High Performing Deep Learning Architectures under Resource Constrained Platforms

Javier Fernández-Marqués

Linacre College

Univeristy of Oxford

DPhil Transfer Report

November 12, 2018

Contents

Chapter 1

Research Proposal

1.1 Introduction

Neural Networks (NNs) are the preferred option when it comes to the design of robust artificial intelligence applications such as those relying on computer vision and voice recognition systems impacting areas like surveillance, health, entertainment or the automotive industry. As a parallel phenomenon to the rise in popularity of NNs in recent years, Internet of Things (IoT) devices such as smart speakers, thermostats or cameras, to name a few, have been embraced by our society with the promise of making our lives easier. With very few exceptions, the development paths of NNs and IoT have exclusively met in *the cloud*, where most of the data collected by these devices gets processed. These devices, often equipped with very constrain CPUs or microcontrollers (MCUs), are not a good fit for modern applications employing complex NNs systems primarily due to the reliance of these on millions of parameters.

The aim of this DPhil thesis is to investigate and develop novel deep learning architectures enabling the execution of NN-based applications in IoT devices. To this end, this thesis will investigate as well compression techniques that make NNs rely more on operations and less on parameters, effectively trading memory for compute. What's the optimal neural network architecture that, while maintaining accuracy levels and with relaxed latency constraints, energy consumption during inference is significantly reduced?

1.2 Motivation

There would be approximately 35 billion MCUs devices sold next year alone and are already flooding the places where we live, work and socialise (e.g. kitchens, supermarkets, factories,

hospitals, airports or gyms) [1]. Today, most of these devices are used in sensing and control applications. Despite their limited memory and compute resources (see Table 1.1), in recent years, MCUs have become the default target platform of a popular deep learning application: Keyword Spotting (KWS). 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. MCUs have become the preferred target platform for these applications primarily due to their good performance per watt profile.

The current state of the art for KWS (Zhang *et al.*, 2017) relies on depth-wise convolutions and results in a model size of 38.6 KB and requires 2.7 M MACs to analyse a one-second clip of audio. These networks are sufficiently small that do not present a challenge when it comes to deployment on constrained devices. Often, 8-bit quantisation is used in these setups as the majority of MCUs do not come with floating-point units, or FPUs. Without built-in support for floating-point these operations are still possible when relying on fix-point arithmetic. Despite their commercial success [2], KWS for personal assistant applications are the only family of DL applications that have made their way into compute-constrained platforms. Other applications, such as those based on vision (e.g. image classification, scene understanding, human pose estimation, etc) require at least two orders of magnitude (see Table 2.1) more compute and memory resources, compared to KWS systems, in order to deliver acceptable results.

Model	Core	Frequency	SRAM	Flash
STM32L151C8T6A	Cortex-M3	32 MHz	32 KB	64 KB
STM32F107RCT6TR	Cortex-M3	72 MHz	64 KB	250 KB
STM32L496VET6	Cortex-M4	80 MHz	320 KB	0.5 MB
ATSAME53N20A-AU	Cortex-M4	120 MHz	256 KB	1 MB
MKV58F1M0VCQ24	Cortex-M7	240 MHz	256 KB	1 MB
ATSAMV71N21B-CB	Cortex-M7	300 MHz	384 KB	2 MB
STM32H743VIT6	Cortex-M7	400MHz	1024KB	2MB

Table 1.1: Commercially available ARM Cortex-M based MCUs

For computer vision applications tasks such as image classification, the machine learning community has been primarily driven by the goal of designing new algorithms that could outperform the state of the art in terms of accuracy levels. This resulted in models with tens of millions of parameters and MACs in the order of billions to analyse a single low resolution 256×256 RGB image. While novel lightweight neural network architectural designs have been proposed, there's a trend of increasing the network's model size in order to surpass the previous state of the art. The current state of the art for image classification requires 21% more parameters than last year's. This trend, although not a problem for smartphones as nowadays have 4 GB

[1] The 2018 McClean Report: http://www.icinsights.com/data/articles/documents/1101.pdf
[2] https://machinelearning.apple.com/2017/10/01/hey-siri.html

of RAM or more, is a step in the wrong direction when it comes to the deployment of efficient architectures in very constrain setups, primarily due to the reliance in more parameters which exacerbated the memory and data movement challenges that MCUs face.

In addition to model size, latency might be one of the main concerns when designing machine learning applications that require real-time processing (e.g. autonomous driving, text to speech applications, etc). MCUs are suitable for real-time scenarios but not if those require the execution of applications operating in high-dimensional (HD) spaces. However, there is a wide range of applications without low-latency requirements that can be benefited from MCUs running deep NNs architectures. A few examples are: on-device crops monitoring, where changes happen at a slow pace, MCUs would be a good choice since they could execute NNs at very slow clock-speeds while on batteries potentially lasting these several weeks; satellite imaging, on-device processing would substantially reduce the amount of data streamed down to Earth, likely to extend satellite's life if running on batteries; medical imaging tests such as *diabetic retinopathy*[3]. For this task, that NNs have proven to be useful at (Yang *et al.*, 2017b; Zhou *et al.*, 2018), a portable device equipped with an MCU could execute such tests in locations with limited access to doctors or hospitals, making the patients wait a few minutes for the image to get processed.

1.3 Challenges and Opportunities

There is an urge to design efficient DNN architectures that would enable the deployment of high performing models in very constrained platforms (e.g. MCUs such ARM Cortex-M). However, current NNs architectures aren't designed to be deployed on these devices primarily due to their reliance on millions of parameters. In this section, we present the main challenges that the design of NNs needs to address when using MCUs as target platforms.

1.3.1 Memory Constraints

MCUs often exclusively rely on SRAM instead DRAM due to their lower energy requirements and much faster read/write operations. This design choice comes at lower KB:$ ratios as well as requiring more area per KB than DRAM does. As a consequence, SRAM in commercially available MCUs is often found in the 32 KB to 1024 KB range. This is one of the primary limitations when it comes to the deployment of complex DNNs-based applications. Existing compression techniques focus on reducing the on-device memory requirements of models by minimising the memory requirements of the network weights (whether this is by employing quantisation, encoding, sparse formats, or other techniques). However, much less attention has been paid to activations, which can easily require as much memory as the totality of the network parameters do. Due to this, there is a need for a holistic NN architecture design by which

[3]https://ai.googleblog.com/2016/11/deep-learning-for-detection-of-diabetic.html

the entire of the inference process is taken into account. In this way, having a very compact representation of the weights would matter as much as the memory footprint of the decoded filters or as much as the memory footprint of the activations that would be generated in every layer.

1.3.2 Energy Constraints

Any device that requires significant power faces a lot of barriers when it comes to its commercialisation. The advantages that smart products have to offer often get outweighed due to power-related inconvenient installation and maintenance processes, whether because they require permanent wiring or frequent docking to charge the battery. In MCUs, data movement is the primary source of energy consumption. Inter-device data movement is costly, even when relying on low power communication mechanisms such as BLE or ZigBee for close-range communications, with energy consumptions in the order of hundreds of milliwatts (Siekkinen *et al.*, 2012). On-device machine learning applications would require fewer communications to a master node (e.g. home router, phone, cloud) alleviating bandwidth bottlenecking issues and become a decisive design choice for ensuring user data privacy. In addition, and unlike displays and radios, CPUs and sensors could use considerably less power: real-time audio processing at 384 KHz requires between 10-25 mW (Hill *et al.*, 2018) using a Cortex-M4; low-resolution image sensors suitable for object tracking applications consume $277\mu W$ (Rusci *et al.*, 2017).

Intra-device data movement also presents a similar distance-energy relationship. Fetching values from DRAM requires over 128× more energy than reading that data from SRAM (Han *et al.*, 2016b). Applications having to rely on flash (i.e. paging) during inference would incur into significant energy costs as well as an increase in latency. An example of such scenario would be a NN which doesn't fit in RAM and therefore only a few layers can be in memory at a given time. The question this rises is *can we replace memory accesses and allocations with more computations in order to design a more energy efficient inference stage?*

1.3.3 Latency Constraints

Compression techniques for image classification has driven a lot of attention in recent years and some frameworks have been even able to reach 500× (Han *et al.*, 2016a) compression ratios for certain networks. However reducing model size alone is not a guarantee for faster inference stages (e.g. Iandola *et al.* (2016) achieves the same accuracy as AlexNet but, while it has 50x fewer parameters, it requires 33% more MACs to analyse a single 256×256 RGB image. A more recent example, MobileNetV2 (Sandler *et al.*, 2018) and AmoebaNet (Real *et al.*, 2018) present a similar behaviour, both require roughly the same number of MACs and offer the same accuracy on ImageNet but the later results in 33% higher CPU latency when evaluated on Google's Pixel-1

phone. These results evidence that not all OPs are created equal (Lai *et al.*, 2018b).

1.3.4 Accuracy

State of the art compression techniques exhibit little to know accuracy degradation in NNs that are overparameterized. When it comes to compressing smaller networks, such as those in the range of 5-10M parameters down to sub-megabyte levels, the literature is very scarce. Even works showcasing architectures relying on aggressive quantisation, i.e. ternary or binary networks, generally do not result in smaller models without sacrificing accuracy (Lin *et al.*, 2017), whether because of the large amount of parameters needed or because the activations of every layer require full-precision representations.

The landscape of NNs resulting in models of < 1 MB for complex applications, such as those using computer vision, hasn't been explored. As a consequence, it is difficult to estimate what's the best architecture or compression technique given the memory constraints of MCUs that maximises that accuracy of the system. In very constrained setups, such as those with 2KB or RAM (Kumar *et al.*, 2017), traditional machine learning techniques outperform to NN-based systems. A combination of both traditional ML and NNs could be a viable solution to ease high dependence of parameters that applications solely relying on NNs have.

1.4 Proposed Research

The focus of this research is the development of novel neural network architectures with minimal energy consumption but capable of achieving high accuracy rates in complex tasks such as image classification when latency constraints are relaxed. Energy consumption, modelled as the amount of data movement required during inference, would be reduced by limiting the number of model parameters that describe the network and therefore the number of accesses to RAM or flash needed (e.g.μSD card) to retrieve those values.

Concretely, in this thesis we are interested in the set of functions that could be used to generate the elements that describe NNs on-the-fly, consequently reducing the impact of parameter fetching in terms of energy consumption. Function Networks (FNs) are NNs whose elements (e.g. filters, activations) are represented or constructed as a combination of *parameters* and *functions*. They differ from regular NNs in the sense that the filters aren't stored directly as part of the model but need to be reconstructed at runtime. The prototypical example of FNs is the work of Ha *et al.* (2016) in which HyperNetworks were introduced. HyperNetworks are NNs that use another NN to construct the filters given a low-dimensional embedding of the filter. Here the *parameters* are the filter embeddings and the *function* is the NN that inflates the filter given the embedding. FNs enable, by increasing the number of operations during inference, the execution of NNs that otherwise would not fit on the device. The design space for FNs hasn't been explored

for constrained platforms and it requires addressing the challenges presented in Section 1.3.

This research will primarily be evaluated on image classification tasks since it's a complex problem with multiple applications that would greatly benefit from being capable of running on MCUs. Our aim is to design application-agnostic architectures and compression techniques, but because image classification systems often do not include other popular types of layers such as LSTMs (Hochreiter and Schmidhuber, 1997) or up-scaling layers, this DPhil will also look, to a lower extent, on to how RNNs and generative networks in applications such as GANs (Goodfellow *et al.*, 2014) and super-resolution, could benefit from using FNs for reasons that would be introduced in 1.4.2 and 1.4.3 respectively.

1.4.1 Approaches to Classification

Classification is arguably one of the most common types of applications when it comes to the analysis of images or sounds. It is also one of the most studied topics in the machine learning community and has experienced a rapid development in recent years. In particular, CNNs have proven to be the most suitable family of NN architectures for this task. We focus on CNN-based architectures and how they could be deployed onto MCUs by making us of FNs.

1.4.1.1 Completed Project

In standard CNNs, filters are learnt end-to-end via back-propagation and stored as part of the model. In certain architectures these can account for a sizeable portion of the total model size, limiting the deployability of these networks in constrained devices. In Tseng *et al.* (2018) we presented an approach to CNNs that learns weighting coefficients of predefined orthogonal binary bases instead of the conventional approach of learning directly the convolutional filters. We generate the filters as a linear combination of orthogonal binary codes that can be generated very efficiently in real time. We employ a set of orthogonal variable spreading factor (OVSF[4]) codes that can be recursively generated on-the-fly in order to reconstruct the convolutional filters for image classification CNNs. Our work lies in the intersection between compression techniques and novel architectural designs. In Fernandez-Marques *et al.* (2018) we explored the suitability of this approach for KWS applications. In addition, we analysed the costs of the filter generation, presented alternative solutions and evaluated them on a Cortex-M7.

