# Compression Frameworks & Bayesian Deep Learning


Milad Alizadeh

Linacre College

University of Oxford

DPhil Transfer Report

Michaelmas 2018

# Contents

# Chapter 1

# Research Proposal

## 1.1 Introduction

This DPhil thesis will focus on the intersection of theoretical methods from Bayesian statistics and applied techniques and architectures used in practice and aims to formalise and improve model compression of Deep Neural Networks (DNNs). The goal is to better understand current compression techniques used in practice, formalise their consequences, and to design more efficient algorithms that could help deploy DNNs to mobile resource-constrained devices. The project is going to look at how Bayesian methods can enable better compression schemes and inversely, how Bayesian models themselves can be accelerated to run on resource-constrained devices.

This report is organised as follows: The remainder of Chapter 1 outlines the research outline and plans for completed and future projects. Chapter 2 provides a literature review of the key methods in model compression of DNNs and highlights some of the limitations and opportunists in the field.

## 1.2 Motivation

Deep Neural Networks are the state of the art models in many tasks across various domains such as computer vision, speech recognition, and natural language processing. However, these models are often deployed using remote cloud infrastructures. There is great interest in expanding usage of DNNs from running remotely in the cloud to performing local on-device inference on resource-constrained devices [Sze et al., 2017, Lane and Warden, 2018]. Examples of such devices are mobile phones, wearables, IoT devices, and robots. This is motivated by the privacy implications of sharing data and models with remote machines, and the appetite to apply DNNs in new environments and scenarios where cloud-inference is not viable. However, requirements of such devices are very demanding: there are stringent compute, storage, memory and bandwidth limitations; many applications need to work in real-time; many devices require long battery life for all-day or always-on use, and there is a thermal ceiling to consider when designing thin and light devices.

On the other hand, the quest for more accurate DNNs has resulted in deeper, more compute-intensive models. This is particularly the case for CNNs. For instance, while the convolutional layers of AlexNet [Krizhevsky et al., 2012] make up only 4% of the model parameters, they are accountable for 91% of the computations at inference time [Louizos et al., 2017a]. Compression and efficient implementation of DNNs are therefore more important than ever. There has been a spate of recent work proposing training and post-training schemes that aim to compress models without significant loss in their performance. Primary examples of these techniques are: pruning, weight sharing, low-rank approximation, knowledge distillation and perhaps most importantly quantisation to lower precisions [Han et al., 2015a, Ullrich et al., 2017, Hinton et al., 2015]. Another critical yet overlooked challenge in using DNNs on-device is the robustness and reliability of sensory inference results. Obtaining uncertainty estimates in such systems, e.g., autonomous cars, is crucial. However, effective and efficient usage of Bayesian DNNs remains an open research question.

## 1.3 Research Questions and Objectives

The work in this DPhil intends to address some of the challenges covered in the previous section and answer the following questions:

- "What are the limitations, theoretical foundations, and best approaches of quantising DNNs to the extreme?"
- "What are the system architectures, hardware architectures, and algorithms that would allow Bayesian deep models to be deployed on constrained devices?"
- "How can DNNs be trained, particularly using Bayesian techniques, to facilitate obtaining on-demand compressed versions?"
- "What properties and information are lost or gained when large models are compressed into smaller versions?"

## 1.4 Research Outline

### 1.4.1 Extreme Quantisation of DNNs

It has been shown that even when quantisation is pushed to the extreme, it is still possible to train DNNs with state-of-the-art results. One example of such methods is training Binary Neural Networks (BNNs) using Straight-Through-Estimator (STE) for backpropagation. However, the training process of this technique is not well-founded. This is due to the discrepancy between the evaluated function in the forward path, and the weight updates in the back-propagation. In other words, the weights updates computed with STE do not correspond to gradients of the forward path.

While there have been attempts to provide theoretical justifications for STE in the literature, their scope has been limited. The first part of this project aims to empirically study the performance of various techniques used in BNNs in order to identify the key parts that enable to them work. The goal in the second part of the project is to formalise and provide theoretical justifications for optimisation of BNNs based on the findings from the empirical study.

### Completed Project

Efficient convergence and accuracy of binary models often rely on careful fine-tuning and various ad-hoc techniques. This project empirically identified and studied the effectiveness of the techniques commonly used in the literature on the accuracy and convergence performance of binary models. It showed that training binary models are harder and slower than the equivalent non-binary model and while the limit of STE's capability can be achieved easily, finding the best set of weights requires longer training. It also provided a series of recommendations and best practices for the efficient training of binary models.

The contributions of the project can be summarised as the following:

- It identifies the essential techniques required for successful optimisation of binary models and shows that end-to-end training of binary networks crucially relies on the optimiser taking advantage of second-moment gradient estimates. Other optimisers can easily get stuck in local minima and can fail to converge.
- It shows that most of the commonly used tricks in training binary models, such as gradient and weight clipping, are only required during the final stages of the training to achieve the best performance. Further, it demonstrates that these tricks lead to much slower convergence in the early stages of optimisation.
- It proposes new procedures for training, making optimisation notably faster by delaying these tricks, or by training a full-precision model first and fine-tune it into a binary model.

This work is currently under review in the International Conference on Learning Representations (ICLRL) 2019 and is included in Appendix A. The early results from this projected was presented as a poster at the ACM International Conference on Mobile Systems (MobiSys) 2018 and is included in Appendix B.

### Future Project

The analysis done in the first part of the project helps in disambiguating what is necessary from what is unnecessary training of binary neural networks, and paves the way for future development of solid theoretical foundations for studying optimisation of quantised models. The second part of this project looks more carefully at the STE and attempts to find proxy loss functions where STE can be seen as the solution in expectation.

### 1.4.2 Acceleration Frameworks for DNNs

Purpose-built hardware accelerators for DNNs are on the verge of going mainstream, and will soon be available in various commodity mobile and embedded devices. This variety of hardware has the potential to perform inference on deep models vastly more efficiently than conventional processors (such as CPUs). But their wide-spread availability will provoke a number of basic questions in system design, processor selection and usage as well as deep model tuning for which we are not yet ready to answer.

#### Completed Project

This project is a collaboration with Nokia Bell Labs in Cambridge and Stony Brook University in New York and aims to provide early answers through in-depth study of currently one of the only commercially-available open neural network accelerators, the Intel Movidius Neural Compute Stick. This study performs a first-of-its-kind systematic measurement study of the latency and energy of this accelerator under a variety of deep convolutional networks, and considers its performance in comparison to processor alternatives for constrained devices; specifically, the DSP, GPU and multi-core CPU available in Qualcomm Snapdragon 820; a platform that is representative of typical mobile and embedded hardware. The project offers a preview of the future in which resource-constrained devices perform on-device deep learning using a rich heterogeneous processor mix that includes hardware accelerators.

This work was lead by Nokia Bell Labs and Stony Brookand and it is currently under review at the International Conference on Information Processing in Sensor Networks (IPSN) 2019 and included in Appendix D.

#### Future Project

The next steps of the project will explore new architectures that work better on commodity accelerator frameworks, but more importantly will propose system architectures for future accelerator with a focus on Bayesian models. Bayesian methods can provide uncertainty estimates for model predictions. However, performing exact Bayesian inference is in general intractable. Bayesian methods often need approximations to make the inference tractable. Among various approximation techniques, variational approximation, or variational inference, tends to be faster and easier to implement. Besides, the variational inference method offers better scalability with large models, especially for large-scale neural networks in deep learning applications.

Using Bayesian models on resource-constrained devices will have fundamentally different requirements. This project will look at system design space, computation blocks, memory and cache architectures, and closed-form approximations that could enable efficient variational inference on Bayesian networks. The project will look at ways to replace the resource-hungry sampling by layer-wise distribution approximations amenable to closed-form representations.

### 1.4.3 Adaptive Single-Shot Compression

**Future Project**

This project will borrow ideas from few-shot learning and meta-learning but rather than applying them to unseen new data points, it aims to train models that can be adopted for various compression levels. The goal is to train multi-objective networks that can achieve their standard best results or target a specific compression via a single or few gradient-steps. Moreover, the project attempts to design methods that can use a single mini-batch from the dataset to prune or determine quantisation-levels for the models before training begins.

### 1.4.4 Lost and Gained Capabilities of Compressed DNNs

**Future Project**

It is well known that compressed model can achieve better generalisation bounds. This project looks at some of the other properties that are gained or lost when DNNs are compressed. For instance, it will look at Dropout as a Bayesian approximation in DNNs and attempts to answer whether these approximations begin to fail when models are compressed.

Another space to explore will be to see the effect of compression on the interpretability of DNNs. The goal is to find models that are not just high-performing but also make explainable predictions and are robust against adversaries. This project attempts to see if these properties are easier to obtain in compressed models.

# Chapter 2

# Literature Review

## 2.1 Overview

There has been a spate of work in recent years that aim to tackle the problem of compression and efficiency of Deep Neural Networks (DNNs) from various viewpoints and with different objectives. This chapter provides a survey and discussion of the most prominent methods proposed in this area.

The main body of work in deriving more efficient deep neural networks can be roughly organised into the following main categories:

- **Simplified Representations** These methods attempt to derive a simpler, smaller representation of the original model. This goal is typically achieved by identifying and pruning away unimportant weights, neurons and convolutions filters, representing parameters with lower-precision and distilling and transferring the learned knowledge of the original model into the smaller model.
- **Efficient Architectures** The methods in this group propose DNN architectures that are designed from the ground up to be more efficient when deployed. These architectures typically aim to have a smaller number of parameters or operations.
- **Bag of Tricks** These methods exploit a combination of various ad-hoc techniques to compress neural networks. For example, they borrow methods from linear algebra, such as Singular Value Decomposition (SVD), Huffman coding from information theory, weight sharing, etc.
- **Hardware Acceleration** This category consists of architectures and available consumer products that use custom-designed hardware and silicon to train and run DNNs more efficiently and with a fraction of energy compared to GPUs and cloud deployments.

In addition to categories above compression techniques can be categorised as training-time methods or post-processing methods. In the following sections of this chapter, we review some of the most important works from these categories followed by a critical discussion of the overall state of model compression literature.

## 2.2 Binary Neural Networks

Representing weights and activations of a neural network by quantising them to lower-preicisons is an obvious solution to derive simpler models. Early quantised models were derived by quantisating full-precision weights of pre-trained models as a post-processing step [Gong et al., 2014]. This approach is widely used in real deployments and enjoys advantages such as the flexibility to apply different levels of quantisation based on the target model size, and not requiring knowledge of model internals. However, it suffers from a significant loss of accuracy. Hubara et al. [2017] showed that in order to maintain model performance, quantisation must be incorporated as part of the training process. This is done by either performing additional training steps to fine-tune a quantised model or by directly learning quantised parameters. This is essential for BNNs where binarising weights of pre-trained models result in significant loss in accuracy.

One appealing quantisation scheme is to push quantisation level to the extreme by representing each weight with a single bit. This approach leads to the emergence of Binary Neural Networks (BNNs). The first successful binarisation-aware training method was proposed in BinaryConnect by Courbariaux et al. [2015]. In their work, the binary weights are not learned directly; instead, full-precision weights are maintained and *learned* during the training as proxies for the binary weights. These proxies are only required during training. During the forward path, binary weights are computed by applying sign function to their corresponding full-precision proxies. Since the sign function is not differentiable BinaryConnect employs Straight-Through-Estimator (STE) [Bengio et al., 2013] for back-propagating gradient estimates to full-precision proxies. The STE estimator simply passes the gradients along as if the non-differentiable operator was not present. In practice, BinaryConnect applied two additional restrictions on vanilla STE: (1) *Gradient clipping* stops gradient flow if the weight's magnitude is larger than 1. This effectively means gradients are computed with respect to hard tanh function. (2) *Weight clipping* is applied to weights *after* gradients have been applied to keep them within a range. To formalise this consider $w_r$ to be a full-precision proxy for binary weight $w_b$. During the forward path (and at the end of the training):

$$w_b = \text{sign}(w_r)$$

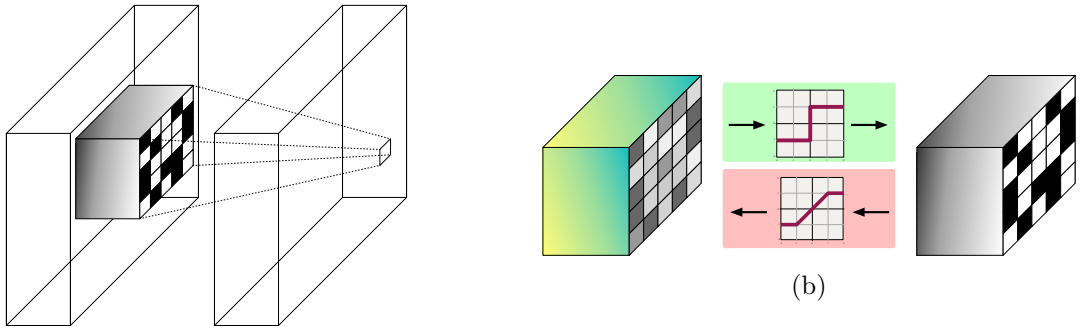
STE with gradient clipping provides an estimate for gradient of this operation:

$$\frac{\partial w_b}{\partial w_r} = \mathbf{1}_{|r|\leq 1} \tag{2.1}$$

In a back-propagation context we assume the gradient of the cost $C$ at the output ($\frac{\partial C}{\partial w_b}$) is available where in computing it the same STE estimator above has been used wherever required. Eq 2.1 enables us to estimate the gradient of the cost at the input ($\frac{\partial C}{\partial w_r}$) and update the proxies:

$$\frac{\partial C}{\partial w_r} = \frac{\partial C}{\partial w_b}\mathbf{1}_{|r|\leq 1}$$

The estimator passes gradients backwards unchanged when proxies are within the {-1,1} range and cancels the gradient flow when the proxy weight has got too positive or too negative. Figure 2.1 depicts how this works for a convolutional kernel in a CNN.



(a) (b)

Figure 2.1: A convolutional kernel in a Binary Neural Network is binary (left) but its values are derived from a full-precision proxy learned using using the STE estimator (right). At the end of the training the proxy kernel is used for one last time to compute final binary values.

There has been several extensions to BinaryConnect's core idea of using STE estimator in binary models. BinaryConnect showed slightly better results when STE was used in stochastic binary neurons. Courbariaux et al. [2016] and Rastegari et al. [2016] expanded BNNs by using the sign function as the non-linearity to achieve binary activations in addition to binary parameters. With this approach full-precision MAC operations in convolution layers can be replaced with cheap XNOR and POPCOUNT binary operations. This results in 58× [Rastegari et al., 2016] improvement in compute-time in addition to the inherit 32× saving in model size that comes from replacing 32-bit floating point parameters with binary ones. XNOR-Net and BWN [Rastegari et al., 2016] managed to scale up BNNs to achieve competitive results on the much bigger ImageNet [Deng et al., 2009] dataset by learning additional full-precision scale factors per-layer. DoReFa-Net [Zhou et al., 2016] used STE in the backpropagation path to quantise gradients and achieve faster training. 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. Lin et al. [2017] achieved state-of-the-art performance by learning a combination of very few binary kernels in each layer.

## 2.3 Classical Network Pruning

The main motivation for pursuing pruning in DNNs comes from observations that deep models are often hevaily over-parameterised, and that pruned models can yield better generalisation bounds [Reed, 1993, Arora et al., 2018]. Therefore, pruning has a long history in the literature on neural networks. Most of the classical methods in network pruning can be placed into two categories [Reed, 1993]:

- **Saliency Criterion** These methods aim to identify and prune a subset of weights from a pretrained network based on some kind of saliency criterion to measure weights importance.

- **Loss Penalty** These methods attempt to directly *induce* sparsity during training by enforcing extra penalties in the loss function.

The simplest approach from the first category is to use magnitude of the weights to estimate their saliency. Methods that use this criterion assume that small weights are less important than large weights [Wan et al., 2009, Hagiwara, 1994]. Magnitude-based pruning techniques are computationally efficient and scale well to large datasets and models but they are also known to unnecessarily remove important parts of the network [Sietsma and Dow, 1988]. Optimal Brain Damage [LeCun et al., 1990] and Optimal Brain Surgeon [Hassibi and Stork, 1993] are two classic papers from the 1990s that expand magnitude-based pruning by using a second-order Taylor approximation of the sensitivity of the loss function to determine unimportant weights.

In Optimal Brain Damage, the saliency for each parameter is computed using a diagonal Hessian approximation. The low-saliency weights are pruned from the network and the resulting network is then re-trained. To formalise, given a network with weights $w$, a small change on the weight vector, denoted by $\delta w$, causes a change on the loss function $L$:

$$\delta L = L(w + \delta w) - L(w) \tag{2.2}$$

This change can be approximated by applying the Taylor expansion:

$$\delta L \approx \delta w \frac{\partial L}{\partial w} + \frac{1}{2}\delta w^T H \delta w \tag{2.3}$$

where H is the Hessian matrix. For a sufficiently trained network, the $\frac{\partial L}{\partial w} \approx 0$ and therefore the change on $L$ is mostly caused by the second order term. The authors further ignore the non-diagonal elements of $H$, i.e. $h_{i,i}$, and estimate the saliency using:

$$\delta L \approx \frac{1}{2}\sum_{i=1}^{K} h_{i,i}\delta w_i^2 \tag{2.4}$$

Optimal Brain Surgeon improves this approach by allowing for general error measures, and proposes a more efficient implementation using dominant eigenspace decomposition that requires less computational and storage. The criteria in both these methods is heavily dependent on the scale of the weights and is designed to be incorporated within the training process. Therefore these methods require many iterations of pruning and retraining steps.

