Memory-Optimal Direct Convolutions for Maximizing Classification Accuracy in Embedded Applications

Albert Gural [1] **Boris Murmann** [1]

Abstract

In the age of Internet of Things (IoT), embedded devices ranging from ARM Cortex M0s with hundreds of KB of RAM to Arduinos with 2KB RAM are expected to perform increasingly sophisticated classification tasks, such as voice and gesture recognition, activity tracking, and biometric security. While convolutional neural networks (CNNs), together with spectrogram preprocessing, are a natural solution to many of these classification tasks, storage of the network's activations often exceeds the hard memory constraints of embedded platforms. This paper presents memory-optimal direct convolutions as a way to push classification accuracy as high as possible given strict hardware memory constraints at the expense of extra compute. We therefore explore the opposite end of the compute-memory trade-off curve from standard approaches that minimize latency. We validate the memory-optimal CNN technique with an Arduino implementation of the 10-class MNIST classification task, fitting the network specification, weights, and activations entirely within 2KB SRAM and achieving a state-of-the-art classification accuracy for small-scale embedded systems of 99.15%.

1. Introduction

Moving machine learning inference from the cloud to energy-efficient edge devices is a research topic of growing interest. Running machine learning models locally may help mitigate privacy concerns associated with a user's raw sensor data and can enable truly autonomous operation by eliminating the need for a data connection. In this application paradigm, often assimilated with the Internet of Things (IoT), the machine learning model is trained on a server and subsequently deployed across a large number of edge devices. Consequently, these platforms must have sufficient local memory to store the weights, biases, activations and configuration parameters associated with typical machine learning algorithms. However, low-cost IoT hardware is severely resource constrained (often just 2-16KB of memory), which stands at odds with the memory-hungry nature of top-performing algorithms, such as deep CNNs.

To address this issue, recent work has looked at memory-efficient alternatives to CNNs and has asked the question: Given a (small) fixed memory budget, what is the maximum classification accuracy that one can achieve? In this quest, Gupta et al. (2017) advocated k-Nearest Neighbor (KNN) models to utilize a memory size as low as 2KB (Arduino UNO platform). Similarly, Kumar et al. (2017) proposed a sparse tree-based algorithm, building on the assumption that a CNN can not run on a device with KB-size memory. However, with the departure from CNNs, which are known to achieve state-of-the-performance, there is an inherent sacrifice in classification accuracy that is difficult to recoup by engineering a new algorithm. For example, the 2KB tree-based approach of Kumar et al. (2017) is limited to 94.38% accuracy on a two-class MNIST-2 dataset (Jose et al., 2013).

This paper presents strategies for implementing CNNs under strict memory resource constraints. While the described techniques are generally applicable, we illustrate their utility through the implementation of a 2KB, four-layer CNN for image classification, thus imposing similar constraints as in Kumar et al. (2017). We show that despite this extreme resource scarcity, a test accuracy of 99.15% is achievable for the original MNIST-10 dataset from LeCun et al. (1998).

The main contributions of this work are: (1) identification of a method for memory-optimal direct convolution along with a proof of its optimality and (2) an example implementation of MNIST-10 classification on a 2KB Arduino platform. The latter should be viewed as an illustrative case study on the asymptotic limits of memory size reduction. As such, it is not necessarily practical or optimized for other performance aspects such as throughput and compute energy. Figure 1 gives a taste of the results of our approach. Code and supplemental material are available here.

[1]Department of Electrical Engineering, Stanford University, Stanford, USA. Correspondence to: Albert Gural <agural@stanford.edu>, Boris Murmann <murmann@stanford.edu>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019.

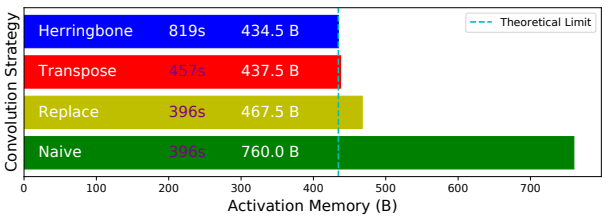

Figure 1. Activation memory versus convolution strategy for the strategies of Section 3.4 applied to our network. Inference time is also given for each strategy applied to all convolution layers. Herringbone is the only method that does not corrupt the data.

2. Related Work

There is a large body of literature dealing with resource-efficient machine learning inference. Here, we review a subset of contributions that focus on similar KB-level memory footprints as considered in our work, as well as a few methods for efficient convolution computation.

There are a few examples of ML techniques targetting KB-size devices. Gupta et al. (2017) presents ProtoNN, a k-Nearest Neighbor model optimized for small-memory footprint by storing a judicious selection of training data. A 2KB version of ProtoNN achieves 93.25% on MNIST-2. Kumar et al. (2017) presents Bonsai, tree-based algorithm utilizing sparse projections from the input data into a lower-dimensional space. A 2KB version of Bonsai achieves 94.38% on MNIST-2. Kusupati et al. (2018) presents FastGRNN, a sequential model making use of residual network connections and low rank matrices. A 6KB version of FastGRNN achieves 98.20% on MNIST-10.

Unlike previous approaches to ML for tiny embedded devices, our work focuses on CNNs. There exists a large body of work on optimizing CNNs. There are algorithmic speedups, such as the Fast Fourier Transform (FFT) (Vasilache et al., 2014) and Winograd Transform (Lavin & Gray, 2016), which convert convolutions into point-multiplies. There are also hardware/software speedups, such as unrolling convolutions into matrix multiplies (Chellapilla et al., 2006; Chetlur et al., 2014) to make use of *gemm* libraries.

However, while there is an abundance of research into CNN speedup optimization, memory optimization research is more sparse. Motivated by CNN unrolling, Cho & Brand (2017) propose a partial unrolling of input features that wastes much less memory in duplications while still taking advantage of BLAS *gemm* speedups. However, as Zhang et al. (2018) points out, even this partial unrolling uses additional memory beyond direct convolution approaches. Instead, Zhang et al. (2018) propose using direct convolutions for "zero-memory overhead" and demonstrate that by carefully reordering the for-loops, one can exceed *gemm* speed. However, it is important to understand that this is "zero-memory overhead" *beyond the memory required to store input and output activations*. In this paper we show that one can in fact do *even better* by overwriting stale input activations to store new output activations.

3. Memory-Optimal Convolutions

We start by restricting our attention to 2D convolutions with odd square kernels, valid padding, and stride of 1. This covers the most common use case for convolutional layers in memory-constrained applications. The valid padding helps reduce activation storage and the other restrictions are commonly found in popular CNN architectures (Krizhevsky et al., 2012; Simonyan & Zisserman, 2014). Extensions to this restricted case are considered in the supplementary material.

The general observation that direct convolutions can be performed in a more memory-efficient manner compared to the "naive" approach of maintaining a separate area of memory for convolution outputs stems from the observation that convolutions are local operations, so pixels in input feature maps are computational dependencies of only a small number of output features. Therefore, after an input pixel has satisfied all of its dependencies, its memory can be deleted and used to store output feature pixels. In the restricted setting described above, two major cases are important for analysis: same/decreasing channel depth, analyzed in Section 3.3 and increasing channel depth, in 3.4.

We additionally note that some of the analyses in the following sections are motivated by an understanding of memory layout. For the following analyses, activations are stored in memory in height, width, channel order, as shown at the top of Figure 2.

3.1. Notation

In this section, we will consider a convolution layer taking an $h_{in} \times w_{in} \times f_{in}$ input feature map to an $h_{out} \times w_{out} \times f_{out}$ output feature map. The convolution kernel size is $k_h \times k_w$ ($k_h = k_w = k$ for the restricted case). The padding and stride restrictions imply $h_{out} = h_{in} - (k_h - 1)$ and $w_{out} = w_{in} - (k_w - 1)$. For convenience, assume $h_{out} \geq w_{out}$ in the following sections (a transpose as described in Section 4.1 can be used if this is not the case).

We refer to input/output feature maps interchangeably with input/output images. A pixel of a feature map is given by its row and column coordinates and includes all channels at that location in the feature map. An output pixel is "processed" or "computed" (opposite "unprocessed") if it has been evaluated and stored in memory. We say an input pixel is "stale" (opposite "live") if, during the direct convolution calculation, it is no longer a dependency of some unpro-

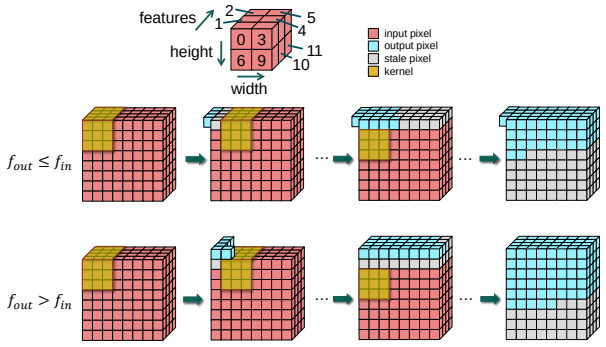

Figure 2. Arrangement of pixels in memory (top) and considerations for where to place output activations based on how f_{in} compares to f_{out}.

cessed output pixel. We use the term "debt" or "cost" to refer to the accrued additional memory requirements on top of the memory required to store just the input feature map. The opposite of "debt" is "payout" or "payback," which is the memory freed when input pixels become stale.

3.2. Assumptions for Memory Optimality

Memory optimality will be proven for lossless direct convolutions in which all input features are loaded in memory (not streamed[1]). We assume that an input pixel does not become stale until all its dependent output pixels have been written to memory. This is in contrast to algorithms like the Winograd (Lavin & Gray, 2016) and FFT transforms that generate intermediate results that have sufficient information to recover the output pixels, therefore allowing input pixels to be deleted[2]. In our analyses, we only count memory used to store activations (ignoring $\mathcal{O}(1)$ additional memory for, eg, loop variables).

