CNN-SLIDE: Sublinear Deep Learning Engine for CNNs

Ishan Chadha
ichadha3@gatech.edu

Raj Janardhan
rjanardhan3@gatech.edu

Manoj Niverthi
manojniverthi@gatech.edu

Abstract

With the rapid rise and proliferation of different forms of visual data, Convolutional Neural Networks (CNNs) have emerged as a powerful deep learning technique applicable across a wide range of tasks, including image classification, object detection, and image segmentation. However, these networks are typically expansive concerning the number of parameters they possess and require expensive hardware, including GPUs and TPUs to train. The purpose of MILF is to accelerate the training time of CNN models in a hardware-agnostic manner. Specifically, this work leverages two previously explored techniques – (1) tensor sketching and (2) locality-sensitive hashing – that have been used to construct a variety of network architectures, such as multi-layer perceptrons, in isolation. This work proposes a unique fusion of these techniques to reduce model training time further. The sketching component focuses on reducing the dimensionality of convolutional kernel filters, while the LSH component lowers the number of filters that are applied. Experimental results showcase MILF's effectiveness in significantly reducing model runtimes. The algorithm achieves $\sim 29\%$ training time speedup on AlexNet versus the usage of filter pruning or sketching unilaterally, demonstrating its potential to accelerate CNN training. Importantly, these efficiency gains come with only marginal decreases in model performance – $\sim 5\%$ training accuracy decrease for AlexNet – emphasizing MILF's ability to strike a favorable balance between speed and accuracy.

1 Introduction

Convolutional Neural Networks (CNNs) have emerged as a groundbreaking paradigm in the field of deep learning, revolutionizing various applications, such as object recognition [4], medical diagnosis [5], and autonomous driving [6]. The significance of CNNs lies in the ability to automatically learn hierarchical representations from data, capturing specific patterns and features. Furthermore, CNNs have demonstrated remarkable transfer learning capabilities, allowing researchers to bring ideas from simulation to reality [7]. Additionally, convolutional networks have shown promise in computing powerful latent representations of complex problems as the dimensions of the network grow in terms of depth, width, and number of kernels; however, this comes with the drawback of being increasingly computationally expensive. Addressing these computational challenges is crucial for making CNNs more accessible and practical for a broader range of applications. This paper mainly focuses on how to algorithmically improve the speed of these convolutional neural networks without sacrificing accuracy.

2 Related Work

2.1 Adaptive Sparsity

In feedforward neural networks, regularization is a family of techniques utilized to decrease variance and increase the bias of a model, allowing for better generalizaiton and less overfitting. One of the

Preprint. Under review.

most prominent regularization techniques for feedforward neural networks is dropout, where penalty terms induce an effective sparsification of the weights by causing neurons to become inactive. Apart from this preventing overfitting the certain neurons, dropout also directly reduces the computational overhead of the network by reducing the number of multiplications that need to be performed. Adaptive sparsification is an extension of this, where a fixed dropout rate is replaced with one that is learned over the duration of model training [8].

2.2 Locality Sensitive Hashing

When we break down the idea of dropout to the granularity of neuron-wise operations, the problem can be viewed as maximization of the inner product between the weight matrix and the input, as a maximal inner product will result in the most neuron activation. A problem that is very similar in construction to maximum inner product (MIPS) is to find if two vectors are very close to each other. We can apply a transform to these vectors prior to comparing them such that if two vectors are found to be close to one another, then the untransformed vectors have a large inner product.

The computation of whether two vectors are close together is often described as nearest neighbor search, calculated in a variety of ways based on the application such as octrees, k-nearest neighbors search, and so on [9]. Exact nearest neighbor search can be expensive, especially as the dimensionality of the search space grows, so an approximate method such as locality sensitive hashing (LSH) is more suitable for this task. To recap, a form of LSH is used where a transform is applied beforehand to alter the search into MIPS, an approach called asymmetric LSH (ALSH), and this approximate MIPS is employed for speeding up dropout computations [10].