1.4.1.2 Current Project

The current project addresses the question: *What are the most suitable set of functions to generate the filters?*. This question, raises another one and, as it will be argued later, both need to be

[4]OVSF codes were introduced for 3G communication systems as channelizations codes aiming to increase system capacity in multi-user access scenarios (Adachi *et al.*, 1997)

answered jointly: *How can we learn the parameters that define these functions?*

The Choice of Functions

For the task of reconstructing the filters in a NN layer given a low-dimensional vector, the choice of functions to perform this task is of great importance and multiple approaches are possible. Because NNs are often regarded as universal function approximators, these could be used in order to generate the filters of our network (effectively resulting in a HyperNetwork). One of the aims of FNs is to rely on those decoding schemes that are efficient, lightweight and with enough representation capabilities to any filter suitable for a specific task, e.g. image classification. Beyond NNs, which are known to rely on many parameters, these are two directions worth exploring:

- **In-Training Filter Shaping:** Filters in NNs generally live in very HD spaces. This property provides enough flexibility during the training process and results in NNs that, although different in terms of parameters, are capable of performing equally well for a given task. This can be used to our advantage by applying constraints on how the filters are learnt and therefore make them suitable to be reconstructed very cheaply. Results from our initial exploration of this idea on a two-layer NN trained on MNIST are shown in Figure 1.2.
- **Recursion**: Relying on recursive functions to reconstruct the filter could greatly reduce the dependence on model parameters. In addition, if it combines deterministically generated elements the filter reconstruction process could be even more compact. The challenge in the design of recursively generated filters is to add flexibility to the generation process. *How can we design a recursive algorithm capable of generating very different types of filters?* This could be framed as a few-to-many RNN-based decoding scheme in which the first input are the *parameters* and the final output after n iterations would be the reconstructed filter. However, due to the reliance on fully-connected layers in RNNs, these would need to

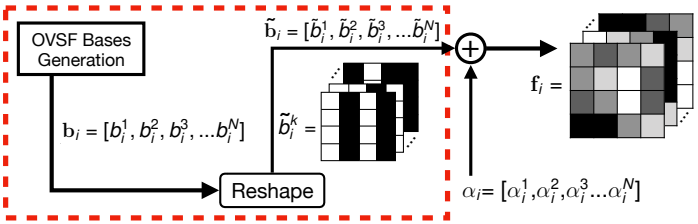


Figure 1.1: Filter generation by combining on-the-fly binary codes with a set of weights learnt. Elements inside the dashed red rectangle are deterministic and therefore can be generated on-the-fly without the need of parameters learnt during training.

be simplified first by using functions that require fewer parameters. Another possibility would be to rely on the notion of fractals and use recursion as a multi-resolution filter design stage.

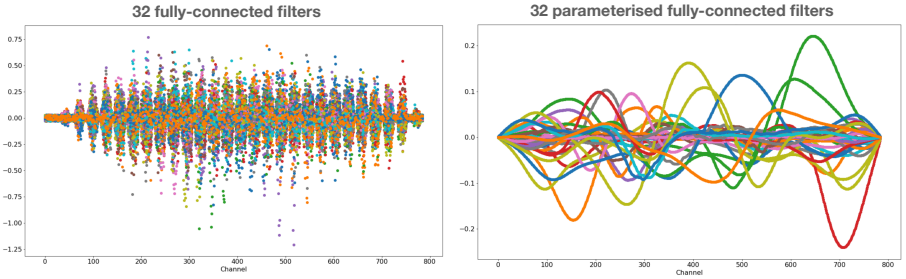


Figure 1.2: On the left, a scatter plot of the filters learnt for the input layer in a standard NN are shown. As expected, values are follow a normal distribution centred at the origin. On the right, the filters of the input layer of the NN (same architecture and hyper-parameters) that get smoothed with a 1-dimensional Gaussian kernel for each filter individually and across the channel dimension. This *smoothing*, performed during training, ensures little in the value of channel-contiguous weights. The filters on the right can be generated using Fourier Series with very few components (16 or less for the example here shown). On the other hand, trying to learn a function capable of generating the filters on the left would be much more challenging. The filters on the right result in a 3% drop in accuracy compared that achieved by the NN using the filters on the left.

Training FNs

Depending on what family of functions are used to reconstruct the filters of a NN at run time, the learning process of such functions or their parameters would vary. Choosing differentiable functions or combinations of these would make the training process suitable for the standard backpropagation algorithm. For any other arbitrary set of functions, where differentiability may not be guaranteed *continue*. This is similar to AutoML approaches such as Neural Architecture Search (Zoph and Le, 2017) in which the optimal network architecture is automatically found given a number of constraints such as model size, number of OPs etc. AutoML techniques applied to FNs differ from these approaches in the sense that it wouldn't be an architecture search (which could be seen as an automated hyperparameter tuning) but a function exploration approach potentially leading to a novel type of NN layer.

1.4.1.3 Future Project

The natural steps to follow when using FNs would be to first load a certain number of parameters, reconstruct the filters given *parameters-function* pairs and then process the input to the network as any standard NN would do. Although this approach might be suitable for some NNs, it might become a limiting factor when executing larger models. As the size of activations generally increases with the network depth, requiring the entire filter to be allocated in memory might result in having to page to disk/flash a portion of the activations tensor. This would greatly increase the energy consumption and latency of the application. The *need for explicit filter reconstruction* is the question that would be addressed in this project.

1.4.2 Approaches to RNNs

RNNs are known to depend on fully-connected layers resulting on little parameter re-use and the need to remember large sequences of events, which translates into a substantial memory usage. In addition to high memory usage during inference, training RNNs is memory intensive. To make this process tractable, the standard training algorithm sets an upper limit to the number of states the network can remember, effectively truncating back-propagation through time (TBPTT) (Werbos, 1990). We believe Function Networks could be a good option to alleviate the memory requirements of RNNs when using TBPTT enabling the training of sequences with dependencies spanning longer time-scales.

1.4.3 Approaches to Generative Networks

Applications that make use of generative approaches, such as NNs for image super-resolution (SR) tasks, require at least 100× more MACs than state of the art image classification NNs. SR networks take as input a low-resolution image and output a high-resolution version of the input. The challenge is to perform this upscaling without introducing image artefacts. In addition to the dramatic increase in operations needed during inference, the size of the activations also gets affected in a similar manner, especially when the input images are upscaled by 4× factors or more. This translates into high run-time RAM usage and limiting the deployability of these systems. Factorisation or compression of activations which, as mentioned in Sec 1.3.1, has not received a lot of attention, would be addressed in this project.

Chapter 2

Literature Review

Neural Networks (NNs) are the preferred option when it comes to the design of robust artificial intelligence (AI) applications such as those relying on computer vision (He *et al.*, 2015; Ren *et al.*, 2015; Zhang *et al.*, 2018b; Goodfellow *et al.*, 2014) and voice recognition systems (Abdel-Hamid *et al.*, 2014; Zhang *et al.*, 2016; van den Oord *et al.*, 2016; Zhang *et al.*, 2017) impacting areas like surveillance, health, entertainment or the automotive industry. Despite their rise in popularity in recent years, and in particular since the success of AlexNet (Krizhevsky *et al.*, 2012) in winning the ImageNet (Deng *et al.*, 2009) competition, the history of NNs has been a roller coaster of enthusiasm and disillusionment since they were first proposed by McCulloch and Pitts (1943). Since their inception and for the following 70 years, four mayor advances paved the way for the development of complex and deeper architectures we use nowadays, namely: the work of Rosenblatt (1957), were the perceptron algorithm for binary classification was first proposed and designed for image recognition tasks; the work of Minsky and Papert (1969), which identifies the technical limitations that perceptrons networks would face with the hardware available at the time, leading to the first AI *winter*; the Back-propagation algorithm (Rumelhart *et al.*, 1986), which helped in making it easier to train NNs; and the work of Hochreiter (1991), which gave insights on why it is hard to train NNs, specifically he identified the *vanishing gradient* problem that results when designing deep NNs. In addition to these and other advancements from the research community, the confluence of two other factors enabled the popularisation of NNs as a viable generic algorithm for a variety of machine learning problems. These two factors are the today's availability of both compute capability (mainly in the form of Graphical Processing Units, GPUs) and large datasets. These two factors enabled training large NNs efficiently and well enough to generalise to unseen examples.

Although in this chapter we will focus on feed-forward networks designed for image classification tasks many of the contents here presented are utilised in greater or or lesser extent in

other applications. The remaining of this chapter is organised as follows: first, we will introduce the most popular datasets used for image classification a long with the most popular DNNs architectural designs and the reasons behind their design choices; second, a detailed overview of the state of the art of techniques for network compression, from traditional methods like SVD to more recent ones such as DeepCompression (Han *et al.*, 2016a) or RedCNN (Wang *et al.*, 2017); third, and overview of the main advantages of low-precision networks and recent works in the area; finally, several of the reference designs for NN accelerators will be presented.

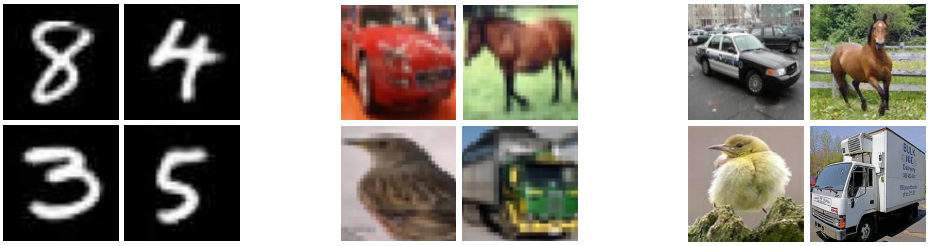
2.1 Neural Network Design Overview

Modern NNs are designed as a stack of operations, often referred as *layers*, through which the input (e.g. an image, a clip of audio) is sequentially transformed by the operation defined in each layer. In this section we will present some of the popular architectural designs for image classification as well as the three datasets that are commonly used to compare these architectures.

2.1.1 Datasets

Prior to describing how network architectures have evolved throughout the years, we should take a look at the different datasets that the research community often employs to compare different architectures to each other. A few images of these datasets are shown n Figure 2.1. From simple to more challenging, the three main datasets are:

- MNIST: This dataset (Lecun *et al.*, 1998) consists of 28×28 grayscale images of handwritten digits, i.e. numbers 0 to 9, with 60k training images and 10k for test. Despite its simplicity, this dataset is often still used by the research community. Most of modern DNNs can reach over 99.5% accuracy on this dataset.
- CIFAR-10: This dataset (Krizhevsky *et al.*, 2009) consist of 32×32 RGB images split into ten classes with 6k images per class. There is a total of 50k training images and 10k test images. The classes are common objects and animals, e.g., cat, dog, ship, airplane, etc. This dataset is more challenging than MNIST and state of the art networks would be capable of reaching +95% accuracy (e.g. Xie *et al.* (2017) achieves 96.42%)
- ImageNet: This dataset (Deng *et al.*, 2009) consists of 256×256 RGB images of a thousand categories. It is divided into training and test sets with 1.28 million and 50k images, respectively. This dataset is by far the most challenging among the three. The current state of the art (Hu *et al.*, 2018) achieves Top-5 4.47% and Top-1 18.68% errors.



(a) MNIST (b) CIFAR-10 (c) ImageNet

Figure 2.1: Sample images from three popular datasets for image classification.

2.1.2 Popular Architectures

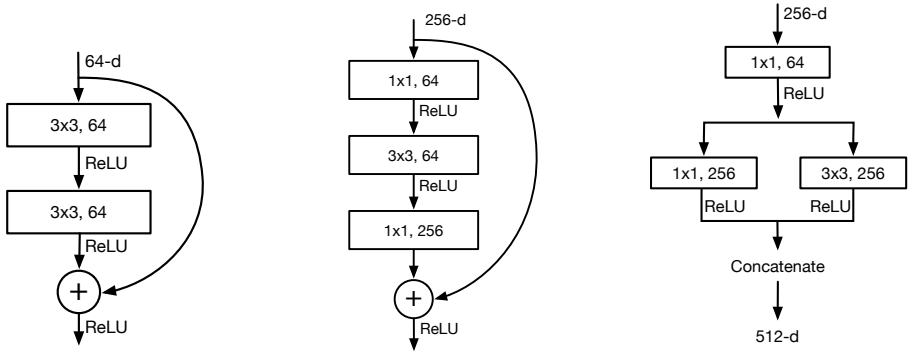
The vast majority of DNNs are designed by combining the same set of layers[1]. These fundamental building blocks are: convolutional, fully-connected, pooling, activation, batch-normalisation layers. Here we present some of the networks that resulted in significant improvement over the state of the art with respect to their predecessors for the task of image classification. An overview of the characteristics of each network architecture and their performance on the ImageNet dataset is shown in Table 2.1.

AlexNet. This architecture is the first successful implementation of a deep convolutional neural network (CNN) on the challenging task of classifying ImageNet images. It won the 2012 ImageNet competition by over 25% compared to the second placed. Up until then, the best performing systems relied in hand-crafted image features such as Local Binary Patterns (LBP) (Ojala *et al.*, 1994), Histogram of Oriented Gradients (HOG) (Dalal and Triggs, 2005) or Fisher Vectors (Perronnin and Dance, 2007), among others.

ResNet. This architecture proposed by He *et al.* (2015) won the ImageNet'15 competition using a network with 152 layers. The main contribution of this work is the introduction *residual* blocks as a way of mitigating the *vanishing gradients* problem that arises when back propagating through many layers. Residual blocks characterise by adding the input of the block to its output, $\mathcal{H}(x) := \mathcal{F}(x) + x$, which adds stability during training. The residual blocks admit different designs, two of the most widely use are shown in Figure 2.2a and Figure 2.2b. The *bottleneck layer* squeezes the input tensor into a lower dimensional space where is less compute intensive to perform the spatial convolution (e.g. using 3×3 filters) and then it is expanded back to the original dimensions so the residual can be added.

