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^*. □

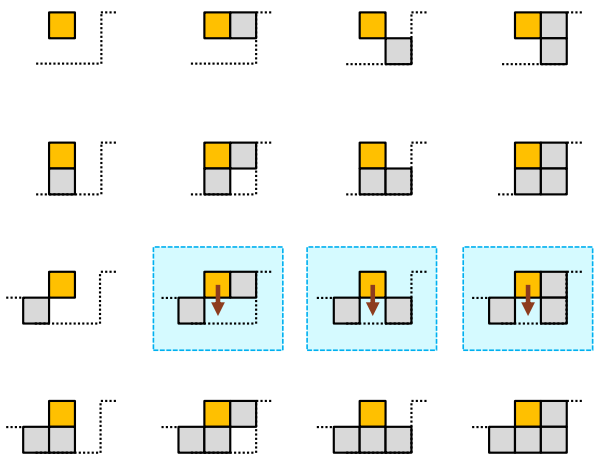

Figure 11. Possible configurations of neighboring pixels to a top-left pixel. The top-left pixel is shown in gold. Other pixels in S are shown in gray with a potential boundary of S indicated by the dotted black line. S may become disconnected in the three highlighted cases.

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.

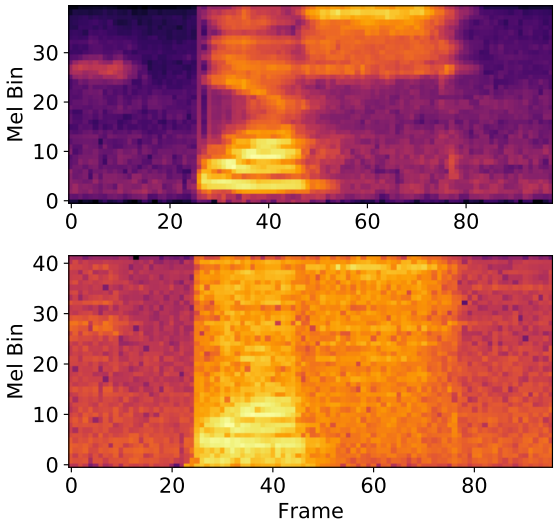

Figure 12. Comparison between standard Log Mel spectrogram features (top) and memory-efficient spectrogram features (bottom) for the word "yes" from the Google-12 dataset (Warden, 2018).

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.

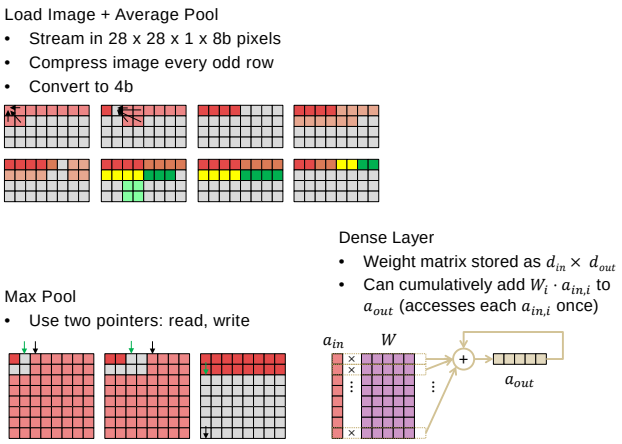

Figure 13. Overview of other layers implemented in the Arduino.

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%