The methods from the second category work by adding sparsity-inducing penalty terms to the loss function. This allows the back-propagation to directly penalise the magnitude of the weights during training. Traditionally the penalty has taken the form of $L_2$ regularisation which is equivalent to the weight-decay [Denker et al., 1987]. Later work [Williams, 1995] proposed using $L_1$ regularisation, which is known to induce sparsity [Tibshirani, 1996, Ng, 2004]. This solution alleviates the need for sophisticated approximations of the Hessian to estimate a parameter's contribution to the loss. Finally, in a recent paper proposed using the simple $L_0$ norm (i.e. the raw number of parameters) during training to encourage weights to become *exactly* zero Louizos

et al. [2017b]. However, since $L_0$ norm of the weights is not differentiable it cannot be embedded directly into the loss function as a regularization term. The authors proposed a combination of variational inference, reparametreisation rick [Kingma and Welling, 2013] and a relaxation of the Concrete distribution [Maddison et al., 2016] to infer stochastic binary gates that determine which weights should be set to zero.

## 2.4 Bayesian Network Pruning

Bayesian methods apply Occam's razor principle and search for the simplest model that can explain the data. As a result Bayesian perspective is inherently aligned with the objective of compressing models. This connection is made explicit in the Minimum Description Length Principle (MDL) principle [Rissanen, 1986, Grünwald, 2007] which is known to be related to Bayesian inference. This section covers some of the early works that aimed to keep neural networks simple and their relation to the Bayesian interpretation of neural networks. It also covers a more recent line of work on Bayesian and variational deep learning and how it has been used to compress models.

One of the earliest works on Bayesian Neural Networks was by Hinton and Van Camp [1993] in the 90s where the problem of overfitting in Neural Networks was framed in terms of MDL framework [Rissanen, 1986, Grünwald, 2007]. The MDL principle has strong links to model compression and can be summarised with the following statement: "The best model is the one that compresses the data best. There are two costs, one for transmitting a model and one for reporting the data misfit.". The model cost can be represented by the number of bits it takes to describe the weights while the data-misfit is represented by the number of bits it takes to describe the discrepancy between the correct output and the output of the model on each training data point.

Hinton and Van Camp applied this framework to neural networks and argued that in order to avoid overfitting, it is important to ensure that there is less information in the weights than there is in the output vectors of the training dataset. Therefore during learning, weights must be kept simple by penalizing the amount of information they contain. The amount of information in a weight can be controlled by adding a source of Gaussian noise. During learning, the amount of noise can be adapted to optimise the trade-off between the expected squared error of the network and the amount of information in the weights. The paper proposed an efficient approach for computing the derivatives of the expected squared error and the amount of information in the noisy weights in a network. The authors also showed how the idea of minimising the amount of information required to communicate the weights leads to a number of interesting schemes for encoding the weights.

The methods used in this paper effectively leads to a Bayesian interpretation of neural networks. Interestingly, the word 'Bayesian' is not mentioned even once in the paper. A more explicit line of work in Bayesian learning is the application of Variational Inference (VI) methods to dropout regularisation. Dropout [Hinton et al., 2012] is one of the most popular and empirically effective

techniques for reducing overfitting in neural networks. The key idea behind this technique is to stochastically remove neurons (along with all their connections) from the model during training. Hinton et al. argue that this approach can prevent units from learning to collaborate too much. During training, dropout effectively samples from an exponential number of different sparse networks. At test time, using the *un*-thinned model approximate averaging over predictions of all thinned models.

In the original dropout scheme (also known as the binary dropout), the decision to drop or keep a unit is determined by sampling from a Bernoulli distribution. The parameter $p$ of the distribution is a fixed hyper-parameter (often 0.5) and shared among all units. However, it was shown in a later work [Srivastava et al., 2014] that a continuous distribution with the same relative mean and variance, such as a Gaussian, works as well or even better compared to the binary dropout. This was expanded by Wang and Manning [2013] who proposed Gaussian Dropout, where instead of sampling to drop weights, the *activations* are directly drawn from their (approximate or exact) marginal distributions.

Using continuous distribution for dropout began to reveal the relationship between dropout and Bayesian inference. Multiplying the inputs by a Gaussian noise is equivalent to putting Gaussian noise on the weights and can be used to obtain a posterior over the model's weights. This connection was made more clear by Kingma et al. [2015] who presented a re-interpretation of the Gaussian dropout as a variational inference method, and proposed a generalisation that they called "Variational dropout". In doing so, they provided a form of Bayesian justification for dropout by deriving its implicit prior distribution and variational objective. This formulation also allowed them to propose several useful extensions to dropout, such as a principled way of making the typically fixed dropout rates $p$ to be *learned* directly from the training data. By maximising the variational lower bound (ELBO) with respect to $a$ they managed to learn a separate dropout rate per layer, per neuron, or even per separate weight.

Interestingly, the variational inference can also be reinterpreted from an MDL point of view by decomposing ELBO into two terms as per Equation 2.5. When training using VI the emphasise is not on the KL divergence term, and it is often seen as a regularisation term. However, if we look at the objective from the viewpoint of compression, the KL term can be viewed as the primary objective while the data misfit is the part that regularises the compression objective.

$$\begin{aligned} \log p(\mathcal{D}) &\geq \underbrace{\mathrm{E}_{q(\mathbf{w})}\left[\log \frac{p(\mathcal{D}, \mathbf{w})}{q(\mathbf{w})}\right]}_{\text{ELBO } \mathcal{L}(\phi)} \\ &= \underbrace{\mathrm{E}_{q(\mathbf{w})}\left[\log p(\mathcal{D}|\mathbf{w})\right]}_{\text{transmitting data misfit}} - \underbrace{KL\left(q(\mathbf{w}) \| p(\mathbf{w})\right)}_{\text{transmitting the model}} \end{aligned} \tag{2.5}$$

Learning dropout rates directly during training and its interpretation in terms of MDL objective can be directly linked to pruning. By carefully choosing the prior distribution and setting hyper-parameters correctly one could push these learned dropout rates to go higher. This approach

then allows us to prune connections whose learned rates are too high safely. However Kingma et al. [2015] were not very successful in achieving that goal. The authors found that very large values of $a$ correspond to local optima from which it is difficult to escape due to large-variance gradients. In order to avoid such local optima they had to constraint $\alpha \leq 1$ during training, i.e., maximise the posterior variance at the square of the posterior mean, which corresponds to a dropout rate of 0.5. While Variational Dropout learns individual alphas but gradients are too noisy, and rates cannot be too large. The initial version of variational dropout could not push the rates too high because the estimator had high variance.

Molchanov et al. [2017] solved this limitation in their "Sparse Variational Dropout" scheme. Their work builds on top of variational Gaussian Dropout but employs a reparametrisation to replace the multiplicative noise with additive to reduce the variance of the estimator. This reparametrisation greatly reduces the variance of the gradient estimator and therefore allows dropout rates to become unbounded. Additionally, they provide a new approximation of the KL-divergence term in Variational Dropout objective which is tight on the full domain. As a result of these improvements they observed that if they allow variational dropout to proceed with dropping irrelevant weights automatically, it it leads to extremely sparse solutions both in fully-connected and convolutional layers. This is the key work that allowed dropout to be adopted to result in pruned models. This approach reduced the number of model parameters up to 280 times on LeNet architectures, and up to 68 times on VGG-like models with almost negligible loss in accuracy.

Finally, another recent work that builds on top of variational dropout methods is "Bayesian Compression" by Louizos et al. [2017a]. In this work the authors introduce two novelties: firstly they employ *hierarchical* priors (Normal-Jeffreys and Horseshoe) which in turn allows them to drop entire neurons (instead of individual weights) or convolutional kernels altogether. Secondly, they use the uncertainty estimates in the posterior to determine the optimal fixed-point precision required to encode the weights. This means if the model is unsure about a parameter, i.e. the posterior has high variance, then there is little sense is representing this parameter with very high precision. This work is interesting because it covers both sides of Bayesian inference for the purpose of model compression: (1) use prior to induce sparsity (2) sse the posterior to decide quantisation levels.

## 2.5 Single-Shot Pruning

In a recent work (under review in ICLR 2019), Lee et al. [2018] proposed SNIP, a single-step pruning scheme using one mini-batch of the dataset. Unlike most pruning schemes, SNIP attempts to directly measure the importance of *connections*, as opposed to weights. Removing dependency on the weights enables pruning to happen at the initialisation point, where the values of the weights are not known yet. After pruning the obtained sparse network is trained in the standard way.

Classical saliency criterion-based methods often attempt to estimate the sensitivity of the loss

function with respect to the weighs of the connections. The main idea in SNIP is that if the weight of the connection could be separated from whether the connection is present or not, then perhaps the importance of each connection could be directly determined by measuring its effect on the loss function.

In order to derive this new saliency criterion, the paper introduces auxiliary indicator variables. These can be seen as a binary mask representing the presence or absence of each connection. Ideally, we would like to evaluate the influence of each connection by toggling each bit in the auxiliary vector from 1 to 0 and look at the change in loss while keeping all other connections intact. However, this approach is not computationally feasible. SNIP proposes estimating this metric by setting all indicator variables to 1 and then computing the gradient of the auxiliary mask values using one stochastic mini-batch of data. This gradient should not be confused with the gradient with respect to the weights, where the change in loss is measured with respect to an additive change in weight. Instead, using the auxiliary indicator variables, the gradient measures the change in the loss function due to multiplicative perturbation of weights.

The pruning method in SNIP has several attractive properties:

- It is simple, requires only a single training step, and eliminates the need for both pre-training as well as the complex pruning schedule.
- It does not require computing or estimating the Hessian of the loss function. Instead it uses the auto-differentiation functionality already available in all deep learning software frameworks to compute the importance score.
- Typically, applying a particular pruning schemes to another type of architecture requires a great deal of modifications. The method proposed by Lee et al. [2018] is interesting because it can be applied to a wider range of architectures. The authors claim to be the first to demonstrate extreme sparsification on residual, convolutional, and recurrent networks without modifying the pruning algorithm or requiring additional hyper-parameters.

In terms of performance, this approach achieves very completive results compared to some of the more more sophisticated approaches covered earlier in this section. It is also interesting to look at this line of work together in light of the Lottery ticket hypothesis covered earlier in Section 2.6 which suggested what SNIP does is not very difficult.

## 2.6 The Lottery Ticket Hypothesis

In previous sections, it was shown how various compression techniques are able to reduce the number of parameter in a neural network significantly with marginal compromise in terms accuracy. If a network can be so compressed, then the function it has learned can be represented by a much smaller network than the original one. Why, then, do we train large models in the first place when we can get similar performance from much smaller models? Many experiments show that large networks are easier to train than small ones [Han et al., 2015b, Bengio et al., 2006, Hinton et al., 2015, Zhang et al., 2016]. Why is it is difficult to train small models from

scratch?

Finding answers to these questions and understanding the differences between small and large models are crucial for designing more efficient models in the future. A recent work that attempts to answer these questions is the "Lottery Ticket" hypothesis by Frankle and Carbin [2018]. It states that any large network that has been successfully trained contains sub-networks that are initialised in a way that, when trained on their own in isolation, can match the accuracy of the original network in at most the same number of training steps.

These sub-networks, called "winning tickets" by the authors, have won the initialisation lottery, i.e. they have initial weights that make training particularly effective. When re-initialised randomly, the discovered winning tickets can no longer match the performance of the original network. This suggests that the the original initialisation is of great importance.

The paper argues that large networks are easier to train because, when randomly initialised, they contain more combinations of sub-networks from which training can recover a winning ticket. The authors present a series of experiments that support their lottery ticket hypothesis and also demonstrate how pruning techniques uncover the winning tickets predicted by the lottery ticket hypothesis. They show that they can consistently find winning tickets that are less than 20% of the size of various fully-connected, convolutional, and residual architectures for MNIST and CIFAR10 datasets.

## 2.7 Model Distillation

A different line of work in model compression, pioneered by Hinton et al. [2015], attempts to derive smaller models by distilling the knowledge of a large and transferring it into a much smaller model.

The optimisation procedure when training a normal model aims to maximise the average log-probability of the correct class. However, a side-effect of this learning process is that the trained model also assigns probabilities to all of the incorrect classes through a softmax layer. Even when these probabilities are very small, some of them are much larger than others. Hinton et al. [2015] argue a large amount of information the model has learned during training is stored in the soft outputs of the last softmax layer.

Hinton et al. [2015] offered a general solution to use this information called "knowledge distillation". The main idea behind this approach is to use those soft outputs as a ground truth dataset for training a smaller model so that the small model can mimic the large one. The authors also show that it often helps to combine this soft-target dataset with the real one-hot training dataset where the loss function is a linear combination of the two. They also showed that the temperature hyperparameter in the softmax function plays a crucial role in the performance of distillation technique.

## 2.8 Efficient Architectures

The majority of the research in machine learning community has been focused on pushing the envelope in terms of model performance and accuracy. Requirements and implications of deploying these models are often second-class objectives. However, there have been architectures designed from the ground up with efficiency in mind. In this section, we cover two of the main examples of these models.

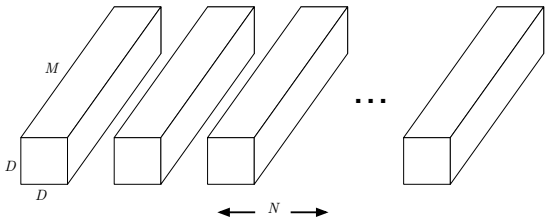
**SqueezeNet** Iandola et al. [2016] proposed a compressed architecture for CNNs called SqueezeNet. This architecture aimed to achieve AlexNet-level accuracy on ImageNet dataset [Krizhevsky et al., 2012] with significantly fewer parameters. The core design decisions made in this architecture were: (1) Replacing 3x3 filters with 1x1 filters. (2) Decreasing the number of input channels to the 3x3 filters. (3) Delaying downsampling the pipeline so that convolution layers have large activation maps.

These decisions resulted in a model that had 50x fewer parameters but still maintained the accuracy level of AlexNet. Moreover, by borrowing compression techniques from DeepCompression [Han et al., 2015a], SqueezeNet was able to compress model size to less than 0.5MB (510x smaller compared to AlexNet). The authors also introduced the notion of "Fire Modules" which consists of: a *squeeze* layer comprising of only 1x1 filters, feeding into an *expansion* layer that has a mix of 1x1 and 3x3 filters.

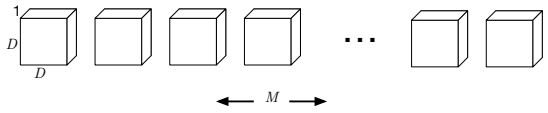
**MobileNet(s)**

Another family of efficient convolutional neural networks is MobileNets. These architectures are small, fast, accurate and come in two versions: MobileNet v1 [Howard et al., 2017] and v2 [Sandler et al., 2018]. The main idea behind the original MobileNet is to split convolutional operation, which is known to computationally expensive, into two sub-routines: a depthwise convolution layer that filters the input, followed by a 1x1 convolution layer (also known as pointwise convolution) that combines the filtered values of the first sub-operator. Together, the depthwise and pointwise convolutions form what authors call a "depthwise separable convolution" layer which performs an approximation of vanilla convolution operation. This approximation leads to 9x less computation in each convolutional layer while maintaining the same accuracy levels. Figure 2.2 depicts different convolution layers in MobileNet.

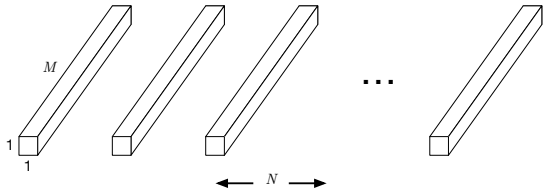
Sandler et al. [2018] slightly modified their architecture in Mobile v2. In the first version of MobileNet the number of pointwise convolution filters was doubled or at least kept the same. In the second version, the authors chose the opposite path and made the number of channels smaller. They called these "projection layers" where high dimensional data is projected into a tensor with much lower number of dimensions. This results in further reduction of the size and the number of Multiply-Accumulate (MAC) operations.




(a) Standard Convolution Filters




(b) Depthwise Convolution Filters




(c) Pointwise 1x1 Convolution Filters

Figure 2.2: Convolution Operators in MobileNet Architectures

## 2.9 Bag-of-Tricks Compression

This section highlights two works that propose frameworks for compressing models via a combination of different approaches rather than an individual algorithm.

A good example of the works in this area is the DeepCompression framework proposed by [Han et al., 2015a]. This architecture consists of a three-stage pipeline: first, a pre-trained model is pruned by learning the importance of connections. This reduces weights by 10x when making use of Compressed Sparse Row (CSR) format. Next, the weights are clustered together using quantisation to enforce weight sharing. After the first two steps network is trained to fine-tune the quantized (to 8 bits or less) centroids of the clusters and the remaining connections. Finally, Huffman coding is applied to compress the model even further. In the case of AlexNet, this framework reduces storage requirements by 35x, from 240MB to 6.9MB without affecting model accuracy. The authors further designed a hardware accelerator called EIE [Han et al., 2016] that operates directly on the compressed model, achieving substantial speedups and energy savings. This framework was successfully used in SqueezeNet resulting in an architecture offering AlexNet performance with a 500x smaller model size.

CNNpack is another framework proposed by Wang et al. [2016]. In this framework, the processing happens in the frequency domain where convolutional filters are treated as images, and their representations in the frequency domain are decomposed into common parts shared among similar filters, and individual unique parts. A large portion of the low-energy frequency components in both parts are then discarded to produce highly-compressed models without significantly

compromising accuracy.

