On-the-fly deterministic binary filters for memory efficient keyword spotting applications on embedded devices

Javier Fernández-Marqués[†], Vincent W.-S. Tseng[‡], Sourav Bhattachara[*], Nicholas D. Lane[†*]

[†] University of Oxford, [‡] Cornell University, [*]Nokia Bell Labs

ABSTRACT

Lightweight keyword spotting (KWS) applications are often used to trigger the execution of more complex speech recognition algorithms that are computationally demanding and therefore cannot be constantly running on the device. Often KWS applications are executed in small microcontrollers with very constrained memory (e.g. 128kB) and compute capabilities (e.g. CPU at 80MHz) limiting the complexity of deployable KWS systems. We present a compact binary architecture with 60% fewer parameters and 50% fewer operations (OP) during inference compared to the current state of the art for KWS applications at the cost of 3.4% accuracy drop. It makes use of binary orthogonal codes to analyse speech features from a voice command resulting in a model with minimal memory footprint and computationally cheap, making possible its deployment in very resource-constrained microcontrollers with less than 30kB of on-chip memory. Our technique offers a different perspective to how filters in neural networks could be constructed at inference time instead of directly loading them from disk.

CCS Concepts

•Computing methodologies → Speech recognition; Neural networks; •Theory of computation → Network optimization;

1. INTRODUCTION

KWS has become a popular always-on feature in smartphones, wearables and smart home devices. It serves as the entry point for speech based applications once a predefined command (e.g. "Ok Google", "Hey Siri", "Alexa") is detected from a continuous stream of audio. Because KWS applications are always running they follow a very efficient architectural design and are often implemented on small dedicated microcontrollers. These devices are constrained in terms of memory and compute capabilities, limiting the complexity and memory footprint of the deployed model.

We present BinaryCmd, read as "binary command", a novel neural network (NN) architecture for audio that represents the weights as a combination of predefined orthogonal binary basis that can be generate very efficiently on-the-fly. This property would enable the off-loading of the convolutional filters from the model, resulting in models with smaller memory footprint and a more efficient inference stage. Inspired by ResNet's *bottleneck* layer [15] and LBCNN [19], where the suitability of sparse binary filters for image classification tasks is proven, we present a vastly reduced architecture from those generally use for vision problems and adjusted it at both macroarchitectural and microarchitectural levels to better capture the temporal dimension of input audio commands. Here we make use of Deterministic Binary Filters, DBFs [29], that are constructed as a linear combination of deterministic binary codes.

We compare our work to *HelloEdge* [34] following their microcontroller classification scheme and particularly focusing on the Small (S) group, where the model size limit is set to 80kB and the maximum number or OPs during inference is 6M. Likewise, we use Google's Speech Commands Dataset [33] to train and evaluate our architecture. BinaryCmd requires significantly less parameters and OPs than the best architecture in [34] that relies in depthwise separable convolutional neural networks (DS-CNN) [16, 9] and that is, to the best of our knowledge, the current state of the art for KWS applications. This work is an extension of [12] and offers the following contributions:

- In [29] we introduced and validated the usage of DBFs in medium sized networks for image classification, here we demonstrate their suitability for audio on architectures with a extremely small memory-footprint.
- We show that our architecture leads to state of the art results in the set of architectures with fewer than 3M OPs and 30kB of model size. We compare BinaryCmd to all the baselines in [34].
- We implemented our deterministic binary code generator on an ARM Cortex-M7 microcontroller and showed that on-the-fly filter generation results in an overhead of just 13ms for our top performer network.

2. RELATED WORK

Deep learning for embedded devices has become increasingly popular in recent years for a variety of applications including image classification, speech recognition and health. Energy efficiency and low computational complexity are two properties that are specially important in memory and compute restricted platforms [22] and they become a major concern when considering the commercialization of such applications. It has proven to be a challenge for the research community to design algorithms and architectures that meet those requirements simultaneously for complex tasks [23]. The existing research addressing these issues can be classified into three categories: NN compressing techniques, novel NN layer architectures and novel system architectures.

