5-2019

Convolution Acceleration: Query Based Filter Pruning with ALSH

Arthur Feeney
Trinity University, afeeney@trinity.edu

Convolution Acceleration: Query Based Filter Pruning with ALSH

Arthur Feeney

A departmental senior thesis submitted to the Department of Computer Science at Trinity University in partial fulfillment of the requirements for graduation with departmental honors.

April 22, 2019

Dr. Yu Zhang Thesis Advisor	Dr. Yu Zhang Department Chair

Michael Soto, AVPAA

TRINITY UNIVESRITY

Convolution Acceleration: Query Based Filter Pruning with ALSH

by

Arthur Feeney

April 2019

Abstract

by Arthur Feeney

The rising ubiquity of Convolutional Neural Networks for learning tasks has led to their use on a variety of devices. CNNs can be used on small devices, such as phones or embedded systems; however, compute time is a critical enabling factor. On these devices, trading high accuracy for improved performance may be worthwhile. This has led to active research in high-level convolution optimizations. One successful class of optimizations is filter pruning, in which filters that are determined to have a small effect on the network's output are deleted. In this work, we present a self-pruning convolution that is intended to accelerate convolutions for use on small devices. We call it an ALSH Convolution because it uses Asymmetric Locality Sensitive Hashing to generate a subset of the convolution's filters that are likely to produce large outputs for a given input. Our methodology is accessible: it generalizes well to many architectures and is easy to use, essentially functioning as a regular layer. Experiments show that a network modified to use ALSH Convolutions can stay within 5% accuracy on CIFAR-10 and 10% on CIFAR-100. Further, on small devices, a network built with our implementation can be 2x faster than the same network composed of PyTorch's convolution.

Contents

Chapter 1

Introduction

Convolutional neural networks (CNN) have become common place in many computer vision applications, including image classification, instance segmentation, pedestrian and car detection, and object localization. Over recent years, the networks for all of these applications have become much deeper, resulting in an increase in the number of network parameters and convolution operations. Such large networks have significant inference costs that become especially apparent when used on embedded sensors or mobile devices, where computational resources may be limited. For small devices, computational efficiency is a critical enabling factor. In fact, any device or service that has time constraints could potentially benefit from an improvement in inference time, if it retains moderately high accuracy.

For many recent CNN models, the majority of the parameters are in the fully-connected layers. A prime example of this is VGG-16, in which the fully-connected layers make up 90% of the total number of parameters, but contribute less than 1% to the total number of floating point operations [33]. Clearly, the amount of work being done during inference is dominated by the convolutional layers of the network. For this reason, most modern optimization efforts focus on the convolutional layers.

One existing optimization for neural networks is weight pruning. Pruning is a well-researched area and was first introduced relatively early in the development of neural networks: Optimal Brain Damage and Optimal Brain Surgeon use a second-order Taylor expansion to select parameters for deletion in fully-connected layers [13, 22]. It is also possible to remove individual weight parameters from convolutions, but doing so generally requires the use of sparse BLAS libraries or even specialized hardware. For this reason, weight pruning is not a popular method for optimizing convolutions. However, deep CNNs often have a significant amount of redundancy in their filters [1, 28]. So, more recently, researchers have been looking into filter pruning as a way to reduce the computational costs of well-trained CNNs. Unlike pruning individual weights, pruning filters does not require the use of sparse BLAS libraries or specialized hardware. Further, the number of filters pruned correlates very strongly with improved performance because it is guaranteed to reduce the scale of matrix multiplications in the network; pruning one filter essentially removes an entire row from the multiplication.

Many filter pruning methods delete filters that are unlikely to produce large outputs. These filters typically have the smallest effect on the final output. A simple but effective method of filter pruning determines which filters to prune based off of their magnitude. It deletes filters that have small weights and retrains the network to account for any change in accuracy [23]. Another method uses a Taylor-expansion that approximates the change induced in the network's loss function by pruning network filters [28]. Their method makes pruning selections with the goal of minimizing this change.

There is a prior work in accelerating fully-connected layers [34] that used approximate maximum inner product search to quickly find a subset of nodes that were likely to produce large outputs. By extending this, and leveraging existing work in filter pruning and maximum inner product search, we introduce a new method to selectively prune filters based on the convolution's input. It functions as a self-pruning layer that we call an *ALSH Convolution*. By using Asymmetric Locality Sensitive Hashing (ALSH) [31, 32] to partition a convolution's filters, we can quickly find a subset of filters that are likely to produce many large inner products for a given input. During each iteration, a new subset of filters is applied to the input, significantly reducing the number of floating point operations. We offer a specific training strategy that allows the network to retain accuracy as fewer filters are used by the network. Unlike other filter pruning methods, the ALSH Convolution is high-level and can be used as a regular network layer, making it relatively accessible.

The central thesis of this work is that *the convolutional layers dominate the number of floating point operations performed by convolutional neural networks. This makes them difficult or impossible to use on small devices, where compute* *time is a critical enabling factor. Our proposed solution to this is the ALSH Convolution. It is a self-pruning layer that uses ALSH to find a subset of filters that are likely to produce many large outputs. It significantly reduces the number of floating point operations performed during convolution inference, while being both easy to use and capable of retaining high accuracy.*

Chapter 2

Related Work

2.1 Convolutional Neural Networks (CNNs)

Convolutional neural networks have become extremely popular for their state-of-the-art results in many domains. Common applications for convolutional neural networks solve problems in computer vision: classifying hand-written digits [21], instance segmentation [24], object detection [10], scene recognition [40], and many others. Convolutional neural networks are even used in reinforcement learning: deep Q-learning is able to achieve very high-scores in simple video games [27]. Inception-ResNet-V2 [37] achieved state-of-the-art results on ImageNet, a huge image classification data set with over a million training images. However, it had over one hundred layers that, in total, had millions of parameters. Networks of this scale require an immense amount of memory and may take weeks to train on large datasets. Even after they have been trained, using them for inference can be taxing and slow. This makes them difficult or even impossible to use on systems with limited computing resources, such as embedded sensors and cell phones where computational and power resources are limited.

2.2 Dropout

There is a traditional challenge when training machine learning algorithms called “Overfitting.” It occurs when a model fits the training set too well and is unable to perform more general inference on new data. Generally, the more parameters that a model has, the easier it is for it to overfit on training data. As neural networks have a large number of parameters, overfitting used to be a serious problem that was difficult to overcome when training [14, 35]. Dropout was developed to alleviate this problem in neural networks. The basic idea of dropout is to drop a random subset of activations from the hidden layers of a network. Doing this can prevent co-adaptation between network parameters [14]. Co-adaptation occurs when two or more of a network's nodes begin to have similar outputs, making them redundant. Dropout allows one node to be updated during a training pass while the other is not. This prevents them from greedily learning the same information and makes it more likely that each node will produce a distinct output. Further, dropout effectively trains a large ensemble of many, thinner networks [35]. In a way, every training example provides gradients for a smaller random subset of the total network. At test time the thin networks that form the ensemble are combined into a single large network. This helps networks generalize when processing new data. If a network is large enough and properly trained, then the network is almost guaranteed to get better results with dropout than it would without dropout.

Adaptive dropout is a variant of dropout. Adaptive dropout introduced a novel way to selectively remove unimportant nodes from a network. It is a direct extension of dropout that gives each node a unique keep probability. These unique keep probabilities are dependent on the layers input [39]. Experiments performed by the authors show that adaptive dropout learns to drop activations based on their magnitude. In these experiments, nodes that tended to produce large activations were consistently given a high keep probability. There is an extreme version of adaptive dropout, called Winner-Take-All [26], that keeps the top $k\%$ of activations and drops the rest. However, finding the best nodes is inefficient because they rely on brute-force techniques. Even though adaptive dropout and its variants reduce the number of nodes used, there are no new computational savings from their use. The primary contribution of the adaptive dropouts to this work is that large activations have greater influence on the output than small activations.

2.3 Improving Neural Network Performance

There has been substantial research into reducing the compute and size complexity of neural networks [6]. The majority of operations in a network are matrix-matrix or matrix-vector multiplications. So, most optimization methods that are intended to accelerate network computations attempt to reduce the scale of these operations. There are many low-level optimizations that can improve a networks compute speed: loop tiling, loop unrolling, replacing floating point numbers with low bit integers, and many others. However, in this section we focus on overviewing higher-level methods for acceleration: quantization and pruning [6].

2.4 Quantization

Quantization techniques have introduced binary and ternary weight networks. These networks have their internal parameters restricted to ± 1 or $\pm 1, 0$ [7]. There has even been work in restricting both the weights and *activations* to ± 1 [15]. Since these networks only use two numbers, they can be compressed much more than networks with floating point parameters. They are also able to maintain very high accuracy on difficult datasets, such as ImageNet.

2.4.1 Pruning

Pruning methods existed far before deep learning's resurgence and have been extensively studied [13, 22]. Pruning can be used in neural networks to remove parameters that are considered unimportant. By doing this, pruning can increase the sparsity of the network significantly. A sparse network requires less storage space and has reduced complexity compared to a regular network. For convolutional neural networks, pruning can be done at varying granularity:

- Fine-grained: any unimportant parameters may be pruned
- Vector-level: vectors are pruned from kernels
- Kernel-level: kernels are pruned from filters
- Group-level: clusters of kernels are pruned from filters
- Filter-level: Entire filters may be pruned from the convolution

As discussed in the introduction, pruning filters correlates very strongly with improved performance because it effectively removes entire rows from the matrix multiplication [6, 28]. However, there is another important optimization: When using filter-level pruning, the channels of the output that would have been computed by the pruned filters will never be used in a computation. This means that the input to the next layer will be smaller, so the next layer will perform less computations than it normally would. So, not only is the current layer optimized, but the following layer is as well. Due to its ability to significantly reduce complexity, filter-level pruning is considered the best for accelerating convolutional neural networks.

There has been successful prior work in filter-level pruning. In an article by Hao Li et al [23], the authors propose a form of filter pruning that selects filters to prune, in a single layer, based off of the sum of its absolute weights $\sum |F_{i,j}|$, the filter's l_1-norm. They then remove some fraction of the filters with the smallest l_1-norms from the layer. They do this because filters with smaller weights tend to produce smaller activations and have a smaller effect on the final output. They also devise a simple method to prune an entire network: iteratively pruning and retraining to account for any loss of accuracy. The authors find a noticeable performance improvement during inference and reduction in the network size with minimal impact on final accuracy. A primary advantage of this method is that it is relatively simple. The next method that we discuss, while possibly making higher quality prunes, appears significantly more difficult to implement using existing libraries.

A more robust method has been proposed by Molchanov et al [28]. They reiterate that removing filters based on l_1 norm can work well, but also claim that pruning based on the mean of output feature maps can be effective. If a filter's mean output is small, then it likely is not useful for the current task. Their main contribution is reformulating pruning as an optimization problem. They use a Taylor expansion to prune parameters that have a nearly flat gradient of the cost function with respect to their output feature map. They are able to stay within 5% accuracy on ImageNet while using only 52% of the network's original filters.

Chapter 3

Background

3.1 Neural Networks

In this section, we quickly cover some of the fundamentals of neural networks. We focus on the forward pass because it will be changed by our methodology. The update will be changed slightly, but only to account for changes in the forward pass.

Forward-propagation is when the inference occurs; the network is given an input and it produces an output. Back-propagation and gradient descent are both parts of updating the network. The network's output is used to calculate a loss. This is used by the back-propagation algorithm to find the gradients of each layer's parameters. Finally, gradient descent is used to update the network's parameters [12].