## 2.10 Discussion

While not comprehensvive, the previous sections cover some of the key approaches in neural networks compression. This section finishes the literature review by highlighting some of the gaps and limitation of current compression methods:

**Small Datasets** In the compression literature, pruning methods are often tested on small datasets such as MNIST and CIFAR-10 [LeCun, 1998, Krizhevsky and Hinton, 2009]. However, most of the state-of-the-art models are evaluated on larger datasets such as ImageNet. This is particularly problematic in case of MNIST where data points have black backgrounds, and the pixels are encoded such that black is mapped to zero. This could impact saliency methods that rely on computing the gradient of the loss function with respect to parameters. An easy way to perform further analysis when testing on MNIST is to invert MNIST (make the images black-on-white, not white-on-black) and see if the proposed method still works. Fashion MNIST [Xiao et al., 2017] is another dataset suffering from the same problem.

**Non-Adaptive** The majority of compression techniques require additional hyperparameters, heuristic design choices, long iterative fine-tuning steps or end-to-end training. Less attention has been paid to augmenting the standard training procedure with compression objective such that compression can be applied as a quick, simple, single-step with arbitrary compression rates.

**Vision-centeric** Most of the compression methods have been designed and tested to work with vision-centric datasets and network architectures. While this makes sense given how deep vision models become, a robust compression method should be generic enough to work on other types of neural networks such as recurrent and residual networks.

**Non-Bayesian** As DNNs slowly find their ways into smaller mobile and IoT devices, ensuring robustness and reliability of sensory inference will become essential. Uncertainty estimates from Bayesian models can be crucial for such applications. However, while Bayesian techniques have been used to train compressed models, Bayesian models themselves have not been subject to compression much.

# Bibliography


Sanjeev Arora, Rong Ge, Behnam Neyshabur, and Yi Zhang. Stronger generalization bounds for deep nets via a compression approach. *arXiv preprint arXiv:1802.05296*, 2018.

Yoshua Bengio, Nicolas L Roux, Pascal Vincent, Olivier Delalleau, and Patrice Marcotte. Convex neural networks. In *Advances in neural information processing systems*, pages 123–130, 2006.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

Jia Deng, Wei Dong, Richard Socher, Li jia Li, Kai Li, and Li Fei-fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

John Denker, Daniel Schwartz, Ben Wittner, Sara Solla, Richard Howard, Lawrence Jackel, and John Hopfield. Large automatic learning, rule extraction, and generalization. *Complex systems*, 1(5):877–922, 1987.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

Peter D Grünwald. *The minimum description length principle*. MIT press, 2007.

Masafumi Hagiwara. A simple and effective method for removal of hidden units and weights. *Neurocomputing*, 6(2):207–218, 1994.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015a.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections

for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015b.

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE, 2016.

Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Geoffrey E Hinton and Drew Van Camp. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual conference on Computational learning theory*, pages 5–13. ACM, 1993.

Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

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. *arXiv preprint arXiv:1704.04861*, 2017.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Diederik P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems*, pages 2575–2583, 2015.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

N. D. Lane and P. Warden. The deep (learning) transformation of mobile and embedded computing. *Computer*, 51(5):12–16, May 2018.

Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.

Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, pages 345–353, 2017.

Christos Louizos, Karen Ullrich, and Max Welling. 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., 2017a.

Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through $l\_0$ regularization. *arXiv preprint arXiv:1712.01312*, 2017b.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. *arXiv preprint arXiv:1701.05369*, 2017.

Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

Russell Reed. Pruning algorithms-a survey. *IEEE transactions on Neural Networks*, 4(5):740–747, 1993.

Jorma Rissanen. Stochastic complexity and modeling. *The annals of statistics*, pages 1080–1100, 1986.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

Jocelyn Sietsma and Robert JF Dow. Neural net pruning-why and how. In *IEEE international conference on neural networks*, volume 1, pages 325–333. IEEE San Diego, 1988.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

Karen Ullrich, Edward Meeds, and Max Welling. Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008*, 2017.

Weishui Wan, Shingo Mabu, Kaoru Shimada, Kotaro Hirasawa, and Jinglu Hu. Enhancing the generalization ability of neural networks through controlling the hidden layers. *Applied Soft Computing*, 9(1):404–414, 2009.

Sida Wang and Christopher Manning. Fast dropout training. In *international conference on machine learning*, pages 118–126, 2013.

Yunhe Wang, Chang Xu, Shan You, Dacheng Tao, and Chao Xu. Cnnpack: packing convolutional neural networks in the frequency domain. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 253–261. Curran Associates Inc., 2016.

Peter M Williams. Bayesian regularization and pruning using a laplace prior. *Neural computation*, 7(1):117–143, 1995.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. *CoRR*, abs/1612.01064, 2016.

# Appendix A

# Conference Paper: ICLR 2019

The following is the submission to the International Conference on Learning Representations (ICLRL) 2019. The paper is titled "A Systematic Study of Binary Neural Networks' Optimisation" and provides an empirical study of the effectiveness of the various ad-hoc techniques commonly used in training Binary Neural Networks using.

# A SYSTEMATIC STUDY OF BINARY NEURAL NETWORKS' OPTIMISATION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Neural networks with deterministic binary weights using the Straight-Through-Estimator (STE) have been shown to achieve state-of-the-art results, but their training process is not well-founded. This is due to the discrepancy between the evaluated function in the forward path, and the weight updates in the back-propagation, updates which do not correspond to gradients of the forward path. Efficient convergence and accuracy of binary models often rely on careful fine-tuning and various ad-hoc techniques. In this work, we empirically identify and study the effectiveness of the various ad-hoc techniques commonly used in the literature, providing best-practices for efficient training of binary models. We show that adapting learning rates using second moment methods is crucial for the successful use of the STE, and that other optimisers can easily get stuck in local minima. We also find that many of the commonly employed tricks are only effective towards the end of the training, with these methods making early stages of the training considerably slower. Our analysis disambiguates necessary from unnecessary ad-hoc techniques for the training of binary neural networks, paving the way for future development of solid theoretical foundations for these. Our newly-found insights further lead to new procedures which make training of existing binary neural networks notably faster.

## 1 INTRODUCTION

There is great interest in expanding usage of Deep Neural Networks (DNNs) from running remotely in the cloud to performing local on-device inference on resource-constrained devices (Sze et al., 2017; Lane & Warden, 2018). Examples of such devices are mobile phones, wearables, IoT devices and robots. This is motivated by privacy implications of sharing data and models with remote machines, and the appetite to apply DNNs in new environments and scenarios where cloud-inference is not viable. However, requirements of such devices are very demanding: there are stringent compute, storage, memory and bandwidth limitations; many applications need to work in real-time; many devices require long battery life for all-day or always-on use; and there is a thermal ceiling to consider when designing thin and light devices. On the other hand, the quest for more accurate DNNs have resulted in deeper, more compute-intensive models. This is particularly the case for CNNs. For instance, while the convolutional layers of AlexNet (Krizhevsky et al., 2012) make up only 4% of the model parameters, they are accountable for 91% of the computations at inference time (Louizos et al., 2017).

Compression and efficient implementation of DNNs are therefore more important than ever. There has been a spate of recent work proposing training and post-training schemes that aim to compress models without significant loss in their performance. Main examples of these techniques are: pruning, weight sharing, low-rank approximation, knowledge distillation and perhaps most importantly quantisation to lower precisions (Han et al., 2017; Ullrich et al., 2017; Hinton et al., 2015). Quantisation is widely used in commercial deployments and its trade-offs and performance improvements for CNNs is well-studied in the lirerature (Krishnamoorthi, 2018). One appealing training-time quantisation scheme (Courbariaux et al., 2015) pushed it to the extreme, by representing each weight with a single bit, while maintaining respectable model accuracy. This paved the way for emergence Binary Neural Networks (BNNs). Courbariaux et al. (2016) and Rastegari et al. (2016) expanded BNNs by using the $\mathrm{sign}$ function as the non-linearity to achieve binary activations in addition to binary parameters. With this approach full-precision MAC operations in convolution layers can be

replaced with cheap XNOR and POPCOUNT binary operations. This results in $58\times$ (Rastegari et al., 2016) improvement in compute-time in addition to the inherit $32\times$ saving in model size that comes from replacing 32-bit floating point parameters with binary ones.

However, as we will describe in Section 2, the common optimisation process used in BNNs is still not fully understood. Moreover, state-of-the-art binary models employ various modifications to conventional training settings in order to squeeze the best performance from the models. Some examples of these modifications are: applying constraints to weights and gradients, changing typical order of operations in a convolutional block, scaling learning rates based on Xavier (Glorot & Bengio, 2010) initialisation values, learning additional parameters for affine transformations of kernels, changing momentum hyper-parameters in Batch Normalisation (Ioffe & Szegedy, 2015) and the choice of optimiser, loss function, learning rate and number of training epochs. In the absence of rigorous mathematical understanding as of yet, it is imperative to empirically study the sensitivity of the optimisation process and the performance of BNNs to these settings and tweaks. Such empirical understanding of the tools will greatly aid any development of solid mathematical foundations for the field. To that end, the main contributions of this work are as follows:

- We identify the essential techniques required for successful optimisation of binary models and show that end-to-end training of binary networks crucially relies on the optimiser taking advantage of second moment gradient estimates.
- We show that most of the commonly used tricks in training binary models, such as gradient and weight clipping, are only required during the final stages of the training to achieve the best performance. Further, we demonstrate that these tricks lead to much slower convergence in the early stages of optimisation.
- We propose new procedures for training, making optimisation notably faster by delaying these tricks, or by training a full-precision model first and fine-tune it into a binary model.
- We provide our reference implementations and training-evaluation source code online [1].

## 2 BACKGROUND

Early quantised models were derived by quantisating full-precision weights of pre-trained models (Gong et al., 2014). This approach is widely used in real deployments and enjoys advantages such as flexibility to apply different levels of quantisation based on the target model size, and not requiring knowledge of model internals. However, it suffers from significant loss of accuracy. Hubara et al. (2017) showed that in order to maintain model performance, quantisation must be incorporated as part of the training process. This is done by either performing additional training steps to fine-tune a quantised model or by directly learning quantised parameters. This is essential for BNNs where binarising weights of pre-trained models result in significant loss in accuracy.

The first successful binarisation-aware training method was proposed in BinaryConnect by Courbariaux et al. (2015). In their work, the binary weights are not learned directly; instead, full-precision weights are maintained and *learned* during the training as proxies for the binary weights. These proxies are only required during training. During the forward path, binary weights are computed by applying $\mathrm{sign}$ function to their corresponding full-precision proxies. Since the $\mathrm{sign}$ function is not differentiable BinaryConnect employs Straight-Through-Estimator (STE) (Bengio et al., 2013) for back-propagating gradient estimates to full-precision proxies. The STE estimator simply passes the gradients along as if the non-differentiable operator was not present. In practice, BinaryConnect applied two additional restrictions on vanilla STE: (1) *Gradient clipping* stops gradient flow if the weight's magnitude is larger than 1. This effectively means gradients are computed with respect to $\mathrm{hard}\tanh$ function. (2) *Weight clipping* is applied to weights *after* gradients have been applied to keep them within a range. To formalise this consider $w_r$ to be a full-precision proxy for binary weight $w_b$. During the forward path (and at the end of the training):

$$w_b = \mathrm{sign}(w_r)$$

STE with gradient clipping provides an estimate for gradient of this operation:

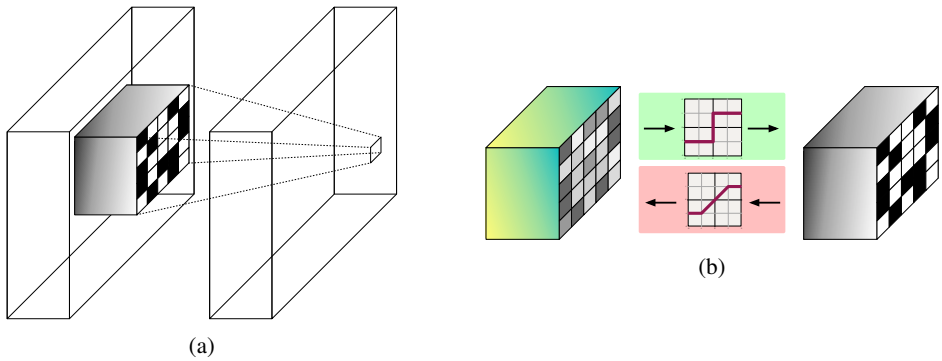
[1] Anonymous GitHub repository

$$\frac{\partial w_b}{\partial w_r} = \mathbf{1}_{|r|\leq 1} \tag{1}$$

In a back-propagation context we assume the gradient of the cost $C$ at the output ($\frac{\partial C}{\partial w_b}$) is available where in computing it the same STE estimator above has been used wherever required. Eq 1 enables us to estimate the gradient of the cost at the input ($\frac{\partial C}{\partial w_r}$) and update the proxies:

$$\frac{\partial C}{\partial w_r} = \frac{\partial C}{\partial w_b}\mathbf{1}_{|r|\leq 1}$$

The estimator passes gradients backwards unchanged when proxies are within the {-1,1} range and cancels the gradient flow when the proxy weight has got too positive or too negative. Figure 1 depicts how this works for a convolutional kernel in a CNN.



(a) (b)

Figure 1: A convolutional kernel in a Binary Neural Network is binary (left) but its values are derived from a full-precision proxy learned using using the STE estimator (right). At the end of the training the proxy kernel is used for one last time to compute final binary values.

There has been several extensions to BinaryConnect's core idea of using STE estimator in binary models. BinaryConnect showed slightly better results when STE was used in stochastic binary neurons. BinaryNet (Courbariaux et al., 2016) used binary activations in addition to binary parameters (as described in Section 1) and made the convolution operation more efficient by using a custom GPU kernel. XNOR-Net and BWN (Rastegari et al., 2016) managed to scale up BNNs to achieve competitive results on the much bigger ImageNet (Deng et al., 2009) dataset by learning additional full-precision scale factors per-layer. DoReFa-Net (Zhou et al., 2016) used STE in the backpropagation path to quantise gradients and achieve faster training. 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. Lin et al. (2017) achieved state-of-the-art performance by learning a combination of very few binary kernels in each layer.

## 3 A Systematic Study of Existing Methodologies in BNNs

There have been several non-empirical attempts to formalise STE and BNNs. Anderson & Berg (2018) took a high-dimensional geometric point-of-view to justify existence of binary solutions irrespective of the optimisation process. Li et al. (2017) provided accuracy guarantees for training binary models under convexity assumptions. However, STE still has not been shown to find the solution of any particular loss function. In the meantime, binary models are achieving acceptable levels of accuracy in practice. Table 1 lists some of the recent binary architectures and their commonly-used training setup.

In this section, we provide an empirical analysis of the main approaches used in these models and help the researchers and practitioners navigate this space. We explore two classes of architectures in our study of binary networks: A CNN inspired by VGG-10 (Simonyan & Zisserman, 2015) on CIFAR-10 (Krizhevsky & Hinton, 2009) dataset and an MLP with three hidden layers with 2048

Table 1: Recent binary architectures and their training setup. The *Reorder* column refers to reordering of blocks in a convolutional layer to make sure pooling layer's input is full-precision. The *1st Layer* column indicates whether the first layer of the network is binary or kept at full precision.

| Network | Optimiser | STE | Clipping | Reorder | 1st Layer | Activation |
|---|---|---|---|---|---|---|
| BinaryConnect | Adam | Yes | Weights & Gradients | Yes | Binary | 32-bits |
| BinaryNet | AdaMax | Yes | Weights & Gradients | Yes | FP | 1-bit |
| XNOR-Net | Adam | Yes | Gradients | Yes | FP | 1-bit |
| BWN | Momentum | Yes | Gradients | Yes | - | 32-bits |
| DoReFa-Net | Adam | Yes | Gradients | Yes | FP | $\geq$1-bit |
| ABC-Net | Momentum | Yes | Gradients | Yes | Binary | $\geq$1-bit |
| HBN | Adam | Yes | Gradients | Yes | FP | $\geq$1-bit |
| Bulat et al. (2017) | RMSprop | Yes | Gradients | Yes | FP | 1-bit |
| Cai et al. (2017) | Momentum | Yes | Gradients | Yes | FP | $\geq$2-bits |
| Xiang et al. (2017) | AdaMax | Yes | Gradients | - | FP | 1-bit |

units and rectified linear units (ReLUs) for MNIST (LeCun, 1998) dataset. We make use of gradient and weight clipping and squared hinge loss unless stated otherwise. We use the last 10% of the training set for validation and report the best accuracy on the test set associated with the highest validation accuracy achieved during training. The results shown are the average of five runs. We have not used early stopping or finite time budget in any of the experiments.

The remainder of this section is organised as follows: We first show that the choice of optimiser matters considerably. We then show the impact of clipping gradients and weights followed by batch normalisation hyper-parameters on convergence speed and accuracy of BNNs. We finish by testing the effectiveness of some of the other commonly used tweaks used in training binary models.

### 3.1 Impact of Optimiser

The majority of recent binary models use an adaptive optimiser in their implementations: BinaryConnect uses ADAM (Kingma & Ba, 2015) for CIFAR-10 and vanilla SGD for MNIST (although in their released source code they used ADAM for both datasets), DoReFa-Net and XNOR-Net use ADAM in their experiments and ABC-Net (Lin et al., 2017) uses SGD with momentum. In this section, we show that this is not accidental and investigate how the optimiser type and its associated hyper-parameters affects the viability of the STE estimator.