EMDL'18, June 15, 2018, Munich, Germany

DOI: https://doi.org/10.1145/3212725.3212731

Compression techniques. Most techniques fixate their efforts in reducing the number of weights and in exploiting sparsity. Existing compression techniques such as *Deep-Compression* [14] and *CNNPack* [30] are a conglomerate of clustering, quantisation and word encoding techniques. A step further is taken by [31] in which, for a given convolutional layer, filters are restricted to produce feature maps with minimal redundancy.

Layer architectures. Works lying in this category include *bottleneck* [15] layers, that reduce the number of channels of the input tensor by using 1×1 filters prior to to convolve it with a spatially larger kernel with the aim of reduce the number or of OPs. Along the same lines, depthwise [16] convolutional layers split the standard convolutional layer into two convolutional layers achieving a reduction in operations of $1/N + 1/D_k^2$, where N is the number of output channels and D_k is the kernel spatial dimensions. [24] proposes a technique to dynamically adjust the number of input channels and filters to use during inference time. [6] exploits kernel sparsification and separation allowing large NN-based models run efficiently on embedded hardware.

System architectures. DRAM accesses severely impact both throughput and energy efficiency [8]. A number of works [13, 7, 11] explore parallel architectures combined with several levels of local memory hierarchy in order to reduce the accesses to DRAM. Such systems, often designed as a coprocessor (FPGA or ASIC) that sits next to the CPU, reach unseen power efficiency levels, as is the case of *Origami* [7] promising 803 GOp/s/W at 0.8 V. It uses an SRAM module of 344 Kbit to store image patches loaded from DRAM and two sectors of registers to temporally store convolutional filters and rows of pixels from the patches in the SRAM, respectively. For a broad evaluation of the current advances in the field of efficient DNNs we refer the interested reader to the Eyeriss Project's survey paper [8].

With respect to KWS, several approaches have been explored including the use of DNN [32], LSTM [27], CNN [28] or CRNN [5], being this last technique able to offer a good trade-off between accuracy, model size and inference costs. [34] provides a complete comparison between several approaches to KWS on constrained platforms in addition to show the suitability of depthwise convolutions for this task with lowest memory usage and the highest accuracy rates among the previously mentioned techniques.

3. A BINARY NETWORK FOR KWS

In this section we describe the design of the evaluated KWS architecture, including how the convolutional filters are constructed and how this novel filter design differs from those found in standard CNN-based architectures.

System Overview. The implemented KWS system is comprised of two fundamental blocks where speech features are first extracted from a $1s$ input voice command and are then fed to a NN-based block that outputs the id of the detected voice command. The system's macroarchitecture is depicted in Figure 1. We follow the same strategy as in [34] to extract an array of 49×10 MFCC [1] speech features from the input speech signal and feed them to our network, BinaryCmd. This NN-based block hierarchically transforms the audio features input into a keyword id that identifies the command detected by the KWS system.

Architecture. We present a novel NN block containing the following elements: three nested *on-the-fly* convolutional layers (they represent BinaryCmd's core, as seen in Figure 2) followed by a standard convolutional with filter dimensions $inCh \times 3 \times 3 \times \ numCls$ (where $numCls$, is the number of classes in the dataset and $inCh$ the number of channels of the input tensor. This parameter varies between each evaluated configuration), max-pooling and fully connected layers. All convolutional layers use ReLu as activation functions and have been trained using batch normalization [17].

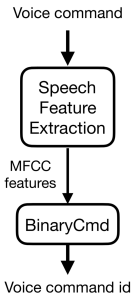

Figure 1: System arch.

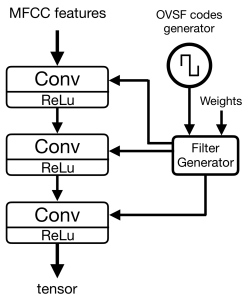

Figure 2: BinaryCmd core.