3.3. Case: Same or Decreasing Channel Depth

Consider processing output pixels in row-major order. The top-left input pixel only has a single output pixel dependent because of valid-padding. Therefore, as soon as the top-left output pixel is processed, the input pixel can be deleted. Since $f_{out} \leq f_{in}$, the output pixel can be stored in the memory that the now-stale input pixel used to occupy. A similar argument holds as additional pixels are processed since there will always be a top-left corner pixel after each input pixel removal.

Overall, an additional f_{out} memory is required over the memory used to store the input activations. The middle row of Figure 2 shows how f_{out} extra memory is initially required, but afterwards, the stale input is sufficient to store output activations. While it is intuitive that this approach achieves optimal memory use, it is nonetheless instructive to analyze, since this builds intuition necessary for Section 3.4.

Claim 1. *The row-major traversal strategy is memory-optimal for direct convolution with same or decreasing channel depth and restrictions described in Section 3.*

Proof. For any method of computing the direct convolution, there will be some point in the procedure when exactly one output pixel has been processed. At any point prior to this computation, all $h_{in} \cdot w_{in} \cdot f_{in}$ activations must stay in memory, since computation dependencies are only removed when all channels of an output pixel are computed. The memory required to store the channels of the output pixel is f_{out}. So no matter what method is used for direct convolution, an absolute minimum of $h_{in} \cdot w_{in} \cdot f_{in} + f_{out}$ memory must be available. Since the proposed algorithm achieves this lower bound, it is optimal. □

3.4. Case: Increasing Channel Depth

When $f_{out} > f_{in}$, additional memory is required, since output pixels will not fit in the vacancy of the input pixels as described in Section 3.3 and as seen in the bottom of Figure 2. Three approaches to dealing with this issue are presented here. In Section 3.4.1 a "replace" strategy that follows the row-major approach of Section 3.3 is described. In Section 3.4.2, a different computation order for output pixels is described and proven to be memory-optimal. In Section 3.4.3, a less computationally-intensive algorithm that is nonetheless near memory-optimal is described. Algorithmic implementation details are described in Section 4.

3.4.1. Replace Strategy

In the replace strategy, output pixels are computed in row-major order. To analyze the memory requirements of this strategy, we keep track of the memory debt accrued at each step. For example, to compute the first output pixel, we accrue a debt of f_{out} then delete the input pixel, which pays back f_{in} memory. For a given row, the sequence of debts and paybacks are $L = [f_{out}, -f_{in}, f_{out}, -f_{in}, \ldots, f_{out}, -k \cdot f_{in}]$, where there are w_{out} copies of f_{out}. Note the last pixel is unique and pays back $k \cdot f_{in}$. Thus, each row accrues debt $D(w_{out}) = w_{out}(f_{out} - f_{in}) - (k-1)f_{in}$, but the peak debt within a row is $D(w_{out}) + kf_{in}$. The total peak debt is:

$$\begin{aligned} D_{rp} &= (h_{out} - 1)D(w_{out}) + D(w_{out}) + kf_{in} \\ &= h_{out}\left(w_{out}(f_{out} - f_{in}) - (k-1)f_{in}\right) + kf_{in} \end{aligned} \tag{1}$$

[1] An interesting alternative to the methods of this paper is to compute sub-feature maps of multiple convolution layers at a time in a pipeline fashion.

[2] Our lossless assumption and basic information theory implies that these methods can not improve on required memory.

This replace method is not memory optimal. Note that after each output pixel of the bottom row is computed, $k-1$ input pixels become stale, but this memory is not conveniently laid out in memory and can not be directly used. This observation motivates the "herringbone" strategy in the next section, which we prove to be memory-optimal.

3.4.2. Herringbone Strategy

Intuitively, accrued debt can be minimized by prioritizing the collection of large payouts near the edges of the image. In the replace strategy, the edges are only encountered after processing w_{out} output pixels. However, after a sufficient number of rows have been processed, edges could be reached faster by processing a *column* of pixels rather than another row of pixels.

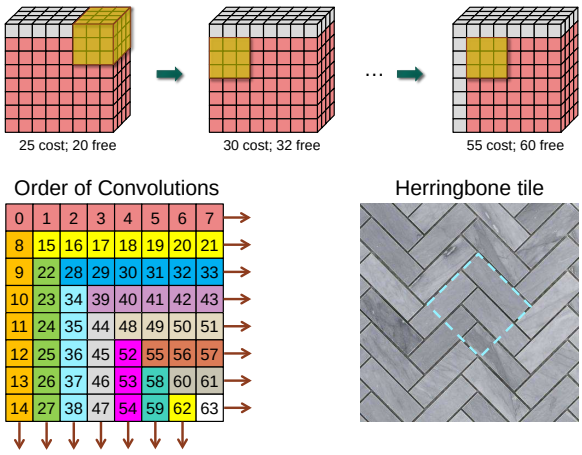

Figure 3. Motivating concept for herringbone (top) and order of pixel traversal (bottom left) with colors indicating rows and columns that are processed sequentially. The name comes from the herringbone tile (bottom right, image of floor tile from Home Depot®) which has the same row-column alternations.

The herringbone strategy proceeds iteratively on rows or columns of the output feature map, greedily taking whichever will accumulate the least debt. The result is an alternating compute procedure of row-column-row-column, which resembles the herringbone tile pattern as seen in Figure 3. Using this strategy, the amount of debt accrued for each row and column decreases as the algorithm progresses on later iterations.

To analyze the memory use of this strategy, we employ the same debt function $D(x) = x(f_{out} - f_{in}) - (k-1)f_{in}$ as in Section 3.4.1, giving the total debt accrued when processing a row or column of x output pixels. Following the herringbone method, the sequence of debts (positive or negative) will be $L_D = [D(w_{out}), D(h_{out}-1), D(w_{out}-1), D(h_{out}-2), \ldots, D(1), D(1)]$ for a square output feature map. If $h_{out} > w_{out}$, L_D will contain $h_{out} - w_{out} + 1$ copies of $D(w_{out})$ at the beginning of the list. Therefore, the peak debt is:

$$D_{hb} = \max_i \sum_{j=1}^{i} L_D(j) + kf_{in}$$

where the additional kf_{in} is due to the input memory required to generate the last output pixel and the $\max_i$ is required since L_D may eventually become negative. One interpretation of $\sum_{j=1}^{i} L_D(j)$ is that it is simply $n_{out}f_{out} - n_{in}f_{in}$ where n_{out} is the number of output pixels that have been computed and n_{in} is the number of stale (non-dependency) input pixels after processing those output pixels, provided we have just finished one of the rows or columns. We can see this formula only depends on the number of processed input and output pixels.

Since L_D is monotonically non-increasing, we can find the maximum prefix sum by finding when $D(x) \leq 0 \Rightarrow x \leq (k-1)f_{in}/(f_{out} - f_{in}) = x^*$. This means that the worst case memory will happen when the remaining output pixels form a square of side length $\lfloor x^* \rfloor$. Therefore:

$$\begin{aligned} D_{hb} &= n_{out}f_{out} - n_{in}f_{in} + kf_{in} \\ &= (h_{out}w_{out} - \lfloor x^* \rfloor^2)f_{out} \\ &\quad - (h_{in}w_{in} - (\lfloor x^* \rfloor + k - 1)^2 - k)f_{in} \end{aligned} \tag{2}$$

Claim 2. *The herringbone strategy is memory-optimal for direct convolution with increasing channel depth and restrictions described in Section 3.*

Proof. For any method of computing the direct convolution, there will be some point in the procedure when $n_{crit} = (h_{out}w_{out} - \lfloor x^* \rfloor^2)$ output pixels have been processed and no others have been processed. Just prior to this point, there are $n_{unproc} = \lfloor x^* \rfloor^2 + 1$ unprocessed output pixels and therefore at least $n_{live} = (\lfloor x^* \rfloor + k - 1)^2 + k$ input pixels are not stale by Lemma 3.1. Therefore, the minimum possible memory deficit just before processing n_{crit} output pixels is:

$$\begin{aligned} D_{min} &= n_{out}f_{out} + n_{in}f_{in} \\ &= (h_{out}w_{out} - \lfloor x^* \rfloor^2)f_{out} - (h_{in}w_{in} - n_{live})f_{in} \\ &= D_{hb} \end{aligned}$$

No matter what method is used for direct convolution, a minimum of $D_{min} = D_{hb}$ additional memory is required. Since the proposed algorithm achieves this lower bound, it is optimal. □

Lemma 3.1. *The minimum number of input pixel dependencies for n^2+1 output pixels and odd kernel size k is $(n+k-1)^2+k$.*

Proof. Let $B(p)$ be a function giving the input pixel dependencies for pixel p and $B(P) = \bigcup_{p \in P} B(p)$. $B(P)$ gives the set of pixels within Chebyshev distance $(k-1)/2$ of any pixel in P. We are interested in finding $|B(S^*)|$, where $S^* = \text{argmin}_S |B(S)|$, subject to $|S| = n^2+1$.