For experiments in this section, we looked at optimisers from four classes in order of increasing complexity: (1) history-free optimisers such as mini-batch SGD that do not take previous jumps or gradients into account, (2) momentum optimisers that maintain and use a running average of previous jumps such as Momentum and Nesterov (Sutskever et al., 2013), (3) Adaptive optimisers that adjust learning rate for each parameter separately such as AdaGrad (Duchi et al., 2011) and AdaDelta (Zeiler, 2012), and finally, (4) optimisers that combine elements from categories above such as ADAM which combines momentum with adaptive learning rate.

Table 2 summarises the best accuracies we achieved using different optimisers. We ran experiments for more epochs than typically required for the datasets (up to 500 epochs depending on the experiment). In each experiment the relevant hyper-parameters were tuned for best results. We observed great variance in convergence speed and model performance as a result of optimiser choice that goes beyond differences seen when training non-binary models. Our first observation is that vanilla SGD generally fails in optimising binary models using STE. We note that reducing SGD's stochasticity (by increasing batch size) improves performance initially. However, it still fails to obtain the best possible accuracy. SGD momentum and Nesterov optimisers perform better than SGD when they are carefully fine-tuned. However, they perform significantly slower compared to optimsing non-binary models and have to be used for many more epochs than normally used for CIFAR-10 and MNIST datasets. Similar to SGD, increasing momentum rate improves training speed significantly but results in worse final model accuracy. In Appendix A we include results for the equivalent non-binary models that show the effect of batch size and momentum are far less substantial.

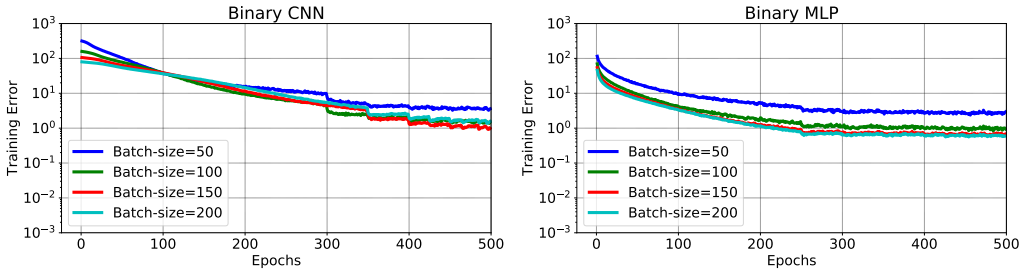
A possible hypothesis is that early stages of training binary models require more averaging for the optimiser to proceed in presence of binarisaton operation. On the other hand, in the late stages of the

training we rely on noisier sources to increase exploration power of the optimiser. This is reinforced by our observation that binary models are often trained long after the training or validation accuracy stop showing improvements. Reducing the learning rate in these epochs does not improve things either. Yet, the best validations are often found in these epochs. In other words, using early stopping for training binary models would terminate the training early on and would result in suboptimal accuracies.

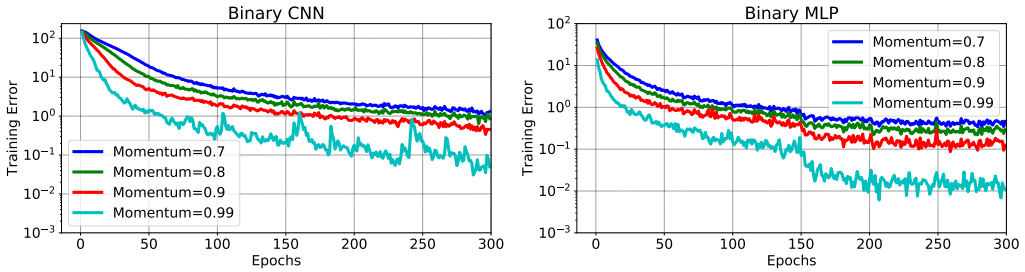
Finally, adaptive optimisers, and specifically ADAM, consistently perform faster and able to achieve better accuracy levels. We experimented with different hyper-parameters in ADAM optimiser (see Figure 2c) and found the decay rate for the *second* moment estimate to play a significant role.

Table 2: Achievable test errors using different optimisers for binary MLP model trained on MNIST and binary CNN model train on CIFAR-10. The hyper-parameters of each optimiser were fine-tuned for best results.

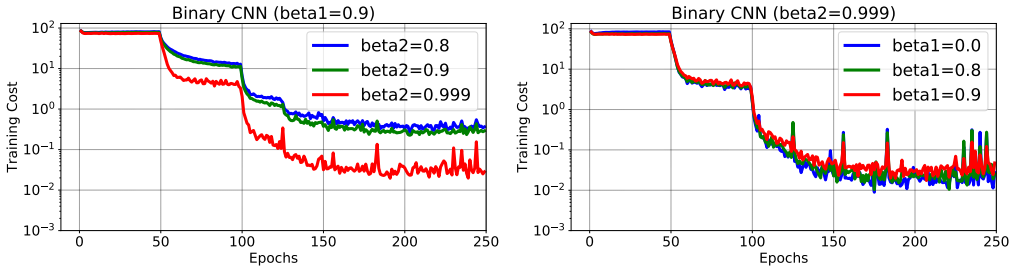
| | SGD | Momentum | Nesterov | AdaGrad | AdaDelta | RMSProp | ADAM |
|---|---|---|---|---|---|---|---|
| MNIST | 4.48% | 1.87% | 1.86% | 1.28% | 1.22% | 1.21% | **1.19%** |
| CIFAR-10 | 17.98% | 12.41% | 12.42% | 10.87% | 10.34 % | 10.33% | **10.30%** |




(a) SGD for Binary CNN and MLP




(b) Momentum SGD for Binary CNN and MLP




(c) ADAM for Binary CNN

Figure 2: Convergence speeds of two binary models trained with different optimisers. We found ADAM to be consistently faster in training BNNs compared to other optimisers. Figure (c) shows effect of various momentum rates for ADAM's first and second moment estimates on convergence of BNNs.

### 3.2 Impact of Gradient and Weight Clipping

The STE variant used in BinaryConnect, XNOR-Net, and most other binary models, is different from vanilla STE introduced by Bengio et al. (2013). In these models the STE stops gradient flow to proxies when the full-precision weights have grown beyond $\pm 1$. Additionally, BinaryConnect clips weights *after* gradient updates have been applied to keep weights within range. Our experiments (summarised in Table 3) show that this technique does indeed result in slight improvements in the accuracy of binary models. We observed 0.07% and 0.54% improvement for MNIST and CIFAR-10 datasets respectively. Clipping weights does generally help when it is combined with gradient clipping but is less effective on its own. In our experiments placing these additional constraints had negligible effects on speed of SGD or Momentum based optimisers. However, ADAM is sensitive to these constraints. We will revisit clipping in Section 4 where we study them again in terms of optimising convergence speed.

Table 3: Impact of gradient and/or weight clipping on the final test accuracy of BNNs.

| Clipping | None (Vanilla STE) | Weights | Gradients | Both |
|---|---|---|---|---|
| MNIST | 1.28% | 1.22% | **1.17%** | 1.18% |
| CIFAR-10 | 10.79% | 10.73% | 10.53% | **10.38**% |

### 3.3 Impact of Batch Normalisation

Batch normalisation (BN) uses mini-batch statistics during training but at inference-time the model is classifying a single data point. Therefore, each BN layer maintains a running average of mini-batch statistics to use during inference. The default momentum rate for this running average is usually large, e.g. 0.99. We noted that some binary models use smaller values for this hyper-parameter. Binary models are typically trained for more epochs than their non-binary counterpart and training is continued even when there is not a meaningful improvement in loss or accuracy. This is consistent with our earlier hypothesis in 3.1. Reducing the momentum rate in BN can help to cancel the effect of long training. The effect is small but consistent. Table 4 shows how different values of BN momentum results in different test accuracies. Krishnamoorthi (2018) also observed that Batch normalisation should be handled differently when training quantised models in order to achieve better performance.

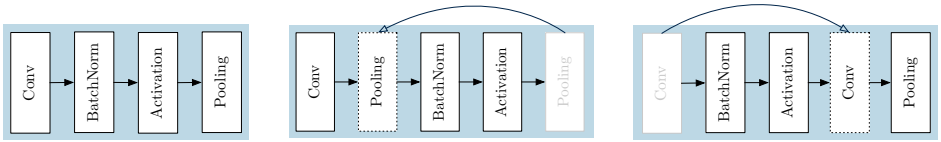
Table 4: Impact of momentum rate in Batch Normalisation's moving average on the final test accuracy of BNNs.

| Momentum | 0.8 | 0.85 | 0.9 | 0.99 |
|---|---|---|---|---|
| MNIST | 1.21% | **1.19%** | 1.22% | 1.23% |
| CIFAR-10 | **10.31%** | 10.35% | 10.53% | 10.61% |

### 3.4 Impact of Pooling and Learning Rate

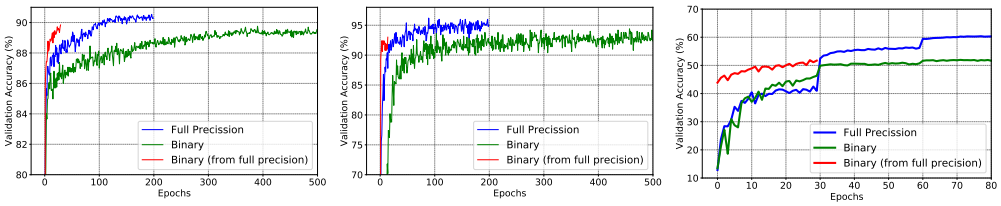
**Reordering Pooling Block.** As can be seen in Table 1 all binary models change the placement of pooling operation within a convolutional layer. This change makes sense intuitively. For instance, applying MaxPooling to a binary vector results in a vector with almost all ones. We have seen two variants of block reordering and in both cases (see Figure 3) pooling is done immediately after the convolution operator where the vector is *not* binary. In our experiments, not making this change resulted in significant accuracy loss.

**Learning Rate Scaling using Xavier.** In BinaryConnect Courbariaux et al. (2015) propose scaling learning rates of each convolutional or fully connected layer by the inverse of Xavier initialisation's variance value. The same value is also used as the range in weight clipping after gradient update. They report noticeable accuracy improvements using these techniques. This modification is interesting because it suggests STE estimator requires an additional dimension. Applying this change effectively makes the slope of the line between $-1$ and $+1$ (see Figure 1b) directly proportional to square root of (Fan-In + Fan-Out) of each layer. In our experiments, this approach helped when used




(a) (b) (c)

Figure 3: Changing the order of pooling operation within a convolutional block is necessary when training binary models.




(a) VGG-10 (b) ResNet-18 (c) AlexNet-like on ImageNet

Figure 4: A binary model (red) is initialised from a full precision model (blue) and reaches top accuracy in a fraction of the epochs that would require to train a binary model (green) end-to-end.

with SGD but we did not see any impact when used with other optimisers. We were able to replicate accuracy levels reported by BinaryConnect without using this technique.

## 4 Training BNNs Faster: Empirical Insights put into Practice

We continue in this section by applying a number of our empirical observations towards optimising BNNs in a faster and more efficient manner. We believe this demonstrates some of the practical implications of our results described earlier that are still to be explored.

In this case-study we consider the well-known observation that training a binary model is often notably slower than its non-binary counter-part, the reasons for which are not well understood. One reason typically cited is that binarisation hinders the use of large learning rates – relative to those adopted in full precision networks. Our experiments show that counter to the conventional wisdom the STE on its own does not affect the training speed of BNNs considerably. The slowdown is mainly caused by the commonly applied gradient and weight clipping, as they keep parameters within the $\{-1,1\}$ range at all times during training. Figure 5 shows how disabling one or both of these clipping schemes affects the training curve of a binary CNN. It can be seen that not clipping weights when learning rates are large can completely halt the optimisation (red curve in Figure 5). On the other hand, using vanilla STE brings the training speed back on par with the non-binary model. This is particularly true for ADAM.

However, this faster convergence comes at the price of a loss in accuracy (see Section 3.2). While weight and gradient clipping help achieve better accuracy, our hypothesis is that they are only required in the later stages of training where the noise added by clipping weights and gradients increases the exploration of the optimiser. We tested this hypothesis by training a binary model in two stages: (1) using vanilla STE in the first stage with higher learning rates and (2) turning clippings back on when the accuracy stops improving by reducing learning rate.

With vanilla STE the gradients are simply passed along to the full-precision proxies and the model is optimised as if the binarisation operations were not present. This, combined with results above, prompts the question whether it is even necessary to apply binarisation from the very beginning of the training. While it might be conceptually attractive to train binary models end-to-end, we are still learning full-precision structures during training. One can define and train an equivalent non-binary model where the binarisation operations are removed. This is useful because in many cases this model is already available. This pre-trained model can then be used to initialise the values of full-precision proxies in the binary model. The model can then be trained using STE and gradient

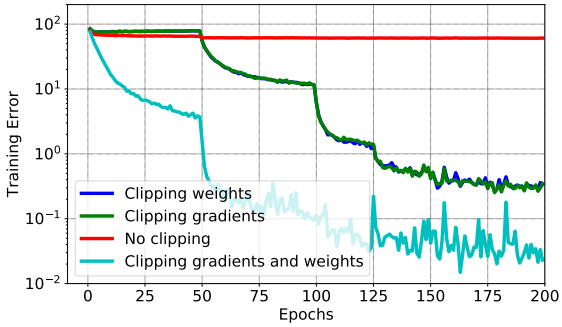



Figure 5: Impact of gradient and weight clipping on convergence speed of binary VGG-10 with large learning rates (0.1).

clipping. Our experiments (see Figure 4) show that this works equally well in terms of accuracy but converges considerably faster for ResNet-18 (He et al., 2016) and VGG-10 architectures than if we had trained these binary models end-to-end. Mishra & Marr (2018) reported similar results.

There is a significant loss in accuracy when this model is binarised for the first time. This can be seen in the starting point of the *Binary (from full precision)* curves in Figure 4. This shows once again why we cannot simply binarise a pre-trained non-binary model and expect it to work well. However, we noted that the number of training steps required to recover the accuracy is very small. This result is encouraging because it turns the problem of learning binary models into a fine-tuning stage that can be applied to available pre-trained models.

It is important to note that while we can quickly get to the point where training and validation accuracies stagnate, there is a small gap between the achieved accuracy and the best possible one. This gap can only be filled by continuing training for many epochs. This difference is often consistent with the gap we observe between (a) the best test accuracy when training for many epochs and (b) the first epoch where validations accuracy stops improving. This reinforces our earlier hypothesis in Section 3.1 that suggests the last mile of model performance has little dependence on the STE's capability and mostly relies on stochastic exploration of the parameter space.

Table 5: Training binary models using pre-trained full-precision models for CIFAR-10 (ResNet-18 and VGG-10) and ImageNet (AlexNet-like) datasets.

| | Binarisation | Best Validation Accuracy | Test Accuracy |
|---|---|---|---|
| Binary ResNet-18 | end-to-end | 94.40% (in epoch 457) | 91.16% |
| | from full-precision | 93.60% (in epoch **17**) | 91.18% |
| Binary VGG-10 | end-to-end | 89.76% (in epoch 391) | 89.18% |
| | from full-precision | 90.16% (in epoch **24**) | 89.32% |
| Binary AlexNet-like | end-to-end | 51.98% (in epoch 88) | — |
| | from full-precision | 51.85% (in epoch **30**) | — |

## 5 Conclusion and Best-Practices

In this work we study the landscape of binary neural networks and evaluate the impact of various techniques on the accuracy and convergence performance of binary models. We show that training binary models is harder and slower than the equivalent non-binary model. Our empirical study suggests that while the limit of STE's capability can be achieved easily, finding the best set of weights requires longer training. For efficient training of Binary models we recommend: (1) using ADAM for optimising the objective, (2) not using early stopping, (3) splitting the training into two stages, (4) removing gradient and weight clipping in the first stage and (5) reducing the averaging rate in Batch Normalisation layers in the second stage.

## REFERENCES


Alexander G. Anderson and Cory P. Berg. The high-dimensional geometry of binary neural networks. In *International Conference on Learning Representations*, 2018.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Adrian Bulat et al. Binarized convolutional landmark localizers for human pose estimation and face alignment with limited resources. In *International Conference on Computer Vision*, 2017.

Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *CVPR*, 2017.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pp. 3123–3131, 2015.

Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

Jia Deng, Wei Dong, Richard Socher, Li jia Li, Kai Li, and Li Fei-fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations*, 2017.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, June 2016.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

Sergey Ioffe and Christian Szegedy. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*, pp. 448–456. JMLR. org, 2015.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

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

N. D. Lane and P. Warden. The deep (learning) transformation of mobile and embedded computing. *Computer*, 51(5):12–16, May 2018.

Yann LeCun. The "MNIST" database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. Training quantized nets: A deeper understanding. In *Advances in Neural Information Processing Systems*, pp. 5811–5821, 2017.

Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, pp. 345–353, 2017.

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

Asit Mishra and Debbie Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. In *International Conference on Learning Representations*, 2018.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pp. 525–542. Springer, 2016.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pp. 1139–1147, 2013.

Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

Karen Ullrich, Edward Meeds, and Max Welling. Soft weight-sharing for neural network compression. In *International Conference on Learning Representations*, 2017.

