many weights we could prune while still maintaining high prediction accuracy, we created several tolerance thresholds below which weights would be cut. At each tolerance level we tracked the prediction accuracy to see if it reached unacceptable levels.

In a real-life example of pruning weights, these weights could be entirely removed from the weight matrix in order to save on space and therefore energy. A large proportion of weights are close to or equal to zero, so this energy efficiency increase can be significant. Furthermore, energy can saved and performance increased because less multiplication operations have to occur between weights. In our operation, we simulate pruning by converting all weights below the threshold level

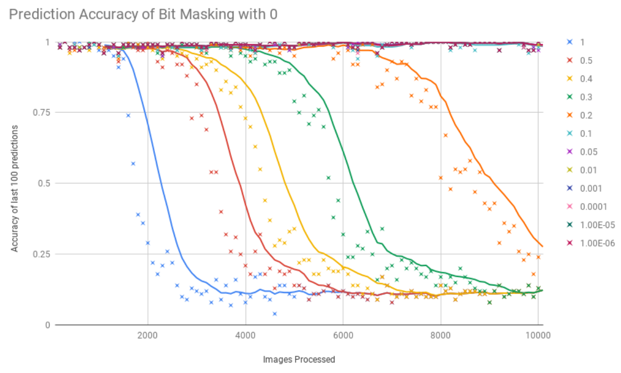

Fig. 3. Prediction Accuracy of Bit Masking with 0.

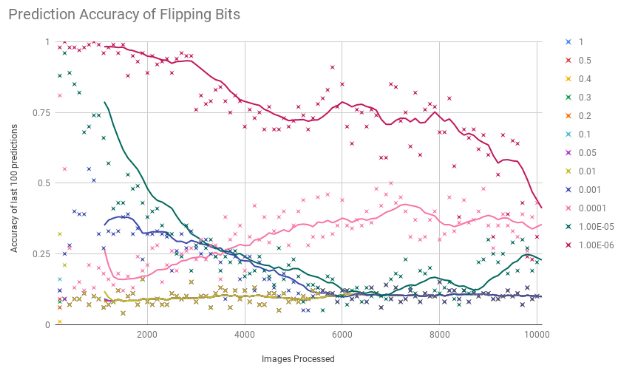

Fig. 4. Prediction Accuracy of Flipping Bits.

to zero. By doing this, we are able to accurately attain the prediction accuracy results that would occur if weights were truly removed.

Quantization In order to experiment on the effect of quantization, we used the same neural network and MNIST dataset as in other experiments. We then ran an iterative set of experiments in which the weights of the already-trained network were bit-shifted by different amounts each experiment. Shifting the bits right and then left allowed us to zero out as many insignificant bits as we wanted, to examine exactly how many bits can be removed before prediction accuracy is affected. This simulates quantization, which stores larger variables in a lower number of bits, with some rounding error. In order to test the effect of quantization on prediction accuracy, we shifted incrementally by 1 bit until prediction accuracy reached unacceptable levels.

One consideration when shifting over a float is the difference in shifting on the mantissa and shifting on the exponent portion of the float. As noted earlier, floats are encoded on the machine in 32 bits, with 23 bits encoding the significant digits of the float, 8 bits encoding the exponent of the float, and 1 bit which holds the sign of the float. These parts of the float are held from left to right as sign, exponent, mantissa. Therefore, when bits are shifted right and then left, the mantissa will be lost first, then the exponent. This is an intentional design choice, as the exponent holds more important information than the mantissa (the difference between $1{*}2^4$ and $1{*}2^{-4}$ is more significant than the difference between $8{*}2^1$ and $2{*}2^1$).

Finally, it is expected that the exponents of the weights will be fairly similar, therefore there may be opportunity for further quantization than just shifting bits. All the weights are within the range -1 to 1, as a result there are likely to be some high-order bits that are always the same and can be discarded. In order to examine this potential effect, we looked at the frequency of high order exponent bits to find any notable trends.

Results. SRAM Faults

As expected, the lower the proposed voltage and the higher the fault rate, the quicker the model's performance declined.