A locality sensitive hash uses hash tables to store points such that vectors that are close together have a monotonically increasing probability of colliding with each other within a certain distance bound. K hash functions and L hash tables are used, where concatenating across the K hash tables can be thought of as an "AND" operation, and the concatenation across multiple hash tables can be thought of as an "OR" operation.

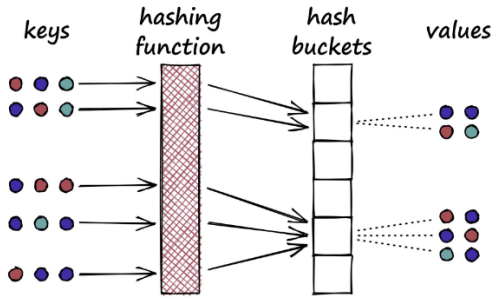

Figure 1: Visual Representation of Locality-Sensitive Hashing

2.3 SLIDE

The current state of the art approach to algorithmic improvements for MLP training speedup is called Sublinear Deep Learning Engine (SLIDE) [1]. SLIDE uses the concept of LSH in its approach, creating several hash tables for the weights in a given layer. For a given input, forward propagation only occurs through weights that hash to the same bucket as the input, indicating a high enough inner product to be activated. Since similarity implies a high activation, the network becomes adaptively sparse through the fact that only select neurons are used. Additionally, these neurons are not random, but rather the most influential neurons, helping to reduce the computational complexity significantly while retaining training performance. This approach may even increase accuracy, as dropout can help to generalize to different testing datasets. SLIDE is unique in its approach, due to its ability to be hardware-agnostic and its exploitation of adaptive sparsity on neural networks using LSH.

2.4 Limitations of Existing Work

SLIDE shows state-of-the-art performance, with a training time speedup on multilayer pereptron models (MLPs) of almost 10x in on CPUs. This approach is only applicable to MLPs: the neurons of each fully-connected linear layer are connected to each of the previous layers' neurons. Because of this, when dropout occurs, the computational complexity drastically decreases. However, the structure of convolutional neural networks is not conducive to improvements that SLIDE suggests. However, the idea of LSH can be applied to different parts of the convolutional network in conjunction with other optimizations.

Therefore, we pivoted to a different direction, taking the core ideas of SLIDE, and applying them to different parts of a convolutional neural network. Therefore, our idea for optimizations lie in a two-pronged solution. The first part is a reduction of the convolutional weight matrix through a concept called sketching. The second part is through filter pruning, where adaptive sparsity is applied through convolutional filters. Both parts of this solution act on the ideas of locality-sensitive hashing and adaptive sparsity.

3 Preliminaries

3.1 Hyperplane Asymmetric LSH

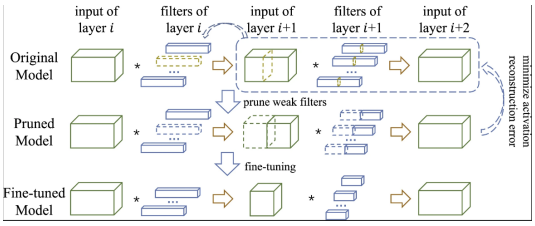

Figure 2: Visual Representation of Filter Pruning

As discussed in section 2.2, asymmetric LSH aims to transform the two vectors such that the two transformed vectors being hashed together in LSH corresponds to the untransformed vectors having a large inner product.