3.1.1 Forward-Propagation

The most fundamental form of neural network is a sequence of L non-linear transformations or "layers" [12]. The layers that compose a standard network are commonly called "affine," "dense," or "fully-connected." To illustrate this type of layer, let $l \in \{1, \ldots, L\}$ index the hidden layers of a network with L fully-connected layers. Forward-propagation through a fully-connected layer can be described as:

$$\begin{aligned} z^l &= W^l y^{l-1} + b^l \\ y^l &= f(z^l) \end{aligned} \tag{3.1}$$

In this sequence of equations, y^{l-1} is the output vector of layer $l-1$ and the input to layer l. Layer l first applies a linear transformation to y^{l-1}; it computes a matrix multiplication with W^l and y^{l-1} and adds b^l to the resulting vector. That sum is z^l. Finally, a non-linear transformation $f(\cdot)$ is applied to z^l, producing y^l. This is the output of layer l and will be the input to the next layer [12]. This series of operations is repeated for each of the remaining layers of the network. The last layer's output is some prediction, such as the class of an image.

3.1.1.1 Convolution Forward-Propogation

At a high-level, a convolutional neural network is essentially the same as a more standard neural network. It is typically a sequence of layers that will produce some output; however, the internal layers, called "Convolutions," perform a different operation than a fully-connected layer.

The type of convolution that is typically used for image classification is composed of filters. It applies these filters to regions of the input, computing an inner product with the filter and each region [12]. It is possible to implement this type of convolution by transforming the input and filters into matrices and then computing their product. This can be done by using the algorithm im2col. It reshapes images into matrices by unrolling patches of the image into the columns of a matrix [2, 17]. To use this with a convolution, one can apply im2col to the input and flatten each filter into a row of a different matrix. Then, one can perform a general matrix multiplication of the convolution's filter matrix and the input column matrix to produce the convolution's output. This product must then be reshaped into the proper dimensions for the convolutions final output.

Algorithm 1: Simplified im2col

```
$input[H][W][C]$
$out[H][W][K][K][C]$
for $h = 0 \dots H$ do
    for $w = 0 \dots W$ do
        for $k_1 = -K/2 \dots K/2$ do
            for $k_2 = -K/2 \dots K/2$ do
                for $c = 0 \dots C$ do
                    $out[h][w][k_1][k_2][c] = input[h + k_1][w + k_2][c]$
Reshape $out$ into a $[M \times W] \times [K \times K \times C]$ matrix.
```

This implementation is a simplified version because it is not strict about how the filters are applied along the input's borders. We primarily include it to show the general idea [2]. It is just taking small squares of the input image and making them the columns of a matrix.

Algorithm 2: Conv2d using im2col

```
input  : An Object $I \in \mathbb{R}^{c \times h_1 \times w_1}$
         Filters $F \in \mathbb{R}^{M \times c \times k \times k}$
output: An object $O \in \mathbb{R}^{M \times h_2 \times w_2}$
begin
    /* Apply Algorithm 1 to the input I                              */
    $cols \leftarrow$ im2col$(I)^\intercal \in \mathbb{R}^{[k \times k \times c] \times [h_1 \times w_1]}$
    $rows \leftarrow$ $F$'s filters transformed into the rows of a matrix in $\mathbb{R}^{M \times [k \times k \times c]}$
    $O^* \leftarrow rows \times cols \in \mathbb{R}^{M \times [h_1 \times w_1]}$
    $O \leftarrow$ reshape $O^*$ into an image in $\mathbb{R}^{M \times h \times w}$
    return $O$
```

After using im2col on the input and flattening the filters into the rows of a matrix, this is essentially the same as a fully-connected layer's forward-pass. It performs a matrix multiplication. Throughout the rest of the paper, we will write our modifications to the convolution algorithm using im2col because it is easier and makes some indexing operations more clear.

3.1.2 Stochastic Gradient Descent (SGD)

Neural networks are updated using gradient descent [12]. Suppose we let a network's parameters at iteration t be θ_t and define a function $\nabla J(\cdot)$ that computes the gradients of the parameters using the back-propagation algorithm. By using the network's parameters and the back-propagation function, we can define SGD as $\theta_{t+1} = \theta_t - l \times \nabla J(\theta_t)$. Where the hyper-parameter l is the learning rate. The network's parameters are updated at every time step by: First, scaling the gradients by the learning rate, l. Second, subtracting the scaled gradients from the network's current parameters.

The learning rate, l, is a required hyper-parameter for gradient descent that affects the magnitude of updates. When updating a network that has a small l, its parameters will be adjusted by a very small amount. On the other hand, if a network has a large l, like 1, then updates will cause its parameters to jump around a lot and it is possible that the parameters will never converge to good values [12].

The momentum method of updating makes a small change to SGD. It essentially gives SGD a "short-term memory" so that some information is preserved preserved from prior updates [36]. In practice, this modification is a major improvement to vanilla SGD.

$$\begin{aligned} v_{t+1} &= \beta \times v_t + \nabla J(\theta_t) \\ \theta_{t+1} &= \theta_t - l \times v_{t+1} \end{aligned} \tag{3.2}$$

In this set of equations, the scalar β is a chosen parameter between zero and one. Common default choices for β are 0.9 or 0.99. the variable v_t, which is initially a zero vector, is the "short term memory" that saves some information about old updates. There are even improvements to SGD with momentum, such as Adagrad and ADAM [9, 18].

3.2 Neighbor Search

We do not perform any form of neighbor search in our methodology. Instead, we perform Maximum Inner Product Search (MIPS). However, they are very strongly related. In fact, the method that we use for MIPS is a direct extension of Locality Sensitive Hashing, a solution for approximate neighbor search. This section begins with nearest neighbor search and builds up to Locality Sensitive Hashing. In the next section, we introduce Asymmetric Locality Sensitive Hashing for MIPS.

3.2.1 Nearest Neighbor Search

The nearest neighbor search problem is important for many fields, including machine learning, pattern recognition, and data compression. It is an example of an *optimization problem*: the goal of nearest neighbor search is to efficiently minimize some objective function [3, 11].

Definition 1 (Nearest Neighbor Search). Given an objective function, $o : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, a set of points $C \subseteq \mathbb{R}^d$ and some query point $q \in \mathbb{R}^d$, return the point $p \in C$ that minimizes $o(q, p)$.

For our purposes, we want to find a point that is close to the query. So, when discussing neighbor searches, we will use $o(q, p) \mapsto ||q - p||_\gamma$ as the objective function, where $||x||_\gamma$ is some arbitrary norm of x and $||q - p||_\gamma$ is the distance between q and p.

3.2.2 Near Neighbor Search

Near neighbor search is very similar to nearest neighbor search; However, instead of returning the point that is closest to the query, one can return any point that is within some predefined distance from the query. For example, in the R-near neighbor search problem, when provided a distance parameter R and a query point q, we begin searching through the collection C. If we find a point $p \in C$ with $||p - q||_\gamma \leq R$ then p is a valid near neighbor of q. If no point in the dataset is within distance R from q, then the search failed [3].

An extension of near neighbor search is the (c, R)-approximate near neighbor search problem. Note that this is approximate: the query time is improved immensely at the expense of accuracy. However, even though it is approximate, for many applications it is likely to find a point that is close enough to the query to still be useful [3, 8, 16].

Definition 2 ((c,R)-Approximate Near Neighbor Search). We are given a collection of d-dimensional points $C \subseteq \mathbb{R}^d$, two parameters $c > 1$ and $R > 0$, and a value $\delta \in [0, 1]$. Build a data structure such that when given a query point $q \in \mathbb{R}^d$, if there exists an R-near neighbor of q in C, it returns any point $p \in C$ such that $||q - p|| \leq cR$ with a probability of success of $1 - \delta$.

Instead of returning a point within distance R from the query, we may now return a point within distance cR from q. This can be simplified by assuming $R = 1$. If $R \neq 1$ then we divide all points in the dataset by R so that R becomes 1. We can do this because dividing by a scalar will not affect the relative ordering of distances between points [3]. When we do this, $cR = c$. This effectively removes the parameter R and allows the name to be simplified to the c-approximate near neighbor search problem. So, with this updated name, we can say that if there is a point within unit distance from the query, there is probability $1 - \delta$ that a point within distance c from the query will be found [3].

3.2.3 Locality Sensitive Hashing (LSH)

Locality sensitive hashing is a solution for the c-approximate near neighbor search problem [3, 4, 8, 16]. As the name suggests, it uses hash tables to store points. It relies on specific hash families that have a high probability of producing the same hash for points that are near each other. If we use one of these families, by hashing the query point we are given a bucket that is likely to contain points that are near the query. We then do a linear search through that bucket for points within distance c. This results in accurate sub-linear search [16].

Definition 3 (Locality Sensitive Hashing). A family $\mathcal{H}$ is said to be (R, cR, p_1, p_2)-sensitive if for any $h \in \mathcal{H}$, two points $p, q \in \mathbb{R}^d$, and $c > 1$

- If $||q - p||_\gamma \leq R$ then $Pr(h(q) = h(p)) \geq p_1$,
- if $||q - p||_\gamma \geq cR$ then $Pr(h(q) = h(p)) \leq p_2$

This says that if q and p are near each other, then they have a high probability of hashing into the same bucket. If they are distant, then they have a lower probability of hashing into the same bucket. It is clear that we want $p_1 > p_2$. If $p_2 > p_1$ then there is a high probability that distant things are hashed into the same bucket, which is not desirable for a near neighbor search. Further, a very important trait for us is that LSH scales very well with increased dimensionality [11, 16]. Other methods for neighbor search, such as kd-trees, begin to perform poorly as the dimension of data increases.

3.2.4 Gap Amplification

One method that we will use in our experiments in Chapter 5 is Gap Amplification [3, 25]. Some implementations of LSH will have two parameters, K and L. The value K is the number of hashes used by each table and L is the number of hash tables. Using the concatenation $g(x) = \text{Concat}(h_1(x), \ldots, h_K(x))$ of K hash functions is effectively an AND operation that increases the gap between p_1 and p_2. Using multiple hash tables is an OR operation that increases both p_1 and p_2. Properly setting these parameters can increase the gap between the threshold probabilities p_1 and p_2 so that p_1 approaches 1 and p_2 approaches 0.

While multiple tables are needed for theoretical guarantees, it is more common to implement LSH using a multi-probe scheme [25]. This uses a single table, but probes multiple buckets. Every bucket is ranked based on their similarity to the hash. Buckets are searched from most similar to least similar.

3.2.5 Example: Hyperplane LSH

In the experiments we perform in chapter 5, we use the Hyperplane hash family [4, 5]. This hash family is typically used for cosine similarity; testing if two vectors have a small angle between them. At the most basic level, it is composed of a function b_i that is defined as

$$b_i(x) = \begin{cases} 1 & a^\intercal x \geq 0 \\ 0 & otherwise \end{cases} \tag{3.3}$$

Where the vector $a \in \mathbb{R}^d$ has elements $a_j \sim \mathcal{N}(0, 1)$. When used with gap amplification, there will be K hash functions that are concatenated together. It becomes a string of K bits, or a 2^K bit integer.

$$g(x) = int(b_K(x), \ldots, b_1(x)) \tag{3.4}$$

$g(x)$ breaks $\mathbb{R}^d$ into different regions using the multiple $b_i(\cdot)$. The vector a is the normal vector of a hyperplane. If a point x is above the this hyperplane, then $a^\intercal x = 1$ is returned for that bit. If it is below the hyperplane, then it is 0.