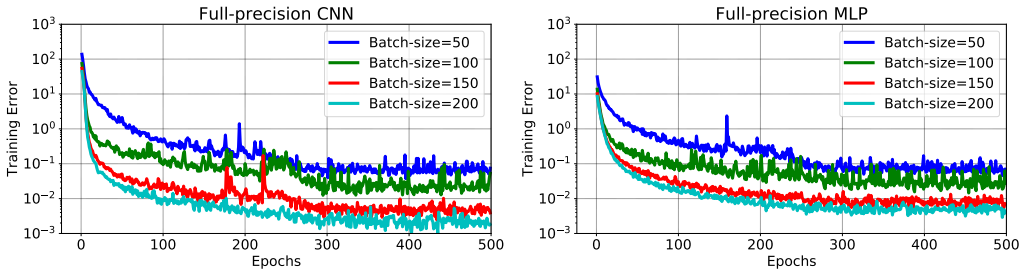
Xu Xiang, Yanmin Qian, and Kai Yu. Binary deep neural networks for speech recognition. In *Proc. Interspeech 2017*, pp. 533–537, 2017.

Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

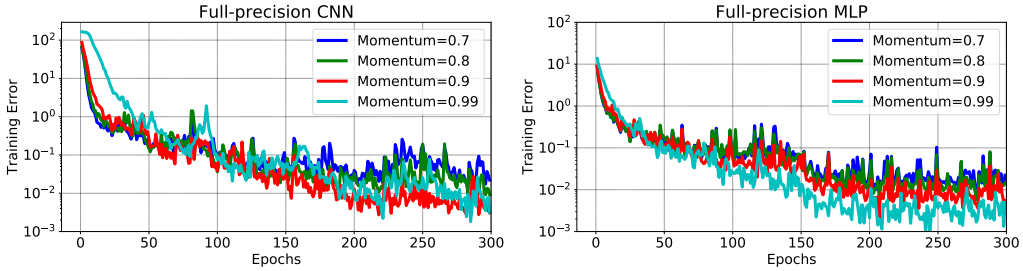
Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.

Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. *CoRR*, abs/1612.01064, 2016.

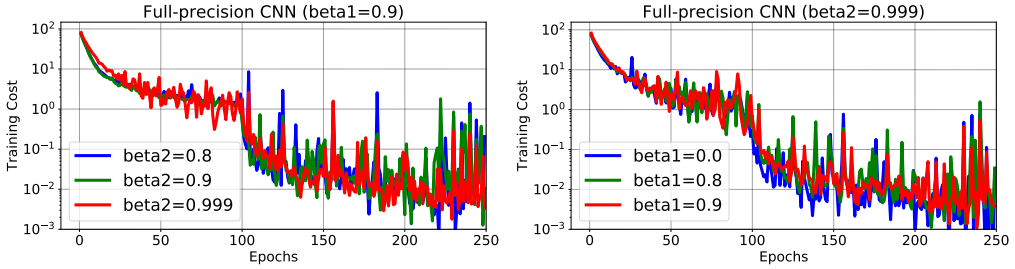
# A IMPACT OF OPTIMISERS IN NON-BINARY MODELS




(a) SGD for Non-Binary CNN and MLP




(b) Momentum SGD for Non-Binary CNN and MLP




(c) ADAM for Non-Binary CNN

Convergence speeds of two full-precision models trained with different optimisers.

# Appendix B

# Conference Poster: MobiSys 2018

The following are the accepted abstract and poster presented at the ACM International Conference on Mobile Systems (MobiSys) 2018. This work includes some of the early studies that later led to submission to ICLR 2019 conference A.

# Poster: Using Pre-trained Full-Precision Models to Speed Up Training Binary Networks For Mobile Devices

Milad Alizadeh[†], Nicholas D. Lane[†*]
[†]University of Oxford, [*]Nokia Bell Labs

## ABSTRACT

Binary Neural Networks (BNNs) are well-suited for deploying Deep Neural Networks (DNNs) to small embedded devices but state-of-the-art BNNs need to be trained from scratch for a long time. We show how weights from a pre-trained full-precision model can be used to speed-up training of binary networks. We show that for CIFAR-10, accuracies within 1% of the full-precision model can be achieved in just 5 epochs.

## CCS CONCEPTS

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

## 1 INTRODUCTION

In recent years DNNs have achieved great success on many tasks across various domains such as computer vision, speech recognition and machine translation [1, 5, 6]. However, less attention has been paid to practical requirements and trade-offs of deploying these models to real-time embedded platforms where there are stringent requirements in terms of available compute power, memory footprint and energy consumption.

Binary networks are suitable candidates for such platforms as binaries are cheap to store and operations involving binary operands are cheap to compute. Initial attempts [4] to derive quantized networks by post-processing a full-precision model suffered from significant loss of accuracy. But it has been recently shown [2, 3, 7] that much better performance can be achieved by training binary networks *end-to-end* instead of post-processing a trained network. The best results are achieved when binary parameters are learned indirectly through full-precision proxies. Binary parameters are derived from these proxies during the forward and backwards paths to respectively compute model predictions and gradients. During the training, the computed gradients are backpropagated and applied to the full-precision proxies.

## 2 PRE-TRAINING FOR BNNS

We show that using full-precision weights from a trained model as initialisation values of full-precision proxies in the equivalent binary network can significantly speed up the training process. While this does not circumvent the need for training stage it has benefits for mobile scenarios:

- Customizing models for different devices and network environments would be valuable at the edge (e.g. cloud-let levels) in reaction to devices coming into range but doing so is not practical without speeding up training.
- Many pre-trained full-precision models are already available for use and can be adopted for mobile applications by replacing training with a quick fine-tuning process.

## 3 EVALUATION

Figure 1 shows the validation accuracies on CIFAR-10 dataset for three scenarios with architectures identical to [2]. The experiments are done using TensrFlow. Training a BNN from scratch is slower than the equivalent full-precision model but using the full-precision values for initialisation boosts the accuracy and achives good performance in few epochs.

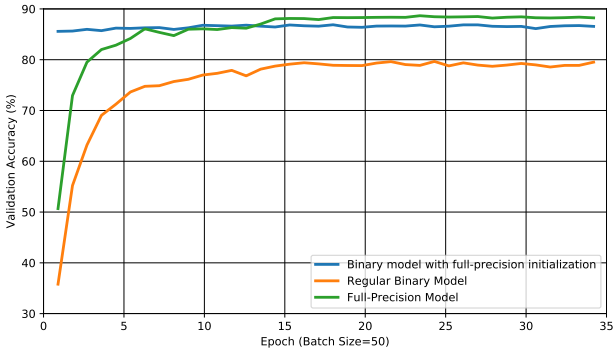



**Figure 1: Validation Accuracy on CIFAR-10**

## REFERENCES

[1] Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[2] Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems* (2015), pp. 3123–3131.
[3] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).
[4] Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[5] Hinton, G., et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine 29*, 6 (2012), 82–97.
[6] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.
[7] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision* (2016), Springer, pp. 525–542.


*MobiSys '18, June 10–15, 2018, Munich, Germany*

ACM ISBN 978-1-4503-5720-3/18/06.
https://doi.org/10.1145/3210240.3210821

# Using Pre-Trained Full-Precision Models to Speed Up Training of Binary Neural Networks

**Milad Alizadeh, Nicholas Lane, Yarin Gal**
Department of Computer Science, University of Oxford
{milad.alizadeh, nicholas.lane, yarin.gal} @ cs.ox.ac.uk

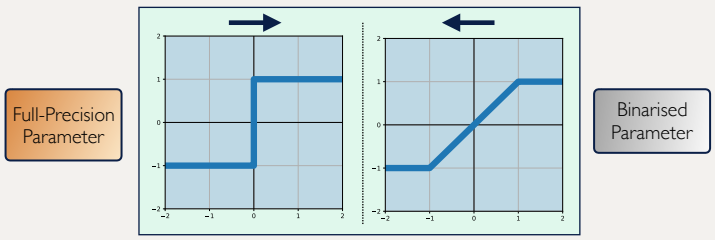
UNIVERSITY OF OXFORD
DEPARTMENT OF COMPUTER SCIENCE
EPSRC Engineering and Physical Sciences Research Council

## TL;DR

- Binary Neural Networks (BNNs) are **great candidates for mobile applications** but best-performing BNNs need to be trained end-to-end. This process is **noisy and slow**.
- The corresponding full-precision network can be used at **initialisation** or **during training** to significantly speed-up and guide the training of the binary network.
- For CIFAR-10, our approach reaches **87% accuracy in just 5 epochs** compared to 80% accuracy in 30 epochs when initialised with XAVIER.

## Background

- Early attempts to derive quantised networks by post-processing a full-precision model suffered from significant loss of accuracy.
- Much better accuracies were achieved by training quantised/binary networks **end-to-end**.
- The best results are achieved when binary parameters are learned **indirectly** through **full-precision proxies**. Binary parameters are derived from these proxies during the forward and backwards paths to respectively compute model predictions and gradients.
- During the training, the **gradients are backpropagated and applied to the full-precision proxies**.
- Because the sign( ) functions is **non-differentiable** it is replaced with a relaxed continuous version for the backwards path. This is known as **Straight-Through-Estimator (STE).** This estimator is biased, noisy and slow.

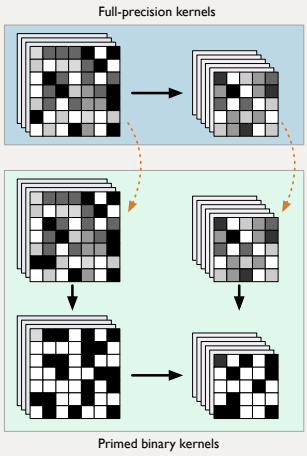



## Applications in Mobile

- Real-time embedded system have stringent requirements in terms of **compute power**, **memory** footprint and **energy** consumption.
- Binaries are **cheap to store**. When binary parameters are combined with **binary activation** as non-linearity, MAC ops in CNNs are reduced to **XOR and POPCNT** instructions.
- Quantising models for different devices and network environments (e.g. **cloud-let levels**) when devices coming into range is desirable but **not practical without speeding up training**.
- Many pre-trained full-precision models are already available for use and can be adopted for mobile applications by replacing training with a **quick fine-tuning process**.

## Distilling Knowledge from Full-Precision Model

- When the associated full-precision model is available one can apply **knowledge distillation** techniques where the full-precision network is the **teacher** and the binary network is the **student**.
- Full-precision proxies in the student network can be **primed** with values from the teacher network at the beginning of the training.
- Outputs from the teacher model can also be used as an **additional dataset with soft-labels**.




## Results

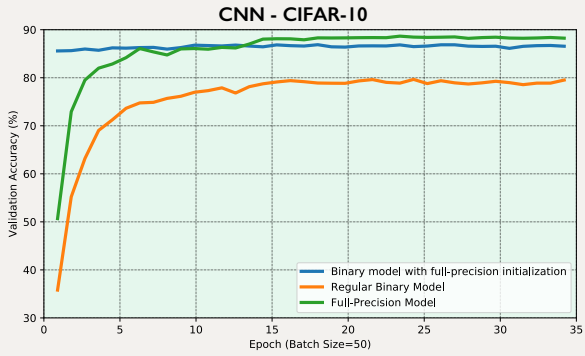
Two scenarios:

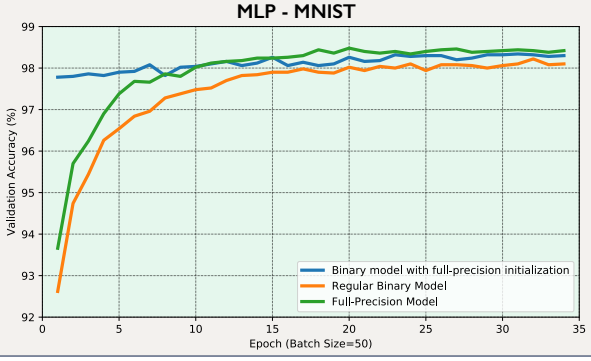
1. MLP model on MNIST (3 hidden layers of 1024 ReLU units)
2. CIFAR-10 using CNN (4 convolutional layers, 2 max-pooling)

For each scenario trained 3 models:

1. Binary model
2. Associated full-precision model
3. Binary model primed by full-precision model from the last step

For CIFAR-10, our approach reaches 87% accuracy in 5 epochs compared to 80% accuracy in the 30 epochs when initialised with XAVIER.




[1] Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In Advances in neural information processing systems (2015).
[2] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1(2016).
[3] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In European Conference on Computer Vision (2016).

# Appendix C

# Collaboration: IJCAI 2018

The following paper was accepted in the International Joint Conferences on Artificial Intelligence (IJCAI) 2018 and is a collaboration with Nokia Bell Labs in Cambridge, UK and Cornell University, US.

Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)

# Deterministic Binary Filters for Convolutional Neural Networks

**Vincent W.-S. Tseng**[‡], **Sourav Bhattachara**[†], **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 mentinoned 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 $1 \times 1$ filters in combination with $3 \times 3$ 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 $1 \times 1$ 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-

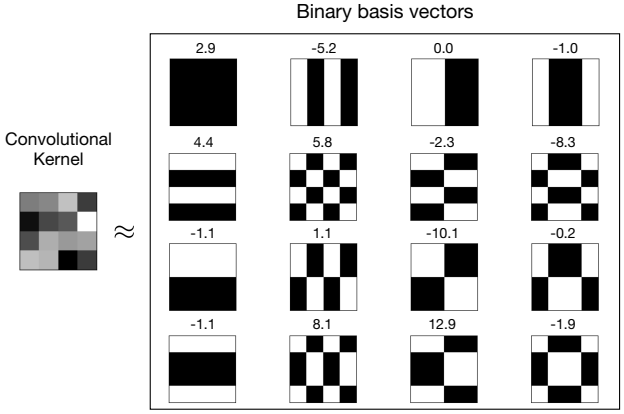



Figure 1: Overview of representing a convolution kernel using a set of binary mutually orthogonal basis vectors. The convolutional kernel (on the left) can be represent accurately using linear superposition of all the binary vectors (patches) presented on the right. The coefficient or strength of a binary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

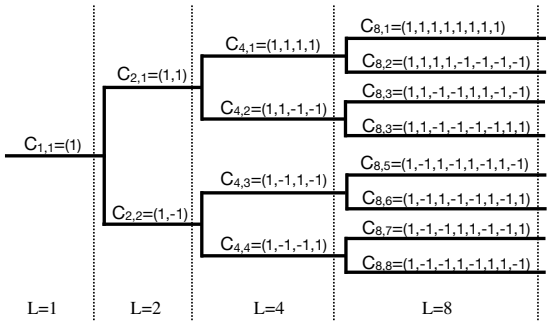



Figure 2: Code Tree for OVSF Code Generation

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, $\alpha_i$ 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 $4 \times 4$ 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 $3D$.

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 $2^l$.

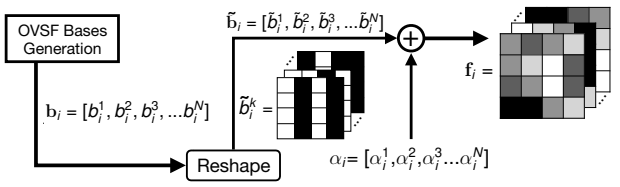



Figure 3: From OVSF codes to DBFs. Each code $b_i^k$ is first reshaped to match the final filter dimensions, becoming $\tilde{b}_i^k$. Then, the reshaped codes in $\tilde{b}_i$ are combined using the weights $\alpha_i$.

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 $l = WHC$, $\boldsymbol{B_k^i}$ is the $i$th OVSF code and $\alpha_j^i \in \mathbb{R}$ its associated weight. $\epsilon$ 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 $\rho$ 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 $3 \times 3$ or $5 \times 5$ 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 $\eta$.
**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 $\alpha_k^i$ 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 $3 \times 3$ or $5 \times 5$. 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

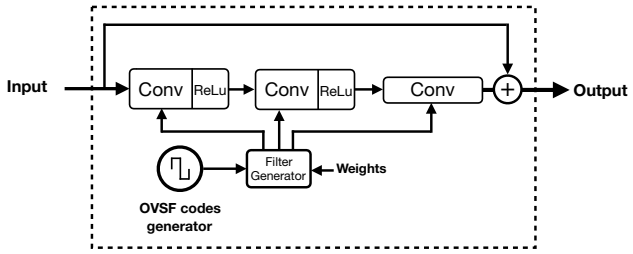



Figure 4: A three layers DBF Module.

as $\rho$. Lower $\rho$ 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 $\rho$ effectively reduces the number of weights, $\alpha$ 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 $\rho$ 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 $\rho$. This parameter can be set independently for each convolutional layer and is used to determine the number of OVSF codes a generator should output 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 $\rho$ for deeper convolutional layer has a lesser impact on accuracy than in the first layers of the network. Concretely, $\rho$ 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 | 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.

| 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 $\rho$ 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 $\rho$ values (%). Accuracy values are shown in Table 2.

To our advantage, $\rho$ 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 $\rho$ 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 $\rho$ 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 $\rho$ 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 $\rho$ 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.

# Appendix D

# Collaboration: IPSN 2019

The following paper is collaborations with Nokia Bell Labs in Cambridge, UK and Stony Brook University in New York, US and is currently under review at the International Conference on Information Processing in Sensor Networks (IPSN) 2019.

# An Empirical Study of Convolutional Neural Network Design for a Commodity Deep Learning Hardware Accelerator

Anonymous Author(s)

## ABSTRACT

Purpose-built hardware accelerators for deep neural networks are on the verge of going mainstream, and will soon be available in various commodity mobile and embedded devices. This variety of hardware has the potential to perform inference on deep models vastly more efficiently than conventional processors (like CPUs). But their wide-spread availability will provoke a number of basic questions in system design, processor selection and usage as well as deep model tuning for which we are not yet ready to answer.