***On-the-fly* convolutions**. Unlike standard CNNs, our architecture does not learn convolutional filters directly. Instead, it learns weighting coefficients of deterministic binary basis that are combined in a linear fashion manner to generate the filters [29]. We call this filters *deterministic binary filters*, or DBFs. We use orthogonal variable spreading factor, OVSF[2], binary codes of length 2^n, $n \in \mathbb{N}$, to generate these basis. OVSF codes are designed in such a way that for any given code length L, there are L different OVSF codes and that are orthogonal to each other. Therefore, to generate a filter of dimensions $dim = W \times H \times C$, our OVSF code generator could output at most dim different codes that would form a basis of the $\mathbb{R}^{dim}$ space. Intuitively, by combining all OVSF codes of a given dimension dim we could perfectly represent any filter of that dimension. On the other hand, using fewer OVSF codes would result in a coarser representation of the target filter. Mathematically, the quality of a filter generated by combination of OVSF codes could be measured as:

$$E_k = \left\| \boldsymbol{f}'_{\boldsymbol{k}} - \boldsymbol{f}_{\boldsymbol{k}} \right\|_2^2 = \left\| \sum_{i=0}^{\lfloor \rho \cdot l \rfloor} \alpha_k^i \boldsymbol{B}_k^i - \boldsymbol{f}_{\boldsymbol{k}} \right\|_2^2 < \epsilon, \qquad (1)$$

where $\rho \in [0, 1]$ is the ratio of codes to use in order to approximate filter f_k, l is the total number of OVSF codes of length $l = WHC$, $\boldsymbol{B}_{\boldsymbol{k}}^i$ is the ith OVSF code and $\alpha_j^i \in \mathbb{R}$ its associated weight. During training, we learn those weights. ϵ is the difference between the the approximated, f'_k and the real filter, f_k. Here f'_k represents a DBF. Intuitively, $\epsilon \to 0$ as we increase the ratio of binary codes used. When $\rho \neq 1$, the product $p \cdot l$ is rounded to the nearest integer value.

[1]Mel-frequency cepstral coefficients (MFCCs) are commonly used as features in speech recognition systems. They are part of the ETSI [10] standard for mobile phones.

[2]OVSF codes were introduced for 3G communication systems as channelizations codes aiming to increase system capacity in multi-user access scenarios[2]. They have been extensively studied in the wireless community [4, 26, 20, 25] and widely used commercially in 3G mobile systems.

These codes can be generated efficiently by a recursive algorithm similarly to how Hadamard matrices are constructed. A detailed explanation of how OVSF codes are generated in our system is presented at the end of this section.

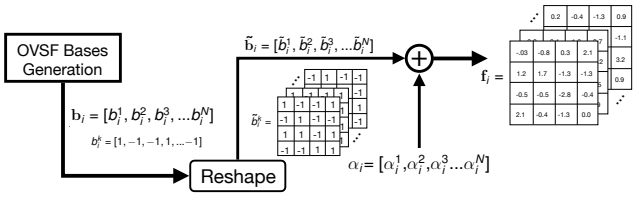

Figure 3: From OVSF codes to DBFs. Each code b_i^k is first reshaped to match the final filter dimensions, becoming $\tilde{b}_i^k$. Then, the reshaped codes in $\tilde{b}_i$ are combined using the learnt weights α_i.

The filter creation process using the OVSF basis can be didactically illustrated as in Figure 3: First the generator outputs a set of 2^n-dimensional arrays of $[+1, -1]$ elements; then the arrays get reshaped to match the dimensions of the convolutional filter (a 4-dimensional tensor); and finally they get combined using the learned weights. We use the generated filters in our convolutional layers. Training filters that are a weighted combination of binary basis can be done using standard backpropagation. Once the OVSF basis and their associated weights are combined, the forward pass evaluation is done in the same way as a standard CNN. During backpropagation, the filters are decomposed in basis and weights pairs and these last ones get updated while maintaining the basis constant.

Network quantisation. Is not uncommon to find embedded devices without floating point units and that are therefore restricted to integer arithmetic. Even though training is generally done at 32-bit precision, existing works [18, 23, 14] have proven the suitability of using 8-bit quantisation in order to reduce model size. Motivated by this, we implemented 8-bit quantisation using Tensorflow's *fake-quantization* operation when training our KWS system. In Section 4 we provide accuracy results of all of our experiments with and without quantisation. A common criticism to systems the take advantage of 8-bit precision for their weights and activations is the lack of support from hardware manufacturers to efficiently operate with low precision arithmetic. The existing software kernels for ARM's Cortex microcontrollers, CMSIS-NN [21], makes it possible to take advantage of using 8-bit words resulting in higher throughputs and more energy efficient inference stages.