If two points are very near to each other, then they will probably be "above" and "below" the same hyperplanes and have the same hash.

3.3 Maximum Inner Product Search (MIPS)

MIPS is similar to the near neighbor search problem [31, 32]. The major distinction is that, instead of searching through a dataset for a point that is near the query, the goal is to find the point that maximizes the inner product with the query. So, given a query vector $q \in \mathbb{R}^d$ and a collection of points $C \subseteq \mathbb{R}^d$, the goal of MIPS is to find the vector $p \in C$ that maximizes $q^\intercal p$. For our methodology, we are going to be finding the filters that are likely to produce large inner products; we want to do a maximum inner product search, not a near neighbor search. Therefore, approximate MIPS is particularly relevant for us. There are a variety of solutions for MIPS, but we have chosen to use Asymmetric Locality Sensitive Hashing.

3.3.1 Asymmetric Locality Sensitive Hashing (ALSH)

Asymmetric locality sensitive hashing is a transformation of LSH that allows it to be used for MIPS. It does this by applying asymmetric transformations prior to hashing that make the distance between the two transformed vectors inversely proportional to the non-transformed vectors' inner product. If the two transformed vectors are close, then the original vectors likely have large inner products [31, 32]. Due to their strong relation, ALSH has nearly the same definition as LSH:

Definition 4 (Asymmetric Locality Sensitive Hashing). A family $\mathcal{H}$ and two functions $Q : \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$ and $P : \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$ are said to be $(R, cR, p1, p2)$-sensitive, if for a hash function $h \in \mathcal{H}$, they satisfy the following for any two points $q, x \in \mathbb{R}^d$ and $c < 1$

- if $q^\intercal x \geq R$ then $Pr(h(Q(q)) = h(P(x))) \geq p_1$
- if $q^\intercal x \leq cR$ then $Pr(h(Q(q)) = h(P(x))) \leq p_2$

The function $Q(\cdot)$ is called the query function. It is only applied when searching the data structure. The function $P(\cdot)$ is called the pre-processing function and is applied when building the data structure. As with LSH, we want $p_1 > p_2$. However, unlike LSH, we must have $c < 1$ so that $cR < R$ [31, 32].

For the Hyperplane LSH discussed in the previous section, good choices for the functions $Q(\cdot) : \mathbb{R}^d \to \mathbb{R}^{d+m}$ and $P(\cdot) : \mathbb{R}^d \to \mathbb{R}^{d+m}$ have already been found [32]. They are defined as

$$\begin{aligned} Q(x) &= \text{Append}_m(x, 0, 0, \ldots, 0) \\ P(x) &= \text{Append}_m(x, 0.5 - ||x||_2^2, 0.5 - ||x||_2^4, \ldots, 0.5 - |x||_2^{2^m}), \end{aligned} \tag{3.5}$$

There is just one assumption that we need for these functions to work for MIPS. If we want to search through the set $\mathbb{X}$, then each $x \in \mathbb{X}$ must have $||x||_2 \leq U < 1$. This can be achieved by scaling every item in the dataset with $U \div \max\limits_{x \in \mathbb{X}} ||x||$. Using these functions $P(\cdot)$ and $Q(\cdot)$, the original authors find the equality:

$$\frac{Q(q)^\intercal P(x)}{||Q(q)||_2 ||P(x)||_2} = \frac{q^\intercal x}{\sqrt{m \div 4 + ||x||_2^{2^{m+1}}}} \tag{3.6}$$

Notice that the value of $||x||_2^{2^{m+1}}$ approaches 0 as m increases because $||x||_2 \leq U < 1$. This means that $q^\intercal x$ is a multiple of the left-hand side of the equality. So, the authors find

$$\underset{x \in X}{\text{argmax}}\ q^\intercal x \simeq \underset{x \in X}{\text{argmax}}\ \frac{Q(q)^\intercal P(x)}{||Q(q)||_2 ||P(x)||_2} \tag{3.7}$$

This says that the x that maximizes the cosine of the angle between $Q(q)$ and $P(x)$ is similar or equal to the x that maximizes $q^\intercal x$ [32]. So, $Q()$ and $P()$ allow LSH to be used for MIPS. Interestingly, it has been shown that asymmetry is not necessary to use LSH for MIPS. For instance, SimpleLSH uses a single symmetric transformation and can outperform Hyperplane ALSH [29]. The base hash function used by Simple-LSH is Hyperplane LSH, but it uses a different $P()$ and $Q()$:

$$Q(x) = P(x) = \text{Append}(x, \sqrt{1 - ||x||_2^2}) \tag{3.8}$$

It is symmetric because $Q(x) = P(x)$. There is a more recent improvement to Simple-LSH called Norm-Ranging LSH [38]. Norm-Ranging LSH breaks the dataset into different sub-partitions based on the norms of each datum. Each sub-partition is hashed separately. Doing this helps to improve the distribution of points in the hash table. Both Simple-LSH and Norm-Ranging LSH have better query times than ALSH methods [29, 38].

Chapter 4

Methodology

Our primary goal is to improve the computational efficiency of convolutional neural networks when used on small devices. We intend to do this by reducing the number of transformations applied by each network layer. The method we have devised is a form of filter pruning that is inspired by another work that showed how ALSH can be used improve the performance of a fully-connected layer [34]. We wish to extend this idea to convolutional layers. Specifically, we want to use ALSH to quickly find a subset of filters that are likely to produce large inner products when applied to the input. The reason that this is restricted to small devices is that it there is an inherently sequential step. We use ALSH to analyze the convolution's input and determine which filters to use, and then apply those filters. It is not possible to efficiently analyze the input and apply the filters at the same time. Due to this being sequential, it would not make sense to use our methodology on a device that supports a large number of threads.

Unfortunately, there are some complications that make our goal a non-trivial extension of the fully-connected version [34]. First, The original paper that applies ALSH to fully-connected layers used a single vector as the network's input. This is not realistic for practical settings because most networks are trained with mini-batches. It is natural for a fully-connected layer's weights to be stored in a matrix. This makes it simple to search for the best weights to use; However, in a standard convolution implementation, a single input is a 3D tensor and each of the convolution's filters are 3D tensors. So, we are working with multi-dimensional tensors that are applied in strange ways; a single input may be treated like many vectors. Second, the author's of the fully-connected version used simple models on small datasets to test their methodology. As we are using convolutions, we want to test our methods on more challenging datasets using more practical models. In the remaining sections of this chapter, we overview our solutions to these problems in what we entitle an “ALSH Convolution.”

There are two main reasons we chose to use ALSH for our methodology: First, we specifically want to find large inner products quickly. ALSH allows us to quickly approximate which filters are likely to produce large inner products. Second, It is also important that ALSH can work well with high-dimensional data [16, 31, 32]. For our application, vectors may have hundreds or thousands of components. When the dimension is that high, many other methods for neighbor search or MIPS may begin to fail. We will explain why we chose to use Hyperplane LSH in our implementation, rather than L2-LSH or SimpleLSH [29, 31], in Chapter 5 because that choice was specific to our implementation, rather than the methodology that we are proposing.

4.1 ALSH Convolution

We break the ALSH Convolution into three parts: 1. the pre-pass, for creating the hash tables; 2. the forward pass, for inference; and, 3. the backward pass, for updating for the network. The pre-pass is a new addition that is done during the convolution’s construction. Its main task is building the ALSH tables that contain references to the convolution’s filters. Part 2 is a major modification to the standard convolution’s forward. Part 3 is a small change in the convolution’s backward pass to account for the changes in the forward pass.

Algorithm 3: ALSH Conv2d Pre-Pass

```
input  : Filters $F \in \mathbb{R}^{M \times c \times k \times k}$
         an LSH family $\mathcal{H}$
         a positive integer $K$
         a positive integer $L$
         an ALSH pre-processing function $P$
output: An ensemble of L ALSH tables
$Tables \leftarrow$ an ensemble of L hash Tables
foreach table in Tables do
    $g(x) \leftarrow (h_1(x), h_2(x), \dots, h_k(x))$ where $h_i \in \mathcal{H}$
    Let $g(x)$ be table's hash function
$F^* \leftarrow$ $F$'s filters transormed into the rows of a matrix
foreach table in Tables do
    foreach row of $F^*$ do
        insert the index of row into table[$g(P(row))$]
return $Tables$
```

The ALSH Conv2d Pre-Pass constructs an ensemble of L hash tables that each use some combination of K hash functions. Then, each of the M filters that were passed into the function are flattened into vectors of length $c \times k \times k$. The indices of the filter-vectors are inserted into each table at the index they are hashed to. These indices act as references into the filter matrix and make it simple to maintain order and avoid selecting the same filter multiple times in the forward pass. After the Pre-Pass has completed the convolution should be ready to use for inference. So, we now introduce the forward-pass of the ALSH Conv2d.

This pseudo-code of the forward-pass uses im2col in an effort to make indexing operations more clear; However, with some minor modifications, the forward pass can also be used with a convolution that is not implemented with im2col.

Algorithm 4: ALSH Conv2d Forward Pass

```
input  : A mini-batch of objects $I \in \mathbb{R}^{N \times c \times h_1 \times w_1}$
         Filters $F \in \mathbb{R}^{M \times c \times k \times k}$
         an ALSH query function $Q$
         an ensemble of hash tables $Tables$ containing the indices into $F$.
output: A mini-batch of objects $O \in \mathbb{R}^{N \times M \times h_2 \times w_2}$
begin
    $\hat{I} \leftarrow$ im2col($I$)
    $F^* \leftarrow$ $F$'s filters transormed into the rows of a matrix
    filtersToUse $\leftarrow \{\}$
    foreach $table \in Tables$ do
        hashCount $\leftarrow$ initially a zero-array
        foreach $column \in \hat{I}$ do
            h $\leftarrow$ table.applyHash($Q$(column))
            increment hashCount[h]
        hash $\leftarrow$ most frequent hash
        insert each item contained in table[hash] into filtersToUse

    /* Defining the active set                                    */
    $A \leftarrow$ rows of $F^*$ indexed by the elements of filtersToUse

    $O^* \leftarrow A \times \hat{I}$
    $O \leftarrow$ reshape $O^*$ into a mini-batch of objects and fill with zeros
    return $O$
```

The ALSH Conv2d forward-pass is essentially the same as the standard convolution forward-pass [12], but it inserts a few lines that find the active set of filters. The algorithm uses the tables that were created in the pre-pass to find a subset of the filters to use in the layer's matrix multiplication. Each table hashes the same regions of the input that the filters scan across. The bucket used by a table corresponds to the most frequently occurring hash value for that table. Using the most common hash is not guaranteed to work well, but we believe that it works well enough in practice. We discuss this in much greater detail in the following section, 4.3. We then use the filters that are inside of the selected buckets in a matrix multiplication with the input. Since we did not use every filter, the output will not have the correct dimension. So, using the indices in filtersToUse and O^*, we can create an object O that initially contains 0's and then fill the proper regions using O^*.