In this work, we begin to provide early answers through in-depth study of currently one of the only commercially-available open neural network accelerators, the Intel Movidius Neural Compute Stick. We perform a first-of-its-kind systematic measurement study of the latency and energy of this accelerator under a variety of deep convolutional networks, and consider its performance in comparison to processor alternatives for constrained devices; specifically, the DSP, GPU and multi-core CPU available in Qualcomm Snapdragon 820 – a platform that is representative of typical mobile and embedded hardware. In this way, our study offers a preview of the future in which resource constrained devices perform on-device deep learning using a rich heterogeneous processor mix that includes hardware accelerators.

## 1 INTRODUCTION

The past four years has seen sizable strides made in the inference-time efficiency of deep learning, and at a cost of few percentage points in accuracy the execution of the best learning algorithms for various tasks (e.g., speech, vision, language) are increasingly becoming feasible for phone, wearable and embedded platforms [21, 23, 29, 33, 38, 48]. This capability provides a range of benefits for network edge devices, not only are they able to use the state-of-the-art models for processing data such as images, audio, and text (instead of weaker, but less complex models) – they are also able to do so when network connectivity is poor (and the cloud is unavailable) and provide a processing option to users where sensitive data (spoken words, image of faces) leaves their personal device. Such benefits have pushed industry towards adoption of on-device deep learning, for instance in Android devices and the iPhone it is used to detect if a user is in a car, walking or running [1] or when they say special hot-key words (e.g., *"Hey Siri"*) [2]. Though it should be noted, due to limited resources many types of deep models and thus possible applications remain out of reach.

Research advances within efficient inference for deep learning has tended to into two major directions, one software-centric and the other related to processor hardware. Examples of software-based innovation include improvements within the learning algorithms themselves, such as more efficient model architectures, or in software system solutions that improve performance through a better understanding of the workload [15, 17, 47, 48]. Efficiency gains in the order of 10× or more using such methods are routine, though usually gains are accompanied with a certain loss in accuracy. On the hardware side, processor designs that are purpose-built for the execution of neural architectures have proliferated [10, 14, 20, 39]. It is common for such accelerators to offer substantial performance gains over mainstream more general-purpose processors like GPUs, CPUs and DSPs (again performance gains of 10× and more are the norm). Though this is often achieved by making strong assumptions as to the neural architecture composition to be executed; this in turn makes offering such performance broadly across a variety of architectures (especially very large and deep models) problematic.

Driven by these intersecting trends, looking forward perhaps the most important question for on-device deep learning will be to understand the role to be filled by purpose-built hardware accelerators. Will accelerators make all neural network models *"cheap"* to run? Are the resource bottlenecks in deep learning inference observed on mainstream CPUs today going to disappear tomorrow as accelerators become more accessible? Core questions of this type have been inaccessible until only recently due to the lack of shipping accelerator hardware. In comparison to the hundreds of proposed accelerator designs and approaches, there had been almost no open accessible accelerator processors suitable for comparing again other processor architecture types or mature enough to run a wide range of deep models. This has now changed with offerings being made from companies that include Nvidia [46], Huawei [27] and Intel [4].

In this paper, we will begin to address these open questions by presenting, to the best of our knowledge, the first performance characterization of how convolutional neural architectures perform under one of the only commercially available hardware accelerators purpose-built to execute deep models efficiently: the Intel Movidius Neural Compute Stick [4] (NCS) – especially, within the increasingly common context of accelerators like the NCS being available on systems that also have other heterogeneous processor hardware (GPUs, DSPs, CPUs) present. The NCS itself delivers exceptional deep model inference performance at a low energy rate through 12 proprietary 128-bit SIMD processors referred to as Vision Processing Units (VPUs). In comparison, a mainstream CPU such as the ARM Cortex-A57 contains just 128-bit SIMD vector units (two, coarsely similar to VPUs). On a per watt power basis, the NCS has been shown to outperform high performance embedded/mobile GPUs like the Nvidia Jetson X2 [43]. Notably, the NCS has many processor design similarities (such as, use of a data flow architecture and hardware support of sparse data structures – see §2) to that of both proposed and already released accelerators like Huawei's Kirin 970 [27] and others, making its study of general importance.

Our empirical study is partitioned into two core elements (corresponding to the results reported in §5 and §4 respectively).

**(1) Accelerator Profiling and Hetero-Processor Comparison.** In the first, we perform detailed comparisons of neural network energy and latency performance under the NCS relative to the distinct processor architecture alternatives (viz. multi-core CPUs, DSPs and

GPUs) present in the Qualcomm Snapdragon 820 [6] – a highly popular mid-range mobile/embedded system-on-a-chip (SoC). To facilitate these experiments we develop a proof-of-concept experimental smartphone platform and software that comprises a OnePlus 3 Android smartphone [5] (that uses the Snapdragon 820 SoC) and NCS with the two communicating across a USB bridge. Such experiments, for instance, identify critical trade-offs that dictate which processor is best for particular deep models and highlight execution bottlenecks in the NCS and other processor architectures.

**(2) Neural Design for Efficient Accelerator Execution.** In our second set of experiments, we how common variations in neural network architecture impact performance goals like latency and energy. We systematically explore deep model parameterization (e.g., network depth, filter dimensions) that have arisen from aforementioned software-centric methods that seek to reduce and shape resource usage. By performing these experiments on the NCS we can identify which of these proliferating software techniques are beneficial when applied under hardware acceleration. Comparisons again performed against other processor architectures (e.g., DSP, GPUs) allow the impact related to processor design of accelerators with respect to more familiar processor types to made clear. Furthermore, such experiments offer insights into the specific design of deep models best suited for accelerators like the NCS.

Significantly in all of these experiments, we consider exclusively convolutional neural networks (CNNs) often used for computer vision related tasks (like object and face recognition or more general scene understanding). Such models and tasks are perhaps the most popular use of mobile and embedded deep learning today – but more importantly, results of the CNN varieties we test generalize strongly to other deep models as they contain network architecture components (e.g., feed-forward layers, skip connections) present in other combinations of model and task; for example, compact fully-connected networks used for on-device audio and speech tasks [44] and neural architecture techniques from CNNs that are adopted in language understanding and machine translation [12].

The scientific contributions of this work are as follows:

- **CNN performance under a commodity accelerator.** Detailed experimental results highlight the performance of various popular CNNs under the NCS. These results highlight performance bottlenecks and shed light as to which model varieties are best suited to execution under accelerator hardware. We anticipate such results will assist in the development of *accelerator-friendly* CNNs as well inform future accelerator designs.
- **Comparisons to hetero-processor alternatives.** As mobile and embedded hardware increasingly offers a variety of processor types, we compare the performance under deep model workloads of the NCS to the GPU, CPU and DSP from a representative Qualcomm SoC. These are fundamental empirical comparisons that inform which processor is best suited certain workloads and performance goals. We believe such comparisons represent some of the first of their kind.
- **Relationship to neural architectural variants.** Because accelerator hardware has been unavailable, the enormous variety of software-based approaches that modify a model to improve its system performance has ignored how these solutions relate to accelerator design. We systematically report the implications of such neural architecture changes and the impact on accelerator performance that results. We expect results of this type will open new opportunities to design software-based efficiency methods aligned with accelerator design principles.
- **Hetero-Processor and Accelerator Platform.** To enable the above series of experiments, we devise an experimental platform and software stack the allows for precise latency and energy measurements of any CNN when executing on any of the native processors present in the Qualcomm Snapdragon 820 (viz. DSP, GPU, CPU) as well as the accelerator present in the NCS. We envision the community at large will be interested in adopting this platform for a wide variety of follow-up experiments, or even in the development of end-to-end applications that require this mixture of processors. This aspect of our work plugs a important methodological hole for researchers, by allowing accelerators and conventional processors to be easily mixed and measured in support of a variety of neural architectures.

## 2 PROCESSORS FOR DEEP LEARNING

In this section, we describe how varieties of processor architectures investigated in our study (viz. CPU, GPU, DSP and accelerators in the form of the Intel Movidius Neural Compute Stick) can be used for deep learning inference. But before we begin this discussion we briefly connect processor design to the inferenece workload of neural networks.

### 2.1 Processor design and Neural Networks

Deep models often have stringent storage and computational requirements and state-of-the-art models are moving towards even bigger and more complex architectures. Convolutional layers – which we focus on in this study – are the commonly-used building blocks in vision models, are very computationally expensive. AlexNet [32] for instance, one of the first successful deep architectures for image classifications, has over 60 million parameters; and while the convolutional layers of AlexNet account for only 4% of the parameters, they are responsible for more than 90% of the computations at inference time [36]. As a result, optimising and designing computation platforms that enable fast and energy-efficient deployment of DNNs are critical for their wide adoption.

The fundamental component of both the convolutional and (fully connected if present) layers are the multiply-and-accumulate (MAC) operations, which can be easily parallelized. In order to achieve high performance, highly-parallel compute paradigms are very commonly used, including both temporal and spatial architectures. The temporal architectures appear mostly in CPUs or GPUs, and employ a variety of techniques to improve parallelism such as vectors (SIMD) or parallel threads (SIMT). Such temporal architecture use a centralized control for a large number of ALUs. These ALUs can only fetch data from the memory hierarchy and cannot communicate directly with each other. In contrast, spatial architectures use dataflow processing, i.e., the ALUs form a processing chain so that they can pass data from one to another directly. Sometimes each ALU can have its own control logic and local memory, called

a scratchpad or register file. We refer to the ALU with its own local memory as a processing engine (PE). Spatial architectures are commonly used for DNNs in ASIC and FPGA-based designs.

## 2.2 Processor Architecture Varieties

After briefly discussing the ties between processor architecture and design with neural network workloads, we provide the background processor design details necessary to appreciate the results of our measurement study.

**CPU: Central Processing Unit.** Mobile CPUs such as ARM processors have Single Instruction Multiple Data (SIMD) capabilities to parallelize compute intensive operations. Most deep learning frameworks on mobile (e.g. TensorFlow Lite, SNPE, etc) use NEON (ARM's enhanced SIMD technique) features to speed up DNN model inference. While mobile CPUs do have some parallelization power for compute intensive applications, they are natively designed for more general tasks and have complex control logic and lower compute density, which are mainly optimized for serial operations (i.e., fewer execution arithmetic units and higher clock speeds). On the other hand, GPUs are originally designed to accelerate graphics applications thus having stronger SIMD capabilities built for massively parallel workloads. Deep learning execution frameworks using GPUs typically utilize many GPU shader cores (usually range from tens to hundreds) SIMD architecture to parallelize the mathematical computations. Figure 1 shows the differences between CPU and GPU SIMD architectures, GPUs are built to have high compute density and high computations per memory access, which are optimized for parallel operations.

There are multiple software libraries designed to provide optimised implementations of linear algebra operations for general-purpose CPUs (e.g. OpenBLAS, Intel MKL) and GPUs (e.g. cuBLAS, cuDNN). Hardware platforms have also started adding support for NN workloads. Two examples are: Arm's NN SDK that enables efficient translation of models from existing frameworks such as TensorFlow and Caffe to run efficiently on Arm Cortex CPUs and Arm Mali GPUs. Qualcomm's Snapdragon Neural Processing Engine (SNPE) framework offers similar feature for its mobile CPUs, GPUs or DSPs. FPGAs are also suitable platforms that can exploit properties of neural network to obtain parallel implementations.

**Accelerators: Intel Movidius Neural Compute Stick.** Intel Movidius Neural Compute Stick (NCS) is one of the first commercially available of such accelerators for mobile, wearable, and embedded systems. It is powered by a low-power, high-performance SoC designed to handle a range of applications such as Deep Neural Network-based classification, object tracking, indoor navigation and 3D vision applications. At the heart of the NCS is the Myriad2 MA2450 Vision Processing Unit (VPU). Myriad2 provides up to 100 GFLOPs of performance at 600MHz within a nominal 1W of power consumption. The Myriad2 features twelve 128-bit vector processing cores called SHAVE (Streaming Hybrid Architecture Vector Engine) which compute most of the neural network load [11]. Figure 2 shows the architecture inside Myriad2.

Critically, emerging deep learning accelerators like the NCS and others use dataflow architecture instead of the SIMD architecture used on CPUs and GPUs. Figure 4 shows the comparison between dataflow and SIMD processor. As you can see each SHAVE core has

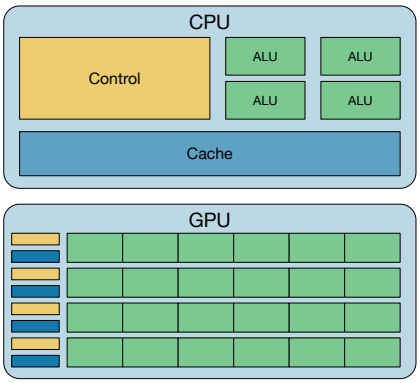



**Figure 1: CPU and GPU SIMD Architecture Comparison**

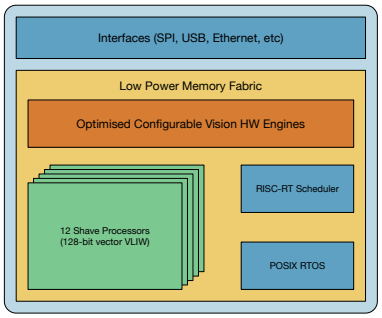



**Figure 2: Myriad VPU Architecture**

a hybrid architecture that combines ideas from RISC, DSP, GPU and VLIW architectures: It supports RISC-style instruction predication, has multiple parallels units and 8/16/32-bit SIMD support controlled via VLIW instructions. It also supports stream processing and Texture Management Units (TMU) similar to GPUs and and provides zero-overhead looping, modulo addressing and FFT units like DSPs.

When we dig into the SHAVE processors, we find each contains wide and deep register files coupled with a variable-length long instruction word (VLLIW) controlling multiple functional units including extensive SIMD capability for high parallelism and throughput at both a functional unit and processor level. The SHAVE processor is a hybrid stream processor architecture combining the best features of GPUs, DSPs, and RISC with both 8-, 16-, and 32-bit integer and 16- and 32-bit floating-point arithmetic as well as unique features such as hardware support for sparse data structures. The architecture maximizes performance per watt while maintaining ease of programmability, especially in terms of support for design and porting of multicore software applications

Thus, sharing data flexibly between SHAVE processors and hardware accelerators via the multiported memory subsystem was a key challenge in the design of the Myriad 2 VPU. In the 28-nm Myriad 2 VPU, 12 SHAVE processors, hardware accelerators, shared data, and SHAVE instructions reside in a shared 2-Mbyte memory block called Connection Matrix (CMX) memory, which can be configured

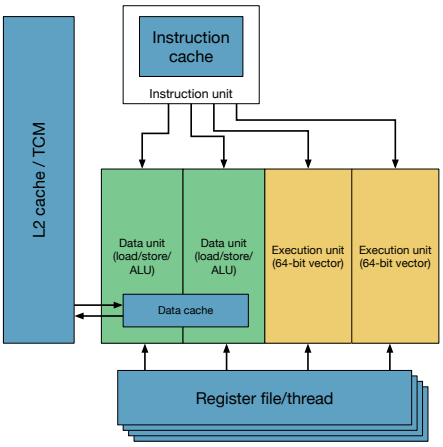



**Figure 3: Hexagon DSP with SIMD extensions**

to accommodate different instruction and data mixes depending on the workload.

**DSP: Digital Signal Processor.** Mobile DSPs (e.g. Hexagon DSP) use SIMD extentions, which enhance the original SIMD by introducing vector execution unit in addition to ALUs as shown in Figure 3. Another characteristic of DSPs is that they usually support integer based operations to further reduce computer clock cycles and energy footprints.

Interestingly, VPU (found in the NCS) and DSPs are more similar in terms of design considerations including hardware parallelization, energy efficiency. VPU are optimized for vision algorithms and in contrast DSPs are optimized for multimedia streaming, encoding/decoding. But fundamentally the computer requirements and hence logic design are closely related.

Compression and efficient implementation of DNNs are more important than ever. There has been a torrent of recent work proposing training and post-training schemes that aim to compress models while avoiding significant loss in their performance. Main examples of such techniques are: pruning, weight sharing, low-rank approximation, knowledge distillation and perhaps most importantly quantisation to lower precisions [22].

**GPU: Graphics Processing Unit.** The basic building block in a GPU is a Streaming Multiprocessor or SM. Different GPU architectures have different numbers and configuration of SMs. For example, each SM in NVIDIA's V100 has 128 cores, 64k registers, 96KB of shared memory, 48KB L1 cache, and up to 2000 threads. The important hardware feature is that the cores in a SM are Single Instruction Multiple Threads (SIMT). This means groups of multiple cores (e.g. 32) execute the same instructions simultaneously, but with different data. This means all threads within an SM are do the same thing at the same time. This is natural for many scientific computing abd graphics processing.

# 3 STUDY METHODOLOGY

We compare the performance of the only commercially available hardware accelerator, NCS, with existing mobile processor architecture—CPU, GPU, and DSP. In this section, we present the measurement setup that is used in subsequent sections (§4 and §5) which report results.

## Convolutional Network Workload

We evaluate the performance of the four process architectures on the most popular neural network workload, that of image recognition. To this end, we study a large number of popular image recognition models: AlexNet [32], SqueezeNet [30], InceptionV3 [41], MobileNetV1 [26], ResNet50 [25], and MobileNetV2 [40]. All of these models use Convolutional Neural Networks [35].

Table 1 summarizes the characteristics of the neural network models. The fields corresponding to FLOPs (Floating-Point OPeration per second) and parameters show the computationally needs of the model. The larger the number of FLOPs and the number of parameters (in millions), the more compute the model requires.

The models represent a diverse range in terms of accuracy, computational requirements, and complexity. Some of the models are designed specifically for mobile devices while others are designed for server-class devices. AlexNet [32] was the watershed model for the deep learning community. SqueezeNet [30] has comparable accuracy to AlexNet [32] but has significantly lower number of parameters making them suitable for mobile deployment. InceptionV3 [41] was developed by Google and further pushed the accuracy higher. MobileNetV1 [26] is a model that is specifically designed for mobile devices, which improves on the efficiency of SqueezeNet [30].