Let us define (c, R)-approximate near neighbor search such that multiplier $c > 1$, distance $R > 0$, and the desired output to query $q \in C$ is point $p \in C$ such that $||q - p|| \leq cR$ with a probability of success $1 - \delta$. Here, $C \subseteq \mathbb{R}^d$ is a collection of d-dimensional points. Let us now define a family of hash functions that accomplish this nearest neighbor search. A family of hash functions is said to be (R, cR, p_1, p_2)-sensitive if every hash function h in the family abides by the following rules for points $p, q \in \mathbb{R}^d$, $c > 1$:

$$||q - p|| \leq R \rightarrow Probability(h(q) = h(p) \geq p_1$$
$$||q - p|| \geq cR \rightarrow Probability(h(q) = h(p) \leq p_2$$

Essentially, within a certain distance, points p and q have high probability of hashing into the same bucket, and outside of a different distance, they have a low probability of hash collision probability. We use K hash functions and L hash tables. The outputs from the K hash functions are concatenated, increasing the probability that $p_1 > p_2$ (p and q are more likely to be hashed into the same bucket if they're closer together). The outputs from the multiple hash tables provide a lower probability that two items are accidentally hashed into the same bucket, increasing both p_1 and p_2.

The specific family of hash functions chosen for this task was hyperplane LSH, where each hash function produced an output of 1 if the dot product of $a \in \mathbb{R}^d$ with elements $a \sim \mathcal{N}(0, 1)$ with input x is greater than 0, and an output of 0 otherwise. The K hash tables produce a K length bit string, where each bit essentially represents a hyperplane in $\mathbb{R}^K$, and if two points are close to one another, they will probably exist on the same side of that hyperplane.

LSH is altered using asymmetric transformations, allowing it to be used for MIPS as opposed to just nearest neighbor search. The definition if very similar, where a query function is applied when searching the data structure and a pre-processing function P is applied when building the

data structure in order to modify the hash tables to contain information that is applicable to MIPS. Functions P and Q that have been shown to structure the data nicely for MIPS in previous work are as follows:

$$Q(x) = Append_m(x, 0, 0, ..., 0)$$
$$P(x) = Append_m(x, 0.5 - ||x||_2^2, 0.5 - ||x||_2^4, ..., 0.5 - ||x||_2^{2^m})$$

For more information on why these modifications allow for MIPS, we refer to [2] to provide additional justification. With P and Q constructed, we can change are LSH formulation to the following:

$$q^T p \geq R \rightarrow Probability(h(Q(q)) = h(P(P)) \geq p_1$$
$$q^T p \leq cR \rightarrow Probability(h(Q(q)) = h(P(p)) \leq p_2$$

With this formulation, p maximizes the cosine of the angle formed between $Q(q)$ and $P(p)$, resulting in the MIPS modification desired for LSH.

3.2 Tensor Sketching

Tensor sketching is an approach to reducing the rank of high-dimensional tensors while attempting to preserve the properties of that tensor.

In this case, the idea is to use random signed matrix $U_n \in \mathbb{R}^{k \times d_n}$ with entries between 1 and -1, scaled down by a factor of $\sqrt{k}$. The goal of tensor sketching is to multiply the weight matrix of a convolution operation by a tensor along a either its input or output dimensions in order to reduce its dimensionality, while retaining the same expected value of the output matrix as well as keeping a bound on the variance of the output. Formalizing the idea of sketching along a certain dimension of a tensor, let tensor $T \in \otimes_{i=1}^{p} \mathbb{R}_i^d$.

We want to sketch along dimension d_n of T, defined as the mode-n sketch $S_n = T \times_n U_n$. We define the mode-n tensor product of tensor T with matrix A as

$$T \times_n A_{i_1 \cdots j \cdots i_N} = \sum_{i_n=1}^{I_n} t_{i_1 \cdots i_N} a_{i_N j}$$

where $T \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ and $A \in \mathbb{R}^{I_N \times J}$

Recall $U_n \in \mathbb{R}^{k \times d_n}$ is the unsigned scaled signed matrix used for sketching. Clearly, k will be significantly smaller than d since we want to reduce the dimension along tensor mode n.

The sketched tensor S_n must have an expected value similar to that of the original weight matrix while maintaining a bound o the variance. The proof for the variance bound is fairly extensive, so we encourage you to read the original paper highlighting these findings [3]. As for the expected value, by construction, $\mathbb{E}[U^T U] = 1$. Thus, $\mathbb{E}[TU^T U] = T\mathbb{E}[U^T U] = T$.