Algorithm 5: ALSH Conv2d Backward Pass

```
input  : A mini-batch of objects $dO \in \mathbb{R}^{N \times M \times h_2 \times w_2}$
         Filters $F \in \mathbb{R}^{M \times c \times k \times k}$
         The last Active Set A
         a set of indices, filtersUsed
         An ensemble of hash tables Tables
         The last input to the forward-pass $\hat{I}$
output: A mini-batch of objects $dI \in \mathbb{R}^{N \times c \times h_1 \times w_1}$
begin
    $dO^* \leftarrow$ reshape $dO$ as a matrix
    $d\hat{I} \leftarrow dO^* A^\intercal$
    $dA \leftarrow \hat{I}^\intercal dO^*$
    $dI \leftarrow col2im(d\hat{I})$
    use $dA$ and filtersUsed to update $F$
    update Tables with the new filters $F$
    return $dI$
```

The final part of the ALSH Convolution is the backward-pass. As with the forward-pass, this algorithm is a minor modification of the standard convolution update [12]. There are a few that points of particular note about this algorithm. One is that it can use the active set, A, to compute $d\hat{I}$. This is essentially a free optimization to the backward pass because A is a subset of F. So, as with the forward pass, the matrix multiplication is smaller than it would normally be. The second thing to note is col2im. This function is the backward pass of im2col and is not unique to the ALSH Convolution. The third aspect of the algorithm to note is that the hash tables are updated with the changed filters. This may seem like a taxing operation, but instead of refilling every table with all of the rows of F, we can just empty the buckets that were used in each table and only reinsert the rows of A. This is possible because the rows of A are the only filters that were updated.

We now offer a possible optimization for the ALSH Conv2d forward pass. In the first pseudo-code of the forward pass, the output is filled with zeros in the kernels where a filter was not applied. However, these zeroed regions are still used in the next layer's hash and convolution operation. This results in many unnecessary multiplications by zero that can be avoided. This variant of the forward pass prevents these unnecessary computations by sharing the current layer's active set with the next layer. We call this "Last-Active-Set Sharing."

Algorithm 6: ALSH Conv2d Forward Pass with Last-Active-Set Sharing

```
input  : A mini-batch of objects $I \in \mathbb{R}^{N \times c' \times h_1 \times w_1}$
         Filters $F \in \mathbb{R}^{M \times c \times k \times k}$
         an ALSH query function $Q$
         an ensemble of hash tables $Tables$ containing the indices into $F$,
         The indices of the previous ALSHConv2d's active set, $LAS$
output: A mini-batch of objects $O \in \mathbb{R}^{N \times M \times h_2 \times w_2}$,
         The set of filters used by this layer, filtersToUse

begin
    $\hat{I} \leftarrow$ im2col($I$) $\in \mathbb{R}^{[c' \times k \times k] \times [N \times h_1 \times w_1]}$

    /* Use only the kernels of each filter that align with the
       kernels of the input $I$.  So, each filter has the same number
       of kernels as the input.                                        */
    $tmp \leftarrow$ use LAS to index the kernels of each filter of F.

    $F \leftarrow$ $tmp$'s filters transformed into the rows of a matrix of the form
      $\mathbb{R}^{M \times [c' \times k \times k]}$
    filtersToUse $\leftarrow \{\}$
    foreach $table \in Tables$ do
        hashCount $\leftarrow$ initally a zero-array
        foreach $column \in \hat{I}$ do

            /* Similarly, only apply parts of the hash function that
               align with the kernels of the input $I$                 */
            $h \leftarrow$ table.applyHash($Q$(column), LAS)

            increment hashCount[h]
        hash $\leftarrow$ most frequent hash
        insert table[hash] into filtersToUse
    $A \leftarrow$ $F^*$ indexed by the elements of filtersToUse
    $O^* \leftarrow A\hat{I}$ // Matrix Multiplication
    $O \leftarrow$ reshape $O^*$ into a mini-batch of objects
    return $O$ and filtersToUse
```

This algorithm is similar to algorithm 4, but it adds a new input parameter LAS and a new output value filtersToUse. The value filtersToUse contains the indices of the filters that were used by the current layer. These are passed into the next convolution as the parameter LAS, or "Last-Active-Set." This LAS parameter is used to determine which kernels the current layer should apply to the input. Algorithm 4 and 6 compute the same thing; however, we consider algorithm 6 to be an optimization. We compare their compute times in Chapter 5.

4.2 Mode Probability

While describing the forward pass algorithm in the previous section 4.2, we claimed that using the mode of a table's hashes is an effective way to determine which bucket to include in the active set. This may be a simple choice, but it is not guaranteed to work. Fortunately, we believe that it is good enough in practice when using the Hyperplane ALSH family [32] and will detail why in the remainder of this section.

For now, we only consider LSH and will find similar probabilities for ALSH later. We want to find the probability that a value x in the dataset $\mathbb{X}$ has the same hash as the most frequently occurring hash in the set $\{h(q) \mid q \in Q\}$, where $h()$ is some concatenation of Hyperplane LSH functions. Essentially, we want to find:

$$Pr\{mode(\{h(q) \mid q \in Q\}) = h(x)\} \tag{4.1}$$

For this problem, we are given a locality-sensitive hash function h, a positive real number $c > 1$, a dataset $\mathbb{X}$, and a set of queries Q. Before we begin, let us briefly recall the definition of LSH [3, 5, 8, 16], discussed in section 3.3.3. For any arbitrary values of q and x that are in the sets Q and $\mathbb{X}$ respectively,

- if $||q - x||_\gamma \leq R$ then $Pr(h(q)) = h(x)) \geq p_1$
- if $||q - x||_\gamma \geq cR$ then $Pr(h(q) = h(x)) \leq p_2$

In addition to these threshold probabilities, we adopt a convention of assuming that the collision probability is monotonically decreasing in distance [31]. Further, we initially assume that the threshold probabilities are a step-function with constant values. So, if points q and x are close, $Pr(h(q) = h(x)) = p_1$. If they are distant, $P(h(q) = h(x)) = p_2$. Doing this makes it easier to model. We will loosen this assumption later. Finally, we define a discrete variable, $V_S(z)$, that is the number of occurrences of a value z in the list of hashes generated from a set S; the frequency of z in the set $\{h(s) \mid s \in S\}$.

Now are ready to begin finding the probability. There are three different types of queries that we must consider: First, when q is close to x. Second, when q is distant from x. Third, when q is neither close or far from x. We define three sets that separate these different classes of queries.

- $M_x = \{q \in Q \mid ||q-x||_\gamma \leq R\}$, so that each q in M_x has $Pr(h(q) = h(x)) = p_1$
- $N_x = \{q \in Q \mid ||q - x||_\gamma \geq cR\}$, where q in N_x has $Pr(h(q) = h(x)) = p_2$
- $O_x = \{q \in Q \mid R < ||q - x||_\gamma < cR\}$, Under our assumption that the collision probability is monotonically decreasing, we know for a q in O_x that $p_2 < Pr(h(q) = h(x)) < p_1$. This value cannot be precisely known in theory; however, we will denote it as p_3.

Each of the hashed elements of M_x are either equal to $h(x)$ with probability p_1 or they are not with probability $1 - p_1$. So, the probability that $V_{M_x}(h(x))$ equals n is a binomial distribution, denoted with the function $b(S, n, p) = \binom{|S|}{n}(p)^n(1 - p)^{|S|-n}$. Thus, we have that M_x has n elements hash to $h(x)$ as

$$Pr\{V_{M_x}(h(x)) = n\} = b(M_x, n, p_1) \tag{4.2}$$

We find similar probabilities for N_x and O_x.

$$\begin{aligned} Pr\{V_{N_x}(h(x)) = n\} &= b(N_x, n, p_2) \\ Pr\{V_{O_x}(h(x)) = n\} &= b(O_x, n, p_3) \end{aligned} \tag{4.3}$$

The resulting hashes of each element of the sets M_x, N_x, and O_x are the results of discrete independent events. So, the probability that $h(x)$ occurs n times in the set $\{h(q) \mid q \in Q\}$ is the sum of all the combinations of probabilities where the number of occurrences of $h(x)$ sums to n across the three sets M_x, N_x, and O_x.

$$Pr\{V_Q(h(x)) = n\} = \sum_{j=0}^{n} \sum_{i=0}^{j} b(M_x, n-j, p_1) \times b(N_x, j-i, p_2) \times b(O_x, i, p_3) \tag{4.4}$$

Before considering the case with an arbitrary number of possible hash values, we will first find the probability that $h(x)$ is the most frequently occurring hash in the case where there are only two possible hash values. This will be useful for us later. As a more succinct notation, we introduce a new function $mode_\beta()$ that is defined as

$$mode_\beta(S) = mode\{(h_1(s), \ldots, h_\beta(s)) \mid s \in S\} \tag{4.5}$$

In the case where $\beta = 1$, we can find the probability that $h(x)$ is the most frequent hash by finding $Pr\{V_Q(h(x)) > |Q| \div 2\}$; the probability that more than half of the values in the list of hashes are equal to $h(x)$. As $V_Q(h(x))$ is discrete, this probability is the summation of $Pr\{V_Q(g(x)) = n\}$ for each n between $|Q| \div 2$ and $|Q|$.

$$Pr\{mode_1(Q) = h(x)\} = \sum_{n=\frac{|Q|}{2}+1}^{|Q|} Pr\{V_Q(h(x)) = n\} \tag{4.6}$$

In reality, we will not be restricted to only two hash values. So, we now want to find a similar probability for the case where there are an arbitrary number of hash functions concatenated together. Rather than being the probability that $h(x)$ occurs more than half of the time, this is the probability that $h(x)$ occurs at least as frequently as all of the other $2^\beta - 1$ values.

$$\begin{aligned} Pr\{mode_\beta(Q) = h(x)\} &= \prod_{y \in \mathbb{X}} Pr\{V_Q(h(x)) \geq V_Q(h(y))\} \\ &= \prod_{y \in \mathbb{X}} \left(\sum_{j=0}^{|Q|} \sum_{i=j}^{|Q|} Pr\{V_Q(h(x)) = i\} \times Pr\{V_Q(h(y)) = j\} \right) \end{aligned} \tag{4.7}$$

Notice that the initial value of i is j. The frequency of $h(x)$ is always greater than the frequency of $h(y)$. Further, by using the symbol $\geq$, we allow for $h(x) = h(y)$. This is important because $h(x)$ may equal $h(y)$, so they will have the same probability. We have now found the probability that a value $x \in \mathbb{X}$ has the same hash as the most frequently occurring hash in the set $\{h(q) \mid q \in Q\}$. We will refer to equation 4.7 as the *mode probability* because it is the probability that $h(x)$ is the mode of $mode_\beta(Q)$. Of course, equation 4.7 can also work for $mode_1$, but equation 4.6 will be easier to use in some instances in the following sections.

4.2.1 Analysis of the Optimal Case

For LSH to be optimal, it must have $p_1 = 1$, $p_2 = 2$, and $c \to 1^+$.

Lemma 4.1. *When $p_1 = 1$, $p_2 = 0$, and $c \to 1^+$, for each $x \in \mathbb{X}$ with a set M_x that contains more than half of the elements of Q, the hash $h(x)$ is guaranteed to be $mode_1(Q)$.*

Proof. Using equation 4.4, when $p_1 = 1$, $p_2 = 0$, $c \to 1^+$, and for any $x \in \mathbb{X}$ with $|M_x| > |Q| \div 2$, we find that $Pr\{mode_1(Q) = h(x)\} = 1$. Similarly, for any $x \in \mathbb{X}$ where $|M_x| < |Q| \div 2$, we find that $Pr\{mode_1(Q) = h(x)\} = 0$. Thus, Theorem 4.1 is true. □

The full derivations of $Pr\{mode_1(Q) = h(x)\} = 1$ and $Pr\{mode_1(Q) = h(x)\} = 0$ can be found in the Appendix 6.1. We can use Theorem 4.1 to find a similar guarantee for an arbitrary number of buckets. We need to consider pairs of $h(x)$ and other buckets individually. If h(x) is the mode among all pairs of the form $(h(x), h(y))$, then $h(x)$ is the mode of the entire hash list.