Define a pixel $p \in S$ at location (x_p, y_p) to be a "top-left corner pixel" if pixel $(x_p, y_p-1) \in S$ and $(x_p+1, y_p) \in S$ and no pixels in S besides p have coordinates (x, y) satisfying both $x \leq x_p$ and $y \geq y_p$. Define "top-right," "bottom-left," and "bottom-right" corner pixels analogously and say that p is a "corner pixel" if it is any one of these four types of corner pixels (refer to Figure 4a for an illustration). Notice that if p is a corner pixel in S, then $|B(S-p)| = |B(S)|-1$, since $|B(p) - B(S)| = |B(p) - (B(p_1) \bigcup B(p_2))| = 1$.

We assume that an optimal S^* is connected and orthogonally convex[3] (Fink & Wood, 1996). Suppose the tight bounding rectangle R of S^* has dimensions $r \times c$, $rc \geq n^2+1$. We can repeatedly remove corner pixels of this rectangle until $rc-(n^2+1)$ corner pixels are removed to restore S^* from R. This can be seen by noting that S^* has pixels on each of the four edges of R, since R is tight, and S^* must have a staircase boundary between adjacent pairs of these edge pixels (Nicholl et al., 1983). It is easy to see that these four staircases can be produced by repeated corner pixel removal, as seen in Figure 4b. Therefore,

$$\begin{aligned} B(S^*) &= B(R) - (\text{\#removed corner pixels}) \\ &= (r+k-1)(c+k-1) - (rc-(n^2+1)) \\ &= n^2 + (k-1)^2 + 1 + (k-1)(r+c) \end{aligned}$$

However, $r+c \geq r+(n^2+1)/r \geq 2\sqrt{n^2+1} > 2n$. Since r and c are integers, $r+c \geq 2n+1$. Then,

$$\begin{aligned} B(S^*) &\geq n^2 + (k-1)^2 + 1 + (k-1)(2n+1) \\ &= (n+k-1)^2 + k \end{aligned}$$

□

Optimality of the herringbone method requires a memory-efficient way to access the stale inputs. This is problematic when computing a column of pixels, as pointed out in Section 3.4.1. Two approaches can be used to solve this problem: shifting and transposing, as summarized in Figure 5.

The first approach is to shift input pixels in memory every time a later input pixel becomes stale to fill in the stale spot and open free memory at the front of the input activations. This algorithm takes $\mathcal{O}(h_{out} w_{out} h_{in} w_{in} f_{in})$.

The second approach is to perform an in-place transpose when switching between row and column processing[4][5]. Because transposes only need to be performed between entire rows and columns, the computational complexity is reduced to $\mathcal{O}((h_{out}+w_{out}) T_{transpose}(h_{in}, w_{in}, f_{in}))$. In Section 4.1, an in-place transpose with complexity $\mathcal{O}(h_{in} w_{in} (c+f_{in}))$ is presented. Empirically, we find that c is usually less than 5 for problem sizes of interest.

Both approaches leave the output in herringbone order. Section 4.2 discusses an efficient solution to return the feature map to row-major order.

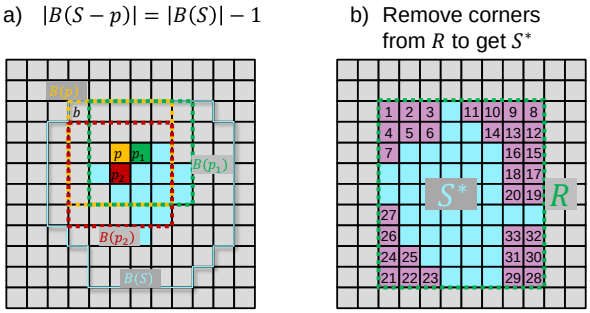

Figure 4. Example of a corner pixel p with $k=5$ (a) and example of a sequence of corner pixel removals to get from R to S^* (b).

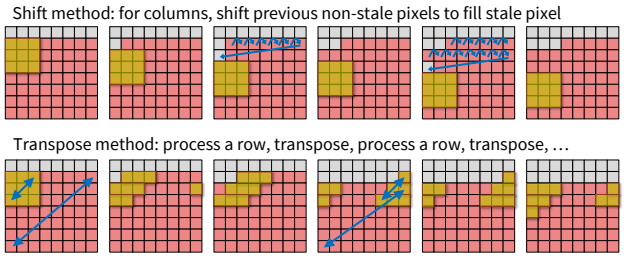

Figure 5. Shifting to claim fragmented stale pixels (top) and using transposes to never have fragmented stale pixels (bottom).

3.4.3. Single Transpose Strategy

While the herringbone method is memory-optimal, it is computationally expensive because of the required memory manipulations. We now demonstrate that a single well-placed transpose can yield most (or sometimes all) of the memory benefits, while significantly reducing computational load. The single transpose method uses standard row-major or-

[3] A proof of this fact is not too insightful and is provided in the supplementary material.

[4] A transpose is equivalent to switching between row-major to column-major storage in memory.

[5] The convolution kernel must be transposed as well.

der for several rows as given in (4), then transposes the remaining input activations and processes the remaining rows (former columns) to completion.

Let r be the number of rows remaining after $h_{out} - r$ rows have already been processed (each processed row adding $w_{out}(f_{out} - f_{in}) - (k-1)f_{in}$ debt, as analyzed in Section 3.4.1). Two cases need to be analyzed separately: either the remaining columns do not add debt, if $r(f_{out} - f_{in}) - (k-1)f_{in} \leq 0$, or they do. An analysis similar to that of Section 3.4.2 shows that the optimal row on which to perform the transpose is $r_1^* = \lfloor (k-1)f_{in}/(f_{out} - f_{in}) \rfloor$ for case 1, or $r_2^* = r_1^* + 1$ for case 2. The total debt is elegantly expressed in terms of the optimal debt D_{hb}.

$$D_{st} = D_{hb} + \min(w_{out}\alpha, (k-1)f_{in}) - r_1^*\alpha \quad (3)$$

where $\alpha = (k-1)f_{in} \bmod (f_{out} - f_{in})$. The first term in the min function is from case 1 ($r = r_1^*$) and the second is from case 2 ($r = r_1^* + 1$). This makes it clear that:

$$r^* = \begin{cases} r_1^* & w_{out}\alpha \leq (k-1)f_{in} \\ r_1^* + 1 & \text{else} \end{cases} \quad (4)$$

From (3), it can be seen that the single transpose method is optimal when $w_{out} = r_1^*$, $(k-1)f_{in} = r_1^*\alpha$, or $\alpha = 0$. This last condition holds when $(k-1)f_{in}$ divides $f_{out} - f_{in}$. Compared to the herringbone computation complexity of $\mathcal{O}((h_{out} + w_{out})T_{transpose}(h_{in}, w_{in}, f_{in}))$, the single transpose method is $\mathcal{O}(T_{transpose}(r^*, w_{in}, f_{in}))$, a factor of approximately $(h_{out} + w_{out})h_{in}/r^*$ times faster.

Figure 10 shows that over a range of randomly generated network architectures, the herringbone method (blue dot) and the transpose method (red dot) have nearly identical memory requirements.

4. Implementation Details

In the herringbone strategy described in Section 3.4.2, the proof of optimality ignored memory requirements of performing transposes. Here, we present a memory-efficient in-place transpose. Using insights gained from this in-place transpose, an inverse herringbone transform is also proposed, to allow efficient "unwrapping" of the output feature map that results from processing in herringbone order.

4.1. Memory-Efficient Transpose

The basic technique for in-place memory manipulations is to decompose the desired memory permutation (such as a transpose) into disjoint cycles, then rotate elements in each of these cycles. To make this algorithm memory efficient, it uses a light-weight successor function mapping each memory location i to another location j, indicating the memory at location j should be moved to i. Recursive application of the successor function thus generates a cycle and the whole cycle can be rotated with one auxiliary memory cell for matrix element storage[6].

All cycles can be rotated by iterating over all elements in memory, to find starting positions, and only rotating cycles that have not yet been visited. Traditionally, extra memory is used to keep track of the visited cycles (Windley, 1959; Laflin & Brebner, 1970). An alternative zero-memory-cost way to ensure unique cycles is to only perform the rotation if the starting position is the minimum element in its cycle. This can be verified by running through the cycle once without any data movement (Morini, 2017).

Finally we address how to compute the successor function. In Laflin & Brebner (1970); Morini (2017), it is given as:

$$f(i; h, w) = (i \bmod h) \cdot w + \lfloor i/h \rfloor \quad (5)$$

where i is the location in memory and $h \times w$ are the image dimensions. To derive this result, we must determine what index j contains the memory needed at index i. Viewed from the transpose perspective, $i = r \cdot h + c$ is located at $(c, r) = (i \bmod h, \lfloor i/h \rfloor)$. From the untransposed perspective, this is $(r, c) = (\lfloor i/h \rfloor, i \bmod h)$, which results in an index $j = r \cdot w + c = f(i; h, w)$, as in (5). Figure 6 illustrates this technique with a small example.

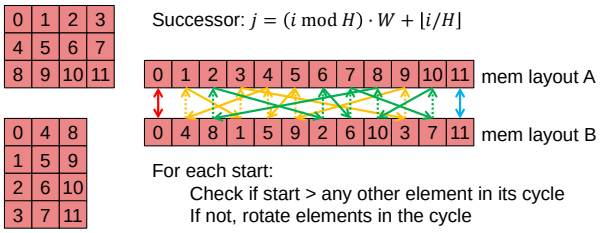

Figure 6. Example for transposing a 3×4 array in-place.

4.2. Memory-Efficient Inverse Herringbone

To invert the herringbone pattern of output pixels, we note that we are simply trying to perform a particular permutation on the memory elements and therefore we can use an in-place permutation again, except with a different successor function. The herringbone pattern is only interesting for the $w \times w$ square at the end of the output activations, so we restrict our attention to this case.

[6]Or zero, if using the XOR swap trick.