[1]For a detailed description of the different types of layers and what their purpose is, we refer the reader to Goodfellow *et al.* (2016)

NASNet. In recent years, the research community has shown interest in developing automated frameworks for neural network discovery. The main contribution behind NASNet (Zoph *et al.*, 2018), which stands for *Neural Architecture Search Network*, is the transferability of the search space for architectural search from one domain (e.g. a particular dataset) to another. By being capable of finding high performing architectures on the relatively small CIFAR-10 dataset and transfer them to larger networks, it is the current state of the art for image classification under ImageNet as well as for object detection on the COCO (Lin *et al.*, 2014) dataset. Networks that rely on automatic architecture network discovery techniques, such as those originally presented in Zoph and Le (2017), often result in fewer parameters and operation required during inference as compared to those architectures designed by humans. This architectures often introduce a higher degree of graph irregularity which could translate into overheads during inference.



(a) Basic residual Layer (b) Bottleneck Layer (c) Fire Module

Figure 2.2: Different layer designs making use of residual connections. Layers (a) and (b) were introduced with the ResNet architecture, and layer (c) in SqueezeNet.

The following architectures did not surpassed the previous state of the art in terms of image classification accuracy but they are capable of reaching competitive accuracy levels while significantly reducing the number of parameters in the model. In the case of MobileNets, the number of MAC operations needed during inference is greatly reduced compared to standard CNN architectures.

SqueezeNet. Inspired by ResNet, and in particular by its *bottleneck layer* (see Figure 2.2b), Iandola *et al.* (2016) focused on reducing the computational cost of AlexNet while maintaining its accuracy levels. This architecture is comprised of a total of nine *Fire Modules* (see Figure 2.2c) containing three convolutional layers inside each. Their architectural design, that relies heavily in 3×3 and 1×1 convolutions, in combination with compression frameworks such as DeepCompression (Han *et al.*, 2016a) results in a model size of $0.47MB$ compared to the original $240MB$ of AlexNet. This is a $510\times$ model size reduction while maintaining the same accuracy on ImageNet.

Architecture	#Layers	#params	MACs	Top-1 Acc.	Top-5 Acc.
AlexNet	8	61 M	0.72 B	57.2 %	80.3 %
VGG-16	16	138 M	15.5 B	71.5 %	90.1 %
ResNet-50	50	25.5 M	3.91 B	75.3 %	92.7 %
ResNet-152	152	60.2 M	11.3 B	77.0 %	93.3 %
DenseNet-264	264	∼33 M	∼15 B	79.2 %	94.7 %
ResNeXt-101 (64 x 4d)*	101	83.6 M	31.5 B	80.9 %	95.6 %
NASNet-A(6 @ 4032)*	-	88.9 M	23.8 B	82.7 %	96.2 %
SqueezeNet	29	1.25 M	0.84 B	57.2 %	80.3 %
0.75 MobileNet	28	2.6 M	0.33 B	68.4 %	-
ChannelNet-v2	28	2.7 M	-	69.5 %	-
ChannelNet-v1	28	3.7 M	0.41 B	70.5 %	-
1.0 MobileNet	28	4.2 M	0.57 B	70.6 %	89.9 %
ShuffleNet (1.5×)	18	-	0.29 B	71.5 %	-
MobileNetV2	24	3.4 M	0.30 B	71.8 %	91.0 %
ShuffleNet (2×)	18	∼5 M	0.53 B	73.7 %	-
NASNet-A(4 @ 1056)	-	5.3 M	0.56 B	74.0 %	91.6 %
MNasNet-92	-	4.4 M	0.39 B	74.8 %	92.1 %

Table 2.1: Performance on ImageNet classification. Architectures on the top half of the table focus on achieving high accuracy levels, while those in the bottom half share the same aim while focusing on reducing the model size and compute costs. *Uses as input larger image patches, instead of the conventional 224×224.

Despite being 50× smaller in terms of model size in its uncompressed form (i.e. without using DeepCompression), it results in 33% in energy consumption (Yang *et al.*, 2017a) during inference compared to the original AlexNet.

MobileNets. Convolutions are important operations for vision applications but, despite their simplicity, they are very compute intensive. MobileNets (Howard *et al.*, 2017) replace conventional convolutional layers with a two-layered convolutional module, often referred to as *depth-wise* convolutional layer. These layers factorise a standard convolution with a depth-wise convolution followed by a 1×1 convolution to combine the outputs of the previous. Suppose a standard convolutional layer has as input a tensor with C_{in} channels and outputs a tensor with C_{out} channels. Using depth-wise convolutions translates into convolving each input channel with a different $k \times k$ filter (where k is generally 3) and then concatenate C_{out} linear combinations,

by means of 1×1 convolutions. Although not mathematically equivalent to the convolution operation, depth-wise separable convolutional layers are capable of significantly reducing the number of operations while still offering acceptable feature extraction capabilities. More recently, MobileNetV2 (Sandler *et al.*, 2018) further reduced the size and MACs of the original network by designing depth-wise convolutional layers as the core of block a bottleneck residual block.

Table 2.1 shows the performance on ImageNet of various network architectures. Other architectures not described in this section are included for completeness. These include: VGG (Simonyan and Zisserman, 2014) which significantly improved over AlexNet and introduced the use of smaller 3×3 filters; DenseNet (Huang *et al.*, 2017), which became the state of the art by extensively using residual connections across the network as oppose to standard ResNets where these were sparse; ResNeXt (Xie *et al.*, 2017) which relies on wider residual layers that learn multiple smaller filters whose activations get aggregated before the residual is added up; ShuffleNet (Zhang *et al.*, 2018a), in which activations are slitted into groups and different filters are applied to each of them; and MNasNet (Tan *et al.*, 2018), which relies on reinforcement learning techniques to find the most suitable architecture given memory and compute constraints.

2.2 Compressing Neural Networks

Traditionally, the machine learning community has been primarily driven by the goal of designing new algorithms that could outperform the state of the art in terms of accuracy levels. With the rise in popularity of data-driven techniques, e.g. DNNs, such fixation on accuracy resulted in algorithms with tens of millions of parameters. Deploying these models in the wild becomes a challenge primarily because of two reasons: first, transferring the model parameters (i.e. the weights) might require significant time and bandwidth. For example, VGG-16 weights result in over 500 MB in file size. Secondly, due to the large amount of OPs needed for inference, without the presence of a GPU these DNNs become unusable in applications where real-time or close to real-time performance is required. In addition to this, having to rely on a GPU severely limits the employability of DNNs-based systems primarily due to energy constrains (e.g. a NVIDIA GTX 1060, a mid-range GPU, requires 120W). In this section we will present an overview of the different approaches presented in recent years aiming to reduce both model size and compute costs during inference.

2.2.1 Layer Design Approaches

The choice of layer designs (g.e. number of filters, size of the filters) and the overall network architecture directly impacts both model size and inference costs. In 2.1.2 we presented some of the widely use layer designs that ease the training of deeper networks (i.e. residual connections), encourage weight sharing (i.e. bottleneck layer), rely on small-sized filters (i.e. 1×1 and 3×3)

and factorisation of the traditional convolutional layer design (i.e. depth-wise convolutions). Here we present other layer designs that, although not widely implemented as a general approach towards the design of lightweight architectures, prove the be advantageous in certain contexts.

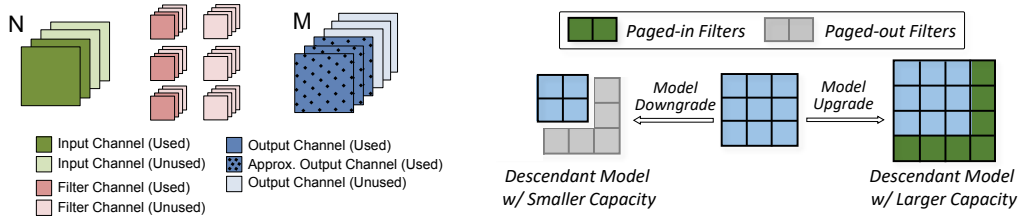
Compared to a standard convolutional layer, depth-wise convolutions significantly reduce the number of operations and parameters. Concretely, the number of operations is reduced by a factor of $r = 1/c_{out} + 1/d_k^2$, where c_{out} is the number of output channels and d_k is the filter dimensions and becomes

$$d_k \times d_k \times c_{in} \times d_f \times d_f + c_{in} \times c_{out} \times d_f \times d_f \tag{2.1}$$

where $d_f \times d_f$ are the dimensions of the input tensor. The second term in Eq. 2.1 represents the dense combination the of feature maps from the first convolution step (i.e. depth-wise). This term is dominated by the number of channels in both input and output tensors. To alleviate this issue, which in deeper layers of the network can result into 1M OPs or more, Gao *et al.* (2018) propose to replace the dense point-wise operations with sparse ones. One approach would be to dived the outputs of the depth-wise convolution into *isolated* g groups and then perform the point-wise convolution by only fusing the feature maps within a group. This effectively splits the feature maps of individually transformed channels and is know as *group convolutions*. Such strict constrain results in a severe degradation in accuracy as observed by Zhang *et al.* (2018a). ChannelNets address this issue by learning a *fusing layer* that combines the features in every group with those in contiguous groups. This is efficiently implemented by convolving with a stride $s = g$ each of the feature maps resulting from the channel-wise convolutions. ChannelNets results in similar performance to MobileNets while relying on fewer parameters and OPs for inference. Results are shown in Figure 2.1.

The design space for NNs has grown considerably in recent years. With the aim of reducing model size Ha *et al.* (2016) introduced HyperNetworks. A hypernetwork is a NN that generates the weights of another NN. In their work, the usage of hypernetworks is framed as a tool to enforce weight sharing across layers resulting in substantially more compact models at the cost of an increase in OPs during both training and inference. Given a convolutional layer L_i in standard CNN architecture, the $c_{in} \times k \times k \times c_{out}$ tensor representing the filters, f_i, can be generated by using an embedding (i.e. a lower-dimensional representation) of the filter, z_i, and a hypernetwork. A hypernetwork is a two-layered NN that takes as input the embedding z_i and outputs an approximation of f_i, $\hat{f}_i \approx f_i$. All the weight embeddings $z = \{z_1, z_2, ..., z_N\}$ have the same dimensions making it possible to share the first layer of the hypernetwork across all layers in the architecture. The second layer depends on the number of output channels c_{out} of each layers, therefore sharing these parameters is limited as in general c_{out} increases with i, the network's depth. HyperNetworks have achieved relatively little success probably due to the impact on latency of the filter generation process. Because of this, applications that require

real-time performance wouldn't chose a hypernetowrk despite its compact model representation. In complex real-world applications such as autonomous driving or activity recognition, image classification usually is not the first task that the system has to accomplish. Other computer vision techniques such as object detection may be required first. We know that standard CNNs apply convolutions uniformly in the feature space regardless of its content, which translates into high computational costs. This problem scales with the input's dimensions and becomes a challenge in real-time applications that requires the processing of high-resolution images. Sparse Block's Networks (Ren *et al.*, 2018) propose a mask-guided approach by which only regions of interest in the input will be transformed by a given convolutional layer. To this end, SBNet modifies the standard bottleneck layer (see Figure 2.2b) by adding a *gather* layer at the top and a *scatter* layer before the performing the residual addition. The use of these binary masks could also be categorised as an *attention mechanism* for faster inference. Other popular object detection architectures such as Faster-RCNN (Ren *et al.*, 2015) rely on similar ideas but are often implemented as a two-step process: first, detect potential locations where objects are present, these are often referred to as *object proposals*; and then, evaluate a image classifying network using as input each of the image patches where an object is believed to be present. SBNet's main motivation is to exploit the sparse nature on LiDAR-based 3D readings on roads by using as mask map information and results in over 2× speed-up and better detections rates than methods densely scanning the entire scene. By relying on a two-step approach, where PSPNet (Zhao *et al.*, 2017) generated object proposals, further speed-ups are achieved.



(a) Effect of using the top 50% of the input channels and filters as proposed in IDP.

(b) Multi-capacity filter size switching in NestDNN.

Figure 2.3: Two layer design approaches that enable the execution of a NN inference stage capable of adjusting to the requirements of the application in terms of compute, memory and latency. These diagrams are extracted from McDanel *et al.* (2017b) and Fang *et al.* (2018).

While PSPNets exploit the sparsity of the scene as a measure to mask out *uninteresting* regions, in other situations using saliency to decide which portions of the image to analyse might be more challenging. As a general approach to dynamically adapt the latency of a given NNs, the work of McDanel *et al.* (2017b) provides a simple mechanism to do so. Incomplete Dot Products (ICP) enables dynamic adjustment of the number of feature maps that a layer would produce. In other words, given a compute budget, ICP adjust the number of filters used in

each layer that maximise accuracy while maintaining the overall number of OPs within the budget. This is possible with the introduction of a profile $\gamma = \{\gamma_1, \gamma_2, ..., \gamma_N\}$ of non-increasing coefficients that weights each input channel differently. This would result in a model that has learnt to rely more on the first channels of the inputs instead of relying on all channels equally. ICP's principles are architecture agnostic and therefore can be equally used in fully-connected, convolutional and depth-wise convolutional layers. Although ICP was evaluated by McDanel *et al.* (2017b) on relatively simple networks and only MNIST and CIFAR-10 datasets, they present a simple yet effective way of trading accuracy for compute without having to modify the network architecture. In Figure 2.3a it is illustrated the impact of using the top 50% channels of the activations in a given convolutional layer. A more holistic approach was recently proposed in NestDNN (Fang *et al.*, 2018). This framework enables the deployment of multi-capacity concurrent NN-based applications on devices depending on memory and compute budget of these at a given time. The motivation is that real world applications (e.g. scene understanding) require the execution of multiple *sub-applications* (e.g. object detection, image classification) and it would be intractable to design networks of different complexity for the wide variety of devices available. NestDNN enables the execution of NNs in a multi-resolution fashion depending on the application requirements and the latency budget of the system. A diagram is shown in Figure 2.3b illustrating this idea.

