Universidade Estadual de Campinas
Instituto de Computação

Matheus Martins Susin

Energy Efficient Inference Computation with Approximate Convolutions

Computação Eficiente de Inferências com Convolução Aproximada

CAMPINAS
2019

Matheus Martins Susin

Energy Efficient Inference Computation with Approximate Convolutions

Computação Eficiente de Inferências com Convolução Aproximada

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Lucas Francisco Wanner

Este exemplar corresponde à versão final da Dissertação defendida por Matheus Martins Susin e orientada pelo Prof. Dr. Lucas Francisco Wanner.

CAMPINAS
2019

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Su81e

Susin, Matheus Martins, 1995-
Energy efficient inference computation with approximate convolutions / Matheus Martins Susin. – Campinas, SP : [s.n.], 2019.

Orientador: Lucas Francisco Wanner.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Computação aproximada. 2. Redes neurais convolucionais. 3. Dispositivos móveis. 4. Computação consciente de energia. 5. Álgebra linear. I. Wanner, Lucas Francisco, 1981-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Computação eficiente de inferências com convolução aproximada
Palavras-chave em inglês:
Approximate computing
Convolutional neural networks
Mobile devices
Energy-aware computing
Linear algebra
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Lucas Francisco Wanner [Orientador]
Guido Costa Souza de Araújo
Luiz Filipe Menezes Vieira
Data de defesa: 25-10-2019
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-7468-2049
- Currículo Lattes do autor: http://buscatextual.cnpq.br/buscatextual/visu

Universidade Estadual de Campinas
Instituto de Computação

Matheus Martins Susin

Energy Efficient Inference Computation with Approximate Convolutions

Computação Eficiente de Inferências com Convolução Aproximada

Campinas, 25 de outubro de 2019

Acknowledgements

We thank all those whose efforts contributed to this project. We especially thank those who developed tools for their own work, but let us borrow and modify the source code for use in our research. This project would not have been possible without their help.

We credit Alex Torres for his work on NNT (5.5).

We credit Caio Rohwedder and Marcos Vinícius Martins for their work in kbench (5.4).

We credit Guilherme Valarini for his aid in this project, helping us find and fix a multitude of pesky bugs throughout the development of our tools.

And finally, we thank all the people at the Laboratory of Computer Systems (LSC) at the University of Campinas (Unicamp) for their support, and the coffee and laughs.

This work was supported by CNPq and Samsung.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001

Resumo

Neste trabalho, descrevemos um ambiente para medição de consumo de energia e tempo de redes neurais convolucionais (CNNs) em dispositivos móveis. Medimos o consumo de um dispositivo enquanto ele realizava múltiplas inferências através do TensorFlow Lite (TFL) executando modelos CNN pré-treinados. Também exploramos técnicas de aproximações que podem ser implementadas na camada convolucional, comparando sua perda de acurácia com a acurácia original do modelo, bem como o ganho de desempenho em tempo e energia correspondente a cada camada aproximada. Projetamos o ambiente e desenvolvemos as ferramentas que o compõem, incluindo programas para coletar e analisar os valores de corrente elétrica; ferramentas para combinar tais valores aos timestamps providos pelo nosso TFL modificado, programas para modificar camadas de CNNs existentes no formato flatbuffer, introduzindo camadas aproximadas; e ferramentas para avaliar a acurácia e desempenho de modelos aproximados. Observamos que, apesar de nem todas as camadas poderem ser aproximadas sem um impacto significativo na acurácia, identificar as camadas que podem ser sujeitadas à estratégia escolhida possibilita economia de tempo e energia que ultrapassa até 40% por camada aproximada. Em pelo menos um caso, conseguimos reduzir o consumo de energia em 5%, mantendo a perda de acurácia abaixo de 1%.

Abstract

In this work, we describe an environment for evaluation of execution time and energy consumption of Convolutional Neural Networks (CNNs) on mobile devices. We profile the performance and energy consumption of a device while it performs multiple inferences running a TensorFlow Lite (TFL) executable and select pre-trained CNN models. We then investigate approximations for the convolutional layer kernels, comparing their accuracy loss on the overall accuracy to their performance gain in time and energy on the device. We designed the environment, and developed the software that comprises it, including a tool to collect and parse electrical current output and match it to custom TFL timestamps, a tool to modify flatbuffer model files to replace layers with approximate versions, and the custom approximate kernels used to compare performance. We found that, whereas not all layers can be approximated without a significant impact on accuracy, identifying the layers that can be subject to the selected strategy can yield gains in time and energy that surpass 40% per approximated layer. In at least one case, we were able to reduce energy consumption by 5%, keeping accuracy loss under 1%.

Contents

Chapter 1

Introduction

1.1 Initial Considerations

Convolutional Neural Networks (CNNs) make powerful image classifiers [18], image "upscalers" that enlarge images while minimizing loss of quality [32], and can even be used for signal processing [24] and graph signal processing [13]. While their training may not be feasible for mobile devices due to performance, memory, and energy constraints, inference is not only possible [26], but an excellent choice for the backend of many applications, especially those that identify and classify characters, faces [36], food [22], pets, wild animals, and other elements in an image from the camera. This type of application, called "image classification", is the subset of CNN tasks that we target in our work.

In the context of neural networks, a generally good model may turn out to not be good enough for a certain application. For instance, strict timing demands may have to be met. That is, each inference must finish before a set amount of milliseconds have elapsed. As an example, if one were to integrate machine learning as part of the frame-rendering process of some Virtual Reality apparatus, they would have to finish the computation in at most 11.11 ms, lest the framerate will drop below 90 Hz, and the user may begin to experience nausea [27]. Additionally, one must be wary of energy consumption to allow a phone users' batteries to last through their day.