4 Methods

Now, we will provide a discussion of how we applied Locality Sensitive Hashing (LSH) and tensor sketching to form our experiments. We used these techniques in tandem to induce further performance improvements in convolutional neural networks beyond using either method unilaterally.

4.1 Locality Sensitive Hashing

Here, we provide a discussion of LSH in the manner that it was applied in our method. The underlying goal of this application of LSH involves inducing sparsity *adaptively* within the network. Namely, we want to selectively choose different components of the convolutional weight matrix that we will activate for different inputs. We now define the manner by which LSH is incorporated into a specific convolutional layer within a neural network.

Initialization To initialize a particular convolutional layer, we start by first initializing the convolutional weight matrix, $W \in \mathbb{R}^{o \times i \times h \times w}$, where i and o are the input and output channels respectively, and h and w form the kernel dimension.

Unique to a regular convolutional layer, the LSH-based convolutional layer incorporates a group of N hash tables. For each table, we construct a `table_hash` that consists of

$$(h_1(x), h_2(x), \cdots, h_L(x))$$

where $h_i \in \mathbb{H}$ for some hash family $\mathbb{H}$ that has L elements and x some vector of arbitrary dimension. Then, over we create o vectors that are $\in \mathbb{R}^{ihw}$, effectively flattening each output filter of the convolutional weight tensor. For every one of these o vectors, we add its index to to the position determined by the `table_hash` operation, namely

$$p = \texttt{table\_hash}(o_1) = (h_1(o_1), h_2(o_1), \cdots, h_L(o_1))$$

where p is an L-dimensional vector that provides an index into one of these hash tables. This provides a unique position in a hash table that is parameterized by L different hash functions. We subsequently add the 0-based index of o_1, which comes from its position in W to the hash table at index p. This operation is then repeated over all the hash tables.

Forward Pass The forward pass of a traditional convolutional layer takes in a image, which is represented by tensor $I \in \mathbb{R}^{i \times h_1 \times w_1}$ and convolves it with the aforementioned weight matrix, $W \in \mathbb{R}^{o \times i \times h \times w}$, resulting in an output tensor $O \in \mathbb{R}^{o \times h_2 \times w_2}$. The LSH-based convolutional layer acts in a similar way, except it utilizes the hash table construct defined during initialization to only act over a subset of the o output channels, reducing the number of convolutional operations that have to be done.

We first convert the input image tensor I into a column-vector format, so we have a w_1 image vectors representing the entire image $\in \mathbb{R}^{i \times h_1}$, which each semantically containing information for one column in the input image. Then, for every single table in our set of N hash tables, we apply a query hash function for each of our w_1 image vectors to "look up" a particular index in the hash table. Using the output channel indices that were stored in this index of the hash table, we are able to determine which output filters we should use. This is performed over all input columns, resulting in final subset of filters that we have shared over all of the N hash tables.

After doing, this we are left with a subset of output filters, $k \leq o$. We then create a new convolutional weight matrix $W' \in k \times i \times h \times w$. From here, we perform a convolution in the manner that it is traditionally done in, resulting in an output tensor $O' \in k \times h_2 \times w_2$. We subsequently fill in the missing output dimensions to arrive at $O \in \mathbb{R}^{o \times h_2 \times w_2}$, which is an output dimension of the correct size.

A note on the usage of this layer – if we know that we are stacking these layers within a neural network, we can elicit further performance improvements by recognizing the fact that the zeroed outputs of the previous LSH-based convolutional layer form the inputs to the next LSH-based convolutional layers. More formally, consider two LSH-based convolutional layers, L_1 and L_2. Suppose L_1, prior to filling in missing output dimensions with zeroes, outputs a tensor that is of $O \in \mathbb{R}^{k_1 \times h_2 \times w_2}$, but is supposed to have an output dimension of size o_1. Then during the LSH convolution that happens in L_2, we do not have to consider inputs into L_2 from output dimensions of L_1 that are guaranteed to be zero if we maintain a global view of its predecessor's pruned convolutional filters. As a result, our input into L_2 can be a tensor of dimension $I \in \mathbb{R}^{k_1 \times h_2 \times w_2}$, meaning that the second LSH convolutional layer, L_2, has to perform an order of

$$\sim \frac{o_1}{k_1}$$

fewer multiplications in calculating its final output.