2.2.2 Post-Training Approaches

The design space of DNNs allows great flexibility on how we stack layers primarily due to high dimensional (HD) nature of the operations performed in those. Despite this flexibility, it has been proven to be a challenge is to design compact architectures that achieve high accuracy levels. To address this, several techniques aim to relax the choice of architecture design following a two-step approach: first, design an architecture that is capable of achieving the desired accuracy levels; and then, once the model has been trained, discard those elements in the network that are redundant while maintaining (or marginally decreasing) the accuracy levels. Because this compression techniques use a pre-trained network, we call them *post-training* compression techniques.

Network pruning (Hassibi and Stork, 1993; Ström, 1997; Han *et al.*, 2015) and factorisation methods (Kim *et al.*, 2016; Denton *et al.*, 2014) have been widely used to compress CNN-based models. DeepCompression (Han *et al.*, 2016a) could be seen as the first compression framework that aggregates classical compression methods, i.e. pruning, clustering and encoding, capable of yielding significant model size reduction rates with no or small drop in accuracy. The DeepCompression framework is split into three stages: the first stage performs standard weight pruning by which connections with magnitude below a threshold value, this reduces weights by 10× when making use of compressed sparse row (CSR) format; next, weights of similar values are clustered together and a code book is generated in order to decode the weights during

inference, this results in an additional 15× to 20× model reduction; the final stage makes use of Huffman encoding (Huffman, 1952) to encode both weight values and look up table indices. DeepCompression is capable of reducing the model size of networks such as AlexNet or VGG-16 by 35× and 49× respectively incurring in a decrease of Top-1 accuracy on Imagenet of less than 1%. In addition, this framework was successfully used in SqueezeNet resulting in an architecture offering AlexNet performance with a 500× smaller model size. A similar framework was proposed by Wang *et al.* (2016), here the processing was done in the frequency domain and relying on discrete cosine transforms. Despite the high compression rates achievable by these methods in certain networks, it is unclear from the literature how does the decoding process of the weights impact latency. In particular, the impact of the look-up tables to decode the Huffman codes and the use of the code book to retrieve the original value of each clustered weight.

Model	Evaluation	Original	SVD	XNOR	Pruning	Perforation	CNNpack	RedCNN
AlexNet	r_{c_1}	1×	5×	32×	35×	1.7×	39×	5.12×
Filters (232 *MB*)	r_{c_2}	1×	1×	32×	1×	1×	1×	2.45×
Maps (2.5*MB*)	r_s	1×	2×	64×	-	2×	25×	4.31×
Multiplications	*top*-1 *err*	41.8%	44.0%	56.8%	42.7%	44.7%	41.6%	42.1%
(7.24×10^8)	*top*-5 *err*	19.2%	20.5%	31.8%	19.7%	-	19.2%	19.3%
VGGNet-16	r_{c_1}	1×	-	-	49×	1.7×	46×	6.87×
Filters (572 *MB*)	r_{c_2}	1×	-	-	1×	1×	1×	3.07×
Maps (52 *MB*)	r_s	1×	-	-	3.5×	1.9×	9.4×	9.63×
Multiplications	*top*-1 *err*	28.5%	-	-	31.1%	31.0%	29.7%	29.3%
(1.54×10^{10})	*top*-5 *err*	9.9%	-	-	10.9%	-	10.4%	10.2%
ResNet-50	r_{c_1}	1×	-	-	-	-	12.2×	4.35×
Filters (97 *MB*)	r_{c_2}	1×	-	-	-	-	1×	3.71×
Maps (40 *MB*)	r_s	1×	-	-	-	-	4×	5.82×
Multiplications	*top*-1 *err*	24.6%	-	-	-	-	-	25.7%
(5.82×10^9)	*top*-5 *err*	7.7%	-	-	-	-	7.8%	8.2%

Figure 2.4: An overall comparison of some of the state of the art post training compression techniques. Here *Pruning* refers to the DeepCompression framework. R_{c_1} stands for compression ratio for convolutional filters, R_{c_2} for compression ration of feature maps, and r_s for speed-up ratio. Results for both Pruning and CNNPack shown have 8-bit quantisation, those for RedCNN are kept at 32-bit. RedCNN is the only compression technique that reduces the memory foot-print of activations without having to rely on heavy quantisation. This table is extracted from Wang *et al.* (2017).

To alleviate the additional processing required to *undo* or decode the weights, other approaches such a RedCNN (Wang *et al.*, 2017) focused on discarding redundant filters instead of redundant individual weights. The intuition behind this work is that the most compact representation of the weights in a layer is that that results in uncorrelated activations coming from different filters. An optimal projection of the inputs should thus not only remove redundancy between feature maps by preserving orthogonality among them, but also preserve the discriminability of the filters that generated them. This framework takes as input a trained network and iteratively modifies the filters in each layer by minimising the projection between activation maps until convergence. Unlike previous approaches, RedCNN does result in a compression in activations which translates

into a reduction of run-time memory during inference. This and the previous techniques described in this section are compared in Figure 2.4 for three popular architectures on the ImageNet dataset. Similarly to Wang *et al.* (2017), the work of Louizos *et al.* (2017) proposes a Bayesian approach to prune filters (instead of weights) and assign per-layer quantisation profiles. This work builds on top previous approaches: the work of Kingma *et al.* (2015), which was the first successful application of Bayesian techniques to inferring the posterior probability of the weights efficiently by making use of the *reparameterisation trick*; and more recently the work of Molchanov *et al.* (2017) which enabled learning high dropout rates (previous works were limited to maximum dropout rates of $r = 0.5$) for each filter in the network leading to high sparsity rates.

All the methodologies here presented focus on compressing large networks such as AlexNet or VGG, or are only evaluated on relatively simple datasets such as MNIST, SVHN or CIFAR-10. We could argue that part of the success of these techniques comes from the fact that the starting point in the compressing pipeline is a heavily overparametersied network. Little attention has been paid to compressing networks that are by design small. Network designs that lie in this category would be those with number of parameters in the range of two to five million, as is the case of MobileNets and other networks shown in the lower part of Table 2.1. AutoML for Model Compression, or AMC (He and Han, 2018), which leverages reinforcement learning (RL) for efficient search given scenario-specific policies. These policies vary between *latency-critical*, *resource-constrained* and *quality-critical* applications. This framework, in addition to be evaluated on large networks such as VGG-16 and ResNets of different depth, it's also evaluated on MobileNet. It is evidenced from the reported results that compressing compact network architectures is considerably more challenging than doing so with an overparameterised net. AMC achieves speed-ups for MobileNet of around 1.5× and 2× when deployed on a NVIDIA TItan XP GPU or Google's Pixel-1 phone respectively with no accuracy degradation.

2.2.3 Using Deterministic Elements

In recent months a few NN-based algorithms that introduce deterministic elements in the network have been presented in top-rated venues. They all share the same motivation for this decision: reduce model size. In the next few lines we will be presenting a few of these works a long with their advantages and limitations.

In convolutional neural networks for image classification, a large portion of the model size account for the convolutional filters of each layer. Reducing the number of filters could damage the performance of the network. In order to do so, Juefei-Xu *et al.* (2017) propose to replace standard convolutional layers with a two-step convolutional layer. The first step convolves the input with a set of deterministically generated filters that match the design of hand-crafted local binary patterns (LBP) traditionally used as a texture descriptor (Ojala *et al.*, 1994; Dalal and Triggs, 2005). This approach receives the name *Local Binary Convolutional Neural Networks*, or

LBCNNs. This first step doesn't require any learnable parameter. The second step applies a linear combination of the activations of the first by learning a single $1 \times l$ array of scalars, where l is the number of LBP filters used in the first stage. These scalars are learnt via back-propagation and achieves competitive results in datasets of the level of complexity of CIFAR-10. A similar idea was presented in DFCNet (Qiu *et al.*, 2018) in which the convolutional filters are constructed as a dense combination of a set of deterministically generated Fourier Bessels basis. This approach is capable of maintaining the accuracy levels of the baseline networks on MNIST and LFW (Huang *et al.*, 2007) while reducing the number of operations required during inference by 3×. Both DFCNet and LBCNNs replace standard convolutional layers in which filters are learnt entirely during back-propagation with a factored convolutional layers in which of term is deterministic and the other has to be learnt. Similarly, in WSNet (Jin *et al.*, 2018) the filters are not learnt directly. However, unlike the previous two methods that rely on factorisation, here the deterministic element in the layer is the sampling process by which each individual filter is generated. WSNet learns a single $A \times B \times N$ filter that is deterministically sample to generated C_{out} filters of shape $C_{in} \times K \times K$, where $K < A$ and $K < B$. The sampling induces weight sharing in both spatial and channel dimensions and, by making use of integral image it saves computing redundant operations. Because the sampling process is deterministic, WSNets can be trained with standard back-propagation. It results in 9× model size reduction on ResNet-50 with a drop of 1% in accuracy on CIFAR-10.

The techniques presented in this section offer an interesting new approach to designing compact architectures for CNN-based image classification applications. However, as evidenced from the results, the reliance on deterministic elements comes at the cost of a drop in how flexible these networks are to unseen images. In other words, it limits the generalisation capabilities of the networks. None of the approaches here presented are capable of providing significant model size reductions while maintaining acceptable accuracy levels on more challenging tasks such as those of the level of complexity as the ImageNet dataset. Similarly to what was stated in 1.4.1.2, the choice of which deterministic elements to use is not trivial.

2.3 Low Precision Networks

Complementary to the compression techniques proposed by the machine learning community in recent years, the design of low-precision networks has attracted considerable interest. This section presents several of the most popular approaches making use of aggressive quantisation with the aim of reducing model size and latency. We will begin by introducing binary networks, then ternary networks, hybrid networks that combine both binary and not-binary weights and finally networks that rely on integer-only arithmetic.

In binary networks, parameters are represented with only one bit, reducing the model size by 32×. Expectation BackPropagation (EBP) (Soudry *et al.*, 2014) proposes a variational

Bayes method for training deterministic Multilayer Neural Networks, using binary weights and activations. This and a similar approach (Esser *et al.*, 2015) give great speed and energy efficiency improvements. However, the improvement is still limited as binarised parameters were only used for inference. Many other proposed binary networks suffer from the problem of not having enough representational power for complex computer vision tasks, e.g. BNN (Courbariaux and Bengio, 2016), DeepIoT (Yao *et al.*, 2017), eBNN (McDanel *et al.*, 2017a) are unable to support the complex ImageNet dataset. On the other hand, BinaryConnect (Courbariaux *et al.*, 2015) and Binary-Weight-Networks (BWN) (Rastegari *et al.*, 2016), were capable of achieving state of the art results on CIFAR-10 and ImageNet datasets respectively. XNORNet (Rastegari *et al.*, 2016) employed the *sign* function as the non-linearity to achieve binary activations in addition to binary parameters. With this approach, 32-bit precision operations in convolutional layers could be replaced with binary XNOR and POPCOUNT operations in supported hardware resulting in a 58× speed-up. BinaryConnect extends the probabilistic concept of EBP, it trains a DNN with binary weights during forward and backward propagation, done by putting a threshold for real-valued weights. It is a common practice in the design of binary networks to not learn the binary weights directly; instead, full-precision weights are maintained and learned during the training as *proxies* for the binary weights. Because the *sign* function is non-differentiable, binary NNs often employ the Straight-Through-Estimator (STE) (Bengio *et al.*, 2013) in order to enable gradient flow during back-propagation and consequently update the full-precision *proxies*. The literature of binary networks presents a variety of ad-hoc techniques with the promise of improving training stability and generalisation. Some of these are: the choice of optimiser, clipping weights and/or gradients and the choice momentum for batch-normalisation layers. We refer the interested reader to Anonymous (2019) for a systematic study of these techniques.

Despite the success of binary networks to match the performance of networks such as AlexNet or VGG-16, they haven't been able to maintain the rapid development seen in regular full-precision networks in terms of accuracy. A series of works have explored the increase in weight resolution as a measure to improve overall accuracy. In TernaryNet (Zhu *et al.*, 2016) quantised parameters to one and a half bits and represented weights using $\{-1, 0, +1\}$. Having zero allows efficient hardware implementations when kernels are sparse. The usage of ternary networks has also been used as a measure to dramatically (over 99.5% in some cases) reduce the number of multiplications needed during inference (Tschannen *et al.*, 2018). Other works, such as ABCNet (Lin *et al.*, 2017) make use of an ensemble of binary convolutional layers as a measure to maintain the accuracy levels of 32-bit precision networks. By doing this, ABCNet achieves ImageNet accuracies of Top-1 68.4% and Top-5 of 88.2% for ResNet-34, a drop of 4.9% and 3.1% compared to the full-precision version of the network.