As before, we consider a "next" element with some unpermuted index whose memory needs to be moved to the "current" element (r, c) with some permuted index. The permuted case is simply row-major form, so $i = r \cdot w + c$ and therefore $r = \lfloor i/w \rfloor$ and $c = i \bmod w$. For the unpermuted case, refer to Figure 3. The indices of a given "shell" can be referenced to the index $w^2 - n^2$, where n is the side-length of the square contained within the shell. Then, behavior can be split between the upper and lower triangular regions for the two halves of the shell: $c \geq r$ and $c < r$. When $c \geq r$, the index is $j = w^2 - (w - r)^2 + c - r$. When $c < r$, the index is $j = w^2 - (w - c - 1)^2 - (w - c - 1) + r - c - 1$. Simplifying, we get:

$$f(i; w) = \begin{cases} r(2w - r - 1) + c & c \geq r \\ c(2w - c - 2) + w + r - 1 & \text{else} \end{cases} \quad (6)$$

5. Case Study

We verify the feasibility of the herringbone method with a case study implementing an MNIST classifier on an Arduino, followed by a discussion of results.

5.1. MNIST on Arduino

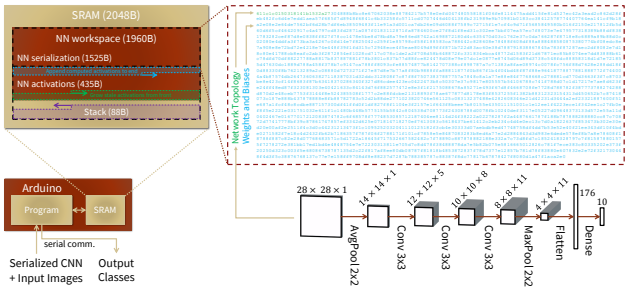

Figure 7. Overview of implemented Arduino system and our CNN architecture. SRAM is split into the network serialization, network activations, and program stack. New layers are built at the front of activations memory (light blue arrow), while current layers are made stale at the back (green arrow).

Figure 7 shows an overview of the hardware setup. A single Arduino based on the ATmega328P chip is employed for the classification task. Network parameters and images are streamed in through serial communications, then the Arduino runs its CNN and returns the output class via serial.

A network architecture search is performed to find the best quantized network by validation performance. Figure 8 shows a random sample of these networks and their float versus quantized performance. After this search, hand tuning is used to optimize the architecture to the one shown in Figure 7. The network is trained in Keras/TensorFlow with Adam (Abadi et al., 2015; Chollet et al., 2015; Kingma & Ba, 2014) for 50 epochs in floating point and 200 epochs with 4-bit quantized training using the straight-through estimator and ALT training (Courbariaux et al., 2016; Jain et al., 2019). Additional techniques used to maximize classification accuracy can be found in the supplemental material.

Table 1. Results Summary.

RESULT	VALUE
MEMORY	434.5 B (ACT) / 1512.5 B (WTS)
TOTAL, SERIALIZED	1960 B
PROGRAM SIZE	6514 B
INFERENCE TIME	684 MS (NOT OPTIMIZED)
ACCURACY	99.11% (DEV) / 99.15% (TEST)

The CNN is then implemented in hardware. Verification is performed both for intermediate activations in a single test image and for the 16-bit output logits for all 10,000 test images to ensure a 100% match. A summary of the results is provided in Table 2. In Figure 1, the benefits of non-naive methods are made clear. For our network, the single transpose method only costs an additional 3B beyond herringbone. However, when including peak stack usage we only have 2B spare, necessitating the use of herringbone.

5.2. Discussion

One might wonder whether the techniques described in this paper are practically useful. We claim that first, the replace method described in Sections 3.3 and 3.4.1 is a generally useful technique that can be applied to any convolutional layer. Second, the herringbone and single-transpose techniques from Sections 3.4.2 and 3.4.3 are useful for 2D or 3D valid-padded CNNs in which (a) channel depth increases, (b) kernel size is greater than 1, and (c) kernel size is not too small compared to feature map width and height. Figure 8 shows that the models near the Pareto frontier tend to have increasing channel depth and 3×3 kernels, satisfying (a) and (b). Meanwhile, our limitation to small devices virtually guarantees that (c) holds. More directly, the bottom left plot of Figure 8 shows that models at the Pareto frontier are disproportionately likely to benefit from the herringbone method.

Figure 10 shows the error versus activation memory requirement across the different convolution computation strategies discussed in Section 3.4. The difference between the naive method and all other methods is substantial and can be qualitatively seen to affect performance. One way to get a more quantitative sense of the expected impact of a decrease in memory efficiency is to look at the slope of the Pareto frontier of this curve. Below 1KB, the validation error appears to rise 10x for every 10x decrease in activation memory (cyan line). In other words, there is a constant (memory

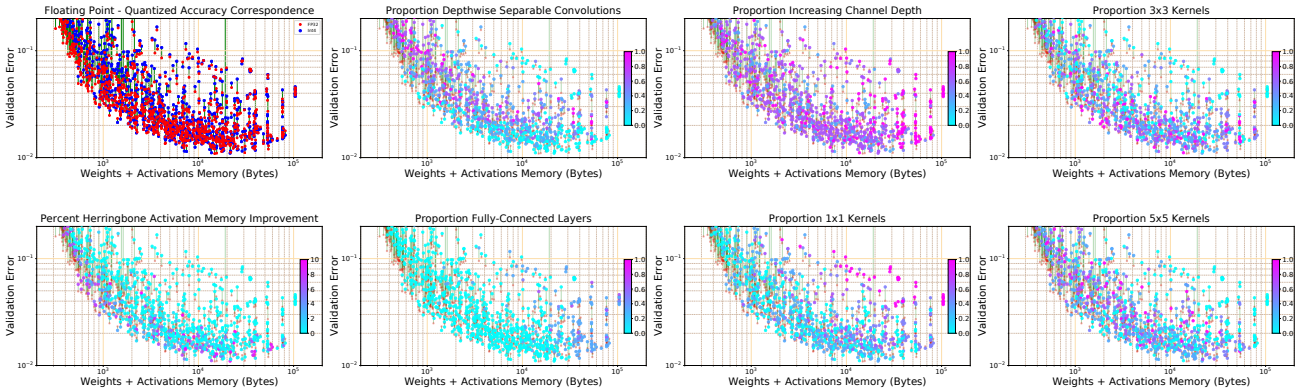

Figure 8. Pareto curves for 1000 randomly selected architectures examining the impact of different architecture features. Models are trained for 5 epochs with floating point weights/activations (red dots), then 5 epochs with quantized weights/activations (blue dots).

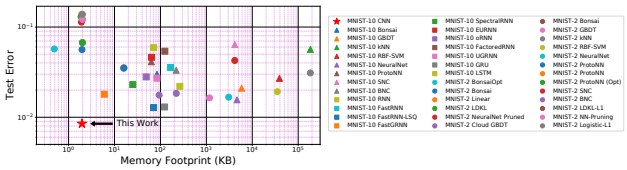

Figure 9. Comparison of our 2KB CNN classifier to results from Kumar et al. (2017); Gupta et al. (2017); Kusupati et al. (2018).

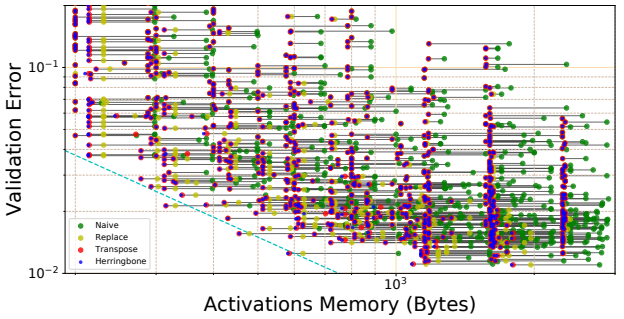

Figure 10. Comparing activation weight storage (4-bit) between four different convolution strategies as described in Section 3 across a range of randomly-generated network architectures. The dotted cyan line gives a rough position for the Pareto frontier.

$\times$ error) product in this regime. Since the naive method has roughly twice the memory use of herringbone (see Figure 1), it may be expected to have twice the error for a given memory constraint. The improvements between the replace method and herringbone are more modest, but appear to be more pronounced for higher-achieving models. So, they could still be predicted to affect performance on average by $\approx 10\%$ relative error.

A Pareto frontier can also be seen in Figure 8. Of particular interest is the existence of a soft knee around 2KB, where extra memory has diminishing returns. This could explain why we achieved such a high accuracy compared to related work in Figure 9 — 2KB is close to the cliff of accuracy degradation, meaning even small memory-use non-optimalities can have a large impact. In contrast to our work, the related works typically take a more balanced approach to the memory-compute trade-off. For the 2KB environment, we may surmise MNIST is an approximate lower bound on problem complexity where our methods would boost accuracy. However, even for simpler classification problems, our methods could still free memory for other processes.

6. Conclusion

In this paper, we analyzed the minimum memory required to run CNN inference so that we could maximize classification accuracy on memory-constrained devices. For a given convolution layer, when channel depth increases, we saw that the herringbone method was optimal and showed that it can be implemented with in-place memory permutations. We used this optimality to show that a single well-placed transpose was nearly or exactly optimal and additionally benefited from a significant reduction in computational complexity.

We then demonstrated these techniques on an Arduino for MNIST classification and achieved a test accuracy of 99.15%, which is state-of-the-art for models with comparable memory constraints. This demonstrates the effectiveness of CNN classification even on small embedded devices. While our focus application was narrow, the replace technique applies to all CNNs and the herringbone/single-transpose techniques apply to many CNNs. MNIST classification in itself may not be all that attractive, but CNNs in conjunction with spectrograms of time-series sensor data could enable a suite of KB-level smart device applications.