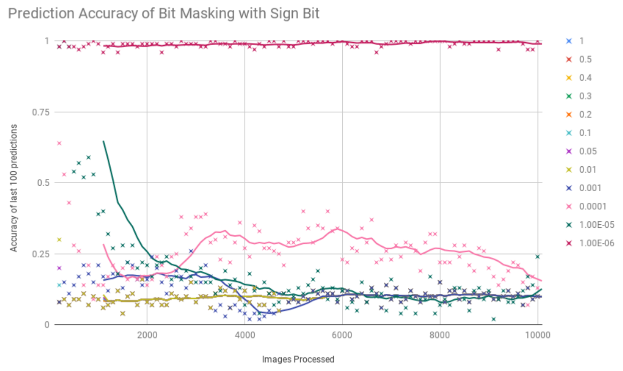

Fig. 5. Prediction Accuracy of Bit Masking with Sign Bit.

Flipping bits, as seen in Figure 4, performed by far the worst. Even at the lowest fault rate, the model's accuracy fell significantly after a 1000 images. This is most likely due to the large changes to a value that can occur when a 0 is flipped to a 1. If a high order exponent bit is suddenly turned on, then the value of that weight can become significantly larger than one, which all but guarantees that the related neuron's activation function is triggered if the weight is positive, or squashed if the weight is negative. Furthermore, allowing the bit to flip means that the weight is always changed in some way, no matter what the flipped bit was originally.

Masking the fault bit with the sign bit produced slightly better results, with the lowest fault rate performing consistently well. However, fault rates higher than 10^{-6} performed similarly poorly to flipping bits, as seen in Figure 5. Unfortunately there is a significant amount of statistical noise when evaluating these two methods, as turning a high order exponent bit on (as described above) can have far reaching impacts, while other flipped bits may be low order and insignificant, or a bit may have already been the same as the sign bit, and therefore unchanged after the fault. Further experimentation is needed to more thoroughly show the advantages of using the sign bit over allowing the bit to flip.

However, despite the statistical noise, these methods are certainly worse than our proposed fault mitigation strategy. Always masking the erroneous bit to 0 yielded a robust model even at the highest fault rate (100%), performing at the high-

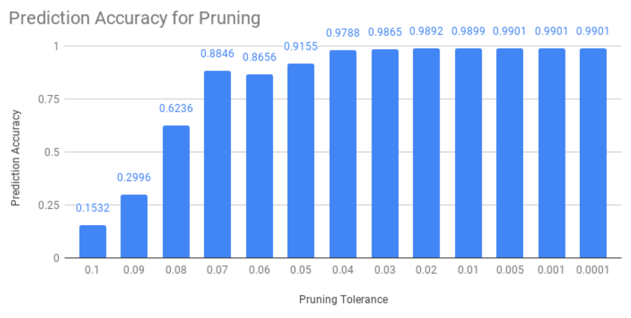

Fig. 6. Pruning Threshold Prediction Accuracy. Prediction accuracy at each tested pruning threshold

est accuracy for 1500 images in the worst case, and performing at the highest level indefinitely for any fault rate below 0.1, as seen in Figure 3. We believe that our method is so much more reliable because the most damage it can do is move a value closer to 0, essentially removing its positive effect, but also preventing it from having a negative effect. Furthermore, for any given bit, there is a decent chance it is already 0, so masking it to 0 changes nothing at all.

Pruning As displayed in Figure 6, our experiments in pruning indicate no significant difference in prediction accuracy between tolerance thresholds of .0001, .001, and .005. This indicates that there are many bits that have negligible effects on the prediction outcome. We see, however, that there is a maximum tolerance threshold of .005 if we are to maintain maximum prediction accuracy of 99.01%. The significant effect that this tolerance has can be seen in the total number of weights pruned - at this tolerance we prune 474,831 weights in our dataset. This number of pruned weights comes out to 15.14% of the total number of weights in our neural network. Our pruning experiments also displayed a predictable fall off in accuracy as the threshold and the number of total weights pruned increases. However, a prediction accuracy of more than 98.9% can still be attained while cutting about 2 million weights, which is much more than half of the weights in the network. Prediction accuracy falls off slowly from a threshold of .01 to .04, however at .05 we see our first major drop off in accuracy as it falls to 91.55%. Continued increases in threshold at the same rate result in even more significant drop offs in total weights pruned and thus in accuracy. This rapid decrease in accuracy as threshold increases is exactly what we expected to see, and demonstrates the limited range at which one can prune while attempting to maintain high levels of prediction accuracy.