4.2 Tensor Sketching

Here, we provide a discussion of tensor sketching in the manner that it was applied in our method. The underlying goal of this application of tensor sketching is to reduce the dimension of the weight tensor by reducing the number of parameters within it by relying on a low-rank approximation.

Initialization To initialize a particular sketching-based convolutional layer, we start by first initializing the convolutional weight matrix, $W \in \mathbb{R}^{o \times i \times h \times w}$, where i and o are the input and output channels respectively, and h and w form the kernel dimension. We also initialize the desired sketch dimension, s, and the size of the sketch dimension, k. Using this and the aforementioned weight matrix, we can create a set of k random signed matrices initialized as

$$R_i \in \mathbb{R}^{o \times k} \sim \mathcal{U}(-1, 1)$$

Note that these matrices are fixed and are not learned at any point during training. Using the signed matrices, we can construct s sketch matrices in the following manner

$$S_i = R_i \times_0 W$$

where the tensor-matrix multiplication happens along the first dimension. This resultant tensor, S_i, is reshaped into a tensor $\in \mathbb{R}^{ihw \times k}$

Forward Pass The forward pass of a traditional convolutional layer takes in a image, which is represented by tensor $I \in \mathbb{R}^{i \times h_1 \times w_1}$ and convolves it with the aforementioned weight matrix, $W \in \mathbb{R}^{o \times i \times h \times w}$, resulting in an output tensor $O \in \mathbb{R}^{o \times h_2 \times w_2}$. The sketch-based convolutional layer acts in a similar way, except it uses its sketched representations of weight matrix to reduce the total number of multiplications that are performed.

We first begin by creating k copies of the input tensor, I, and unfolding across this dimension, resulting in a new input tensor $I' \in \mathbb{R}^{k \times ihw \times L}$, where L is a free parameter that ensures the size of the input remains invariant and h and w in this case represent the kernel dimensions of the convolutional weight tensor.

Then, we perform two operations, sequentially, using I':

1. Tensor-matrix multiplication with the sketched matrices
2. Tensor-matrix multiplications with the signed matrices

First, we construct an intermediate matrix,

$$O'' = I' \times_2 S \in \mathbb{R}^{s \times k \times L}$$

where S represents all k tensor sketches. We take this resulting matrix and construct

$$O' = O'' \times_2 R \in \mathbb{R}^{s \times o \times L}$$

We can then expand this matrix along its final dimension and then take the mean along the first dimension, result in

$$O \in \mathbb{R}^{o \times h_2 \times w_2}$$

Averaging a number of sketches provide additional robustness in model performance, since the expectation of the sketch-signed matrix product approaches the original weight matrix. Furthermore, in a traditional convolutional layer, we see that we have on the order of

$$O(hwih_2w_2o)$$

operations, since for every point in the final output tensor $O \in \mathbb{R}^{o \times h_2 \times w_2}$, we must perform a multiplication with an input-channel-sized convolutional weight tensor. This implementation is able to significantly reduce this cost given a good selection of k and s. We see that the number of operations instead is on the order of

$$O(shwih_2w_2k + sh_2w_2ko)$$

This is due to the separation of the weight matrix into the sketch and signed components, as well as the folding primitives used during the operation. As a result, we see that since $k << o$, this can lead to a speedup on the order of

$$O(\frac{o}{k})$$

with respect to the number of operations required, since the first term that reflects the product of the input matrix with the sketch matrix will dominate the number of operations. Additionally, we can tune the number of sketches, s, as required depending on the tradeoff demanded between the number of parameters and the correctness of the sketch approximation.