Theorem 4.2. *When $p_1 = 1$, $p_2 = 0$, and $c \to 1^+$, for the $x \in \mathbb{X}$ with the largest set M_x, the hash $h(x) \in \mathbb{N}$ is guaranteed to be the mode of $mode_\beta(Q)$.*

Proof. Suppose we have the $x \in \mathbb{X}$ with the largest set M_x. Now, consider a pairs of hashes of the form $(h(x),\ h(y))$ where $y \in \mathbb{X}$. Because $p_1 = 1$ and $p_2 = 0$, everything in the set $(N_x \cup N_y) \setminus (M_x \cup M_y)$ will not vote for $h(x)$ or $h(y)$. As $c \to 1^+$ we know that $|O_x| = 0$, so we do not need to consider it. To find the mode among $h(x)$ and $h(y)$, we only need to consider elements in the set $Q \setminus ((N_x \cup N_y) \setminus (M_x \cup M_y))$. Everything in this set is guaranteed to vote for $h(x)$ or $h(y)$. From Theorem 4.1, we know that

$$Pr\{mode_1(Q \setminus ((N_x \cup N_y) \setminus (M_x \cup M_y))) = h(x)\} = 1 \tag{4.8}$$

Since y is arbitrary, we know that this is true for any $y \in \mathbb{X}$. As $h(x)$ wins the vote mode among all pairs with other buckets, it is guaranteed to be the mode of the entire hash set. We can now conclude that if $|M_x| > |M_y|$ for all $y \in \mathbb{X}$, then

$$Pr\{mode_\beta(Q) = h(x)\} = 1 \tag{4.9}$$

Thus we know that Theorem 4.2 is true. □

Now, in this optimal case, we can loosen our assumption that the probabilities p_1 and p_2 are exactly the collision probabilities. To do so, we must think back to the standard definition of LSH [3, 4, 11, 16] that states that

- if $||q - x||_\gamma \leq R$ then $Pr(h(q)) = h(x)) \geq p_1$ and
- if $||q - x||_\gamma \geq cR$ then $Pr(h(q) = h(x)) \leq p_2$

In optimal case, we have $p_1 = 1$ and $p_2 = 0$. So, when using the standard definition of LSH, because $Pr(h(q) = h(x)) \geq p_1$ and $p_1 = 1$, we know that $1 \leq Pr(h(q) = h(x)) \leq 1$. Therefore, if $||q - x||_\gamma \leq R$ then $Pr(h(q) = h(x)) = 1$ for all $y \in \mathbb{X}$. Similarly, if $||q - x||_\gamma \geq cR$ then $Pr(h(q) = h(x)) = 0$ for all $y \in \mathbb{X}$. Therefore, the optimal case is true for LSH and does not require that p_1 and p_2 are exact.

4.2.2 Extending to ALSH

We have just looked at the probabilities for LSH; however, our methodology uses ALSH. So, we need to show that these same equations can be derived for ALSH. Recall that the definition of ALSH [31, 32] states that

- if $q^\intercal x \geq R$ then $Pr(h(Q(q))) = h(P(x))) \geq p_1$
- if $q^\intercal x \leq cR$ then $Pr(h(Q(q)) = h(P(x))) \leq p_2$

We will make similar assumptions based on this. We assume that the collision probability is monotonically increasing in $q^\intercal x$ and that the threshold probability are exact. So, if points q and p have a large inner product, the probability that

$Q(q)$ and $P(x)$ is large. This allows us to derive probabilities in a similar way to how we derive equations 4.3 and 4.4.

Because ALSH uses a query function, $Q(\cdot)$, we represent the set input queries as Y instead of Q. The dataset that we are searching through is still $\mathbb{X}$.

We can now redefine the sets M_x, N_x, and O_x to derive the new probabilities.

- $M_x = \{q \in Y \mid q^\intercal x \geq R\}$, so that $q \in M_x$ all have $Pr(h(Q(q)) = h(P(x))) = p_1$
- $N_x = \{q \in Y \mid q^\intercal x \leq cR\}$, so that $q \in N_x$ all have $Pr(h(Q(q)) = h(P(x))) = p_2$
- $O_x = \{q \in Y \mid cR < q^\intercal x < R\}$, Using our assumption that the collision probability is monotonically decreasing, we know for a $q \in O_x$ that $p_2 < Pr(h(Q(q)) = h(P(x))) < p_1$.

Using these, we can derive probabilities that are *identical* to equations 4.3 and 4.4. We will not show them here, because the derivation really is the same. While the probabilities look the same, they model different things. Both probabilities model the probability that, for an input set of queries, a bucket x is selected. However, the probabilities that use LSH have a dependence on the distance between the queries and x. While the probabilities that use ALSH have a dependence on the inner products of the input queries and x.

4.3 Training Strategy

The training strategy that we propose is similar to that used by other filter pruning methods. These methods iteratively prune filters and retrain to account for changes in accuracy [23, 28]. Our method has a slight distinction though. We iteratively "replace" and retrain. We replace a standard convolution with an ALSH Convolution every n epochs during retraining. The filters of the ALSH Convolution are initialized so that it has the same weights as the convolution that it is replacing. If we replace convolutions with ALSH Convolutions without retraining, there is a noticeable dip in accuracy; eventually becoming random guesses. We show in our experiments that retraining is sufficient to bring the accuracy back to a decent level.

This algorithm shows one possible way to implement our training strategy. There are some potential improvements that could be made, such as gradually increasing the gap between replacement during training. Or, using a validation set and making a replacement when the loss has stopped decreasing. For our experiments, we stick with a simple algorithm that is very similar to the one shown that sets the number of epochs between replacements to be a constant value.

Algorithm 7: ALSH Conv2d Train-and-Replace

```
input  : A model $M$,
         number of epochs $E$,
         and the number of epochs between each replacement $gap$
output: $M$

depth ← number of feature layers in $M$
for $n$ in $\{1, \ldots, E\}$ do
    if $n \% gap = 0$ then
        replace M.features[depth] with an ALSH Conv2d
        depth ← the next “deepest” convolution in M.features.
    train(M)
while M.weights have not converged do
    train(M)
return $M$
```

Chapter 5

Experiments

To test our proposed methodology, we perform experiments using three popular image classification datasets:

- **CIFAR-10** consists of 60,000 images from ten classes [19]. Each class contains 6,000 images that are 32×32 RGB pixels. 50,000 images are used for training and the remaining 10,000 images compose the test set. We did not use a validation set while training.
- **CIFAR-100** is very similar to CIFAR-10, but has 100 classes instead of ten [19]. There are 500 training images for each class and 100 test images for each class. Again, we do not use a validation set while training.
- **MNIST** has 70,000 images of hand-written digits. 60,000 are used for training and 10,000 are used for testing [21].

We test our methodology using multiple network models. Namely, AlexNet [20] and VGG-11 [33]. These models are relatively small for modern convolutional neural networks. Our methodology is specifically intended to be used with small devices, so it would not make sense to use networks on the scale of Inception-Resnet-V2.

For both of these models, we use the implementations from torchvision.models that are pre-trained on Imagenet-12 and do not use batch normalization [30]. We retrain both models on CIFAR-10 and CIFAR-100 until they achieve high accuracy on each dataset. After they get high accuracy, we apply our train-and-replace strategy. The main architectural detail of note about these models is the number of convolutions that they have. AlexNet has five convolutions and VGG-11 has eight [20, 30, 33]. In addition to using AlexNet and VGG-11 on CIFAR-10 and CIFAR-100, we perform additional tests on MNIST [21] with an extremely small custom model that has three convolutions.

5.1 Implementation Details

In order to highlight that our methodology can be implemented using existing high-level deep learning libraries, the bulk of the implementation of ALSH Conv2d is written using PyTorch Tensor operations. Additionally, a majority of the implementation of Hyperplane ALSH uses PyTorch Tensor operations [30]. An interesting point that may help to highlight the trade-off between hashing every region and dropping filters is that we implemented the Hyperlane LSH [5] function using PyTorch's functional convolution. The Hyperplane LSH is composed of functions b_i that are defined as

$$b_i(x) = \begin{cases} 1 & a_i^\intercal x \geq 0 \\ 0 & otherwise \end{cases} \tag{5.1}$$

In one sense, we can treat a_i like a filer of a convolution because it computes an inner product with x. If the table's hash function is defined as $g(x) = (b_1(x), \ldots, b_n(x))$, we can make the vectors a_i the filters of a convolution. We can then compute $g(x)$ by applying that convolution to the input and then determine the bit values and final hash value from that convolution's output. Since our hash function is implemented using a convolution, as long as the number of hash tables and hash functions per table is smaller than the number of filters, there is potential for a speedup. However, there is overhead when finding the most frequent hashes in order to generate the active set.

5.1.1 Hash Family

We chose to use the Hyperplane ALSH family purely because it was possible to implement it using a convolution. This allowed us to keep our implementation high-level. We are not aware of any other ALSH families that can be implemented in a similar fashion. This is because of the $Q()$ function that is used by ALSH. For Hyperplane ALSH, $Q()$ appends zeros to its input [32]. However, other ALSH families require $Q(x)$ to append some function of the norm of x [29, 31, 38]. Consider some input: since the regions of the input that a convolution scans across often overlap, it is not possible to append the norm of these regions to the input. So, while these $Q()$ functions could work with lower-level implementations, they cannot work with the high-level implementation we chose.

For each ALSH Convolution in every test, we use the same parameters to create the hash tables. We found that using three hash tables with a hash function that is the concatenation of five of the $b_i()$ functions works well and is small enough to provide a significant speedup. For our hashing setup, we chose to use multiple tables. In practice, while multiple tables is necessary for theoretical guarantees, it is common for ALSH to use a multi-probe scheme [25]. However, our tables are not large enough for storing redundant data to be a major concern.

Further, we use a table's top five most frequent hash values to determine which buckets to generate the active set with. Finally, we set $m = 2$ for the asymmetric functions, $Q(x)$ and $P(x)$, in all of our tests [31, 32]. These settings make it so that about fifty percent of the filters will be used on average.

5.2 Experimental Settings

There are some universal settings that we used in all of our tests. We will specify any settings that are unique to a test when we get to them. One of the first things to note is the data augmentation that we used in our experiments. We normalize each image using the settings recommended by PyTorch so that they have a mean of $(0.485, 0.456, 0.406)$ and deviation of $(0.229, 0.224, 0.225)$ [30]. For the training data, we take a 224×224 random resized crop of each image and perform a random horizontal flip. For the test data, we resize the images to be 256×256 pixels and then make a 224×224 center crop. The batch size that we use is dependent on the model and dataset, but the training batch and testing batch sizes are always the same. All of the models that we used are from torchvision.models [30]. They have been pre-trained on ImageNet-12, but we retrain them on each dataset. Finally, our timing tests were run on an Intel Xeon ES-2695 v4.

5.3 Experimental Results

5.3.1 Compute Time

We show the average batch time during testing using a batch size of 64 images. Table 5.1 show the times on each dataset and network without the ALSH Convolution. For each model, we scale the images in the dataset to 224 × 224. So, the times should be similar for each dataset. We include all of them though.