Generating OVSF codes. DBFs can be generated on-the-fly by combining OVSF codes. The choice of using OVSF is not arbitrary. These codes have the property of being able to be constructed recursively given a *seed* code in the form of a squared matrix:

$$H_n = \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix} \tag{2}$$

where H_n is the $L \times L$ Hadamard matrix, with $L = 2^n$, $n \in \mathbb{N}$. An example of constructing $H_{n=2}$ is shown below:

$$H_0 = \begin{bmatrix} 1 \end{bmatrix}, H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, H_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{3}$$

Each row of these matrices are OVSF codes. By design, they are orthogonal to each other, making it possible to use them as basis of $\mathbb{R}^L$. Therefore, each Hadamard matrix verifies the following property:

$$H_n H_n^\intercal = 2^n I_n \tag{4}$$

where I_n is the $n \times n$ identity matrix. These codes can be also generated as a recursive process in a binary tree [3] form making it possible to retrieve individual (leaf or node) codes without having to explicitly generate the entire tree. We use this approach in our implementation.

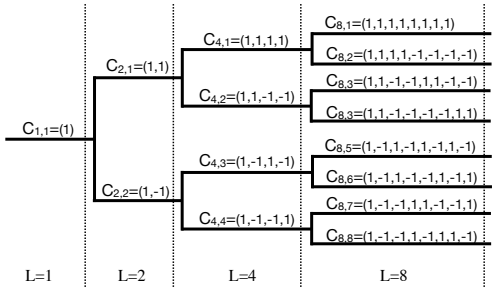

Figure 4: Code Tree for OVSF code generation. Note that the codes of the first three levels are exactly those in Eq. 3.

Mutual orthogonality and "recursiveness" were the two properties that made these codes an attractive option to fit our purpose of generating DBFs. In Section 4.4 we analyse the costs of generating OVSF codes and in Section 5 we highlight some limitations of these codes.

4. EVALUATION

4.1 Dataset and Method

The Google's Speech Commands dataset is comprised of 65k one-second long single-word audio clips from thousands of different people. There are 30 different keywords that can be classified into three groups: know expression, commands like "yes", "no" or "up", "down"; silence (i.e. a audio clip with only background noise); and unknown commands (e.g. "happy", "Sheila", "cat") for the remaining 20 keyword classes. Summarising, this results in 12 different classes.

We evaluate three configurations of BinaryCmd with a focus on reducing on-device memory footprint and number of OPs per inference pass while maintaining acceptable accuracy rates that outperform other models [32, 28, 5, 27] found in the recent literature with comparable number of OPs. The three configurations share the majority of network parameters and only differ in the number of filters, stride and ratio parameters used in our on-the-fly convolutional layers. The parameters used in our experiments are shown in Table 1. The spatial dimensions of the filters in the on-the-fly module kept fixed at: 4×8, 4×4 and 4×4.

Config	#Filters	Strides $[x, y]$	Ratios (ρ)
A	$[64, 8, 32]$	$[2, 2], [2, 2], [1, 1]$	$[1.0, 1.0, 1.0]$
B	$[64, 16, 16]$	$[2, 2], [2, 2], [1, 1]$	$[1.0, 0.5, 1.0]$
C	$[16, 16, 16]$	$[2, 2], [1, 1], [1, 1]$	$[1.0, 1.0, 1.0]$

Table 1: Parameters in BinaryCmd for each configuration. All parameters are given in triplets since there are three on-the-fly convolutional layers (see Figure 2).

This work has been implemented in TensorFlow [1] using as base the source code provided in [34]. We therefore maintained the use of Adam optimizer, batch size of 100, 30K learning iterations and initial learning rate of 5×10^{-4} and decreased by factors 0.2 and 0.5, after 10k iteration and 20k iterations respectively. We maintained the same dataset splitting ratios where 80% of the commands are used for training, for 10% validation and the remaining for testing.