Quantization

The results of our quantization experiment demonstrate that most bits in the float representations of weights can be cut without significant losses to prediction accuracy. In fact, when removing the least significant 17 bits of the significand, there is absolutely no loss in prediction accuracy, which is a remarkable result in and of itself. 17 bits is more than half the length of a float, and with more than 3 million weights in the neural network, this is a significant amount of wasted

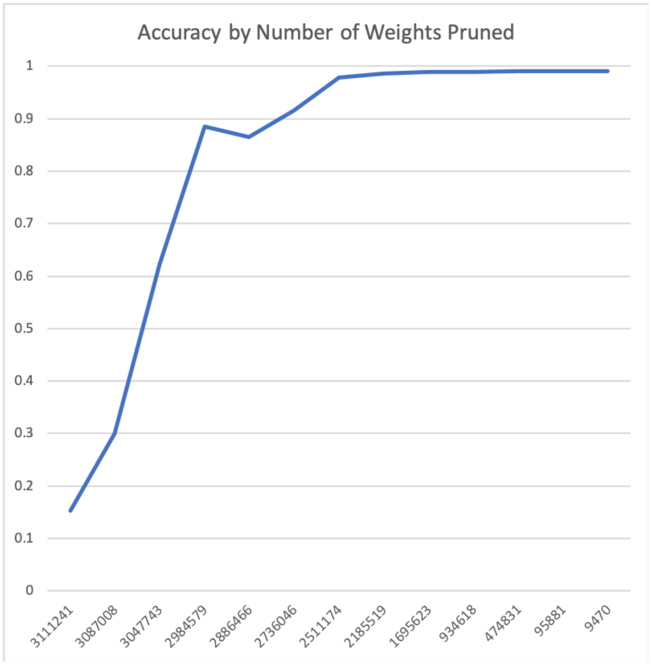

Fig. 7. Prediction Accuracy vs Number Pruned. Total number of weights pruned and corresponding accuracy

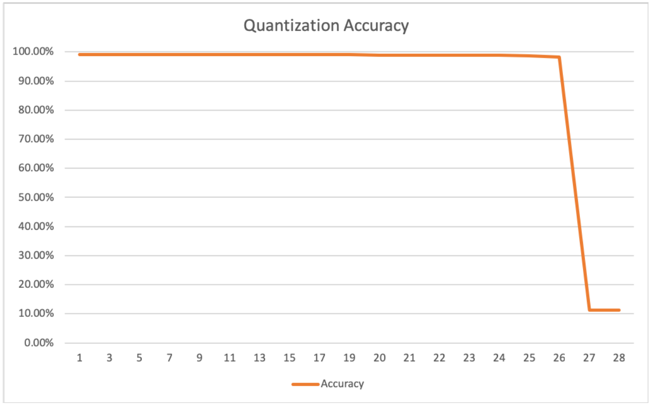

Fig. 8. Quantization Accuracy.

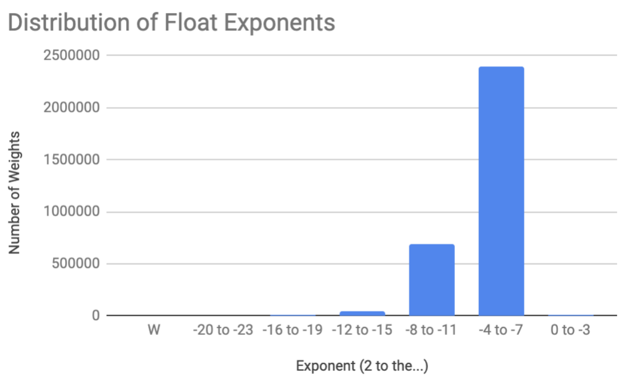

Fig. 9. Quantization Table.