In addition to the challenges of training binary neural networks and the apparent limitation they have to achieve state of the art results, the lack of efficient software packages that take advantage of binary operations is another reason why binary networks are not widely used

beyond research. For this reason, part of the machine learning community believe that 8-bit networks is the right balance between data precision, accuracy and efficient compute. Jacob *et al.* (2018) present a training framework by which networks can be trained end-to-end using integer-arithmetic only with minimal loss in accuracy compared to their full-precision counterpart networks. This is possible by quantising during training to 8-bits both weights and activations while enabling the bias terms (which represent a negligible portion of the total parameters) as 32-bit integers. An efficient implementation, part of Arm's CMSIS library (Lai *et al.*, 2018a), that takes advantage of 8-bit quantised networks on 32-bit capable hardware enables the execution of NNs on Arm Cortex-M based systems that support SIMD instructions.

Chapter 3

Completed Work

This section presents a novel technique to model compression that enables the generation of convolutional filters on-the-fly by making use of a set of codes that can be generated deterministically and form a base for the filter. This technique has been validated for keyword spotting applications resulting in models of 15.8kB while still achieving over 91% accuracy on Google's Speech Commands Dataset (Warden, 2017). This work was first presented in SysML-18 and in the International Workshop on Embedded and Mobile Deep Learning co-located with ACM MobiSys-18. It has been also validated for image classification tasks and published in IJCAI-18.

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

Javier Fernández-Marqués[†], Vincent W.-S. Tseng[‡], Sourav Bhattacharya[*], 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].

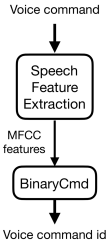


Figure 1: System arch.

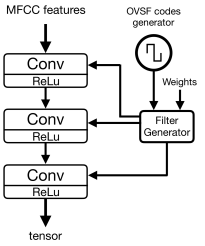


Figure 2: BinaryCmd core.

***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:

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

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.

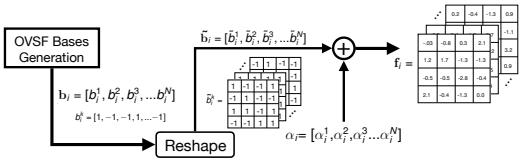


Figure 3: From OVSF codes to DBFs. Each code b_i^k is first reshaped to match the final filter dimensions, becoming $\tilde{b}_i^k$. Then, the reshaped codes in $\tilde{b}_i$ are combined using the learnt weights α_i.

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:

$$H_n = \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix} \quad (2)$$

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:

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

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:

$$H_n H_n^\intercal = 2^n I_n \quad (4)$$

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.

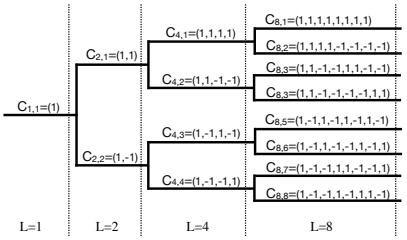


Figure 4: Code Tree for OVSF code generation. Note that the codes of the first three levels are exactly those in Eq. 3.

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].

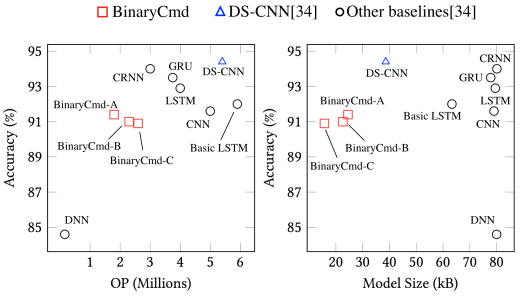


Figure 5: Results comparison against architectures in [34] for the category of small microcontrollers. DS-CNN has never been tuned below 38.6kB and 5.4M OPs. All other configurations in [34] result in larger and computationally more expensive models.

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.

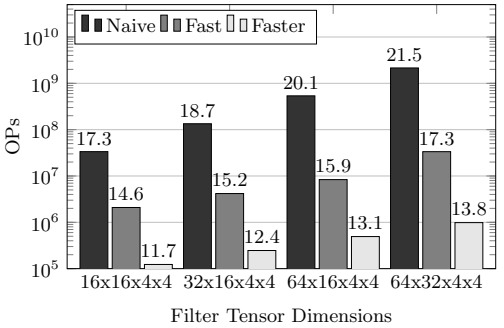


Figure 6: Number of OPs required for each filter generation method for different filter dimensions.

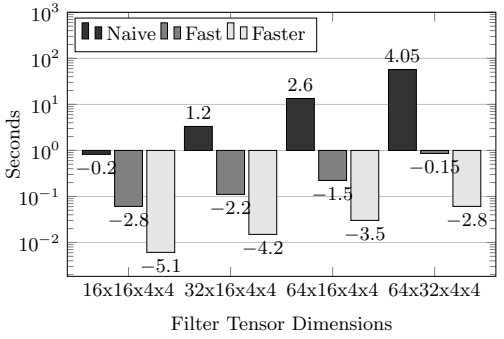


Figure 7: Execution time (seconds) required for each filter generation method for different filter dimensions on ARM Cortex-M7.

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).

Deterministic Binary Filters for Convolutional Neural Networks

Vincent W.-S. Tseng[‡], **Sourav Bhattacharya**[†], **Javier Fernández-Marqués**[*], **Milad Alizadeh**[*], **Catherine Tong**[*] and **Nicholas D. Lane**[†*]

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

Abstract

We propose *Deterministic Binary Filters*, an approach to Convolutional Neural Networks that learns weighting coefficients of predefined orthogonal binary basis instead of the conventional approach of learning directly the convolutional filters. This approach results in architectures offering significantly fewer parameters ($4\times$ to $16\times$) and smaller model sizes (up to $32\times$ due to the use of binary rather than floating point precision). We show our deterministic filter design can be integrated into well-known network architectures (such as ResNet and SqueezeNet) with as little as 2% loss of accuracy under datasets like CIFAR-10. Under ImageNet, they are used in an architectures $3\times$ smaller compared to sub-megabyte binary networks while reaching comparable accuracy levels.

1 Introduction

Since the success of AlexNet [Krizhevsky *et al.*, 2012], convolutional neural networks (CNN) have become the preferred option for computer vision related tasks. While traditionally the research community has been fixated on goals such as model generalization and accuracy in detriment of model size. Recently, multiple approaches attempt to reduce model's on-device memory footprint while still maintaining high levels of accuracy. Such approaches could be subdivided into two main categories: new network compression techniques and novel layer architectural designs. Multiple network compression techniques [Wang *et al.*, 2016; Han *et al.*, 2015; Frosst and Hinton, 2017] have been proposed as post-training stages. In addition, several approaches, [Courbariaux and Bengio, 2016; Rastegari *et al.*, 2016], proved the suitability of aggressive data quantisation techniques as a way to reduce the memory and compute requirements during inference by replacing 32-bit parameters with 8-bit and/or binary values. Examples of novel layer design are [He *et al.*, 2015; Howard *et al.*, 2017] aiming all of them to offer alternative approaches to the traditional convolutional layers, being their advantages more noticeable when operating with very high-dimensional feature maps in deeper layers of the network.

In this work, we present *Deterministic Binary Filters* (DBF), an approach to Convolutional Neural Networks that learns weighting coefficients of predefined orthogonal binary bases instead of the conventional approach of learning directly the convolutional filters. We generate the filters as a linear combination of orthogonal binary codes that can be generated very efficiently on real time. We achieve this by using a popular orthogonal binary code generator that has been extensively studied for over two decades in the wireless community and widely used in mobile cellular systems. Our work lies in the intersection between the previously mentioned categories: compression techniques and novel architectural designs.

Our approach results in $4\times$ to $16\times$ reduction in the number of convolutional layer parameters to be learned, and more than $32\times$ savings in model size due to the use of binary weights instead of floating point parameters. Unlike most of the network compression techniques, our method allows learning compressed models directly. We demonstrate our deterministic filter design can be integrated into well-known network architectures (such as ResNet [He *et al.*, 2015] and SqueezeNet [Iandola *et al.*, 2016]) with as little as 2% loss of accuracy under CIFAR-10. With fewer parameters such models are less prone to over-fitting and can be potentially trained with significantly less compute operations and memory needs. DBFs can also offer improved efficiency for inference on microprocessors and embedded devices. Experiments show the suitability of DBFs and their usage in networks with model size up to $3\times$ smaller compared to already optimized binary networks while offering comparable accuracy levels for datasets like ImageNet, which has 1000 classes.

We believe DBFs are a first step in the development of efficient architectures relying less on large amounts of trainable parameters and more on deterministic data structures. Models with such characteristics would be more suitable for applications on resource constrained embedded devices requiring high accuracy rates but minimal compute complexity. In this work we offer the following contributions:

- A new module that performs convolution filters using a weighted combination of orthogonal binary bases that offers significantly reductions on the amount of *learnable* parameters required for the network.
- We find that such a module is able to offer nearly equal accuracy levels under common network architectures

and datasets, making it a viable model choice, and providing insights into filter design moving forward.

- The ability to trade-off model size for low-complexity compute, this is a unique characteristic important for low-memory platforms.
- The number of parameters needed to be updated during training can be greatly reduced since we only need to update the weights and not the entire filter, leading to a faster training and inferences stages.

2 Related Work

Our study of DBFs for CNNs touch upon the following areas of deep neural network research.

Novel Filter Design. The design of filters within convolutional networks is critical to the effectiveness of such networks in discriminative tasks, and have significant downstream implications for efficiency (e.g., requirements for memory and compute). Earlier work [Jarrett *et al.*, 2009] showed random kernels with no learning achieving decent performance in Caltech-101. Similarly, other works [Saxe *et al.*, 2011; Pinto *et al.*, 2009] make use of random filters to show that in addition to the convolutional filters, the network architecture plays a fundamental role in the learning process. Moreover, [Saxe *et al.*, 2011] argued that some performance of certain state-of-the-art methods can be attributed to the their architecture alone. All of these demonstrate the ability for filters despite not being learned from data during training. More closely to our filter design is the LBCNN (Local Binary CNN) module [Juefei-Xu *et al.*, 2017] that use pre-defined sparse local binary filters that also do not need to be updated during training. However, a critical difference in our design is our ability to generate DBFs on the fly through efficient algorithms that enable significantly smaller model sizes as light-weight compute operations replaces in-memory overhead (critical for embedded and mobile scenarios with low memory footprints). LBCNN modules also cannot directly replace conventional filters in existing architectures as requires a two stage approximation of convolution. This may not be applicable to all architectural designs. In comparison, our DBFs can be trivially applied to common architectures.

Binary Networks. Adoption of network architecture designs that include binary filters and weights are also a promising direction. Under this approach parameters are represented with only one bit, reducing the model size by $32\times$. Although such methods offer small model size and inference efficiency they do not necessarily reduce the amount of parameters as offered by our deterministic binary filters. Expectation BackPropagation (EBP) [Soudry *et al.*, 2014] proposes a variational Bayes method for training deterministic Multilayer Neural Networks, using binary weights and activations. This and a similar approach [Esser *et al.*, 2015] give great speed and energy efficiency improvements. However, the improvement is still limited as binarised parameters were only used for inference. Many other proposed binary networks suffer from the problem of not having enough representational power for complex computer vision tasks, e.g. BNN [Courbariaux and Bengio, 2016], DeepIoT [Yao *et al.*, 2017], eBNN [McDanel *et al.*, 2017] are unable to support the complex ImageNet dataset seen in our results.

Network Architecture Optimization. Many attempts towards optimizing network architectures for more efficient training, inference and parameter exist. One direction in quantization involves taking a pre-trained model and normalizing its weights to a certain range. This is done in [Vanhoucke *et al.*, 2011] which uses an 8 bits quantization to store activations and weights. Other works such as [Han *et al.*, 2015] and [Wang *et al.*, 2016] are a conglomerate of multiple clustering, quantisation and word encoding techniques that have been proven to work well in large architectures such as AlexNet and ResNet. In addition to compressing weights in neural networks, researchers have also been exploring more light-weight architectures: SqueezeNet uses 1×1 filters in combination with 3×3 filters, reducing the model to $50\times$ smaller than AlexNet while maintaining the same accuracy levels; *bottleneck* layers, introduced in ResNet, that aims to reduce the number operations and parameters of convolutional layers by reducing the number of channels of the input tensor using 1×1 filters; or *MobileNets* [Howard *et al.*, 2017] that make use of depthwise convolutional layers and result in lightweight networks suitable for embedded vision applications. SparseSep [Bhattacharya and Lane, 2016b] adds sparsification to both convolutional and dense layers resulting in highly compact model representations.

Deep Learning for embedded platforms. Energy efficiency and low computational complexity are two major requirements that algorithms must fulfil when deploying them in memory and compute restricted platforms [Lane *et al.*, 2015] and they become a major concern when considering the commercialization of such applications. Embedded deep learning applications, often instantiated as wearables, exist for a diverse range applications including vision [Mathur *et al.*, 2017; Suleiman *et al.*, 2017], audio [Fernandez-Marques *et al.*, 2018; Georgiev *et al.*, 2017] and activity recognition [Tahavori *et al.*, 2017; Bhattacharya and Lane, 2016a]. We refer the interested reader to [Lane *et al.*, 2017] and [Sze *et al.*, 2017] for a deeper evaluation of the challenges associated with this area of research.