The more recent model, ResNet50 [25], is the state-of-the-art model in terms of accuracy, but is computationally intensive (see Table 1). MobileNetV2 [40] is an improved version of MobileNetV1 that leverages the residual blocks in ResNets [25] to improve accuracy but still maintaining a smaller number of parameters and FLOPs

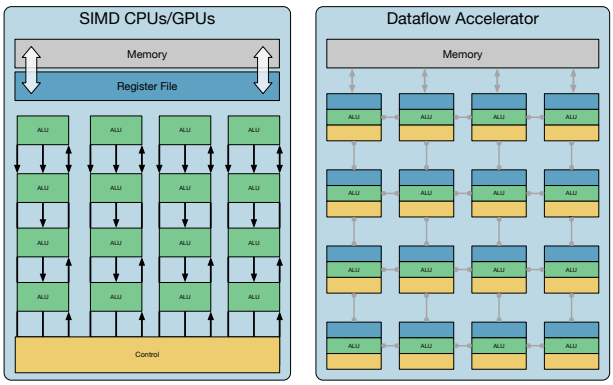



**Figure 4: SIMD and Dataflow Architecture Comparison**

| Name | Layers | Million FLOPs | Million Params | Top1 Accuracy |
|---|---|---|---|---|
| AlexNet | 8 | 1574.9 | 50.31 | 57.2% |
| SqueezeNet | 10 | 1683.1 | 1.25 | 57.5% |
| InceptionV3 | 48 | 5744.4 | 25.57 | 78.0% |
| MobileNetV1 | 29 | 1146.0 | 4.23 | 70.9% |
| ResNet50 | 50 | 7022.3 | 25.56 | 75.2% |
| MobileNetV2 | 21 | 608.8 | 3.51 | 71.9% |

**Table 1: The six image classification neural network models considered in this study. The table shows the number of layers, the number of floating point operations per second (FLOPS), the number of parameters (in millions) and the accuracy of these models. MobileNetV1 and MobileNetV2 are designed specifically for mobile devices and have a lower number of parameters.**


**Figure 5: Experiment setup: The figure shows how the NCS is connected to the phone. The entire unit is connected to the Monsoon Power Monitor for power measurements.**

## Latency and Energy measurement

We compare the performance of different processors based on latency and energy, two critical metrics on mobile devices. Since we run the same model on different processors, the accuracy remains the same. The *inference latency* is the time between inputing the image and finishing the classification. The energy consumption is measured during the inference time. All reported numbers are averaged over 10 runs.

We use Monsoon Power Monitor [3] to measure the power for the smartphone. To measure the power of NCS, we use a on-the-go cable to connect the NCS into the phone.

## Device Specification

We compare performance of CNN model performance on the Movidius NCS and an OnePlus 3 Android smartphone [5]. The NCS USB stick internally has a Myraid 2 Vision Processing Unit (VPU), accompanied with 1 GB RAM, 4GB storage. The smartphone is running Android 8.0.0 and has a Qualcomm 820 CPU, Adreno 530 GPU and Hexagon 680 DSP. It also has a 6 GB RAM and 64 GB storage. The NCS is connected to a PC as instructed by the SDK.

## Software Runtime and Tools

We use the available TensorFlow [42] implementation for all models. The software used to run the inference depends on the processor architecture. Each of the software used is necessary to support individual experiments and model/hardware needs. We perform the necessary steps to ensure we can compare results across software.

The NCS inference is performed on the *Movidius Neural Compute Software Developer Kit* (NCSDK) [7]. It includes software tools for compiling and profiling TensorFlow models. On the CPU, we run the inference on *TensorFlow Lite* (TFLite) [8]. TensorFlow's lightweight solution for mobile and embedded devices. It enables on-device machine learning inference with low latency and a small binary.

Finally, for the DSP and GPU experiments, we use *Snapdragon Neural Processing Engine* (SNPE).

# 4 PERFORMANCE STUDY OF THE CNN EXECUTION ON MOBILE PROCESSORS

We start by investigating the latency and energy consumption of running size neural network models (Table 1) over NCS, CPU, GPU, and DSP.

The key takeaways are:

- Takeaway 1: The NCS accelerator has been specifically designed for neural models. However, while NCS improves latency over CPU, the latency of NCS is more than that on the GPU and DSP. Interestingly, the relative performance of the models remain the same across the four processors.
- Takeaway 2: NCS does improve the energy consumption for inferencing compared to CPU and GPU. However, te energy consumption of DSP is lower than that of NCS. As before, the relative energy consumption across the models remain the same.
- Takeaway 3: Increasing the number of parallel (SHAVE) cores in NCS improves the inference latency, but then reduces. The per-model consumed energy first decreases as more SHAVE cores are used, but then increases when using more than a certain number of cores, which means the speedup gains are not able to compensate for the increased energy consumption of more cores operating simultaneously.

Overall, we find that NCS behaves more like a DSP but not better than DSP for the studied CNN workloads. We also notice that even conventional processors like DSPs can well suit CNN workloads although being purposely designed for multimedia and signal processing applications.

## 4.1 Latency and Energy Characterization

Figure 6 shows the inference latency of running the CNN models on the NCS compared with the smartphone CPU/GPU/DSP. Both GPU/DSP and the NCS demonstrates speedups over the CPU baseline in all cases. However, to our surprisingly interest, the NCS

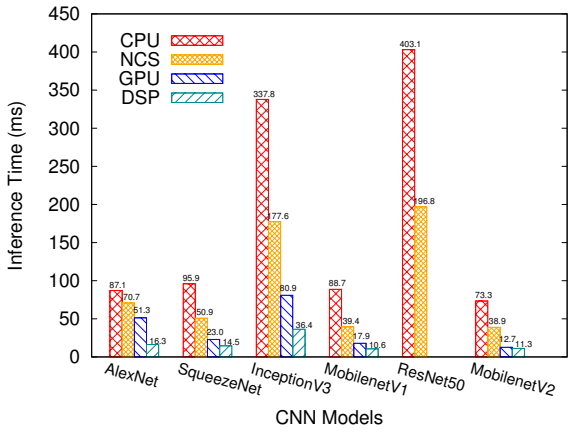



**Figure 6: Inference latency on NCS and the other three smartphone processors for the six neural network models. Note that SNPE does not support ResNet50 yet, so GPU/DSP latency are not shown.**

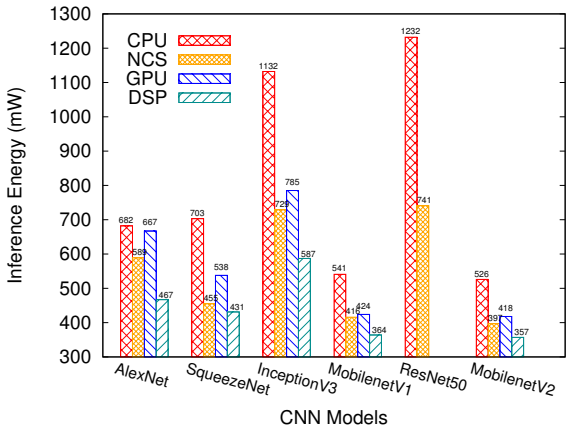



**Figure 7: Energy consumption of studied 6 CNN workloads on NCS and smartphone CPU/GPU and DSP processors**

armed with a commodity deep learning accelerator turns out to be not better than conventional mobile GPU and DSP. Another interesting point is that different CNN model architectures do not affect the processor speedup performance (compared to CPU), i.e. the case where one model runs faster and the other slower does not exist in our study.

Figure 7 shows the network energy consumption of over the four processors. DSP consumes the minimum energy across all models. However NCS consumes lower energy compared to the GPU and CPU. The fact that GPU and DSP are faster than CPU is understandable since they have much better parallelization capabilities (i.e, many cores, simpler control logics, optimized compute cycles). However, NCS has more energy footprints than DSP and longer inference time for the studied models.

Based on the latency and energy study, the recent general notion [11, 13, 37] that specialized DNN hardware accelerator has short latency and impressive power efficiency does not hold in our case. Conventional mobile processors like GPU and DSP have highly optimized both hardware and software stack(compiler toolchain, extensively tested code quality). DSPs can be used as the accelerator choice for DNNs, at least the design experience for DSPs can be borrowed to new accelerator design.

## 4.2 AlexNet Layer Latency Breakdown

To further study the reason, we also perform the layer latency for the AlexNet model [32]. Table 2 shows AlexNet inference time breakdown at each layer. We find that there is little difference in the relative performance across layers. For example, in each layer, the GPU latency is lower than NCS resulting in an overall reduction in latency. The one exception of fc6 (the sixth layer which is fully connected), where the NCS latency is slightly better than GPU. This is due to the large feature map processing after the convolution layers. The Phone GPU needs to allocate the memory (shared with CPU) on demand while the NCS can prealloacte the memory on the DRAM and possible cache them in each SHAVE core's local SRAM memory. We also see that the slowdown is not due to any model side factors or bottlenecks, thus being more likely the lack of software SDK optimization.

Table 2 also shows convolution layers being much less efficient for the NCS. Relative speedup for the NCS over CPU on the convolution layers vs. its speed on fully connected is extreme, which indicates new architectures that are fully convolutional would be great for the NCS.

| Layer | NCS | GPU | DSP | CPU |
|---|---|---|---|---|
| conv1 | 3.961 | 1.563 | 1.707 | 8.904 |
| pool1 | 0.319 | 0.044 | 0.086 | 0.548 |
| conv2 | 9.824 | 4.136 | 1.578 | 21.854 |
| pool2 | 0.201 | 0.03 | 0.039 | 0.256 |
| conv3 | 4.036 | 2.245 | 0.704 | 9.043 |
| conv4 | 6.549 | 4.602 | 1.505 | 17.244 |
| conv5 | 6.754 | 3.284 | 1.024 | 11.692 |
| pool5 | 0.068 | 0.011 | 0.027 | 0.042 |
| fc6 | 20.018 | 22.959 | 4.142 | 7.091 |
| fc7 | 15.046 | 11.316 | 2.222 | 5.646 |
| fc8 | 3.71 | 1.02 | 0.824 | 2.907 |
| output | 0.201 | 0.12 | 0.16 | 0.041 |
| total | 70.69 | 51.33 | 14.02 | 85.27 |

**Table 2: AlexNet inference time (ms) breakdown by layers**

## 4.3 Performnce of NCS with multiple cores

Figure 6 shows the AlexNet inference latency using different number of SHAVE cores on the NCS. The inference time plateaus as the number of core increases, with most of the speed up occuring between 4 and 6 cores. Figure 9 shows the corresponding energy consumption numbers of running AlexNet using different number of SHAVE cores. It can be seen that using 6 to 8 cores gives the best energy performance. This is because fewer cores result

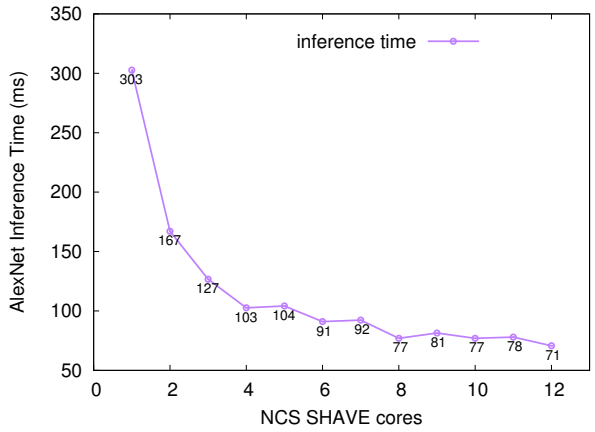



**Figure 8: AlexNet inference latency on NCS when using different number of SHAVE cores.**

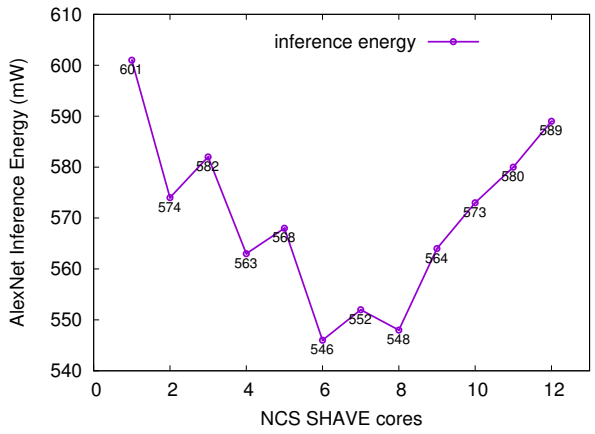



**Figure 9: AlexNet inference energy on NCS when using different number of SHAVE cores**

in longer latency increasing energy while a large number of cores consume energy for computation. This SHAVE core behavior indicates always using all available cores may not be the optimal choice, furthermore, different workloads may need to be tuned to use different number of cores to achieve the best latency and energy trade-offs.

# 5 EFFECT OF MODEL PARAMETERS AND OPTIMIZATIONS

In this section we study the effect of using different models on its latency and energy performance of the hardware accelerator. We do this by warying the model parameters and apply existing model optimizations to the six models. To study the effect of model configurations, we vary three of the most common model hyperparameters in CNN models: filter size, network width, and network depth. In terms of optimization, we study the effect of two common model optimizations: low precision and pruning.

Our key takeaways in this are:

- Takeaway 1: CPU and NCS prefer "fat-wide" style networks (i.e. larger filter size, more filters) while GPU and DSP prefer a moderate style (i.e. both filter size and number of filters are neither big nor small).
- Takeaway 2: DSP is not sensitive to filter size under changing network depth. On the GPU, the performance is best under moderate conditions of not too deep network and not too big filter. still prefer moderate style network (i.e., neither too deep nor too big filter).
- Takeaway 3: Increasing the network input size will slowdown the inference for all 4 processors. NCS slowdown is between the GPU and DSP.
- Takeaway 4: Existing known hardware CNN optimization techniques like low precision and pruning optimizations does not provide benefits for either GPU or NCS.

| | Model Name | stage 3 |
|---|---|---|
| Filter size vs. Width | fw1 | $874C1 \times 2 - 256C1$ |
| | fw3 (baseline) | $256C3 \times 3$ |
| | fw5 | $128C5 \times 2 - 256C5$ |
| Filter size vs depth | fd2 | $256C2 \times 6$ |
| | fd3 (baseline) | $256C3 \times 3$ |
| | fd5 | $256C5$ |
| Width vs Depth | wd6 (baseline) | $256C3 \times 3$ |
| | wd8 | $160C3 \times 5 - 256C3$ |
| | wd11 | $128C3 \times 8 - 256C3$ |
| | wd13 | $108C3 \times 10 - 256C3$ |
| | wd18 | $96C3 \times 15 - 256C3$ |
| | wd25 | $80C3 \times 22 - 256C3$ |
| | wd38 | $64C3 \times 35 - 256C3$ |

**Table 3: Various configurations in final stages of AlexNet under constrained time complexity. The first stages of the models are the same in all configurations, i.e. (64C7)-(MP3)-(128C5)-(MP2). The notation $(s, n)$ represents the filter size and the number of filters. "$/2$" represents stride = 2 (default 1). "$\times m$" means the same layer configuration is applied k times (not sharing weights). The numbers in the pooling layer represent the filter size and also the stride. All convolutional layers are with ReLU.**

## 5.1 Changing model parameters

Table 3 shows the parameters we vary across filters, network width, and network depth. All the networks have a $224 \times 224 \times 3$ input image size. We group the network layers into three stages. Stage 1 has one convolutional layer having 64 $7 \times 7$ filters with a stride 2, followed by a $3 \times 3$ max pooling layer with a stride 3. Stage 2 consists of a convolutional layer that has 128 $5 \times 5$ filters, followed by a $2 \times 2$ max pooling layer with a stride 2. In the third stage, for the baseline model, it has three convolutional layers all have 256 $3 \times 3$ filters, for other models, they are shown in table 3. After the stage 3, there is a $3 \times 3$ max pooling layer with a stride 3, followed

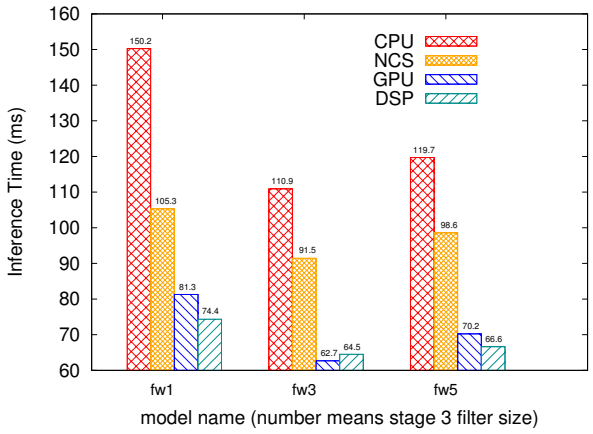



**Figure 10: CNN filter size vs network width**

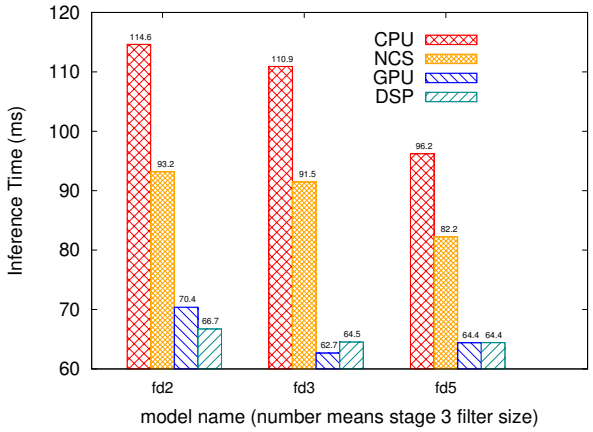



**Figure 11: CNN filter size vs network depth**

by two 4096-d fully connected layers and a 1000-d fully connected layer with softmax as the output.