Acknowledgements

We thank Daniel Bankman, Alex Chen, Elaina Chai, Dan Villamizar, Lita Yang, Ernest So, Bryan He, Peter Cuy, Saketh Are, Scott Wu, Hoa Gural, and Kenneth Gural for helpful comments and advice. Albert Gural is supported by a National Science Foundation Graduate Research Fellowship and by Analog Devices via the Stanford SystemX Fellow Mentor Advisor (FMA) program.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

Allen, J. Short term spectral analysis, synthesis, and modification by discrete fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 25(3):235–238, 1977.

Chellapilla, K., Puri, S., and Simard, P. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Cho, M. and Brand, D. Mec: memory-efficient convolution for deep neural network. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 815–824. JMLR. org, 2017.

Chollet, F. et al. Keras. https://github.com/fchollet/keras, 2015.

Cooley, J. W. and Tukey, J. W. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

Fayek, H. M. Speech processing for machine learning: Filter banks, mel-frequency cepstral coefficients (mfccs) and what's in-between. https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html, 2016. Accessed: 2019-01-10.

Fink, E. and Wood, D. Fundamentals of restricted-orientation convexity. *Information Sciences*, 92 (1):175 – 196, 1996. ISSN 0020-0255. doi: https://doi.org/10.1016/0020-0255(96)00056-4. URL http://www.sciencedirect.com/science/article/pii/0020025596000564.

Gupta, C., Suggala, A. S., Goyal, A., Simhadri, H. V., Paranjape, B., Kumar, A., Goyal, S., Udupa, R., Varma, M., and Jain, P. Protonn: compressed and accurate knn for resource-scarce devices. In *International Conference on Machine Learning*, pp. 1331–1340, 2017.

Hasan, M. R., Jamil, M., Rahman, M., et al. Speaker identification using mel frequency cepstral coefficients. *variations*, 1(4), 2004a.

Hasan, M. R., Jamil, M., Rahman, M., et al. Speaker identification using mel frequency cepstral coefficients. *variations*, 1(4), 2004b.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Hershey, S., Chaudhuri, S., Ellis, D. P. W., Gemmeke, J. F., Jansen, A., Moore, R. C., Plakal, M., Platt, D., Saurous, R. A., Seybold, B., Slaney, M., Weiss, R. J., and Wilson, K. W. CNN architectures for large-scale audio classification. *CoRR*, abs/1609.09430, 2016. URL http://arxiv.org/abs/1609.09430.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL http://arxiv.org/abs/1704.04861.

Jain, S. R., Gural, A., Wu, M., and Dick, C. Trained uniform quantization for accurate and efficient neural network inference on fixed-point hardware, 2019.

Jose, C., Goyal, P., Aggrwal, P., and Varma, M. Local deep kernel learning for efficient non-linear svm prediction. In *International conference on machine learning*, pp. 486–494, 2013.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Kumar, A., Goyal, S., and Varma, M. Resource-efficient machine learning in 2 kb ram for the internet of things. In *International Conference on Machine Learning*, pp. 1935–1944, 2017.

Kusupati, A., Singh, M., Bhatia, K., Kumar, A., Jain, P., and Varma, M. Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. In *Advances in Neural Information Processing Systems*, pp. 9031–9042, 2018.

Laflin, S. and Brebner, M. A. Algorithm 380: In-situ transposition of a rectangular matrix [f1]. *Commun. ACM*, 13(5):324–326, May 1970. ISSN 0001-0782. doi: 10.1145/362349.362368. URL http://doi.acm.org/10.1145/362349.362368.

Lavin, A. and Gray, S. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Morini, M. Transpose a matrix without a buffering one. Software Engineering, May 2017. URL https://softwareengineering.stackexchange.com/a/271722. Version: 2017-05-23.

Nicholl, T. M., Lee, D.-T., Liao, Y.-Z., and Wong, C.-K. On the xy convex hull of a set of xy polygons. *BIT Numerical Mathematics*, 23(4):456–471, 1983.

Saxe, A. M., McClelland, J. L., and Ganguli, S. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *CoRR*, abs/1312.6120, 2013. URL http://arxiv.org/abs/1312.6120.

Simard, P. Y., Steinkraus, D., and Platt, J. C. Best practices for convolutional neural networks applied to visual document analysis. In *null*, pp. 958. IEEE, 2003.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556.

Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., and LeCun, Y. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.

Warden, P. Speech commands: A dataset for limited-vocabulary speech recognition. *CoRR*, abs/1804.03209, 2018. URL http://arxiv.org/abs/1804.03209.

Windley, P. F. Transposing matrices in a digital computer. *The Computer Journal*, 2(1):47–48, 1959. doi: 10.1093/comjnl/2.1.47. URL http://dx.doi.org/10.1093/comjnl/2.1.47.

Zhang, J., Franchetti, F., and Low, T. M. High performance zero-memory overhead direct convolutions. *arXiv preprint arXiv:1809.10170*, 2018.

A. Optimality of Connected Orthogonally Convex Pixel Dependents

Lemma 3.1 in the main paper assumes that there exists a set of $n^2 + 1$ pixels $S^* = \text{argmin}_S|B(S)|$ that is connected and orthogonally convex. Here we provide a proof of a generalization of that claim.

Lemma A.1. *For all N, there exists a selection S^* of output pixels such that $S^* = \text{argmin}_S|B(S)|$, $|S^*| = N$, and S^* is connected and orthogonally convex.*

Proof. Suppose S satisfies optimality condition $S = \text{argmin}_{S'}|B(S^{'})|$ and cardinality $|S| = N$. We assume such an optimal S will be connected. We will then show that there is a series of pixel swaps that can be performed which will preserve connectivity, optimality, and cardinality, and will eventually lead to an orthogonally convex S.

Define an "elbow" pixel to be $p \notin S$ such that it is bordered by at least two pixels in S on adjacent edges of p. It is easy to see that if p is added to S, $B(S + p) \leq B(S) + 1$, since the bordering pixels capture most of p's input dependencies. Define the "top-left" pixel to be the unique pixel $p \in S$ whose x value is minimal among all pixels whose y value is maximal in S. Similar to the analysis of corner pixels, $B(S - p) \leq B(S) - 1$.

There are 16 possible configurations for cells neighboring the top-left pixel, as shown in Figure 11. Of these 16, only 3 have the potential to disconnect S if the top-left pixel is removed. However, these disconnections can be avoided by swapping pixels according to the arrows in the figure and selecting the next top-left pixel. One can easily check that shifting according to those arrows also preserves cardinality and optimality ($B(S - p_{tl} + p_{tl}^{'}) \leq B(S)$). Therefore, we can always eventually remove a top-left pixel without disconnecting S and still achieve $B(S - p_{tl}) \leq B(S) - 1$.

From these observations, we arrive at the following procedure. Repeatedly select an elbow pixel that is not in the top row of pixels in S, then fill it with the top-left pixel in the manner suggested earlier. Each step preserves optimality since $B(S - p_{tl} + p_{elbow}) \leq B(S) - 1 + 1 \leq B(S)$. Since there always exists a top-left pixel, this algorithm must terminate due to lacking elbow pixels, which happens when S is a rectangle, except for possibly the top row. For such a shape, it is always more optimal to have the top row pixels connected to each other rather than spread apart. The resultant shape is orthogonally convex. Therefore, through a series of connectivity, optimality, and cardinality preserving operations, we transformed S into a connected orthogonally convex S^*. □

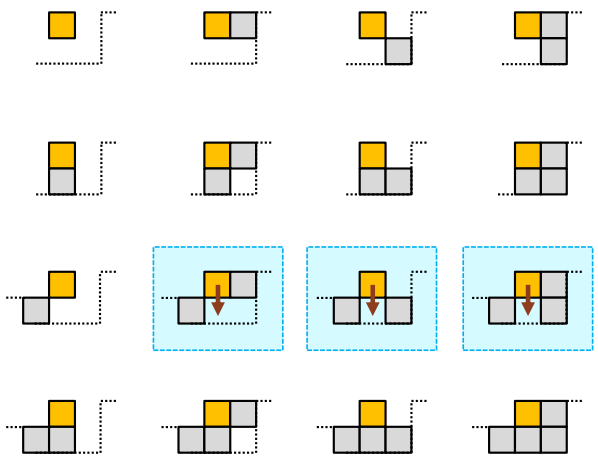

Figure 11. Possible configurations of neighboring pixels to a top-left pixel. The top-left pixel is shown in gold. Other pixels in S are shown in gray with a potential boundary of S indicated by the dotted black line. S may become disconnected in the three highlighted cases.

B. Single Transpose Debt Analysis

Let r be the number of rows remaining after $h_{out} - r$ rows have already been processed (each processed row adding $w_{out}(f_{out} - f_{in}) - (k-1)f_{in}$ debt, as analyzed in Section 3.4.1). Two cases need to be analyzed separately: either the remaining columns do not add debt, if $r(f_{out} - f_{in}) - (k-1)f_{in} \leq 0$, or they do.