3 Deterministic Binary Filters

In this section, we introduce a novel approach of performing convolution operations and present an efficient algorithm for training the parameters. Specifically, we design a convolution layer, where all filter kernels are generated using a linear superposition of a predefined bases set of binary orthogonal vectors. Fewer number of tunable parameters generates a model with a smaller memory footprint, which is particularly suitable for deployment on resource-constrained wearable or IoT devices. The properties of the binary codes used to generate the filters also makes it possible to implement convolu-

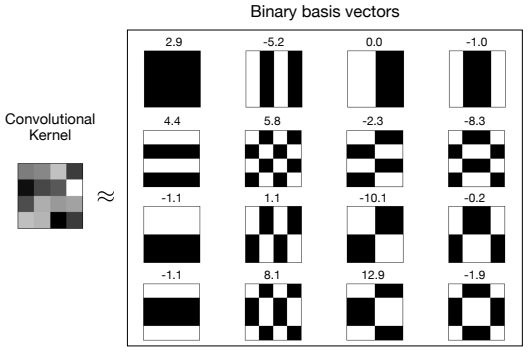


Figure 1: Overview of representing a convolution kernel using a set of binary mutually orthogonal basis vectors. The convolutional kernel (on the left) can be represent accurately using linear superposition of all the binary vectors (patches) presented on the right. The coefficient or strength of a binary vector used in the reconstruction is given on top of the patches.

tions without explicitly having the filters allocated in RAM, therefore allowing efficient runtime on embedded devices.

In this work we employ *orthogonal variable spreading factor* (OVSF) codes to generate filters for the convolution layer. The presented technique can also be applied to the fully connected layer parameters, however, we only focus on convolution layers in this work. In the following three sub-sections we present the main intuition behind representing convolution kernels with OVSF codes, present technique to efficiently use the codes to generate kernels and lastly describe the feature-maps generation process.

3.1 OVSF Codes: Overview

A point $\boldsymbol{x} \in \mathbb{R}^N$ can be represented by the span of a set of N mutually orthogonal set of vectors or bases $\{\boldsymbol{B_i}\}_{i=1}^{N}$ ($\boldsymbol{B_i} \in \mathbb{R}^N$), where $\forall i, j$, and $i \neq j$, $\boldsymbol{B_i} \perp \boldsymbol{B_j}$. In other words, any point in $\mathbb{R}^N$ can be presented as a linear combination of the basis vectors as:

$$\boldsymbol{x} = \sum_{i=1}^{N} \alpha_i \cdot \boldsymbol{B_i}, \tag{1}$$

where, α_i is the coefficient or strength for the i^{th} basis vector.

Without a loss of generality, we can apply the same linear superposition strategy when generating the hypermatrix or tensor that would become our convolutional filter. To gain computational or representational benefits, we can enforce certain properties on the bases set. For instance, in this work we only consider binary basis vectors, i.e., $\boldsymbol{B_i} \in \{-1, +1\}^N, \forall i$. For illustration, in Figure 1 we present a scenario when a random filter of dimension 4×4 is represented accurately by a set of 16 binary and mutually orthogonal basis vectors. The coefficients for individual binary vectors are presented on the top of each patches. Note that, for illustration purpose we only consider $2D$ filters, whereas in practice the filters used in convolution layers are $3D$.

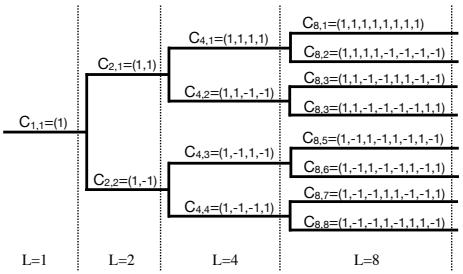


Figure 2: Code Tree for OVSF Code Generation

To achieve this filter representation, we require a techniques for efficiently generating the basis vector set. Specifically, given the dimensionality of a filter, the bases generation technique should output all the orthogonal binary vectors for the convolution parameter space. This bases generator should have the following properties: (i) capable of generating all the bases for any space regardless of its dimensionality, (ii) employs a deterministic procedure, i.e., for a given dimension, the basis vectors set remains invariant, (iii) being efficient such that it can be used in real-time applications on embedded devices.

OVSF Codes

For the purpose of generating convolutional kernels, while meeting the above mentioned conditions, we use the algorithm presented in [Adachi *et al.*, 1998]. The OVSF codes have been extensively studied in the wireless community [Andreev *et al.*, 2003; Rintakoski *et al.*, 2004; Kim *et al.*, 2009; Purohit *et al.*, 2013] and widely used in W-CDMA based 3G[1] mobile cellular systems to provide multi-user network access. Their simplicity and efficiency on-silicon implementation makes them suitable for real-time implementation on power-constrained devices. OVSF codes are binary $\{-1, +1\}$, orthogonal to each other and of length $L = 2^l, l \in \mathbb{N}$. Figure 2 shows OVSF bases at different l values generated as a recursive process in a binary tree [Adachi *et al.*, 1997].

3.2 Filter Generation Process

Unlike in standard CNNs, our architecture does not learn convolutional filters directly. Instead, it learns the coefficient for the basis vectors needed to generate the convolutional filters. Note that the dimension of the OVSF code is the same as the N filters, which is $W \times H \times C$, where W and H are the width and height of the filters[2], and C is the number of channels.

For any given code length L, there are L different OVSF codes (as observable in Figure 2). Therefore, to generate a filter of dimensions $dim = W \times H \times C$, our generator could output at most dim different codes that would form a basis of $\mathbb{R}^{dim}$. Intuitively, by combining all OVSF codes of

[1] 3GPP TS 25.213, v 3.0.0, Spreading and modulation (FDD), Oct. 1999

[2] In this work we only consider square filters with dimension of the form 2^l.

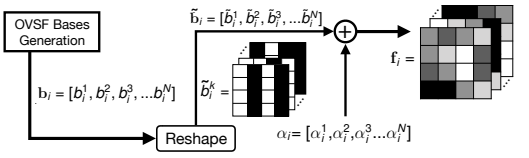


Figure 3: From OVSF codes to DBFs. Each code b_i^k is first reshaped to match the final filter dimensions, becoming $\tilde{b}_i^k$. Then, the reshaped codes in $\tilde{b}_i$ are combined using the weights α_i.

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:

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

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. ϵ is the difference between the the approximated DBF, f_k', and the real filter, f_k. 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.

Filter Generation Stages

During training, the set of weights $\{\alpha\}_{i=1}^N$ that pre-multiply each of the OVSF codes are learnt via backpropagation [LeCun *et al.*, 1989]. At inference time, the generated filter $\boldsymbol{f_k'}$ can be treated as any other standard floating-valued convolutional filter. The filter generation process using OVSF bases and the learned weights is didactically illustrated in Figure 3. This process involves: generation of $\lfloor \rho \cdot l \rfloor$ OVSF codes of length $l = W \times H \times C$; reshape each code in order to match the shape of the filter; and combine them using the learnt weights $\{\alpha\}_{i=1}^N$.

The Importance of Ratios

One of the main focus of our evaluation is the study of how ρ impacts on the performance of our models. This parameter, that can be independently set for each convolutional layer in the network, is directly proportional to the number of learnable parameters N in a given layer. As an example, when $\rho = 0.5$, the filter would be generated using half of the OVSF codes and, therefore, our network would only require to learn half to the weights.

OVSF Limitations

By design, OVSF codes must 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. We overcome this limitation by only using a portion of the elements in each OVSF code. For example, in order to construct a $3 \times 3 \times 1 \times 1$ filter, we would first generate OVSF codes of length $4 \times 4 \times 1 \times 1$; then keep 9 out of the 16 dimensions; and proceed with the reshape and combination stages as shown in Figure 3. This approach results in pseudo-OVSF codes that are no longer orthogonal to each other. We call these codes *square-pseudo* OVSF, sp-OVSF for brevity. In Section 4, we empirically show that the generated filters perform well even though the codes used are no longer mutually orthogonal.

Algorithm 1 Training with DBFs

Input: A minibatch of inputs labels and labels($\boldsymbol{X}$, $\boldsymbol{Y}$), a dictionary of orthogonal binary bases $\{B_1, B_2, ..., B_k\}$ for each convolution filter and learning rate η.
Output: Updated coefficients $\{\alpha_1^{t+1}, \alpha_2^{t+1}, ..., \alpha_k^{t+1}\}$ for each of the binary filters.
for *l* = 1 to L **do**
  1. Forward Propagation:
  $\{\boldsymbol{B}_1, \boldsymbol{B}_2, \ldots, \boldsymbol{B}_k\} \leftarrow$ OVSF(n, k)
  $\boldsymbol{f}^t \leftarrow \alpha_1^t \boldsymbol{B}_1 + \alpha_2^t \boldsymbol{B}_2 + \ldots + \alpha_k^t \boldsymbol{B}_k$
  Compute $\boldsymbol{X} * \boldsymbol{f}^t$ ▷ $*$ is the convolution operation.
  2. Backward Propagation:
  for i=1 to k, **do**
  $\frac{\partial L}{\partial \alpha_i^t} = \sum_{j=1}^n \frac{\partial L}{\partial \boldsymbol{f}_j^t} \frac{\partial \boldsymbol{f}_j^t}{\partial \alpha_i^t}$
  3. Coefficient Update:
  for i=1 to k
  $\alpha_i^{t+1} \leftarrow \alpha_i^t - \eta \frac{\partial L}{\partial \alpha_i^t}$

3.3 Model Training and Optimization

In the following we describe the main steps involved in the training of the proposed architecture. We use stochastic gradient descent (SDG) to update all the tunable parameters in the architecture and an overview of the training process is presented in Algorithm 1. During the forward pass, we first generate individual filters, and then follow the conventional CNN inferencing to compute the loss. However, during the backward pass, we only update the coefficients $\{\alpha\}_{i=1}^N$, but not the binary basis vectors. The loss propagates to each of the layers from the output layer, and the gradient of each coefficient with respect to the the total loss is calculated using chain rule, i.e.:

$$\frac{\partial L}{\partial \alpha_i} = \sum_{j=1}^{n} \frac{\partial L}{\partial \boldsymbol{f}_j} \frac{\partial \boldsymbol{f}_j}{\partial \alpha_i} \quad (3)$$

where L is the loss, $\boldsymbol{f}_j$ is the convolution filter. During the forward propagation in the next iteration, the filters is generated using the updated coefficients.

Convolution kernel generation using OVSF codes as binary basis vectors can be easily integrated to existing architectures, such as fully CNNs, ResNet or any architecture employing convolutions. Therefore, existing architectures can be trained faster, as we have smaller number of free parameters to update, and can have better inference time.

3.4 Inference

The convolution operations within a layer, using the set of OVSF codes, can be summarized as:

$$F_k^O = \left(\sum_{i=1}^{\lfloor \rho \cdot l \rfloor} \alpha_k^i \cdot \boldsymbol{B}_k^i \right) * F^I, \tag{4}$$

where, F^I is the input feature-map and F_k^O is the output feature-map for filter k, $\boldsymbol{B}_k^i$ and α_k^i are the binary vector and corresponding coefficient while representing the k^{th} filter. Interestingly, we can use the linearity of the convolution operation and compute the same output feature-map as follows:

$$F_k^O = \sum_{i=1}^{\lfloor \rho \cdot l \rfloor} \alpha_k^i \cdot \left(\boldsymbol{B}_k^i * F^I \right), \tag{5}$$

These two formulations allow two distinct architecture deployment methods that are suitable in two different application scenarios. In the first case, we generate a static version of the model, employing Equation 4, and in the latter case, for resource constrained devices, we instantiate a dynamic version of the architecture where the OVSF codes are generated on the fly and then used during convolution convolutions as in Equation 5. In the following we describe the two cases.

Explicit Filter Generation. In scenarios with sufficient on-device memory to store the model in memory, the DBFs could be generated, and therefore allocated in memory, as part of a initialization stage during model deployment and set-up. After this, the inference stage would be identical to that of any other CNN architecture.

On-the-fly convolutions. Due to the nature of the filter generation process by combining OVSF codes using weighting coefficients learned during training we could bypass the generation and allocation of the filters during model deployment. These filters would no longer need to be explicitly generated. Instead, since our model has the weighting coefficient and we can generate OVSF codes very efficiently, we can subdivide the operations of a convolutional layer into smaller operations that are less memory taxing. Effectively, this strategy trades memory for computations.

4 Evaluation

In this section we validate the usage of DBFs in convolutional networks for the task of image classification. Here we:

- Compare the performance of two popular CNNs architectures when using filters generated from OVSF codes against standard fully-learnable filters.
- Make use of DBFs a as model size reduction strategy.
- Validate the usage of sp-OVSF codes that would permit the usage of filter with arbitrary dimensionality.

4.1 Datasets

We conducted our experiments on three popular datasets, ImageNet, CIFAR-10 and MNIST. ImageNet is a large-scale image classification dataset, which contains 1000 categories and a total of 1.33 million color images. These images vary in dimension and resolution and are generally resized and cropped to 224×224 images. The dataset is divided into training and validation data, with 1.28 million images and 50,000 images, respectively. The CIFAR-10 dataset contains 60,000 32×32 color images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images, with equal number of images per class in both training set and test set. The MNIST dataset consists of 28×28 grayscale images of handwritten digits, with 60,000 training images and 10,000 test images.

4.2 Experimental Setup

Filters generated using OVSF codes can be used in any CNN for both training and inference. In order to validate the representation capabilities of these filters, we substitute the standard convolutional layers of two popular CNN architectures, ResNet and SqueezeNet, and train them on CIFAR-10 for 250 epochs. We used batch size of 128, initial learning rate of 0.1 with decay factor of 0.1 at epochs $\{90, 150, 190, 220\}$. We applied standard dat augmentation: random image cropping, random mirroing and image normalization. We evaluate our architecture on two configuration of ResNet with 18 and 34 layers. The architectures evaluated in this section were originally designed for ImageNet, here, they have been adapted to the dimensionality of the CIFAR-10 dataset. This adaptation consist on reducing the filter dimensions and stride of the input convolutional layer.