As the buckets do not have a great distribution, the variation between iterations is fairly small. However, because the hashes used are random, the distribution in the buckets can change across different runs. So, during some runs the average number of filters used may change quite a lot. So, we report the average compute time of 20 batches of 64 images that are 224x224 for five trials. We show the average of the trials for each dataset and network.

Trial	Model	CIFAR-10 Time(s)		CIFAR-100 Time(s)	
		Default	ALSH	Default	ALSH
1	Alexnet	2.93	1.81	2.94	1.87
	VGG-11	25.49	18.88	25.23	17.20
2	Alexnet	2.95	1.86	2.94	1.87
	VGG-11	25.24	17.32	25.23	18.61
3	Alexnet	2.94	1.74	2.94	1.78
	VGG-11	25.24	18.41	25.25	18.55
4	Alexnet	2.94	1.87	2.94	2.00
	VGG-11	25.23	19.50	25.34	17.75
5	Alexnet	2.93	1.89	2.95	1.98
	VGG-11	25.43	18.57	25.29	18.80
Ave.	Alexnet	2.94	1.83	2.94	1.90
	VGG-11	25.33	18.54	25.27	18.18
Stdev.	Alexnet	.006	.05	.004	.08
	VGG-11	.11	.71	.04	.61

Table 5.1: Average Time per Batch of 64 Images

For both networks, we replaced all of the convolutions with ALSH Convolutions. We performed these timing tests using PyTorch 0.4.0 [30]. With this version of PyTorch, the ALSH Convolution is faster on average. The greater deviation of times could be caused by the random nature of ALSH. The filters will always be partitioned in different ways in each trial and that could impact which subset ends up being used.

We perform similar tests using PyTorch 1.0.1 [30]. In these tests, we show how the number of ALSH Convolutions in a network affects the compute time. To do this, we show recorded the times that the network took to process a single batch. We replace an existing convolution with an ALSH Convolution every 20 iterations. Since the images are scaled to be the same size, even though the datasets are different, the average time to process a batch should be the same for each dataset. So, for these tests we arbitrarily decided to use CIFAR100.

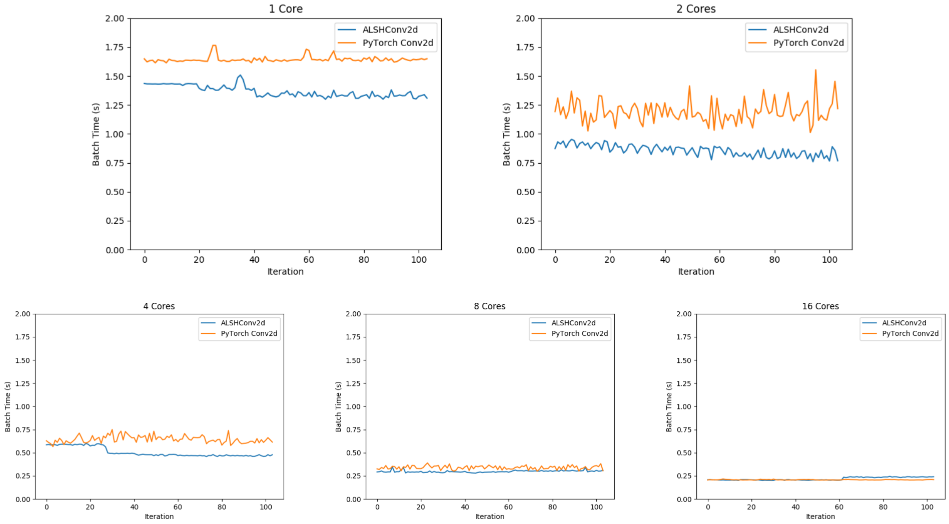

Figure 5.1: AlexNet on CIFAR100 with All Features

Figure 5.1 shows the times that different implementations of AlexNet took to compute a batch for 100 batches. One implementation, shown in orange, is composed purely of PyTorch's Conv2d. The other, in blue, has an ALSH Convolution replace PyTorch's Conv2d every twenty iterations. The ALSH Convolution used in these plots does not use Last-Active-Set sharing. As desired, our methodology offers a small speedup to PyTorchs' Conv2d when there are a small number of cores. As the number of cores increases, the speedup becomes negligible.

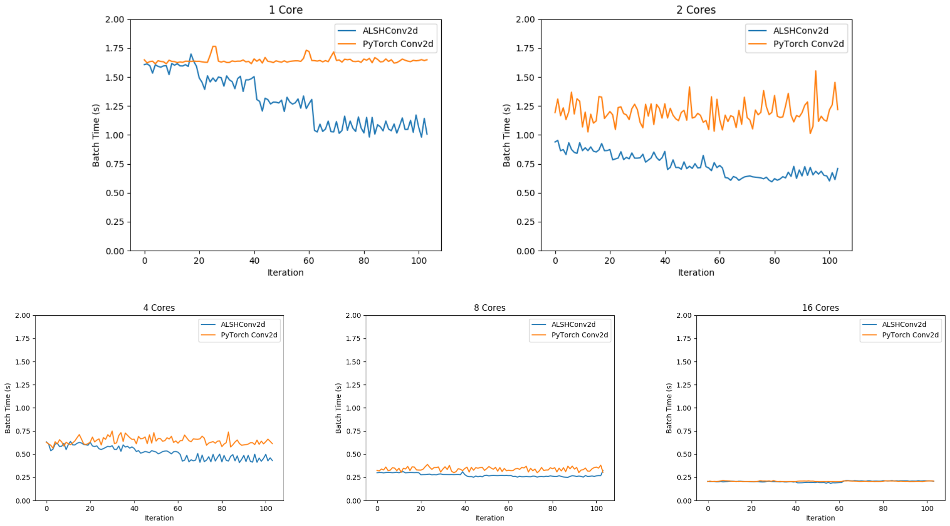

FIGURE 5.2: AlexNet on CIFAR100 with Last-Active-Set Sharing

Figure 5.2 is similar to 5.1; it also shows time to compute batches for two versions of AlexNet. The difference is that the ALSHConv2ds used in 5.2 employ Last-Active-Set Sharing. These plots indicate that Last-Active-Set sharing offers greater speedup as the number of ALSH Convolutions in the networks increases. Again, as the number of cores increases, the speedup becomes insignificant.

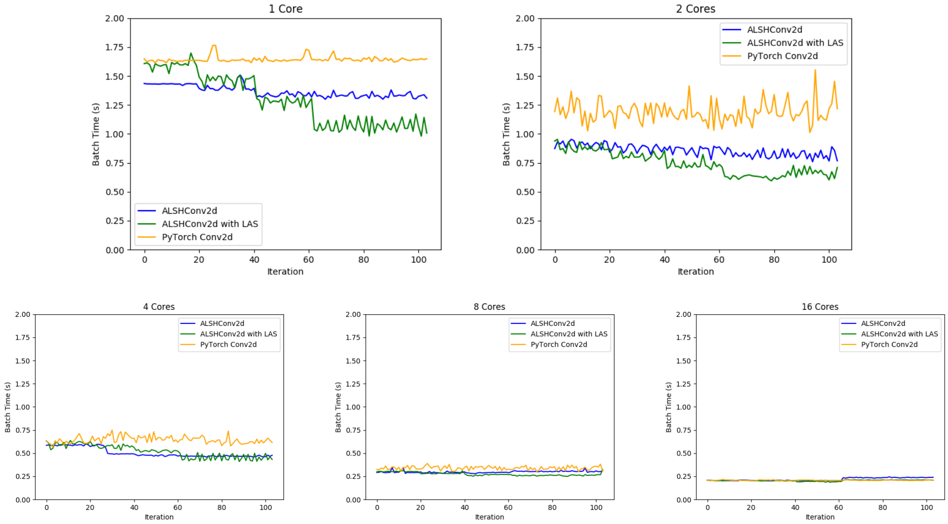

FIGURE 5.3: AlexNet on CIFAR100

Figure 5.3 shows the data from figures 5.1 and 5.2 plotted together. We can see that when the AlexNet has four ALSH Convolutions with Last-Active-Set Sharing, it has better compute time compared to vanilla ALSH Convolutions. There also seems to be a greater correlation between the number of layers that use the ALSHConv2d and a speedup. The green line shows a clear step down in compute time when an ALSHConv2d is inserted into the network.

We now make similar comparisons for VGG-11. Again, these use PyTorch 1.0.1 [30]. As with the previous plots, figure 5.4 shows the time that VGG11 takes to compute a batch of 64 images that are 224 × 224 pixels. We replace a standard convolution with an ALSH Convolution every 20 iterations.

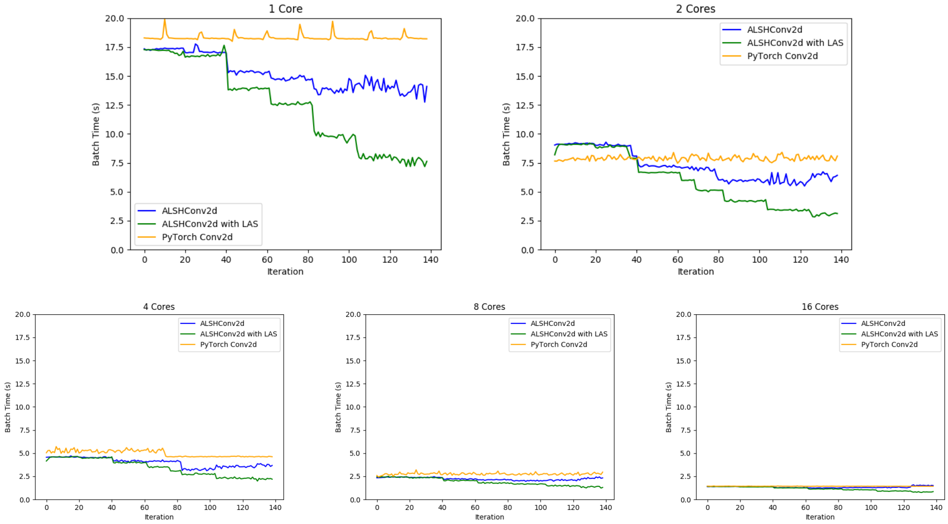

FIGURE 5.4: VGG11 on CIFAR100

When using Last-Active-Set sharing, there is a noticeable improvement to the regular ALSH Convolution implementation. Our methodology offers a substantial speedup to VGG11 when there is a small number of cores. When 1 core is used, there is approximately a 2× speed up. Even when using 16 cores, there is a consistent speed up when using Last-Active-Set sharing. It scales surprisingly well.

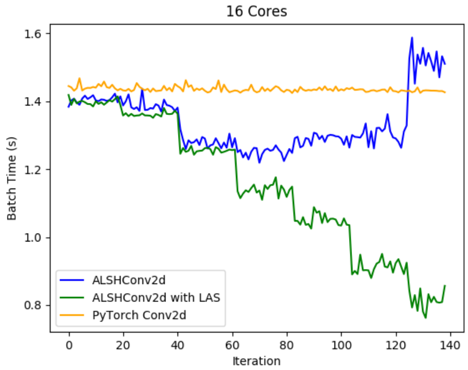

FIGURE 5.5: Close-up of VGG11 on CIFAR100 with 16 cores

5.3.2 Accuracy

In this section, we show how our methodology impacts the accuracy of a network. The "Normal" accuracy that we report is the initial accuracy of the network before we performed our train-and-replace training methodology to construct the ALSH network. Essentially, the "Normal" versions are pure CNN architectures without ALSH Convolutions. To achieve the normal network accuracies, we used a model that was trained to get high accuracy on ImageNet-12 and then retrained it on each dataset until it attained reasonably high accuracy [30]. The "ALSH" accuracy that we report is the same model, but after our train-and-replace strategy has been applied. For instance, Table 5.2 reports the model's accuracy with only a single ALSH Convolution replacement.