4.3 `MILF` Approach

The `MILF` approach synthesizes the two previous approaches together. Note that the LSH operations and the tensor sketching operations act on two discrete portions of the convolution operation:

1. The tensor sketching approach *reduces the number of parameters within a single convolutional filter*
2. The LSH module *reduces the number of filters that must be applied to complete the convolution*

Therefore, in order to couple the approaches together, with an initialization that includes all components required in both the LSH convolution (e.g. set of hash tables) and the tensor sketching convolution (e.g. set of sketches and random signed matrices). Then, to determine which filters to use, we reconstruct the weight matrix using the random signed matrices and the set of sketches, since

$$W = S \times R \in \mathbb{R}^{ihw \times o}$$

which can be trivially reshaped to the weight matrix that the LSH convolution expects.

From this, using the methods defined in the forward pass of the LSH convolution layer, we can determine the set of active filters that should be forward propagated in the exact same manner described in section 4.1.

Finally, given an active filter set of size α we can select the relevant dimensions from the sketch tensor. Recall the sketch tensor is of dimension $S \in \mathbb{R}^{s \times k \times o}$. We can choose a subset of the final dimension, α, and arrive at a reduced sketch tensor

$$S' \in \mathbb{R}^{s \times k \times \alpha}$$

We then proceed forward doing a sketch convolution as described above with the adjusted parameter set. With this construction, we combine both of the aforementioned methods together and see performance benefits induced by a combination of the two. We investigate these results through experimentation below.

5 Experiments

In these experiments, we are going to be testing the performance of each part and the total improvement of our approach. In terms of some of the experiment parameters, they are as follows:

Experiment Details	
Datasets	CIFAR-10, MNIST
Epochs	10
Learning Rate	1×10^{-3}
Machine	A100 GPU
Loss	Cross-Entropy Loss
Optimizer	Adam

Before testing out the approach, we wanted to baseline the approach with a general convolutional neural network.

In terms of the networks that we were testing, we tested a 3-layer convolutional neural network, AlexNet, 8-Layer CNN, and a 10-layer VGG. In each of these networks, 4 tests were conducted. The first test was with the original network. The second test was to replace a select number of layers with sketching. The third test was to add filter pruning to a few selected layers within the network. The final test was to add both filter pruning and sketching within the convolution network. Within each of these tests, the runtimes and accuracies were used to compare the speedup that was able to be achieved by the respective models and the corresponding accuracy.

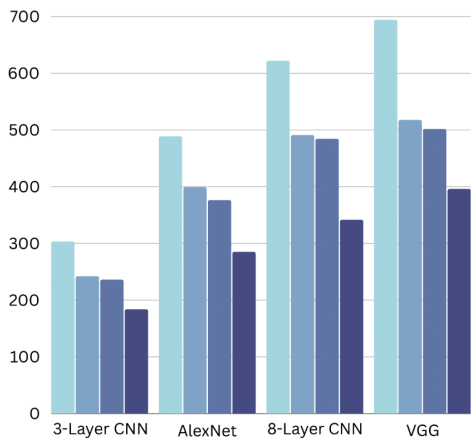

Figure 3: These are the relative runtimes of each of the networks. From left to right, the columns are the vanilla version, the sketched version, the LSH version, and Sketch + LSH version. The y axis is training time in seconds and the x axis is the architecture chosen

5.1 Architectures

As stated above, the 4 main architectures that were chosen were the: 3-layer convolutional neural network, the AlexNet architecture, an 8-Layer Convolutional Network, and a 10-layer VGG network. The main reason the 3-layer convolutional network was used was to see the efficacy of replacing every single layer of a small network. Most practical use cases of the network convolves replacing specific layers within a large network. however, testing with a small network and replacing every single layer helps see the true efficacy of our solution. We then tested on the AlexNet architecture, due to its efficacy on image classification and widespread usage. The third test of an 8-layer convolutional network is another test on larger network. In this network, every single layer is not replaced by our approach. Rather, every even layer is replaced by our approach. The last architecture that was tested was a 10-layer VGG. The idea behind using this specific architecture revolved around using the biggest CNN that could be used on our datasets in a reasonable timeframe with the amount of compute at hand.