Quality of OVSF filters. Given that OVSF codes are limited to be of length $2^l, l \in \mathbb{N}$, we have modified the spatial dimensions (width and height) of all the 3×3 and 7×7 the convolutional filters in ResNet and SqueezeNet, and replace them with 4×4 and 8×8 filters, respectively. In Table 1 (right) we compare the accuracy levels reached for each architecture when learning filters directly and when using the proposed OVSF filter generation stage.

sp-OVSF: Overcoming 2^l limitation. Despite not being the focus of this work, we evaluated the suitability of OVSF codes to generate filters whose dimensions are not a 2^l, e.g. those with spatial dimensions 3×3 or 5×5. To achieve this, each time we require an OVSF code of l', we first generate the shortest OVSF code of dimensionality $2^l > l'$, and then clip it. The resulting set of sp-OVSF codes are no longer orthogonal to each other, limiting the performance of the generated filters. In Table 1 (left) we should that these codes can still be use to generate convolutional filters.

The impact of ratios. The nature of the filter generation process using OVSF codes permits, by means of a hyperparameter, choose how many bases use to generate each convolutional filter. This hyperparamter is represented in Eq. 2

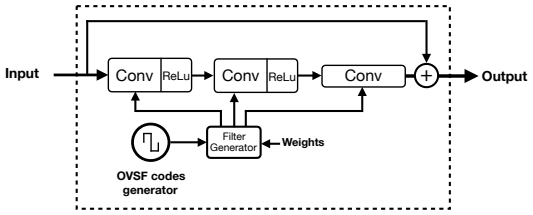


Figure 4: A three layers DBF Module.

as ρ. Lower ρ values results in fewer trainable parameters and a more efficient inference stage at the cost of generating coarser filters and a potential drop in accuracy. We evaluated the effect of using coarser filters on ResNet-18, ResNet-34 and SqueezeNet. These results are shown in Table 2. Reducing ρ effectively reduces the number of weights, α in Eq.2, needed to generate the filter and therefore resulting in a smaller model, as shown in Table 3. An architecture with DBF and $\rho = 1$ has the same model size as it would have with standard filters.

A new CNN architecture. We designed a new CNN architecture using DBFs. We will refer to it as *DBFNet*. Macroarchitecturally, it borrows from SqueezeNet in the sense that after an initial convolution and max-pooling layer, the remaining of the pipeline is comprised of a three cascades of convolutional modules separated by max-pooling layers. We call these blocks *DBF Modules* and consist of three stacked OVSF convolutional layers with a bypass connection. Our network is comprised of eight DBF Moudles arranged similarly to SqueezeNet's *FireModules*. Figure 4 shows a generic DBF Module in isolation with explicit filter generation. When part of a network, a single OVSF code generator is enough to generate the convolutional filters of the entire network. This architecture is evaluated on MNIST, CIFAR-10 and ImageNet at different ρ values. The results are shown in Table 4. On ImageNet we used learning rate of 0.1 and decay factor of 0.1 every 30 epochs for a total of 100 epochs using bach size 64. For CIFAR-10 and MNIST datasets we used batch size of 128, the same initial learning rate and decay factor but decaying it at epochs $\{40, 70, 90\}$. No pre-processing or data augmentation was applied.

4.3 Results

We have proven that, despite the simplicity of the filter generation process using binary OVSF codes, CNNs can perform as well as if filters were learnt directly, like most CNNs do. As we described in Eq.2, the proposed DBFs are as good as any other standard convolutional filter. This is shown in Table 1 (left). We have validated this on two popular deep convolutional architectures, ResNet and SqueezeNet.

Using Coarser DBFs

Networks using DBFs in their convolutional layers can adjust their memory footprint by tuning ρ. This parameter can be set independently for each convolutional layer and is used to determine the number of OVSF codes a generator should output

Architecture	Acc. (%)
ResNet-18	91.15
ResNet-18$_{\text{DBF}}$	91.02
ResNet-34	92.46
ResNet-34$_{\text{DBF}}$	92.32
SqueezeNet	91.16
SqueezeNet$_{\text{DBF}}$	91.33

Architecture	Acc. (%)
ResNet-18	90.68
ResNet-18$_{\text{sp-OVSF}}$	89.12
ResNet-34	92.53
ResNet-34$_{\text{sp-OVSF}}$	91.30
SqueezeNet	91.22
SqueezeNet$_{\text{sp-OVSF}}$	90.25

Table 1: Evaluation on CIFAR-10 when filters are (left) either of dimensionality 2^l and (right) when maintaining the original filter dimensions. DBFs are generated with $\rho = 1$. In each table, every pair of architectures (e.g. ResNet-18 and ResNet-18$_{\text{DBF}}$) uses the same filter dimensions accross layers and the same model size.

in order to generate a given convolutional filter. We demonstrate that even a $4\times$ reduction in the number of parameters of popular architectures such as ResNet and SqueezeNet can result in only 2% accuracy loss. In our experiments we found that reducing ρ for deeper convolutional layer has a lesser impact on accuracy than in the first layers of the network. Concretely, ρ was set to 6.25% for the last two layers in ResNet-34$_{\text{DBF}}$ in the set up where, on average, $\rho = 0.25$. On the other hand, shallower layers kept $\rho = 1$.

Architecture	100%	75%	50%	35%	25%
ResNet-18$_{\text{DBF}}$	91.02	90.46	89.34	89.11	88.02
ResNet-34$_{\text{DBF}}$	92.32	92.92	91.43	91.31	89.88
SqueezeNet$_{\text{DBF}}$	91.33	91.28	91.17	89.89	89.22

Table 2: Evaluation of different architectures using DBFs generated with different average ρ values (%) on CIFAR-10.

Architecture	100%	75%	50%	35%	25%
ResNet-18$_{\text{DBF}}$	1.37	1.02	0.69	0.48	0.34
ResNet-34$_{\text{DBF}}$	3.39	2.54	1.70	1.19	0.85
SqueezeNet$_{\text{DBF}}$	4.41	3.31	2.21	1.54	1.10

Table 3: Model size (MB) of architectures using DBFs with different average ρ values (%). Accuracy values are shown in Table 2.

To our advantage, ρ and the number of learnable parameters are tightly correlated for any architecture using DBFs. This is evidenced in Table 3. Convolutional filters of deeper layers tend to be considerably larger than those in shallower layers, as is the case in ResNet and SqueezeNet. Consequently, these filters represent a seizable portion of the total model size. By means of the parameter ρ their impact in model size can be lessened and, as previously exemplified for the case of ResNet-34$_{\text{DBF}}$, it is possible to achieve $16\times$ memory impact reduction of certain layers while maintaining good accuracy results.

Finally, we show that the benefits of using an architecture with OVSF-based filters are also applicable when the image classification task is considerable more challenging, as is the case with in the ImageNet dataset. Table 4 shows the performance of DBFNet on various image classification dataset at different ρ values.

Dataset	100%	70%	50%	25%
MNIST	99.6	99.6	99.6	99.5
CIFAR-10	89.7	88.3	88.0	85.5
ImageNet	55.1/78.5	53.3/77.3	50.4/74.8	40.5/65.7
Size (MB)	10.32	7.25	5.16	2.58

Table 4: Accuracy (%) of our DBFNet in three popular image datasets at different ρ values (expressed as %). For ImageNet, accuracy values are shown as Top-1/Top-5.

5 Benefits of Deterministic Binary Filters

In the following section we present the main benefits of using the binary basis vectors for the construction of convolution filters and its effect during inference and training time.

5.1 Fewer Parameters

Results from Section 4 show that networks using DBFs offer significant parameter savings. Table 2 shows that $4\times$ reduction in the number of total learnable parameters is possible. By means of parameter ρ we can adjust the memory impact of each layer individually, resulting in $16\times$ memory savings in the deeper layers of the networks while maintaining their dimensionality. We believe these results can be improved by forcing the set of coefficients that pre-multiply each OVSF bases to be sparse in a layer by layer.

We demonstrated the feasibility of DBFs to reduce the model size of already small architectures, in the order of 1-4 MB in size. Our technique brings model size to levels achievable by binary networks. In Table 5 we compare ResNet-18 using DBFs to two popular binary networks. BinaryConnect's results use deterministic binarization. Our technique is capable of providing similar levels of accuracy while reducing the model size to sub-MB levels. Similarly, we compare in Table 6 BinaryConnect and BWN [Rastegari *et al.*, 2016] against our DBFNet when evaluated on ImageNet. DBFNet performs better than BinaryConnect while being $3\times$ smaller.

Architecture	Accuracy (%)	Size (MB)
BinaryConnect	90.1	0.73
BNN	89.85	0.73
ResNet-18$_{\text{DBF}}$ ($\rho = 0.35$)	89.11	0.48

Table 5: Comparison in terms of accuracy and model size of binary architectures and floating-valued architectures using DBF. Results are shown for CIFAR-10.

Architecture	Top-1 (%)	Top-5 (%)	Size (MB)
BinaryConnect	35.4	61.0	7.8
DBFNet ($\rho = 0.125$)	36.5	61.9	2.2
BWN	56.8	79.4	7.8
DBFNet ($\rho = 0.7$)	53.3	77.3	7.3

Table 6: Comparison in terms of accuracy and model size of binary architectures and DBFNet. Results are shown for ImageNet.

5.2 Inference Efficiency

During inference, the overhead of using binary bases is marginal. Bases can be generated once and used across all convolutional layers. This does not impose a big memory footprint as bases are binary and they can be densely packed into bytes and occupy $8\times$ less space.

When these bases are expanded and combined to form the full-precision kernel we do not need to store any of the intermediate values and the run-time memory required is the same as any normal convolutional layer. However, since convolution operation is distributive and associative with scalar multiplication one can change the order of operations and do convolution of input with binary bases first and then scale and combine the results. While this approach normally does not make sense given the significant cost of convolution operation, it can be efficient in architectures with binary inputs where the convolution operation is reduced to XORing and bit counting [Rastegari *et al.*, 2016].

5.3 Training Efficiency

The main benefits of using Deterministic Binary Filters come from their ability to reduce memory and computation footprints without a significant drop in the recognition accuracy. From the previous section we see that across different datasets the proposed architecture can achieve high accuracy while only considering a fraction of the OVSF codes. This allows for a significant reduction int the number of tunable convolution parameters, in our case only the coefficients, and generates a very compact model size, which is ideal for embedded deployment. As we need a smaller number of parameters to tune, the model becomes less prone to overfitting than the corresponding static version of the architecture. An architecture with DBFs runs faster backward pass, thereby reducing the overall training procedure significantly.

6 Conclusion

We have presented *Deterministic Binary Filters*, an new approach to constructing modules within CNNs that only requires learning of weighting coefficients with respect to a predefined orthogonal binary basis. Significant savings result in comparison to conventional convolutional filters that are learned entirely from data. With fewer parameters such models are less prone to over-fitting and can be potentially trained with significantly less compute operations and memory needs. DBFs provides important new insights in the design of low-complexity models that maintain high accuracy level for discriminative image tasks with implications for training and inference efficiency.

Acknowledgements

This work was supported in part by the UK's Engineering and Physical Sciences Research Council (EPSRC) with grants EP/M50659X/1, EP/N509711/1 and EP/R512333/1.

References

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

[Adachi *et al.*, 1998] Fumiyuki Adachi, Mamoru Sawahashi, and Hirohito Suda. Wideband ds-cdma for next-generation mobile communications systems. *IEEE communications Magazine*, 36(9):56–69, 1998.

[Andreev *et al.*, 2003] Boris D. Andreev, Edward L. Titlebaum, and Eby G. Friedman. Orthogonal code generator for 3g wireless transceivers. In *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, GLSVLSI '03, pages 229–232, New York, NY, USA, 2003. ACM.

[Bhattacharya and Lane, 2016a] S. Bhattacharya and N. D. Lane. From smart to deep: Robust activity recognition on smartwatches using deep learning. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6, March 2016.

[Bhattacharya and Lane, 2016b] Sourav Bhattacharya and Nicholas D. Lane. 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*, SenSys '16, pages 176–189, New York, NY, USA, 2016. ACM.

[Courbariaux and Bengio, 2016] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[Esser *et al.*, 2015] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V. Arthur, and Dharmendra S Modha. Backpropagation for energy-efficient neuromorphic computing. In *Advances in Neural Information Processing Systems 28*, pages 1117–1125. Curran Associates, Inc., 2015.

[Fernandez-Marques *et al.*, 2018] Javier Fernandez-Marques, Vincent T.-S. Tseng, Sourav Bhattacharya, and Nicholas D. Lane. On-the-fly deterministic binary filters for memory efficient keyword spotting applications on embedded devices. In *Proceedings of the 2nd International Workshop on Deep Learning for Mobile Systems and Applications*, EMDL '18. ACM, 2018.

[Frosst and Hinton, 2017] Nicholas Frosst and Geoffrey E. Hinton. Distilling a neural network into a soft decision tree. *CoRR*, abs/1711.09784, 2017.

[Georgiev *et al.*, 2017] Petko Georgiev, Nicholas D. Lane, Cecilia Mascolo, and David Chu. Accelerating mobile audio sensing algorithms through on-chip gpu offloading. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 306–318, New York, NY, USA, 2017. ACM.