In these tests, some of the settings that we used for our train-and-replace strategy are consistent and are fairly standard. For instance, We always use the Cross Entropy loss function because it works well for multi-class classification. We also use SGD with momentum that had $\beta = 0.9$ and an initial learning rate of 0.001. The learning rate is static while training-and-replacing. After all of the ALSH Convolutions have been inserted, we decay the learning rate every thirty epochs. While performing tests, we noticed that the filters do not change buckets in the hash tables frequently. So, rather than updating the hash tables every iteration, we update them after each epoch. Some settings are different for each network/model combination. Specifically, the number of replacements, the number of epochs between replacements, the number of epochs run after the final replacement.

Model	Dataset	#Replace	Gap	Post-Epochs
AlexNet	CIFAR-10	4	35	40
AlexNet	CIFAR-100	4	30	50
VGG-11	CIFAR-10	7	35	40
VGG-11	CIFAR-100	7	35	50

Table 5.2: Unique Settings for ALSH Models

The term "Gap" is the number of epochs in-between replacements and "Post-Epochs" is the number of epochs after every ALSH Convolution has been inserted into the network. These values were not rigorously determined and can likely be improved.

To begin our analysis of how our methodology affects model accuracy, we looked at the case when only one layer in the network is an ALSH Convolution. We replace the "top" convolution before the classification layers of the network.

Model	CIFAR-10		CIFAR-100	
	Normal	ALSH	Normal	ALSH
AlexNet	92.52%	91.81%	72.83%	70.12%
VGG-11	94.58%	94.10%	77.81%	76.34%

Table 5.3: Accuracy: Replacing only the "Top" Convolution

We can see from this table that the by replacing the last convolution in the "feature" layers of the network that there is the expected dip in accuracy, but it is always within 3% of the original accuracy. As we did not exert much effort trying to find optimal settings, it is likely that this dip in accuracy can be reduced. Next, we look at the accuracy when we make replacements deep into the network.

Model	CIFAR-10		CIFAR-100	
	Normal	ALSH	Normal	ALSH
AlexNet	92.52%	87.02%	72.83%	62.31%
VGG-11	94.58%	88.69%	77.81%	68.34%

Table 5.4: Accuracy: "Deep" Replacements.

With AlexNet, we replaced four of the five convolutions and for VGG-11, we replaced seven of the eight convolutions. We can see that there is a much larger decrease in the classification accuracy compared to the single replacement. Unfortunately, even though our training settings are likely non-optimal, the current dip in accuracy is still quite large. In fact, VGG-11 with ALSH Convolutions is slower and less accurate than the normal implementation of AlexNet. Again, finding optimal settings was not a major concern for us. So, it is likely that the accuracy that we report can be improved.

There are a few methods, beyond parameter selection, that could potentially improve the accuracy. First, we could make the replacements more gradual. At the moment, replacing an existing convolution with an ALSH Convolution causes, very suddenly, about half of the filters to be used. One way to make the replacement more gradual would be to start with a large number of tables used and gradually remove some. In our tests, we always used three tables. Instead, we could start with ten tables and remove one every epoch until three remain. This is closer to the training strategy used by other filter pruning methods that remove a filter every iteration [23, 28].

Changing the frequency of replacements could also improve accuracy. These images are of the training loss. There is a peak in the loss when a replacement happens. By retraining after a replacement, the loss is able to stabilize some.

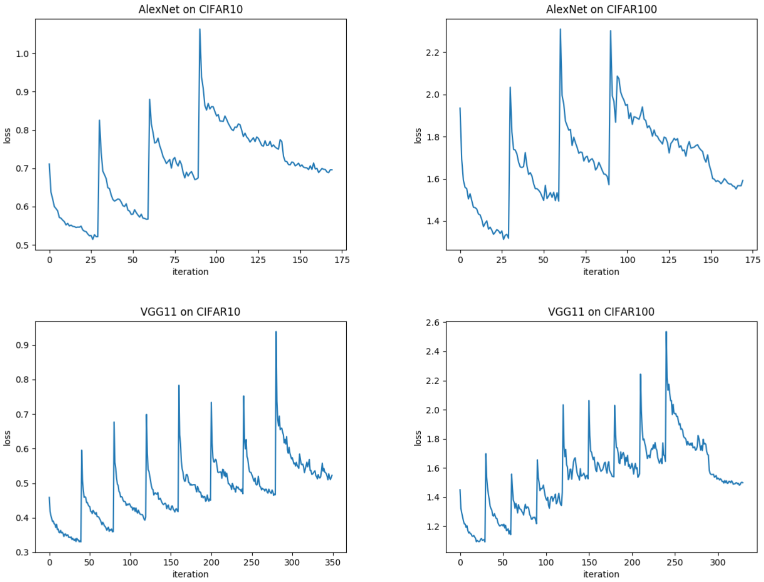

FIGURE 5.6: Training Losses

Judging from the training loss, it appears that in some cases the gap between replacements could have been larger. It may be beneficial to use a smarter function to set the gap between replacements, rather than a constant number. This could mean using a validation set and making replacements when the validation loss stops decreasing. Or, it could just be a function that gradually increases the gap between replacements.

5.3.3 Bucket Statistics

During our tests on CIFAR-10 and CIFAR-100 [19], we noticed that the number of filters used did not vary substantially across iterations during inference. While the number of filters used was not always the same, there was a subset of filters that were almost always being used. Hyperplane ALSH is known to have a poor distribution of bucket sizes [29, 38], but we were not sure if this was the cause. It is possible that it is being caused by the ReLU hash function that makes a convolution's output semipositive. We have decided that using an activation function that applies a similar transformation to positive and negative values may be beneficial. So, we perform more experiments using the Softshrink, Hardshrink, and

Tanh activation functions in an effort to fix the poor bucket distribution that we experience when using ReLU.

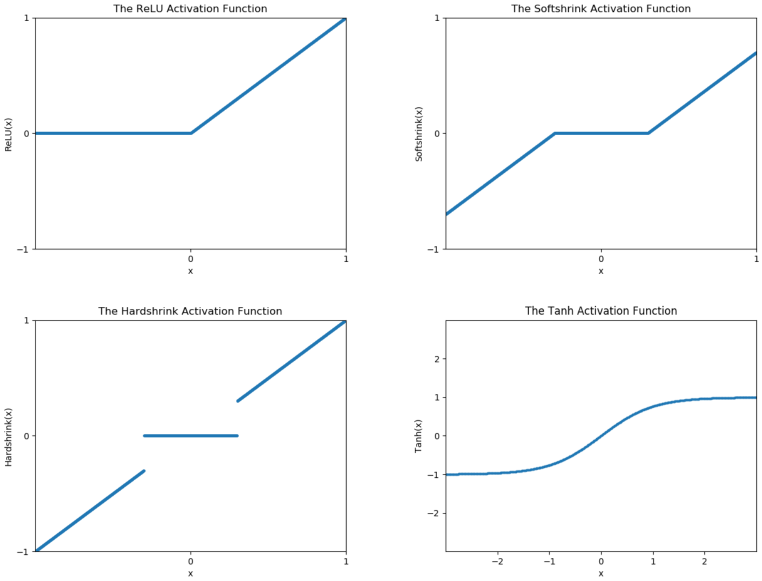

Figure 5.7: ReLU V.S. Softshrink

For these tests, we employ a small model on MNIST made of three convolutions that each have 32 filters. We are not concerned with performance or accuracy in these tests, we just want to compare the frequency that certain hashes occur when using different activation functions.

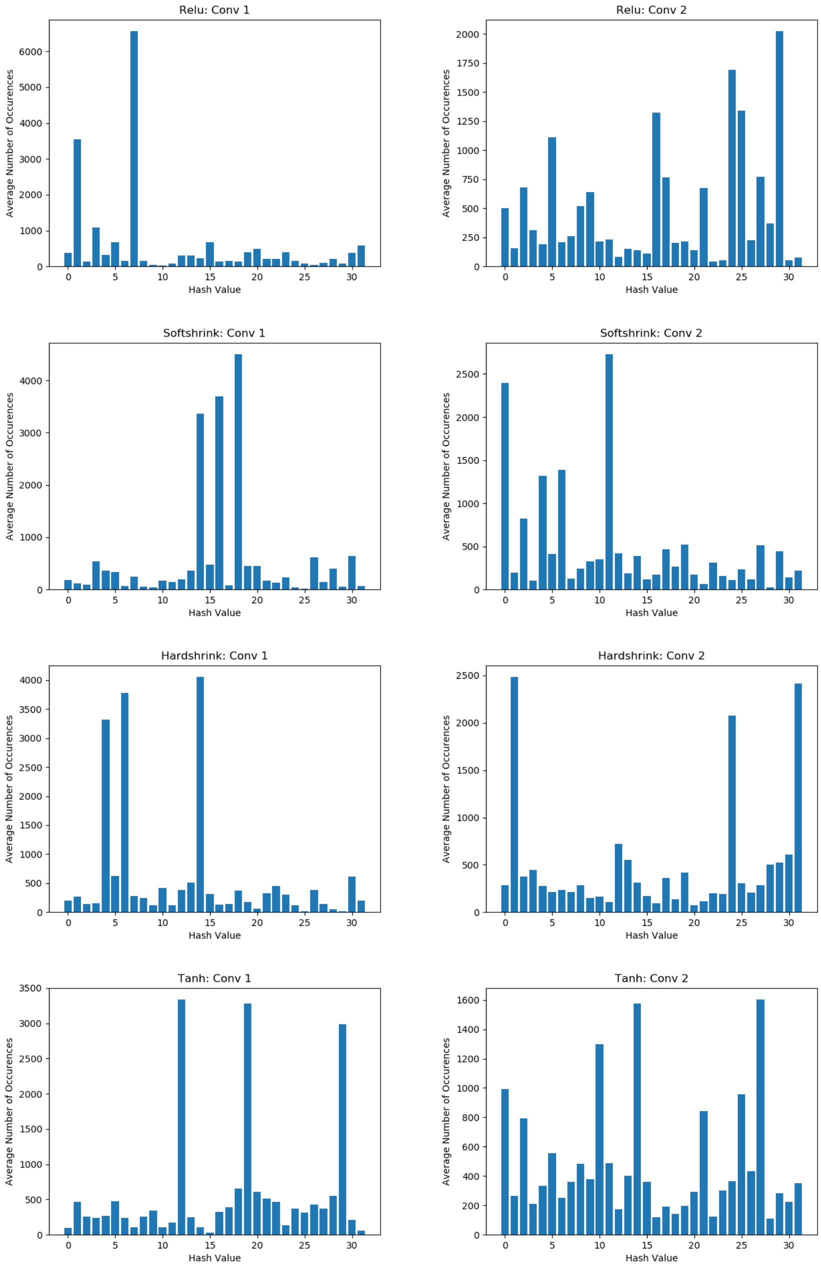

Figure 5.8: Average of hashes during inference

We looked at the number of times that each hash occurred during an iteration. Figure 5.7 displays the average number of times a hash occured across an entire test cycle. In the left column of the plots, "Conv 1" refers to the middle convolution of the three in the network. "Conv 2" in the right column refers to the top convolution. Every activation function has a few buckets that contain many more points than the other buckets. So, the ReLU activation function is not entirely responsible for the poor distribution. This is consistent with prior work on ALSH.