Case 1: If the remaining columns do not add debt, the worst case debt happens immediately before completing the processing of $h_{out} - r$ rows of output pixels. Therefore, the total maximum debt is:

$$\begin{aligned} D_{st1}(r) &= (h_{out} - r)D(w_{out}) + kf_{in} \\ &= (h_{out} - r)\left(w_{out}(f_{out} - f_{in}) - (k-1)f_{in}\right) \\ &\quad + kf_{in} \end{aligned} \tag{7}$$

Equation (7) is minimized when r is maximized while satisfying $r(f_{out} - f_{in}) - (k-1)f_{in} \leq 0$. Therefore, $r_1^* = \lfloor (k-1)f_{in}/(f_{out} - f_{in}) \rfloor$ and:

$$\begin{aligned} D_{st1} &= D_{st1}(r_1^*) \\ &= h_{out}w_{out}(f_{out} - f_{in}) - r_1^* w_{out}(f_{out} - f_{in}) \\ &\quad - (h_{out} - r_1^*)(k-1)f_{in} + kf_{in} \end{aligned} \tag{8}$$

Case 2: If the remaining columns add debt, the worst case debt happens after processing all output pixels. Therefore, the total maximum debt is:

$$\begin{aligned} D_{st2}(r) &= (h_{out} - r)D(w_{out}) + w_{out}D(r) + kf_{in} \\ &= (h_{out} - r)\left(w_{out}(f_{out} - f_{in}) - (k-1)f_{in}\right) \\ &\quad + w_{out}\left(r(f_{out} - f_{in}) - (k-1)f_{in}\right) \\ &\quad + kf_{in} \\ &= h_{out}w_{out}(f_{out} - f_{in}) \\ &\quad - (h_{out} + w_{out} - r)(k-1)f_{in} + kf_{in} \end{aligned} \tag{9}$$

Equation (9) is minimized when r is minimized while satisfying $r(f_{out} - f_{in}) - (k-1)f_{in} > 0$. Therefore, $r_2^* = \lceil (k-1)f_{in}/(f_{out} - f_{in}) \rceil = r_1^* + 1$ and:

$$\begin{aligned} D_{st2} &= D_{st2}(r_1^* + 1) \\ &= h_{out}w_{out}(f_{out} - f_{in}) \\ &\quad - (h_{out} + w_{out} - r_1^* - 1)(k-1)f_{in} + kf_{in} \end{aligned} \tag{10}$$

Overall, $D_{st} = \min(D_{st1}, D_{st2})$. More interesting than this result, however, is a comparison to the optimal herringbone method. To do this, we consider the margin non-optimality of the single transpose method to the herringbone method.

$$\begin{aligned} D_{\Delta TH} &= D_{st} - D_{hb} \\ &= \min(D_{st1} - D_{hb}, D_{st2} - D_{hb}) \end{aligned} \tag{11}$$

Again, we analyze the two cases separately. As a reminder, $D_{hb} = (h_{out}w_{out} - \lfloor x^* \rfloor^2)f_{out} - (h_{in}w_{in} - (\lfloor x^* \rfloor + k - 1)^2 - k)f_{in}$ where $x^* = \lfloor (k-1)f_{in}/(f_{out} - f_{in}) \rceil = r_1^*$. Therefore, $D_{hb} = h_{out}w_{out}(f_{out} - f_{in}) - (r_1^*)^2(f_{out} - f_{in}) - (h_{out} + w_{out} - 2r_1^*)(k-1)f_{in} + kf_{in}$.

Case 1:

$$\begin{aligned} D_{\Delta TH1} &= D_{st1} - D_{hb} \\ &= h_{out}w_{out}(f_{out} - f_{in}) - r_1^* w_{out}(f_{out} - f_{in}) \\ &\quad - (h_{out} - r_1^*)(k-1)f_{in} + kf_{in} \\ &\quad - h_{out}w_{out}(f_{out} - f_{in}) \\ &\quad + (r_1^*)^2(f_{out} - f_{in}) \\ &\quad + (h_{out} + w_{out} - 2r_1^*)(k-1)f_{in} - kf_{in} \\ &= r_1^*(r_1^* - w_{out})(f_{out} - f_{in}) \\ &\quad + (w_{out} - r_1^*)(k-1)f_{in} \\ &= (w_{out} - r_1^*)\left[(k-1)f_{in} - r_1^*(f_{out} - f_{in})\right] \\ &= (w_{out} - r_1^*)\alpha \end{aligned} \tag{12}$$

Where:

$$\begin{aligned} \alpha &= \left[(k-1)f_{in} - r_1^*(f_{out} - f_{in})\right] \\ &= (k-1)f_{in} \mod (f_{out} - f_{in}) \end{aligned} \tag{13}$$

Case 2:

$$\begin{aligned} D_{\Delta TH2} &= D_{st2} - D_{hb} \\ &= h_{out}w_{out}(f_{out} - f_{in}) \\ &\quad - (h_{out} + w_{out} - r_1^* - 1)(k-1)f_{in} + kf_{in} \\ &\quad - h_{out}w_{out}(f_{out} - f_{in}) \\ &\quad + (r_1^*)^2(f_{out} - f_{in}) \\ &\quad + (h_{out} + w_{out} - 2r_1^*)(k-1)f_{in} - kf_{in} \\ &= -(r_1^* - 1)(k-1)f_{in} \\ &\quad + (r_1^*)^2(f_{out} - f_{in}) \\ &= (k-1)f_{in} - r_1^*\left[(k-1)f_{in} - r_1^*(f_{out} - f_{in})\right] \\ &= (k-1)f_{in} - r_1^*\alpha \end{aligned} \tag{14}$$

Overall,

$$\begin{aligned} D_{\Delta TH} &= D_{st} - D_{hb} \\ &= \min(D_{\Delta TH1}, D_{\Delta TH2}) \\ &= \min(w_{out}\alpha, (k-1)f_{in}) - r_1^*\alpha \end{aligned} \tag{15}$$

From (15), it can be seen that the single transpose method is optimal when $w_{out} = r_1^*$, $(k-1)f_{in} = r_1^*\alpha$, or $\alpha = 0$. This last condition holds when $(k-1)f_{in}$ divides $f_{out} - f_{in}$.

C. Extensions to Memory-Optimal Convolutions

Section 3 detailed memory optimality in the most common use cases for convolutional layers. This section covers minor extensions beyond the restrictions of that section, including the effect of non-square convolution kernels C.1, padding C.2, stride C.3, and the popular residual connection layer C.4 and depthwise separable convolution layer C.5. Combinations of these extensions are generally not considered. For example, the valid padding restrictions are assumed when analyzing different strides.

C.1. Non-square Kernels

For non-square kernels, the decreasing channel depth case is still easy to deal with, since output pixels can be stored directly in the input pixels that become stale — at least one is guaranteed per output pixel that is processed. However, the corresponding herringbone and single-transpose methods change slightly.

C.1.1. Non-square Kernel Herringbone

The general principle of the herringbone method is to greedily choose the row or column that will result in the smallest debt accrual. With square kernels, the debt function $D(x) = x(f_{out} - f_{in}) - (k-1)f_{in}$ is the same for rows and columns and only depends on the length x. With a non-square kernel, there are two debt functions: $D_r(x_r) = x_r(f_{out} - f_{in}) - (k_w - 1)f_{in}$ and $D_c(x_c) = x_c(f_{out} - f_{in}) - (k_h - 1)f_{in}$ for rows and columns, respectively. We can equate these two to find when to switch from rows to columns and vice versa:

$$\begin{aligned} 0 &= D_r(x_r) - D_c(x_c) \\ &= x_r(f_{out} - f_{in}) - (k_w - 1)f_{in} \\ &\quad - x_c(f_{out} - f_{in}) + (k_h - 1)f_{in} \\ x_c &= x_r + \frac{k_h - k_w}{f_{out} - f_{in}} f_{in} \end{aligned} \tag{16}$$

Letting $x_0 = (k_h - k_w)f_{in}/(f_{out} - f_{in})$, the result of (16) implies that we should start by processing $\lfloor x_0 \rfloor$ rows (or $\lceil -x_0 \rceil$ columns), then process the remainder with the standard herringbone sequence of alternating row-column-row-column.

C.1.2. Non-square Kernel Single Transpose

As in Section 3.4.3, let r be the number of rows remaining after $h_{out} - r$ rows have already been processed. Two cases need to be analyzed separately: either the remaining columns do not add debt (when $r(f_{out} - f_{in}) - (k_h - 1)f_{in} \leq 0$) or else they do. Letting $r_1^* = \lfloor (k_h - 1)f_{in}/(f_{out} - f_{in}) \rfloor$, $r_2^* = r_1^* + 1$, and $\alpha = (k_h - 1)f_{in} \bmod (f_{out} - f_{in})$,

$$\begin{aligned} D_{st1} &= (h_{out} - r_1^*)D_r(w_{out}) + k_w f_{in} \\ &= (h_{out} - r_1^*)\left(w_{out}(f_{out} - f_{in}) - (k_w - 1)f_{in}\right) \\ &\quad + k_w f_{in} \end{aligned} \tag{17}$$

$$\begin{aligned} D_{st2} &= (h_{out} - r_2^*)D_r(w_{out}) \\ &\quad + w_{out}D_c(r_2^*) + k_h f_{in} \\ &= (h_{out} - r_2^*)\left(w_{out}(f_{out} - f_{in}) - (k_w - 1)f_{in}\right) \\ &\quad + w_{out}\left(r_2^*(f_{out} - f_{in}) - (k_h - 1)f_{in}\right) + k_h f_{in} \\ &= D_{st1} - w_{out}\alpha + (k_h - 1)f_{in} \end{aligned} \tag{18}$$

As before,

$$\begin{aligned} D_{st} &= \min(D_{st1}, D_{st2}) \\ &= (h_{out} - r_1^*)\left(w_{out}(f_{out} - f_{in}) - (k_w - 1)f_{in}\right) \\ &\quad + k_w f_{in} + \min(0, (k_h - 1)f_{in} - w_{out}\alpha) \end{aligned} \tag{19}$$

Therefore, we take the single transpose when

$$r = \begin{cases} r_1^* & w_{out}\alpha \leq (k_h - 1)f_{in} \\ r_1^* + 1 & \text{else} \end{cases} \tag{20}$$

C.2. Effect of Padding

Section 3 analyzed memory-optimal convolutions with valid padding. One other popular padding style is "same" padding, in which the output feature map has the same height and width as the input feature map. This is equivalent to valid padding applied to an input feature map that has been padded with $(k-1)/2$ zeros on all sides[7]. Therefore, we can immediately see that an upper bound on the debt accrued can be found by applying the analysis of Section 3 to an input image of size $(h_{in} + k) \times (w_{in} + k)$ and adding the zero-pad debt: $(h_{in} + k)(w_{in} + k) - h_{in}w_{in}$. It is possible to do better than this bound (see Section C.4, which uses same-padded convolutions). We omit a more detailed analysis.