4.2 Tuning BinaryCmd

Unlike in [34], where each architecture has been optimally trained after performing an exhaustive search for feature extraction and NN model hyperparameters, the work here presented only modifies the number of training steps from the default parameters provided in [34] source code, leaving room for more efficient training set-ups. Furthermore, our implementation applies 8-bit quantisation to all layers except the first convolutional layer, meaning that further reducing the model's memory footprint is also possible. In Table 2 we present the accuracy values reached by each of the evaluated configurations with and without 8-bit quantisation.

Configuration	32-bit			8-bit		
	Train	Val.	Test	Train	Val.	Test
BinaryCmd-A	95.64	92.24	92.83	94.47	90.79	91.40
BinaryCmd-B	96.22	92.69	93.05	93.97	91.11	91.21
BinaryCmd-C	96.37	92.76	92.86	94.97	90.53	90.97

Table 2: Accuracies of each set for each of the evaluated configurations with and without 8-bit quantisation.

4.3 Comparisons to State of the Art

We compare BinaryCmd against DS-CNN and all the baselines analysed in [34]. Our configurations explore the void space of 1M-3M OPs and 10kB-25kB. Our preliminary results, Figure 5, show the potential of our binary architecture: up to 59% model size and 67% number of OPs reduction at the expense of no more than 3.4% accuracy loss when compared to DS-CNN. All three of our configurations simultaneously achieve top *accuracy-to-size* (A2S) and *accuracy-to-OPs* (A2OPs) ratios meaning that BinaryCmd is a good first step towards the design of architecture capable of providing over 90% accuracy levels with minimal memory footprint and low computational costs.

We believe it is worth pointing out that while the DNN baseline model requires an order of magnitude less OPs during inference, the levels of accuracy reached by this system are considerable below the rest of the baselines. This is capture by the large gap between A2OPs and A2S ratios. We maintained it in our evaluation for the shake of completeness when comparing to [34].

Model	Acc.	Memory	OPs	A2S/A2OPs
DS-CNN [34]	**94.4%**	38.6kB	5.4M	2.45/17.48
CRNN [34]	94.0%	79.7kB	3.0M	1.18/31.33
GRU [34]	93.5%	78.8kB	3.8M	1.19/24.6
LSTM [34]	92.9%	79.5kB	3.9M	1.17/23.82
Basic LSTM [34]	92.0%	63.3kB	5.9M	1.45/15.59
CNN [34]	91.6%	79.0kB	5.0M	1.16/18.32
BinaryCmd-A	91.4%	24.5kB	**1.8M**	3.73/**50.78**
BinaryCmd-B	91.2%	22.8kB	2.3M	4.00/39.57
BinaryCmd-C	91.0%	**15.8kB**	2.6M	**5.76**/34.96
DNN [34]	84.6%	80.0kB	**0.16M**	1.05/**528.75**

Table 3: Comparison of three BinaryCmd configurations against DS-CNN, the current state of the art for KWS applications, and other baselines presented in [34].

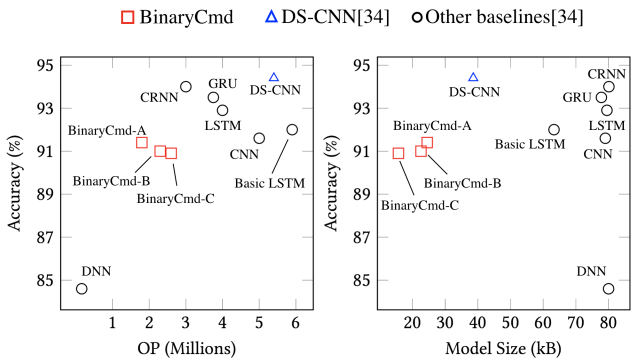

Figure 5: Results comparison against architectures in [34] for the category of small microcontrollers. DS-CNN has never been tuned below 38.6kB and 5.4M OPs. All other configurations in [34] result in larger and computationally more expensive models.

4.4 Filter Generation Overhead

In Sec. 3 we described the creation process of DBFs: a convolutional layer with N $K \times K$ filters and whose inputs have C channels, requires a 4-dimensional tensor of shape $dim_f = N \times C \times K \times K$. Such tensor can be generated by combining OVSF codes of that dimensions. Because of the recursive construction process of Hadamard matrices, there are a total of $L = NCKK$ codes of shape dim_f, as shown in Fig. 4. Given these premises, we could generate our DBF of dimensions dim_f, DBF_{dim_f}, by combining L codes of dimensions dim_f. The complexity of such operation grows quadratically with dim_f, i.e. $\mathcal{O}((NCKK)^2)$, and would significantly slowdown the inference process when generating the filters on-the-fly, as observable in Fig. 6 and Fig. 7.