[Han *et al.*, 2015] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[He *et al.*, 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[Howard *et al.*, 2017] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[Iandola *et al.*, 2016] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[Jarrett *et al.*, 2009] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *IEEE 12th International Conference on Computer Vision, ICCV 2009*, 2009.

[Juefei-Xu *et al.*, 2017] Felix Juefei-Xu, Vishnu Naresh Boddeti, and Marios Savvides. Local binary convolutional neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, volume 1, 2017.

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

[Krizhevsky *et al.*, 2012] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[Lane *et al.*, 2015] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. 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*, IoT-App '15, pages 7–12, New York, NY, USA, 2015. ACM.

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

[LeCun *et al.*, 1989] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, December 1989.

[Mathur *et al.*, 2017] Akhil Mathur, Nicholas D. Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 68–81, New York, NY, USA, 2017. ACM.

[McDanel *et al.*, 2017] Bradley McDanel, Surat Teerapittayanon, and H. T. Kung. Embedded binarized neural net-

works. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks, EWSN 2017, Uppsala, Sweden, February 20-22, 2017*, pages 168–173, 2017.

[Pinto *et al.*, 2009] Nicolas Pinto, David Doukhan, James J. DiCarlo, and David D. Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLOS Computational Biology*, 5:1–12, 11 2009.

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

[Rastegari *et al.*, 2016] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.

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

[Saxe *et al.*, 2011] Andrew Saxe, Pang W. Koh, Zhenghao Chen, Maneesh Bhand, Bipin Suresh, and Andrew Y. Ng. On random weights and unsupervised feature learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. ACM, 2011.

[Soudry *et al.*, 2014] Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *Advances in Neural Information Processing Systems 27*, pages 963–971. Curran Associates, Inc., 2014.

[Suleiman *et al.*, 2017] A. Suleiman, Y. H. Chen, J. Emer, and V. Sze. Towards closing the energy gap between hog and cnn features for embedded vision. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017.

[Sze *et al.*, 2017] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *CoRR*, abs/1703.09039, 2017.

[Tahavori *et al.*, 2017] Fatemeh Tahavori, Emma Stack, Veena Agarwal, Malcolm Burnett, Ann Ashburn, Seyed Amir Hoseinitabatabaei, and William Harwin. Physical activity recognition of elderly people and people with parkinson's (pwp) during standard mobility tests using wearable sensors. In *Smart Cities Conference (ISC2), 2017 International*, pages 1–4. IEEE, 2017.

[Vanhoucke *et al.*, 2011] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[Wang *et al.*, 2016] Yunhe Wang, Chang Xu, Shan You, Dacheng Tao, and Chao Xu. Cnnpack: Packing convolutional neural networks in the frequency domain. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 253–261. Curran Associates, Inc., 2016.

[Yao *et al.*, 2017] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek F. Abdelzaher. Compressing deep neural network structures for sensing systems with a compressor-critic framework. *CoRR*, abs/1706.01215, 2017.

Bibliography

Abdel-Hamid, O., Mohamed, A., Jiang, H., Deng, L., Penn, G., and Yu, D. (2014). Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, **22**(10), 1533–1545.

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

Anonymous (2019). A systematic study of binary neural networks' optimisation. In *Submitted to International Conference on Learning Representations*. under review.

Bengio, Y., Léonard, N., and Courville, A. C. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, **abs/1308.3432**.

Courbariaux, M. and Bengio, Y. (2016). Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, **abs/1602.02830**.

Courbariaux, M., Bengio, Y., and David, J. (2015). Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, **abs/1511.00363**.

Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, pages 886–893, Washington, DC, USA. IEEE Computer Society.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

Denton, E., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, pages 1269–1277, Cambridge, MA, USA. MIT Press.

Esser, S. K., Appuswamy, R., Merolla, P., Arthur, J. V., and Modha, D. S. (2015). Backpropagation for energy-efficient neuromorphic computing. In *Advances in Neural Information Processing Systems 28*, pages 1117–1125. Curran Associates, Inc.

Fang, B., Zeng, X., and Zhang, M. (2018). Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, pages 115–127, New York, NY, USA. ACM.

Fernandez-Marques, J., T.-S. Tseng, V., Bhattacharya, S., and Lane, N. D. (2018). On-the-fly deterministic binary filters for memory efficient keyword spotting applications on embedded devices. In *Proceedings of the 2nd International Workshop on Deep Learning for Mobile Systems and Applications*, EMDL '18. ACM.

Gao, H., Wang, Z., and Ji, S. (2018). Channelnets: Compact and efficient convolutional neural networks via channel-wise convolutions. In *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Ha, D., Dai, A., and Le, Q. (2016). Hypernetworks.

Han, S., Pool, J., Tran, J., and Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pages 1135–1143, Cambridge, MA, USA. MIT Press.

Han, S., Mao, H., and Dally, W. J. (2016a). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. (2016b). Eie: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254.

Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 164–171. Morgan-Kaufmann.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, **abs/1512.03385**.

He, Y. and Han, S. (2018). ADC: automated deep compression and acceleration with reinforcement learning. *CoRR*, **abs/1802.03494**.

Hill, A. P., Prince, P., Piña Covarrubias, E., Doncaster, C. P., Snaddon, J. L., and Rogers, A. (2018). Audiomoth: Evaluation of a smart open acoustic device for monitoring biodiversity and the environment. *Methods in Ecology and Evolution*, **9**(5), 1199–1211.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, **9**(8), 1735–1780.

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

Hu, J., Shen, L., and Sun, G. (2018). Squeeze-and-excitation networks. In *IEEE Conference on Computer Vision and Pattern Recognition*.

Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Huang, G. B., Ramesh, M., Berg, T., and Learned-Miller, E. (2007). Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst.

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, **40**(9), 1098–1101.

Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, **abs/1602.07360**.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Jin, X., Yang, Y., Xu, N., Yang, J., Jojic, N., Feng, J., and Yan, S. (2018). WSNet: Compact and efficient networks through weight sampling. In J. Dy and A. Krause, editors, *Proceedings of*

the 35th International Conference on Machine Learning, volume 80 of *Proceedings of Machine Learning Research*, pages 2352–2361, Stockholmsmässan, Stockholm Sweden. PMLR.

Juefei-Xu, F., Boddeti, V. N., and Savvides, M. (2017). Local binary convolutional neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, volume 1.

Kim, Y.-D., Park, E., Yoo, S., Choi, T., Yang, L., and Shin, D. (2016). Compression of deep convolutional neural networks for fast and low power mobile applications.

Kingma, D. P., Salimans, T., and Welling, M. (2015). Variational dropout and the local reparameterization trick. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2575–2583. Curran Associates, Inc.

Krizhevsky, A., Nair, V., and Hinton, G. (2009). Cifar-10 (canadian institute for advanced research).

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

Kumar, A., Goyal, S., and Varma, M. (2017). Resource-efficient machine learning in 2 KB RAM for the internet of things. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1935–1944, International Convention Centre, Sydney, Australia. PMLR.

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

Lai, L., Suda, N., and Chandra, V. (2018b). Not all ops are created equal! *CoRR*, **abs/1801.04326**.

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

Lin, T., Maire, M., Belongie, S. J., Bourdev, L. D., Girshick, R. B., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft COCO: common objects in context. *CoRR*, **abs/1405.0312**.

Lin, X., Zhao, C., and Pan, W. (2017). Towards accurate binary convolutional neural network. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 345–353. Curran Associates, Inc.

Louizos, C., Ullrich, K., and Welling, M. (2017). Bayesian compression for deep learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3288–3298. Curran Associates, Inc.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, **5**(4), 115–133.

McDanel, B., Teerapittayanon, S., and Kung, H. T. (2017a). Embedded binarized neural networks. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks, EWSN 2017, Uppsala, Sweden, February 20-22, 2017*, pages 168–173.

McDanel, B., Teerapittayanon, S., and Kung, H. (2017b). Incomplete dot products for dynamic computation scaling in neural network inference. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 186–193.

Minsky, M. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA.

Molchanov, D., Ashukha, A., and Vetrov, D. (2017). Variational dropout sparsifies deep neural networks. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2498–2507, International Convention Centre, Sydney, Australia. PMLR.

Ojala, T., Pietikainen, M., and Harwood, D. (1994). Performance evaluation of texture measures with classification based on kullback discrimination of distributions. In *Proceedings of 12th International Conference on Pattern Recognition*, volume 1, pages 582–585 vol.1.

Perronnin, F. and Dance, C. (2007). Fisher kernels on visual vocabularies for image categorization. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8.

Qiu, Q., Cheng, X., robert Calderbank, and Sapiro, G. (2018). DCFNet: Deep neural network with decomposed convolutional filters. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4198–4207, Stockholmsmässan, Stockholm Sweden. PMLR.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, **abs/1603.05279**.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2018). Regularized evolution for image classifier architecture search. *CoRR*, **abs/1802.01548**.

Ren, M., Pokrovsky, A., Yang, B., and Urtasun, R. (2018). Sbnet: Sparse blocks network for fast inference. *CoRR*, **abs/1801.02108**.

Ren, S., He, K., Girshick, R., and Sun, J. (2015). Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems (NIPS)*.

Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton : (Project Para)*. Cornell Aeronautical Laboratory, Buffalo, NY.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, **323**, 533 EP –.

Rusci, M., Rossi, D., Farella, E., and Benini, L. (2017). A sub-mw iot-endnode for always-on visual monitoring and smart triggering. *IEEE Internet of Things Journal*, **4**(5), 1284–1295.

Sandler, M., Howard, A. G., Zhu, M., Zhmoginov, A., and Chen, L. (2018). Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, **abs/1801.04381**.

Siekkinen, M., Hiienkari, M., Nurminen, J. K., and Nieminen, J. (2012). How low energy is bluetooth low energy? comparative measurements with zigbee/802.15.4. In *2012 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pages 232–237.

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

Soudry, D., Hubara, I., and Meir, R. (2014). Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *Advances in Neural Information Processing Systems 27*, pages 963–971. Curran Associates, Inc.

Ström, N. (1997). Phoneme probability estimation with dynamic sparsely connected artificial neural networks.

Tan, M., Chen, B., Pang, R., Vasudevan, V., and Le, Q. V. (2018). Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, **abs/1807.11626**.

Tschannen, M., Khanna, A., and Anandkumar, A. (2018). StrassenNets: Deep learning with a multiplication budget. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4985–4994, Stockholmsmässan, Stockholm Sweden. PMLR.

Tseng, V. W.-S., Bhattacharya, S., Marqués, J. F., Alizadeh, M., Tong, C., and Lane, N. D. (2018). Deterministic binary filters for convolutional neural networks. In *IJCAI*.

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. In *Arxiv*.

Wang, Y., Xu, C., You, S., Tao, D., and Xu, C. (2016). Cnnpack: Packing convolutional neural networks in the frequency domain. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 253–261. Curran Associates, Inc.

Wang, Y., Xu, C., Xu, C., and Tao, D. (2017). Beyond filters: Compact feature map for portable deep model. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3703–3711, International Convention Centre, Sydney, Australia. PMLR.

Warden, P. (2017). Speech commands: A public dataset for single-word speech recognition.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, **78**(10), 1550–1560.

Xie, S., Girshick, R. B., Dollár, P., Tu, Z., and He, K. (2017). Aggregated residual transformations for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 5987–5995.

Yang, T., Chen, Y., and Sze, V. (2017a). Designing energy-efficient convolutional neural networks using energy-aware pruning. *IEEE Conference on Computer Vision and Pattern Recognition*.

Yang, Y., Li, T., Li, W., Wu, H., Fan, W., and Zhang, W. (2017b). Lesion detection and grading of diabetic retinopathy via two-stages deep convolutional neural networks. *CoRR*, **abs/1705.00771**.

Yao, S., Zhao, Y., Zhang, A., Su, L., and Abdelzaher, T. F. (2017). Compressing deep neural network structures for sensing systems with a compressor-critic framework. *CoRR*, **abs/1706.01215**.

Zhang, X., Zhou, X., Lin, M., and Sun, J. (2018a). Shufflenet: An extremely efficient convolutional neural network for mobile devices. *IEEE Conference on Computer Vision and Pattern Recognition*.

Zhang, Y., Pezeshki, M., Brakel, P., Zhang, S., Laurent, C., Bengio, Y., and Courville, A. (2016). Towards end-to-end speech recognition with deep convolutional neural networks. In *Interspeech 2016*, pages 410–414.

Zhang, Y., Suda, N., Lai, L., and Chandra, V. (2017). Hello edge: Keyword spotting on microcontrollers. *arXiv preprint arXiv:1711.07128*.

Zhang, Y., Li, K., Li, K., Wang, L., Zhong, B., and Fu, Y. (2018b). Image super-resolution using very deep residual channel attention networks. In *ECCV*.

Zhao, H., Shi, J., Qi, X., Wang, X., and Jia, J. (2017). Pyramid scene parsing network. In *CVPR*.

Zhou, K., Gu, Z., Liu, W., Luo, W., Cheng, J., Gao, S., and Liu, J. (2018). Multi-cell multi-task convolutional neural networks for diabetic retinopathy grading kang. *CoRR*, **abs/1808.10564**.

Zhu, C., Han, S., Mao, H., and Dally, W. J. (2016). Trained ternary quantization. *arXiv preprint arXiv:1612.01064*.

Zoph, B. and Le, Q. V. (2017). Neural architecture search with reinforcement learning. *International Conference on Learning Representations (ICLR)*.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *Computer Vision and Pattern Recognition (CVPR), 2018 IEEE Conference on*, volume 1.