[7] For even k, two of the sides will have one fewer padding than the other two sides.

C.3. Effect of Stride

With stride (r_s, c_s), every output pixel that is processed makes $r_s \times c_s$ input pixels stale. Mathematically, this looks a lot like having $f_{in}^{'} = r_s c_s f_{in}$ for the purposes of deciding whether channel depth is increasing or not. If $f_{out} \leq r_s c_s f_{in}$, then we can use the methods of Section 3.3 and we are done. Otherwise, we can compute the row-wise and column-wise debt functions:

$$D_r(x_r) = x_r(f_{out} - r_s c_s f_{in}) + r_s k_w f_{in} \quad (21)$$
$$D_c(x_c) = x_c(f_{out} - r_s c_s f_{in}) + c_s k_h f_{in} \quad (22)$$

Equating these two to find when to switch between rows and columns for the herringbone method.

$$\begin{aligned} 0 &= D_r(x_r) - D_c(x_c) \\ &= x_r(f_{out} - r_s c_s f_{in}) + r_s k_w f_{in} \\ &\quad - x_c(f_{out} - r_s c_s f_{in}) - c_s k_h f_{in} \\ x_c &= x_r + \frac{r_s k_w - c_s k_h}{f_{out} - r_s c_s f_{in}} f_{in} \end{aligned} \quad (23)$$

This is the same form as (16) and so the methods of Section C.1.1 can be used with $x_0 = (r_s k_w - c_s k_h) f_{in} / (f_{out} - r_s c_s f_{in})$. Equations (21) and (22) can also be used to analyze the single transpose method as in Section C.1.2.

C.4. Residual Connections

Residual connections (He et al., 2015) comprise a range of different building blocks. A popular block is the two-layer block defined as $y_2 = W_2 * \sigma(y_1) + b_2 + x$, where $y_1 = W_1 * x + b_1$, σ is the ReLU function, and W_1 and W_2 are convolution kernels with kernel dimensions 3×3. We focus our attention on this particular two-layer residual connection building block, but similar techniques could be used for other building blocks.

Memory-efficient residual connection convolutions can be performed using $3w_{out} + 2$ additional memory. First note that in residual connections, channel size stays the same, meaning we can use the techniques of Section 3.3, except here we need "same" padding. For convolutions that compute edge pixels, zero pads are not explicitly added in memory. Instead, we need to keep track of when to multiply the kernel with zeros versus input features.

We will proceed by computing rows of the intermediate pixels y_1 and the final output pixels y_2 noting that input pixels only go stale after their dependency to y_2 is complete. To simplify the description of the algorithm, we need only describe the sequence of rows of pixels to compute (and which of y_1/y_2 are being computed).

Number the rows from 1 to $h_{out}+3$. We begin by computing y_1 in rows 2 and 3. Then we compute y_2 in row 1 using the y_1 dependencies in rows 2 and 3 and the input x dependency in row 4. To avoid collisions with y_1, y_2 must be offset by two additional pixels. Now row 4 is stale and the next row of y_1 can be computed, which means row 2 of y_2 can be computed, again with an offset of two pixels. The process repeats until completion — y_1 in row 5, y_2 in row 3 (with two pixel offset), y_1 in row 6, and so on.

C.5. Depthwise Separable Convolutions

A traditional convolution takes an $h_{in} \times w_{in} \times f_{in}$ feature map to an $h_{out} \times w_{out} \times f_{out}$ feature map with a $k_h \times k_w \times f_{in} \times f_{out}$ kernel. Depthwise separable convolutions (Howard et al., 2017) save weight memory and compute by splitting the standard convolution into two separate convolutions. First, it applies a set of f_{in} distinct $k_h \times k_w \times 1 \times m$ kernels to the input feature map, one per channel, to get an intermediate $h_{out} \times w_{out} \times m$ f_{in} feature map. Then it applies a $1 \times 1 \times m$ $f_{in} \times f_{out}$ kernel to get the final $h_{out} \times w_{out} \times f_{out}$ feature map. Herringbone can be applied to the first of these two convolutions if $m > 1$ relatively straightforwardly. In fact, because the f_{in} kernels apply distinctly to separate input channels, input memory becomes stale after every m output pixels is computed (rather than after every m f_{in}), potentially allowing for lower peak memory debt.

D. Memory Efficient Spectrograms

While the focus of this paper is memory-optimal 2D convolutions for embedded systems, we note that embedded systems more often deal with one-dimensional time-series data. For example, a common application is to recognize patterns in accelerometer data or classify acoustic signals from a microphone. 2D convolutions can still be relevant, however, because time series data can be converted to a meaningful 2D representation using a spectrogram, with Mel-spectrograms being especially popular in acoustic classification tasks (Hasan et al., 2004b; Hershey et al., 2016).

In this section, we present a memory-efficient spectrogram implementation amenable to hardware in order to establish the feasibility of fitting the entire classification pipeline on one embedded device.

D.1. Spectrograms

A spectrogram is generated by splitting time series data into overlapping frames and finding the spectrum of each frame using a Fourier transform, generating a matrix X whose elements are

$$X_{nm} = \sum_{k=0}^{T-1} w(nD-k)x(k)e^{2\pi jkm/T} \tag{24}$$

where x is the input, w is a windowing function (usually the Hamming window), and $n \in [0, N-1]$ gives the time index and $m \in [0, M-1]$ gives the frequency index of X. Frames are T samples long and are offset by D samples (Allen, 1977). As a minor detail, we are often more interested in the power of the signal $P_{nm} = |X_{nm}|^2$.

One way to compute X is by running one FFT per frame, a computational cost of $\mathcal{O}(T \log T)$ per frame (Cooley & Tukey, 1965) and a maximum memory cost of $\mathcal{O}(NM+T)$, representing the storage costs for X and the FFT algorithm. However, in many applications, only a few frequency components are required. For example, speech applications may use anywhere from 64 bins (Hershey et al., 2016) to 32 bins (Kusupati et al., 2018) to 16 bins or even lower (Hasan et al., 2004a). Additionally, the bins are often logarithmically spaced (Hasan et al., 2004b), necessitating a much larger FFT to resolve frequencies at the bottom of the spectrum. This means $T \gg M$ and we should prefer algorithms that do not have a factor of T in their memory complexity.

D.2. Proposed Algorithm

Our proposal is the computation of an approximate spectrogram by keeping running sums of D-length chunks of the input signal dotted with a periodic signal at each of the M frequency bins of interest. Because running sums are used, T input data points can be stored using a constant number of accumulators per frequency bin rather than having to store all T samples for use in an FFT. For hardware efficiency, we propose using a square wave as the periodic signal and analyze the implications for the resulting power spectrum in Appendix D.3.

The proposed algorithm works by first maintaining two intermediate arrays $A_I^{(n)} \in \mathbb{R}^{H \times M}$ and $A_Q^{(n)} \in \mathbb{R}^{H \times M}$, which at the n^{th} chunk, represent the quadrature components of the signal for the past $H \approx 5$ chunks at each of M frequencies of interest. The first column of $A_{I/Q}^{(n)}$ are computed as:

$$A_{I,m,1}^{(n)} = \sum_{k=0}^{D-1} x(nD+k) f_{sq}\left(\frac{2\pi}{T}(nD+k)m - \frac{\pi}{2}\right) \tag{25}$$

$$A_{Q,m,1}^{(n)} = \sum_{k=0}^{D-1} x(nD+k) f_{sq}\left(\frac{2\pi}{T}(nD+k)m\right) \tag{26}$$

where f_{sq} is a square wave with period 2π and amplitude 1. Note that $A_I^{(n)}$ and $A_Q^{(n)}$ can be computed in real time as samples are streamed in and therefore do not require any raw sample storage. Every D samples, the contents of $A_{I/Q}^{(n)}$ can be shifted right one column to get $A_{I/Q}^{(n+1)}$ so that partial accumulations as in (25) and (26) can be stored in the first column.

Every new chunk, the spectrogram power components P_{mj} at time j can be computed as:

$$P_{mj} = \left(\sum_{i=0}^{H-1} w(i) A_{I,mi}\right)^2 + \left(\sum_{i=0}^{H-1} w(i) A_{Q,mi}\right)^2 \tag{27}$$

where w is a windowing function that operates on chunks rather than individual time samples and is a down-sampled version of popular windows such as the Hamming window. After N chunks have been processed, P is complete and can be used as input features for classification[8].

[8] In practice, we would store log-components of P rather than P itself and use those for classification.