The computational costs of generation process described (that we will refer to as *naive*), can be easily reduced by making a few small modifications. Instead of combining OVSF codes of dim_f we could split the generation process and generate N filters of dimensions $C \times K \times K$ separately and then concatenate them so we obtain a final tensor/filter of dimensions dim_f. This *fast* approach is $\mathcal{O}(N(WHC)^2)$. We can even better by splitting each filter of dimensions $C \times K \times K$ into C matrices of dimensions $K \times K$ and then concatenate them. This *faster* approach is $\mathcal{O}(NC(WH)^2)$. Figure 6 shows the computational costs for each technique at different filter dimensions. In Figure 7 we show the execu-

tion time measured on the ARM Cortex-M7 microcontroller when generating each of the filters.

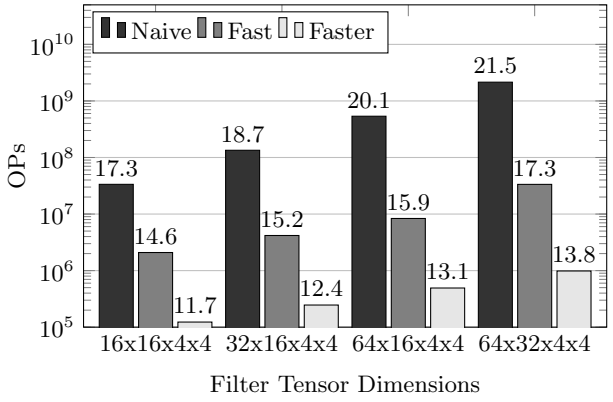

Figure 6: Number of OPs required for each filter generation method for different filter dimensions.

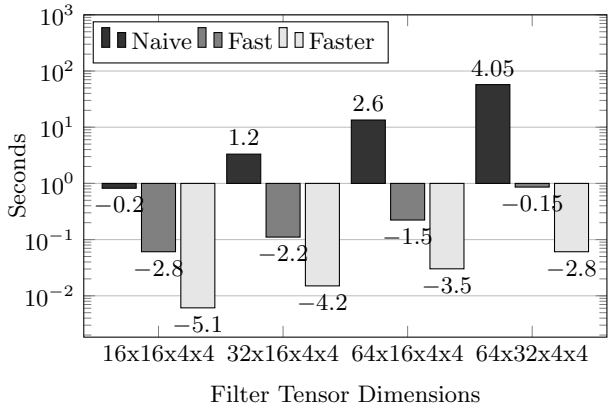

Figure 7: Execution time (seconds) required for each filter generation method for different filter dimensions on ARM Cortex-M7.

It is worth mentioning that the execution times shown in Figure 7 include both the filter generation itself (i.e. performing the linear combination of OVSF codes) and the creation of those OVSF codes as a recursive tree (see Figure 4). Because *faster* needs to generate OVSF codes of dimensions $K \times K$, we only need $(KK)^2$ bits to store all of them. In other works, by caching the OVSF codes we were able to further reduce the filter generation stage by ∼50%.

These measurements use a lightly optimised OVSF code generator written in C and using 32-bit weights. We believe further optimisations are possible, including the usage of 8-bit weights for filter generation.

Finally, in Table 4 we show the total number of operations required for our BinaryCmd configurations when generating filters on the fly. By using the *faster* filter generation technique and caching the OVSF codes, we are able to generate filters by increasing the inference time less than 14ms. We also estimated the execution time required for inference (excluding filter generation) in the M7.

Model	OPs Inference		Time Inference	
	Forward	Generation	Forward	Generation
BinaryCmd-A	1.77M	0.40M	32.67ms	13.6 ms
BinaryCmd-B	2.27M	0.40M	41.9ms	13.6 ms
BinaryCmd-C	2.63M	0.25M	48.6ms	10.3 ms

Table 4: Inference costs of running each BinaryCmd configuration on an ARM Cortex-M7 using 32-bit weights. Forward inference time is approximated using the baselines measurements in [21].