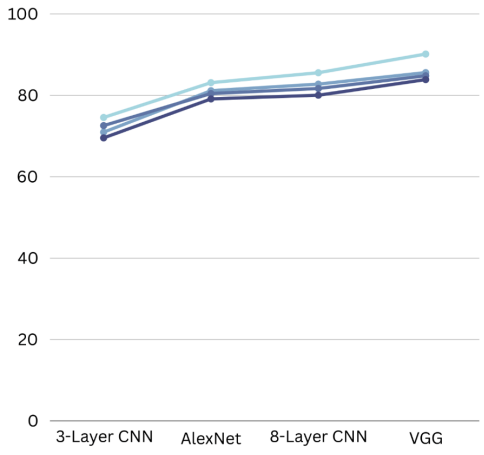

Figure 4: These are the relative accuracies of each of the networks. From top to bottom for each network, the points correspond to the vanilla version, the sketched version, the LSH version, and Sketch + LSH version. The y axis is accuracy (%) and the x axis is the architecture chosen

5.2 Datasets

The two main datasets that were used were CIFAR-10 and MNIST. Both of these datasets are some of the most popular image processing datasets used in benchmarking computer vision models. The MNIST dataset at its core is a set of handwritten digits and the corresponding label, referring to the digit that is written. Most of the standard benchmarks for MNIST have used convolutional neural networks. The CIFAR-10 dataset refers to a set of 60,000 32x32 images that are classified into 10 separate categories.

6 Discussion

Quantitatively, for each of the specific networks, there was a significant dropoff of about 30 % in regards to runtimes of each of the networks, when comparing the vanilla convolution network to the Sketching and Locality Sensitive Hashing Approach that was introduced. In comparison to both the Sketching approach and LSH approach individually, our approach showed about a 15 % dropoff in runtime. This speaks to our hypothesis of sketching and filter pruning compounding their effect to provide a significant dropoff in runtime. However, the optimizations in runtime mean nothing, if the accuracy dropoff is too immense. The first comparison in accuracies are between the sketching and filter pruning individually with respect to the vanilla network. Each of these approaches show consistently about a 2-3 % accuracy dropoff with all of the architectures tested. When comparing the vanilla network to our approach, we saw about a 5 % accuracy dropoff. This type of dropoff provides options for the user dependent on the use case, as if there is a tolerant accuracy bound, the actual computational complexity of the network can be significantly.

When looking at the results between the regulation network, the sketching network, the LSH network, and fully optimized network, there are a couple main observations that can be made. The first is that the assumption that is made through the approach that both the LSH and the filter pruning would work in conjunction to provide a significant decrease in computational accuracy seemed to be correct. Sketching and filter pruning on their own provide very comparable computational improvements. However, when combined, the computational improvements show significant improvement than each on their own. In terms of the acuracy, the accuracy does encounter a bigger dropoff than each individually, but they do not compound fully.

In terms of the applications of these approaches, it is important to look at the computational complexity and accuracy dropoff. As shown in our approach, we can show significant improvements in computational accuracy of a neural architecture if the error margins in a specific use case are slightly larger. The accuracy dropoff and performance improvement seem to have an inverse relationship and this can be utilized to optimize based on the use case of a convolutional network. For example, in an automated face detection system, it is important to make sure that classification is as close to 100 % accuracy as possible. In that situation, the tradeoff of using more compute is necessary to achieve a goal. However, in situations where a 3-4 % accuracy dropoff is acceptable, the computational complexity can be significantly reduced.