storage. Figure 8 shows that there is no more than a percent loss in prediction accuracy when 26 bits are cut. This means that 6 bits are capable of representing a single weight with a .78% loss in prediction accuracy. It also indicates that the exponent is far more important than the significand. However, once another bit is removed from the float, the neural network's prediction approaches the random level of 10%. This can be explained by the general nature of weights in our neural network. Most weights, by far, are between 2^{-7} and 2^{-4}, which in float representation has the exponent binary value of 011110(00) (the last two values affect whether it is 2^{-4}, 2^{-5}, 2^{-6} or 2^{-7}). When the value is shifted by 27, the exponent value becomes 011100(00), which is equivalent to values of (2^{-12}, 2^{-13}, 2^{-14}, or 2^{-15}). This means that weight values that are fairly large become much smaller, minimizing their effect on the final result. Since more than ¾ of weights are affected, this is very significant to the prediction accuracy. Also significant is the fact that some bits in the remaining 6 bit word are not used for anything meaningful. Three of the six bits are exactly the same in every single weight, meaning they could be removed, leaving just 3 bits (the sign bit, and the last two remaining exponent bits). With this information we can retain all of the relevant information for any weight, and still achieve more than 98% prediction accuracy. These are spatial savings of more than 90%, which is massive.

Conclusion and Future Work. The next obvious step in our future work is to get energy results that match with the optimizations that we attained. This would involve building and testing hardware, but having energy improvements to match our expectations would go a long way in showing how effective hardware optimizations can be in machine learning systems. Specifically, it would allow practitioners to analyze the tradeoffs between energy and prediction accuracy, to make deliberate choices about how to optimize their systems.

Another element we would like to measure is the intersection of optimizations studied in this paper. In the paper by Han et al. they found that by combining quantization and pruning, they achieved better prediction accuracy than by running either separately, we believe that this is a potential path forward in analyzing the hardware optimizations studied in this paper.

Fully outfitted neural networks have a significant amount of deadweight, by simplifying we can save computation cost for as good or near as good accuracy. There are sacrifices in performing unsafe optimizations such as the ones conducted in this paper, but with minimal losses in accuracy, we showed that there are significant spatial and energy gains to be had. Even though neural networks are costly, specific hardware can help loosen the constraints of certain applications. By reducing voltage in the SRAM, quantizing bits, and pruning weights, there are large energy gains to be had. For example, we believe that a system that runs with lower SRAM voltage and prunes and quantizes after training would perform much better energy-wise than a comparable system that is holding on to useless information stored in floats and near-zero weights. As neural networks continue to become more complex and cumbersome, hardware optimization techniques such as these will become increasingly necessary.

Bibliography

1. Gysel P, Motamedi M, Ghiasi S. Hardware-oriented Approximation of Convolutional Neural Networks. ArXiv160403168 Cs. April 2016. http://arxiv.org/abs/1604.03168. Accessed March 8, 2019.

2. Reagen B, Adolf R, Whatmough P, Wei G-Y, Brooks D. Deep Learning for Computer Architects. Synth Lect Comput Archit. 2017;12(4):1-123. doi:10.2200/S00783ED1V01Y201706CAC041

3. Han S, Mao H, Dally WJ. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. ArXiv151000149 Cs. October 2015. http://arxiv.org/abs/1510.00149. Accessed March 8, 2019.

4. Biswas A, Chandrakasan AP. Conv-RAM: An energy-efficient SRAM with embedded convolution computation for low-power CNN-based machine learning applications. In: 2018 IEEE International Solid - State Circuits Conference - (ISSCC). ; 2018:488-490. doi:10.1109/ISSCC.2018.8310397

5. Convolutional Neural Networks Tutorial in PyTorch. Accessed April 8, 2019. https://adventuresinmachinelearning.com/convolutional-neural-networks-tutorial-in-pytorch/