5. LIMITATIONS AND FUTURE WORK

Implicit in Section 4 and Section 3, the usage of OVSF codes comes with two limitations:

OVSF dimensionality. The nature of OVSF codes limits them to be of length $L = 2^l, l \in \mathbb{N}$. This means that commonly used filter dimensions such as 3×3 or 5×5 are not a possibility. This is a reason why our filters are 4×8 and 4×4. Note that a 4×4 filter has $1.78\times$ more parameters than a 3×3 filter. We think this is an important limitation of our approach.

Using OVSF codes efficiently. Being able to generate very cheaply a basis of $\mathbb{R}^L$, where $L = 2^l, l \in \mathbb{N}$, makes OVSF codes a powerful tool. However, we find that this usage of OVSF codes to build a basis comes with a no negligible drawback: the need to assign (learn) a magnitude to each of the basis, as shown in Eq. 1. To effectively reduce the number of *learnable* parameters we will need to set $\rho < 1$ and, as shown in [29], it is a viable solution for larger networks.

The next iteration of this work will explore two main paths that we believe would make OVSF codes more advantageous:

Less paging. Microncontrollers often have less than 1MB of memory and slightly more flash storage (e.g. our Nucleo-144 comes with an ARM Cortex-M7 has 512 kB of SRAM and 2MB of flash) severely limiting the complexity of the application that these devices can run. Although in a normal scenarios we would like the totality of our network to fit in memory, we might be interested in running a larger model to perform, for example, a more complex task. Running such application would require paging (i.e. temporally storing main memory data in flash memory) the network parameters (i.e. filters) and such I/O operations would considerably slowdown the inference process. By using OVSF codes, $\rho < 1$ and generating the filters as described in this work, we could reduce the time taxing access to flash memory without changing the structure of the network.

New architectures. In this work we have limited our study of DBFs in a "traditional" CNN. Giving the binary and deterministic nature of OVSF codes, we believe a more in-depth search for a better architectural design should be carried out. We are particularly interested in designing binary end to end architectures that can exploit the construction simplicity of Hadamard matrices. In addition, we used as input MFCC *hand-crafted* features as they have become standard for KWS applications. The performance of other features or the usage of the raw audio waveform should be evaluated.

6. CONCLUSION

We have applied a new type of convolutional layer to KWS and shown that they are capable of giving near start of the art accuracy levels even in architectures requiring a frac-

tion of model parameters and considerable less operations per inference pass compared to architectures of similar complexity. We make this possible by crafting convolutional filters on-the-fly using binary orthogonal codes that can be generated efficiently and can reduce the number of trainable parameters. We believe our approach offers a simple yet powerful technique that could evolve into an new network architecture capable of reducing memory accesses by relying on deterministic data structures, the OVSF codes.

Acknowledgements

This work was supported in part by the UK's Engineering and Physical Sciences Research Council (EPSRC).

7. REFERENCES

[1] Abadi, M., et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Adachi, F., et al. Wideband ds-cdma for next-generation mobile communications systems. *IEEE communications Magazine 36*, 9 (1998), 56–69.

[3] Adachi, F., Sawahashi, M., and Okawa, K. Tree-structured generation of orthogonal spreading codes with different lengths for forward link of ds-cdma mobile radio. *Electronics Letters 33*, 1 (Jan 1997), 27–28.

[4] Andreev, B. D., et al. Orthogonal code generator for 3g wireless transceivers. In *Proceedings of the 13th ACM Great Lakes Symposium on VLSI* (New York, NY, USA, 2003), GLSVLSI '03, ACM, pp. 229–232.

[5] Arik, S. Ö., et al. Convolutional recurrent neural networks for small-footprint keyword spotting. *CoRR abs/1703.05390* (2017).

[6] Bhattacharya, S., and Lane, N. D. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM* (New York, NY, USA, 2016), SenSys '16, ACM, pp. 176–189.

[7] Cavigelli, L., et al. Origami: A convolutional network accelerator. *CoRR abs/1512.04295* (2015).

[8] Chen, Yu-Hsin and others. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers* (2016), pp. 262–263.