The computational complexity is $\mathcal{O}(DM + HM)$, which is the sum of the accumulations required to compute $A^{(n)}_{I/Q}$ and the number of accumulations required to compute $P_{:,j}$. The maximum memory cost is $\mathcal{O}(NM + HM)$, representing the storage costs for P and A.

D.3. Square Wave Analysis

A square wave can be decomposed into its Fourier components as:

$$f_{sq}(x) = \frac{4}{\pi} \sum_{\ell=1,3,5,\ldots} \frac{1}{\ell} \sin(x\ell) \tag{28}$$

If we let $X^{(n)}_m$ represent the m^{th} frequency bin of the DFT of $\{x(nD), x(nD+1), \ldots, x(nD+D-1)\}$, and let $X^{(n)}_m = X^{(n)}_{R,m} + jX^{(n)}_{I,m}$ where $X^{(n)}_{R,m}$ and $jX^{(n)}_{I,m}$ are both real. Then,

$$\begin{aligned} A^{(n)}_{I,m,1} &= \sum_{k=0}^{D-1} x(nD+k) f_{sq}\left(\frac{2\pi}{T}(nD+k)m - \frac{\pi}{2}\right) \\ &= \frac{4}{\pi} \sum_{\ell=1,3,5,\ldots} \frac{1}{\ell} \sum_{k=0}^{D-1} \\ &\quad x(nD+k) \cos\left(\frac{2\pi}{T}\ell(nD+k)m\right) \\ &= \frac{4}{\pi} \sum_{\ell=1,3,5,\ldots} \frac{1}{\ell} X^{(n)}_{R,\ell m} \end{aligned} \tag{29}$$

$$A^{(n)}_{Q,m,1} = \frac{4}{\pi} \sum_{\ell=1,3,5,\ldots} \frac{1}{\ell} X^{(n)}_{I,\ell m} \tag{30}$$

Let $P^{(n)}_{a,b}$ represent the covariance of signals $X^{(n)}_a$ and $X^{(n)}_b$. In the case when $a = b$, $P^{(n)}_{a,a} \equiv P^{(n)}_a$ is the power of $X^{(n)}_a$. We find:

$$\begin{aligned} &\left(A^{(n)}_{I,m,1}\right)^2 + \left(A^{(n)}_{Q,m,1}\right)^2 \\ &= \left(\frac{4}{\pi}\right)^2 \sum_{k,\ell=1,3,5,\ldots} \frac{1}{k\ell} \left(X^{(n)}_{R,km} X^{(n)}_{R,\ell m} + X^{(n)}_{I,km} X^{(n)}_{I,\ell m}\right) \\ &= \left(\frac{4}{\pi}\right)^2 \sum_{k,\ell=1,3,5,\ldots} \frac{1}{k\ell} P^{(n)}_{km,\ell m} \\ &= \left(\frac{4}{\pi}\right)^2 \left[P^{(n)}_m + \frac{2}{3} P^{(n)}_{m,3m} + \frac{2}{5} P^{(n)}_{m,5m} + \cdots + \frac{1}{9} P^{(n)}_{3m} + \cdots\right] \end{aligned} \tag{31}$$

Equation (31) shows that a significant proportion of power in bin m using the square wave can come from cross-terms between the signal in the m^{th} frequency bin and its odd harmonics. We hypothesize that this level of signal corruption is still acceptable for neural network classification.

D.4. Comparison of Techniques

The proposed spectrogram method is qualitatively compared to a standard Log-Mel spectrogram method (Fayek, 2016), as seen in Figure 12. Compared to the standard method, the memory-efficient version is much noisier. This is expected, since the standard method has the benefit of multi-FFT-bin averaging and no harmonic leakage. In Section F, we see that despite the lower fidelity, networks are still able to perform audio classification.

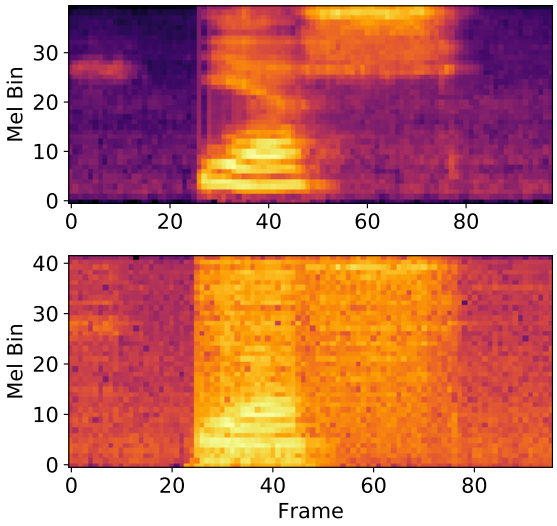

Figure 12. Comparison between standard Log Mel spectrogram features (top) and memory-efficient spectrogram features (bottom) for the word "yes" from the Google-12 dataset (Warden, 2018).

E. Tricks for Small Networks

We found the following techniques helpful for maximizing MNIST accuracy.

E.1. Herringbone

Using the herringbone method for convolutions allowed us to select a larger and more accurate model. Compared to the best model that would have fit with the replace method, herringbone improved accuracy by $\approx 0.2\%$ over the replace method or $\approx 0.5 - 1\%$ better than a naive method, after utilizing all of the other tricks. Note that this improvement would be much more pronounced in a scenario where only activation memory is constrained, as opposed to both weights and activations.

E.2. Multi-Train Selection

In contrast to larger networks, we hypothesize that smaller networks are more susceptible to bad initializations and training steps. Therefore, we run training multiple times and evaluate performance on the validation set every epoch to decide on the best model parameters. Specifically, we run 50 epochs floating point + 200 epochs quantized training a total of 10 times and select the best quantized network by validation loss. On average, this results in $\approx 0.2\%$ improvement.

E.3. Data Augmentation

Elastic distortions (Simard et al., 2003) are a powerful augmentation technique for MNIST. We found that just applying them directly actually reduced accuracy and hypothesize that this is because the small network gets confused by bad training examples. To fix this, we train a more powerful network and use that to decide on "hard" versus "easy" training examples. We discard any training example it misclassified. This results in $\approx 0.2\%$ improvement.

E.4. Orthogonal Initialization

We hypothesize that one of the potential failure modes for bad initialization is when weights between different channels are too highly correlated, significantly limiting the a priori span of the weights. To fix this, we apply orthogonal initialization (Saxe et al., 2013) and see $\approx 0.1\%$ improvement on average (just above the noise floor of test accuracy).

E.5. Other Layers

There are no major tricks required to get the non-convolutional layers working, as they are much simpler to implement and are not the memory bottlenecks of our network. Figure 13 briefly illustrates how these other layers are implemented.

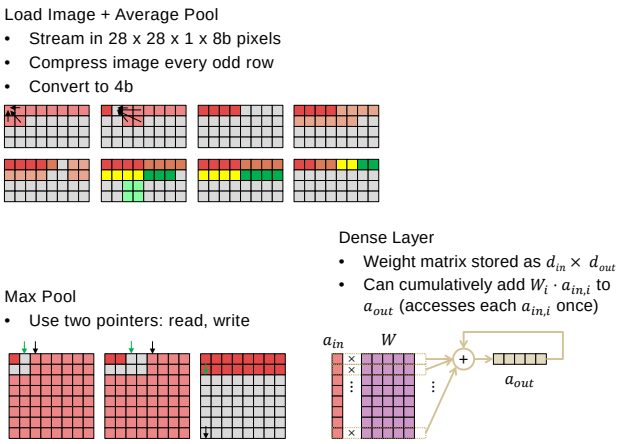

Figure 13. Overview of other layers implemented in the Arduino.

F. Other Experiments

The goal of our case study was to establish the feasibility of implementation of our proposed convolution methods. Accordingly, we focused attention on just the single MNIST experiment. However, the approach should be extensible to any small-scale 2D classification task. Here we look at a few more experiments, with quantized performance validated in software.

The first two examples fit all weights and activations in SRAM, while the other examples use SRAM only for activation memory, writing the network weights to 32KB Flash. Data augmentation as described in Section E is used in all experiments. Google-12 uses preprocessing as described in Section D.1 based off of the preprocessing described by Kusupati et al. (2018) of the Google keyword spotting dataset (Warden, 2018).

Table 2. Simulated Results on Other Datasets.

DATASET	ARCHITECTURE	ACCURACY
MNIST-10 2KB W+A	AVG POOL 2×2 CONV $k : 3 \times 3, s : 1 \times 1, f_o : 5$ CONV $k : 3 \times 3, s : 1 \times 1, f_o : 8$ CONV $k : 3 \times 3, s : 1 \times 1, f_o : 11$ MAX POOL 2×2 DENSE 10	99.15%
CIFAR-10 2KB W+A	AVG POOL 2×2 CONV $k : 3 \times 3, s : 1 \times 1, f_o : 7$ CONV $k : 3 \times 3, s : 1 \times 1, f_o : 12$ MAX POOL 2×2 DENSE 10	55.78%
CIFAR-10 2KB A	CONV $k : 2 \times 2, s : 2 \times 2, f_o : 12$ RESUNIT $k : 3 \times 3, f_o : 12$ MAX POOL 2×2 D-S CONV $k : 3 \times 3, m : 5, f_o : 80$ DENSE 10	71.07%
GOOGLE-12 2KB A	INPUT 24×96 CONV $k : 1 \times 4, s : 1 \times 4, f_o : 6$ CONV $k : 3 \times 3, s : 2 \times 2, f_o : 24$ CONV $k : 3 \times 3, s : 1 \times 1, f_o : 28$ RESUNIT $k : 3 \times 3, f_o : 28$ RESUNIT $k : 3 \times 3, f_o : 40$ DENSE 12	85.29%