The poor bucket distribution is likely a failure of Hyperplane ALSH [29, 38] rather than being caused by ReLU.

While each activation function has a poor distribution, we notice that Tanh has a slightly better distribution than ReLU. In Conv 1, Tanh peaks at about 3500 elements while ReLU peaks at over 6000. In Conv2, Tanh peaks at 1600 and ReLU peaks at a bit over 2000. So, while the activation function does not solve the problem, it does appear to have an affect. We will look at this more deeply. To judge the quality of the bucket distribution, we can look at the max and the deviation. The lower the max and deviation, the better the distribution should be.

Trial	Statistic	ReLU		Soft		Hard		Tanh	
		1	2	1	2	1	2	1	2
1	Max	6559	2021	4496	2726	4346	2215	3334	1601
	Stdev	1236	500	1077	620	1151	543	860	397
2	Max	4126	2655	3787	2075	4378	2372	3199	1792
	Stdev	1111	645	955	514	1057	460	804	392
3	Max	3776	2893	3857	1802	3875	2552	3732	2162
	Stdev	911	713	971	441	992	628	951	434
4	Max	3985	2607	4134	2194	3629	2092	3720	2084
	Stdev	1005	663	993	455	942	434	937	403
5	Max	4002	2670	3682	2126	3524	2173	3378	1968
	Stdev	979	596	988	485	574	574	875	497
Ave	Max	4489	2569	3991	2328	3950	2280	3472	1921
	Stdev	1048	623	994	503	943	527	885	424

TABLE 5.5: Bucket Distributions across Trials

Again, we found the average of the number of times a hash occurred across an entire test. Table 5.5 displays the maximum and the standard deviation of the buckets sizes in the average. We test using the same four activation functions before the middle and top convolutions, numbered 1 and 2, for five trials. We also show the average of each trial. For each column we color the largest max red and the smallest max blue.

This table indicates that the choice of activation function does affect the bucket distribution. For convolution 1, Tanh's worst performance is better than ReLU's best. In addition, the averages reported for Tanh are lower than the other three activation functions. This warrants more rigorous future work examining how the choice of activation function affects the quality of the ALSH Convolution.

Chapter 6

Conclusion

In this work, we set out to improve the computational efficiency of convolutional neural network inference on small devices. To that end, we were partially successful. We were able to substantially accelerate inference for the networks that we tested without severely damaging the network's ability to accurately classify images. Unfortunately, the speedup offered by our current implementation is still not enough to make this a viable method for networks trained to classify images; PyTorch's implementation of AlexNet is faster and more accurate than our ALSHConv2d version of VGG-11.

Even so, there is still room for our current implementation to be put into use. Compared to other forms of filter pruning [23, 28], our methodology can be treated as a regular network layer, so it is relatively simple to use and generalizes well to many architectures. Further, there are many applications of neural networks that have not received the same level of attention as image classification. For such applications, there may only be one accessible models that has been fully trained by experts. So, if one wants to use a neural network on an embedded device for some niche task, the ALSH Convolution can still be beneficial.

For future work, it may be worthwhile to explore using SimpleLSH or Norm-Ranging LSH instead of Hyperplane ALSH, because they improve the bucket distributions [29, 38]. As discussed in Chapter 5, doing this will likely require developing a slightly lower-level implementation. It may also be interesting to analyze how the choice of activation function affects accuracy since they appear to influence the bucket distributions. We also believe that making the replacement process more gradual, by slowly decreasing the number of tables, and using a more thorough methodology to decide when to make a replacement, rather than using a constant gap, are both important and warrant deeper inquiry. We are confident that any combination of these could lead to improved accuracy and plan to pursue these ideas in the future.

Appendix A

A.1 Finding Best-Case Mode Guarantee

The following is the full derivation of $Pr\{mode_2(Q) = h(x)\}$ from Chapter 4. Recall the equality, where f is a binomial distribution.

$$Pr\{vote_2(Q) = h(x)\} = \sum_{n=\frac{L}{2}+1}^{L} \left(\sum_{j=0}^{n} \big(f(M_x, n-j, 1) \times f(O_x, j, p_3) \big) \times f(N_x, 0, 0) \right) \tag{A.1}$$

We replace the function $f(\cdot,\cdot,\cdot)$ with the actual binomial distribution.

$$\sum_{n=\frac{L}{2}+1}^{L} \left(\sum_{j=0}^{n} \binom{|M_x|}{n-j} (p_1)^{n-j} (1-p_1)^{|M_x|-(n-j)} \times \binom{|O_x|}{j} (p_3)^j (1-p_3)^{|O_x|-j} \right) \times \binom{|N_x|}{0} (0)^0 (1-0)^{|N_x|-0} \tag{A.2}$$

We notice that as $c \rightarrow 1^+$ the set O_x will become empty. Thus, $|O_x| = 0$. So, the only case where $\binom{|O_x|}{j}$ is non-zero is when $j = 0$. In this case, $\binom{|O_x|}{j} = \binom{0}{0} = 1$. So, we find:

$$\sum_{n=\frac{L}{2}+1}^{L} \binom{|M_x|}{n} (p_1)^n (1-p_1)^{|M_x|-n} \times \binom{|N_x|}{0} (0)^0 (1)^{|N_x|} \tag{A.3}$$

We know that $\binom{|N_x|}{0}(0)^0(1)^{|N_x|} = 1$ so it can be removed.

$$\sum_{n=\frac{L}{2}+1}^{L} \binom{|M_x|}{n} (p_1)^n (1-p_1)^{|M_x|-n} \tag{A.4}$$

Remember that $p_1 = 1$. So, as we have $0^0 = 1$, it is the case that $(1 - p_1)^{|M_x|-n} = 0$ when $|M_x| - n \neq 0$. This implies that, we must have $|M_x| = n$. So, when $|M_x| > \frac{L}{2}$, we can derive:

$$\begin{aligned} Pr\{mode_2(Q) = h(x)\} &= \sum_{n=\frac{L}{2}+1}^{L} \binom{|M_x|}{n} (p_1)^n (1-p_1)^{|M_x|-n} \\ &= \binom{|M_x|}{|M_x|} (1)^{|M_x|} (1-1)^{|M_x|-|M_x|} \\ &= 1 \times 1 \times 1 = 1 \end{aligned} \tag{A.5}$$

Similarly, when $|M_x| \leq \frac{L}{2}$, we get $Pr\{vote(Q) = h(x)\} = 0$. Thus, in the case where $p_1 \to 1^-$, $p_2 \to 0^+$, and $c \to 1^+$, if a bucket contains an element that has an M_x that contains more than fifty percent of the total number of queries, then that bucket is guaranteed to be selected.

Bibliography

[1] *Speeding Up Convolutional Neural Networks with Low Rank Expansions* (May 2014).

[2] Anderson, A., Vasudevan, A., Keane, C., and Gregg, D. Low-memory gemm-based convolution algorithms for deep neural networks, September 2017. https://arxiv.org/abs/1709.03395.

[3] Andoni, A., and Indyk, P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM 51*, 1 (Jan. 2008), 117–122.

[4] Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I., and Schmidt, L. Practical and optimal lsh for angular distance. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Cambridge, MA, USA, 2015), NIPS'15, MIT Press, pp. 1225–1233.

[5] Charikar, M. S. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2002), STOC '02, ACM, pp. 380–388.

[6] Cheng, J., Wang, P.-s., Li, G., Hu, Q.-h., and Lu, H.-q. Recent advances in efficient computation of deep convolutional neural networks. *Frontiers of Information Technology & Electronic Engineering 19*, 1 (Jan 2018), 64–77.

[7] Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (Cambridge, MA, USA, 2015), NIPS'15, MIT Press, pp. 3123–3131.

[8] Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (New York, NY, USA, 2004), SCG '04, ACM, pp. 253–262.

[9] Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12*, Jul (2011), 2121–2159.

[10] Everingham, M., Van Gool, L., Williams, C. K. I., Winn, J., and Zisserman, A. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision 88*, 2 (June 2010), 303–338.

[11] Gionis, A., Indyk, P., and Motwani, R. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1999), VLDB '99, Morgan Kaufmann Publishers Inc., pp. 518–529.

[12] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] Hassibi, B., Stork, D. G., and Wolff, G. J. Optimal brain surgeon and general network pruning. *IEEE International Conference on Neural Networks* (March 1993), 293–299.

[14] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. Improving neural networks by preventing co-adaptation of feature detectors, 2012. https://arxiv.org/pdf/1207.0580.pdf.

[15] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 4107–4115.

[16] Indyk, P., and Motwani, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1998), STOC '98, ACM, pp. 604–613.

[17] Jia, Y. Learning semantic image representations at a large scale.

[18] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *CoRR abs/1412.6980* (2014).

[19] Krizhevsky, A. Learning multiple layers of features from tiny images. Tech. rep., 2009.

[20] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[21] LeCun, Y., and Cortes, C. MNIST handwritten digit database, 2010.

[22] LeCun, Y., Denker, J. S., and Solla, S. A. Optimal brain damage. *Advances in Neural Information Processing Systems*, 2 (January 1990), 598–605.

[23] Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. *International Conference on Learning Representations 2017* (March 2017).

[24] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollr, P., and Zitnick, C. L. Microsoft coco: Common objects in context. In *European Conference on Computer Vision (ECCV)* (Zurich, 2014). Oral.

[25] Lv, Q., Josephson, W., Wang, Z., Charikar, M., and Li, K. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (2007), VLDB '07, VLDB Endowment, pp. 950–961.

[26] Makhzani, A., and Frey, B. J. k-sparse autoencoders. *CoRR abs/1312.5663* (2014).

[27] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. Playing atari with deep reinforcement learning. *CoRR abs/1312.5602* (2013).

[28] Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. Pruning convolutional neural networks for resource efficient inference. *International Conference on Learning Representations 2017* (June 2017).

[29] Neyshabur, B., and Srebro, N. On symmetric and asymmetric lshs for inner product search. In *International Conference on Machine Learning 32* (2015).

[30] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W* (2017).

[31] Shrivastava, A., and Li, P. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2321–2329.

[32] Shrivastava, A., and Li, P. Improved asymmetric locality sensitive hashing (alsh) for maximum inner product search (mips). In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence* (Arlington, Virginia, United States, 2015), UAI'15, AUAI Press, pp. 812–821.

[33] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *CoRR abs/1409.1556* (2014).

[34] Spring, R., and Shrivastava, A. Scalable and sustainable deep learning via randomized hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2017), KDD '17, ACM, pp. 445–454.

[35] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res. 15*, 1 (Jan. 2014), 1929–1958.

[36] Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28* (2013), ICML'13, JMLR.org, pp. III–1139–III–1147.

[37] Szegedy, C., Ioffe, S., and Vanhoucke, V. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI* (2016).

[38] Yan, X., Li, J., Dai, X., Chen, H., and Cheng, J. Norm-ranging lsh for maximum inner product search. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 2952–2961.

[39] Zhai, K., and Wang, H. Adaptive dropout for training deep neural networks. *NIPS'13 Proceedings of the 26th International Conference on Neural Information Processing Systems 2* (December 2013), 3084–3092.

[40] Zhou, B., Lapedriza, A., Khosla, A., Oliva, A., and Torralba, A. Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2017).