[9] Chollet, F. Xception: Deep learning with depthwise separable convolutions. *CoRR abs/1610.02357* (2016).

[10] David Pearce, C. d. Speech processing, transmission and quality aspects (stq); distributed speech recognition; front-end feature extraction algorithm; compression algorithms.

[11] Du, Z., et al. Shidiannao: Shifting vision processing closer to the sensor. *SIGARCH Comput. Archit. News 43*, 3 (June 2015), 92–104.

[12] Fernández-Marqués, J., Vincent, W.-S. T., Bhattachara, S., and Lane, N. D. Binarycmd: Keyword spotting with deterministic binary basis. *SysML* (2018).

[13] Gokhale, V., Jin, J., Dundar, A., Martini, B., and Culurciello, E. A 240 g-ops/s mobile coprocessor for deep neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops* (June 2014), pp. 696–701.

[14] Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR abs/1510.00149* (2015).

[15] He, K., et al. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[16] Howard, A. G., et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR abs/1704.04861* (2017).

[17] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR abs/1502.03167* (2015).

[18] Jacob, B., et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR abs/1712.05877* (2017).

[19] Juefei-Xu, F., et al. Local binary convolutional neural networks. *CoRR abs/1608.06049* (2016).

[20] Kim, S., Kim, M., Shin, C., Lee, J., and Kim, Y. Efficient implementation of ovsf code generator for umts systems. In *2009 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing* (Aug 2009), pp. 483–486.

[21] Lai, L., Suda, N., and Chandra, V. CMSIS-NN: efficient neural network kernels for arm cortex-m cpus. *CoRR abs/1801.06601* (2018).

[22] Lane, N. D., Bhattacharya, S., Georgiev, P., Forlivesi, C., and Kawsar, F. An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices. In *Proceedings of the 2015 International Workshop on Internet of Things Towards Applications* (New York, NY, USA, 2015), IoT-App '15, ACM, pp. 7–12.

[23] Lane, N. D., Bhattacharya, S., Mathur, A., Georgiev, P., Forlivesi, C., and Kawsar, F. Squeezing deep learning into mobile and embedded devices. *IEEE Pervasive Computing 16*, 3 (2017), 82–88.

[24] McDanel, B., et al. Incomplete dot products for dynamic computation scaling in neural network inference. *CoRR abs/1710.07830* (2017).

[25] Purohit, G., Chaubey, V. K., Raju, K. S., and Reddy, P. V. Fpga based implementation and testing of ovsf code. In *2013 International Conference on Advanced Electronic Systems (ICAES)* (Sept 2013), pp. 88–92.

[26] Rintakoski, T., Kuulusa, M., and Nurmi, J. Hardware unit for ovsf/walsh/hadamard code generation [3g mobile communication applications]. In *2004 International Symposium on System-on-Chip, 2004. Proceedings.* (Nov 2004), pp. 143–145.

[27] Sun, M., et al. Max-pooling loss training of long short-term memory networks for small-footprint keyword spotting. *CoRR abs/1705.02411* (2017).

[28] Tara N. Sainath, C. P. Convolutional neural networks for small-footprint keyword spotting. *Sixteenth Annual Conference of the International Speech Communication Association* (2015).

[29] Tseng, V. W.-S., et al. Deterministic binary filters for convolutional neural networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18* (2018).

[30] Wang, Y., et al. Cnnpack: Packing convolutional neural networks in the frequency domain. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 253–261.

[31] Wang, Y., et al. Beyond filters: Compact feature map for portable deep model. In *Proceedings of the 34th International Conference on Machine Learning* (International Convention Centre, Sydney, Australia, 06–11 Aug 2017), D. Precup and Y. W. Teh, Eds., vol. 70 of *Proceedings of Machine Learning Research*, PMLR, pp. 3703–3711.

[32] Wang, Z., et al. Small-footprint keyword spotting using deep neural network and connectionist temporal classifier. *CoRR abs/1709.03665* (2017).

[33] Warden, P. Speech commands: A public dataset for single-word speech recognition.

[34] Zhang, Y., et al. Hello edge: Keyword spotting on microcontrollers. *CoRR abs/1711.07128* (2017).