Another main discussion topic is the use of the CIFAR-10 and MNIST datasets. The idea behind the optimizations are to isolate the main convolutions that are making an impact on a specific network's task and to take those convolutions and compress them to a more computationally inexpensive aspect of the network. CIFAR-10 and MNIST are two of the most common datasets used in image classification, an important usage of convolution networks. The CIFAR dataset requires classification into 10 different classes, so it is possible that the performance speedup and minimal accuracy dropoff are accentuated by the simplicity of the dataset. In the future, we intend to see how our approach works in more adversarial use cases, with more classes for classification, noisier images if applicable, and more complex datasets in general.

7 Conclusion

In conclusion, the escalating prevalence of Convolutional Neural Networks in domains such as image classification, object detection, and self-driving cars, has brought its computational demand to the forefront. The inherent expansiveness of CNNs usually calls for high-performance hardware, such as GPUs and TPUs, but those are not always available. This study takes a different approach, applying

algorithmic and hardware-agnostic improvements to convolutional neural networks without a major accuracy dropoff.

Our approach combines Multi-level Tensor Sketching and Locality Sensitive Hashing to expedite CNN training. The sketching aspect of the approach reduces the dimensionality of specific kernel filters. The LSH aspect looks at the specific filters used and selects only the most valuable set of filters during forward propogation. Experimental analysis has indicated that our approach allows for approximately a 30 % decrease in runtime with about a 5 % decrease in accuracy. Our approach speaks to ongoing opportunities for algorithmic improvements in all types of machine learning and AI. Convolutional networks have been used as the SOTA for a wide variety of tasks in the past 10 years, and the usage of these networks has become prevalent despite its heavy computational cost. It is important to note that these types of networks still present opportunities for significant algorithmic improvements in computational complexity. Overall, our approach is able to show significant improvements in computational complexity from a purely algorithmic approach with a minimal loss in accuracy.

References

[1] Chen, B., Medini, T., Farwell, J., Gobriel, S., Tai, C., Shrivastava, A.. (2020). SLIDE : In Defense of Smart Algorithms over Hardware Acceleration for Large-Scale Deep Learning Systems.

[2] Feeney, A. (2019). Convolution Acceleration: Query Based Filter Pruning with ALSH. Computer Science Honors Theses.

[3] Kasiviswanathan, S., Narodytska, N., Jin, H.. (2017). Deep Neural Network Approximation using Tensor Sketching.

[4] Everingham, M., Van Gool, L., Williams, C., Winn, J., Zisserman, A. (2010). The Pascal Visual Object Classes (VOC) Challenge. International Journal of Computer Vision, 88(2), 303–338.

[5] Yadav, S., Jadhav, S. (2019). Deep convolutional neural network based medical image classification for disease diagnosis. Journal of Big Data, 6(1), 113.

[6] Bojarski, M., Choromanska, A., Choromanski, K., Firner, B., Ackel, L., Muller, U., Yeres, P., Zieba, K. (2018). VisualBackProp: Efficient Visualization of CNNs for Autonomous Driving. In 2018 IEEE International Conference on Robotics and Automation (ICRA) (pp. 4701–4708).

[7] Bird, J., Faria, D., Ekárt, A., Ayrosa, P. (2020). From Simulation to Reality: CNN Transfer Learning for Scene Classification. In 2020 IEEE 10th International Conference on Intelligent Systems (IS) (pp. 619–625).

[8] Ba, J., Frey, B. (2013). Adaptive dropout for training deep neural networks. In Advances in Neural Information Processing Systems. Curran Associates, Inc..

[9] Potamias, M., Athitsos, V. (2008). Nearest neighbor search methods for handshape recognition. In Proceedings of the 1st international conference on PErvasive Technologies Related to Assistive Environments (pp. 1–8). Association for Computing Machinery.

[10] Shrivastava, A., Li, P.. (2014). Improved Asymmetric Locality Sensitive Hashing (ALSH) for Maximum Inner Product Search (MIPS).