To isolate the effect of each of these parameters we vary one parameter keeping the rest of the network a constant. The evaluations are performed over AlexNet [32]. only using SqueezeNet for network input size study, reason is commented out below.

*5.1.1* **Filter Size versus Network Width.** Table 3 shows that convolutional layers with smaller filters have more filters which means wider network. Figure 10 shows the inference time of varying the filter size versus network width. On both NCS and CPU, the performance improves with larger filter sizes. However, on the GPU and DSP, the performance is best when the filter size is not too big or small. We guess this is due to the memory/cache limit of GPU and DSP, both CPU and NCS have more available memory to use or cache if more filter weights come in within a layer's computation time.

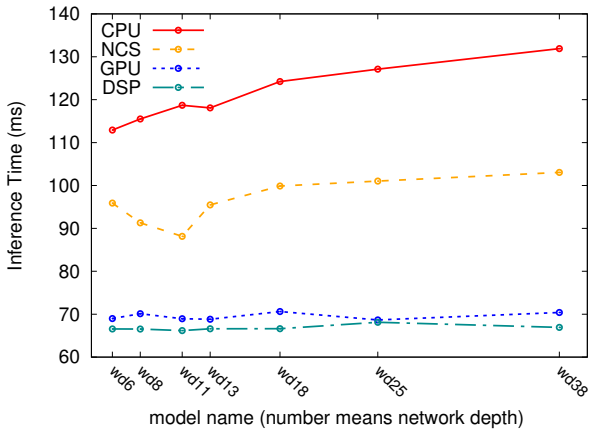



**Figure 12: CNN network width vs network depth**

*5.1.2* **Filter Size versus Network Depth.** Similar to filter size versus network width variations, convolutional layers with smaller filters have more layers in the network which means deeper network. Figure 11 shows the inference time of varying the filter size versus network depth. Interestingly, DSP is not sensitive to the filter size and network width tradeoffs, this can attribute to the fact that the inference is already fast enough and the computation complexity stays the same across the variants. NCS along with CPU performs best with large filter size, while the GPU performs best with a filter size that is not too big or small.

*5.1.3* **Network Width versus Network Depth.** To further see the effects of network depth and width variations, we designed a set of network architectures ranging from "wide-shadow" style to "deep-thin" style as shown in Table 3.

Figure 12 shows that GPU and DSP are not sensitive to the network width and depth alterations. This again is because the inference is fast enough and the computation complexity stays the same across the variants. For CPU, the performance on the wide-shadow network is better since as the network becomes deeper and thinner the inference latency becomes larger. More interestingly, on NCS the performance is best under not too wide-shadow nor too deep-thin networks conditions. The inference latency is lowest at 11 layers. Stacking more layers will cause slightly longer inference time but not as aggressive as on the CPU. This is possibly because NCS is not tuned for more general CNN models as the official SDK only supports a limited number of published networks.

*5.1.4* **Network Input Size.** Figure 13 shows the SqueezeNet inference slowdown for increasing input size. As the input size to the network becomes larger, NCS slowdown is between the GPU and DSP. We guess this is due to the memory characteristics, GPU shares the memory with CPU (which is fast when CPU offloads the computation to GPU), NCS VPU has its own DRAM and SRAM for caching. DSP, however, only has L2 cache (small size compared to VPU SHAVE core SRAM) for model weight reusing.

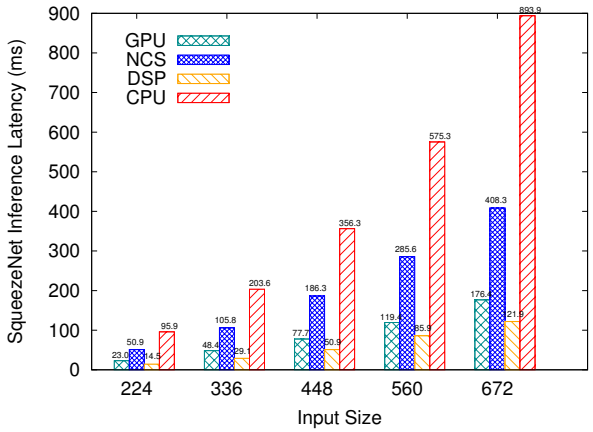



**Figure 13: SqueezeNet inference latency on NCS and smartphone processors when input size increases**

**Figure 14: SqueezeNet inference latency vs input size.**

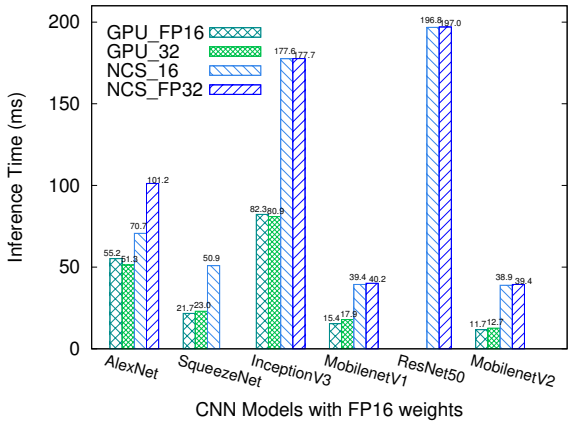



**Figure 15: Latency comparison for CNNs with FP16 and FP32 precision weights**

## 5.2 The effect of Optimizations

We applied two sets of optimizations, i.e., low precision and pruning to the MobileNetV1 network. We could not study the effect of other common optimizations including quantization [] because they are not currently supported in NCS.

*5.2.1 Low Precision Inference.* Figure 12 shows that GPU and NCS inference latency for low precision model weights. For GPU, AlexNet and InceptionV3 perform slightly better using the full precision weights while their mobile variants – SqueezeNet and MobileNet(V1 and V2) slightly prefer half precision model weights. For NCS, we found that although the internal VPU supports low precision arithmetics such as INT8/FP16 operations, the software stack is not fully optimized for the hardware. We also found the NCSDK internally represents the model weights in half precision format even though it supports full precision as input and output.

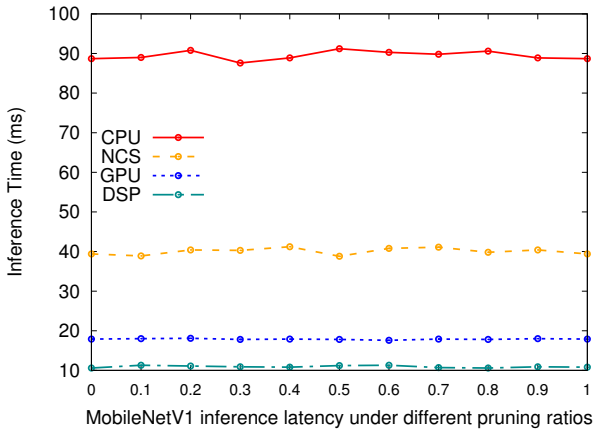



**Figure 16: MobileNetV1 inference latency w.r.t to pruning ratio**

Note that AlexNet full precision is much lower than the half precision counterpart, and SqueezeNet full precision takes infinite time to run, we found this is due to the NCSDK software problem.

*5.2.2 Weights Pruning.* The purpose of pruning is to avoid computations for zero-value parameters (model weights), which is especially effective for CNN accelerators as shown in [19, 20, 48].

The MobileNet architecture consists of one standard convolution layer acting on the input image, a stack of depthwise separable convolutions, and finally averaging pooling and fully connected layers. 99% of the parameters in the MobileNetV1 are in the 1x1 pointwise convolution layers (74.6%) and fully connected layers (24.3%). In this study, we set the weight pruning threshold to $10^{-3}$. We do not prune the parameters in the one standard convolution layer and in the depthwise convolution layers since there are very few parameters in these layers (1.1% of the total number of parameters).

Figure 16 shows the inference latency of MobilenetV1 when applied with different pruning ratio. Both GPU and NCS show no significant changes in latency. No performance gains for GPU is understandable since it's mainly designed for graphics rendering and general parallel computations. But for NCS, our hypothesis is that, even though NCS has a CNN accelerator chip, the VPU itself is not optimized for sparse CNN computations.

## 6 RELATED WORK

As there are very few measurement studies like the one we perform in this paper, we survey results found in software-based architecture optimization research and accelerator design studies. These represent the most salient forms of related work.

**Software-based architecture optimization.** Early models with quantized parameters were derived by applying quantization operation to weights of pre-trained models [18]. This approach is common but suffers from significant loss in accuracy. [28] showed that in order to maintain model performance, quantization must be an integral part of the training process. This can be achieved

by either performing additional fine-tuning steps after training is finished or by directly learning quantized parameters.

The core idea is to use new coding or scaling techniques for the float point real numbers reside in DNN weights. The extreme form form of quantization, binarization, can constrain network parameter to 1-bit representation [15] (+1 and -1) so that parameter updates and calculating activations can be implemented through X-nor [38] operations. For example, floating point multiplications are supplanted with bitwise XNORs and left and right bit shifts.

Although DNN models normally require a vast number of parameters to guarantee their superior performance, significant redundancies have been reported in their parameterizations [16]. Therefore, with a proper strategy, it is possible to compress these models without significantly losing their prediction accuracies. Among existing methods, network pruning appears to be an outstanding one due to its surprising ability of accuracy loss prevention.

There are various pruning approaches with respect to different criteria, deep compression [22] applies pruning to reduce model size and starts by learning the connectivity via normal network training and prune the small-weight connections(all connections with weights below a threshold are removed from the network) and followed by retraining the network to learn the final weights for the remaining sparse connections. [48] proposes a new fine-grained pruning algorithm that specifically targets energy-efficiency. It aggressively prune the layers with the highest energy consumption with marginal impact on accuracy. Moreover, the pruning algorithm considers the joint influence of weights on the final output feature maps, thus enabling both a higher compression ratio and a larger energy reduction.

**Processor Hardware and System Optimization.** As the compute cost of machine learning models continues to grow, and their adaptation increases, more optimisations are required to deploy them in practice. There is also an increase desire for ASIC accelerators specifically designed for NN workloads. Many of these optimisatuins are achieved using custom hardware acceleration blocks. Most of these solutions are designed to speed up inferences, while others aim to accelerate the training phase. Given popularity of DNNs it's not surprising that there has been a spate of publications, prototypes, and commercial hardware accelerations. While less flexible than other platform they can offer better energy-efficiency and smaller silicon area footprint.

[20] proposed an energy efficient inference engine called EIE. The engine performs inference on a model compressed using weight pruning and weight sharing. The result is accelerated using sparse matrix-vector multiplication with weight sharing, skipping zero activations from ReLUs, exploiting sparsity and going from DRAM to SRAM. [45] presented a scalable hardware prototype on FPGA for using with DNNs. The DLAU accelerator employs three pipelined processing units to improve the throughput and utilizes tile techniques to explore locality for deep learning applications.

Mobile GPUs provide another optimization avenue. CNNDroid [34] both showed a mobile GPU can be used to improve the CNN/DNN execution time. In CNNDroid, they reported more than 10-fold speedup for AlexNet model on CIFAR-10 dataset. DeepMon [29] showed careful optimization like caching or sharing partial results on mobile phones' GPU could achieve promising real time execution of deep vision models.

AI benchmarks [31] studies eight computer vision workloads for various mobile SoCs. [24] characterizes the CNN latency and throughput for mobile vision tasks. Both of them don't study the CNN performance for commodity hardware accelerators. [9] discusses various deep learning workloads for training and inference on server-level processors.

## 7 CONCLUSION

Motivated by the looming arrival of neural network accelerators in embedded and mobile devices, we have conducted one of the first systematic measurement studies that considers an early open commercially available neural network accelerator: the Intel Movidius Neural Compute Stick. Because, only until very recently, accelerators have been available hardware simulations or limited supply prototypes many of our empirical results are the first time. Along with accelerators, heterogeneous compute options (such as, GPUs, multi-core CPUs, DSP) are becoming commonplace in constrained platforms – many of which have been shown to be viable processors onto which to perform efficient deep learning inference. For this reason, one core element of our investigation as been to compare the efficiency of neural architectures under such processor design alternatives. We perform these experiments using the collection of processors available in the Qualcomm Snapdragon 820 – a platform that is representative of typical mobile and embedded hardware. Finally, due to the proliferation of software-based deep architecture variations for low resource situations – virtually none of which consider processor designs of accelerators; we systematically study commonly manipulated architecture parameters to under the NCS.

## REFERENCES


[1] 2018. Google APIs for Android. https://developers.google.com/android/reference/com/google/android/gms/location/DetectedActivity. (2018).

[2] 2018. Hey Siri: An On-device DNN-powered Voice Trigger for AppleâĂŹs Personal Assistant. https://machinelearning.apple.com/2017/10/01/hey-siri.html. (2018).

[3] 2018. Monsoon Power Monitor. https://www.msoon.com/LabEquipment/PowerMonitor/. (2018).

[4] 2018. Movidius Neural Compute Stick. https://developer.movidius.com/. (2018). accessed 23-March-2018.

[5] 2018. OnePlus 3. https://www.oneplus.com/3. (2018).

[6] 2018. Qualcomm Snapdragon 820. https://www.qualcomm.com/products/snapdragon/processors/820. (2018).

[7] 2018. Software Development Kit for the Neural Compute Stick. https://github.com/movidius/ncsdk. (2018). accessed 21-April-2018.

[8] 2018. TensorFlow Lite. https://www.tensorflow.org/. (2018). accessed 8-April-2017.

[9] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2016. Fathom: Reference workloads for modern deep learning methods. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 1–10.

[10] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 1–13.

[11] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma. 2015. Always-on Vision Processing Unit for Mobile Applications. *IEEE Micro* 35, 2 (Mar 2015), 56–66. https://doi.org/10.1109/MM.2015.10

[12] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. 2017. Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906* (2017).

[13] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 367–379.

[14] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[15] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[16] Misha Denil, Babak Shakibi, Laurent Dinh, Nando de Freitas, et al. 2013. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*. 2148–2156.

[17] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. 2014. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115* (2014).

[18] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. 2014. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115* (2014).

[19] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 75–84. https://doi.org/10.1145/3020078.3021745

[20] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 243–254.

[21] S. Han, H. Mao, and W. J. Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *ArXiv e-prints* (Oct. 2015). arXiv:cs.CV/1510.00149

[22] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[23] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 123–136.

[24] Jussi Hanhirova, Teemu Kämäräinen, Sipi Seppälä, Matti Siekkinen, Vesa Hirvisalo, and Antti Ylä-Jääski. 2018. Latency and Throughput Characterization of Convolutional Neural Networks for Mobile Computer Vision. In *Proceedings of the 9th ACM Multimedia Systems Conference (MMSys '18)*. ACM, New York, NY, USA, 204–215. https://doi.org/10.1145/3204949.3204975

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[26] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[27] Huawei. 2018. Kirin 970. http://www.hisilicon.com/en/Media-Center/News/Key-Information-About-the-Huawei-Kirin970. (2018).

[28] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research* 18 (2017), 187–1.

[29] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 82–95.

[30] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360* (2016).

[31] A. Ignatov, R. Timofte, W. Chou, K. Wang, M. Wu, T. Hartley, and L. Van Gool. 2018. AI Benchmark: Running Deep Neural Networks on Android Smartphones. *ArXiv e-prints* (Oct. 2018). arXiv:cs.AI/1810.01109

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[33] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. 2016. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 1–12. https://doi.org/10.1109/IPSN.2016.7460664

[34] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. 2016. CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android. In *Proceedings of the 2016 ACM on Multimedia Conference (MM '16)*. ACM, New York, NY, USA, 1201–1205. https://doi.org/10.1145/2964284.2973801

[35] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.

[36] Christos Louizos, Karen Ullrich, and Max Welling. 2017. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*. 3288–3298.

[37] David Moloney, Brendan Barry, Richard Richmond, Fergal Connor, Cormac Brick, and David Donohoe. 2014. Myriad 2: Eye of the computational vision storm. In *Hot Chips 26 Symposium (HCS), 2014 IEEE*. IEEE, 1–18.

[38] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[39] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 267–278.

[40] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.

[41] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[42] TheTensorFlowAuthers. 2018. TensorFlow. https://www.tensorflow.org/. (2018). accessed 8-May-2018.

[43] Nvidia TX2. 2018. Nvidia TX2. https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/. (2018).

[44] Ehsan Variani, Xin Lei, Erik McDermott, Ignacio Lopez-Moreno, and Javier Gonzalez-Dominguez. 2014. Deep neural networks for small footprint text-dependent speaker verification.. In *ICASSP*, Vol. 14. Citeseer, 4052–4056.

[45] Chao Wang, Qi Yu, Lei Gong, Xi Li, Yuan Xie, and Xuehai Zhou. 2016. DLAU: A scalable deep learning accelerator unit on FPGA. *arXiv preprint arXiv:1605.06894* (2016).

[46] Nvidia Xiaver. 2018. Nvidia Xiaver. https://developer.nvidia.com/embedded/buy/jetson-xavier-devkit. (2018).

[47] Jian Xue, Jinyu Li, and Yifan Gong. 2013. Restructuring of deep neural network acoustic models with singular value decomposition.. In *Interspeech*. 2365–2369.

[48] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Jul 2017). https://doi.org/10.1109/cvpr.2017.643