On the other hand, while neural networks are expected to not be 100% accurate, we still expect them to perform well under the scenarios they were trained for. Simplifying existing models may lead to gains in computational performance, but can we always afford the possibility of lowering accuracy, and the cost of re-training the network?

With these considerations in mind, we set out to identify the main sources of delay and battery drainage in popular CNNs, and to explore how approximate computing can be employed to shave off some milliseconds and millijoules from the execution of an inference.

1.2 Contributions

This project's goal is to profile and optimize energy consumption of a mobile device running inferences using a convolutional neural network model. We extended TensorFlow Lite to support profiling the time taken by each operation and, by matching timestamps of the profiled times with power measurements, we were also able to profile energy consumption. We also explored ways to approximate a convolution operation, trading off accuracy for efficiency.

First, we present our findings regarding time and energy consumption of existing networks. We argue that optimizing the convolution kernel is at the core of speeding up inference on mobile devices. Then, we show the results of applying selected approximate computing techniques from literature that we implemented in TensorFlow Lite.

We decided to focus on the implementations available in TensorFlow Lite (TFL) [39], a lightweight counterpart to TensorFlow (TF) [8] that is designed to only run inferences on previously trained networks, and not to train the models from scratch. TFL implements a subset of TF's operators, whose implementations are also referred to as "kernels". This subset is sufficient to support a large number of popular models, if not always directly, at least by converting operators or sequence of operators into other operators or sequence of operators, a task performed by the TensorFlow Lite Converter (TOCO).

We show opportunities to reduce power consumption by replacing an operator that causes great impact on battery usage with computationally cheaper versions, specifically by approximating the kernel for convolution using algorithms from linear algebra (Chapter 3) to the weights that were learned by image classification networks. We both evaluate the impact in model accuracy after the optimizations are applied, and argue for the upper bound in savings that can be achieved with the best strategy that we found, and how to achieve it. This strategy, in its current implementation is able to save up to roughly 40% of the energy consumed per layer, and keeping the accuracy drop within 1%, we obtained a 5% reduction to energy consumption of a model

We hope to not only inspire other researchers to build upon our results, but to share the tools that we used to obtain them. We were satisfied to hear that both the inference profiler and the modules that comprise the approximation pipeline have seen use in other research projects that feature CNNs and TFL at their core. The code is available at `https://github.com/SusinMat/Convolution-Profiling-and-Approximation`

Chapter 2

Related Work

In this chapter, we summarize what has been presented in papers that relates to designing efficient CNNs; improving or profiling the efficiency of machine learning frameworks; and optimizing or approximating operations commonly found in CNNs. We also briefly explore the extent of our project, comparing it to existing work.

2.1 Efficient convolutional networks

Our work focuses on Convolutional Neural Networks designed for image classification. While powerful tools for this task exist, achieving acceptable latency on less powerful devices still proves challenging. In this section, we present the basic ideas behind the three CNN architectures which spanned the seven models that we selected for our benchmarks.

SqueezeNet [20] achieves the same level of accuracy as AlexNet [25], the breakthrough network that won Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) [6] with a large margin over its competitors. SqueezeNet has 50x less parameters, and the ability to be compressed to 0.47 MB, 510x smaller than AlexNet. This is done by using more 1x1 filters while reducing the number of 3x3 filters, while also decreasing the number of channels passed to 3x3 filters, and delaying downsampling to preserve accuracy. In [16], a drop in accuracy of less than 1% was observed when using 8-bit data types instead of 32.

MobileNet [19] is a CNNs that aims to reduce latency of the inference, and the techniques employed also reduced its size in comparison to AlexNet, not necessarily at the expense of accuracy. The main pillar of its architecture is replacing what would be 3x3 convolutions with a 3x3 depthwise convolution that applies the same filter to all channels in one layer, followed by another lawyer that performs 1x1 convolution that linearly combines the outputs of the depthwise convolution. It then uses batchnorm and ReLU for non-linearity at the end of each sequence of convolutions, except for the last one, in which softmax is used. There are also hyper-parameters for linearly reducing the number of input and output channels of each layer. One of the targets of our work are the first and the second [35] versions of MobileNet.

Inception [38] is the codename to the architecture that set the new state of the art for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge

2014 (ILSVRC14), submitted by the team known as GoogLeNet. It was designed to increase depth and width while maintaining a computational budget, in this case, measured in multiply-and-accumulate floating point operations (MACs). For most of the experiments, the budget was set to 1.5 billion MACs. The module of the original Inception architecture has one 3x3 max pooling operation, one 3x3 convolution, one 5x5 convolution, and four pointwise (1x1) convolutions, which are then concatenated before the next layer or module. In our work, we analyzed and targeted with approximations later versions of Inception, which replaced some max pooling operations with average poolings. This proved an interesting choice, since, in contrast with Mobilenet, whose modules are a 3x3 depthwise convolution followed by a 1x1 convolution, Inception features 3x3 and 5x5 convolutions that we can apply approximations to.

2.2 Software-level optimizations

This section presents strategies designed to speed up CNNs without the need for specialized hardware. Applications that use such models already expect an accuracy that is significantly lower than 100% due to the nature of machine-learning, making them good candidates for the approximation of certain operations. As such, we chose to focus on works that put emphasis on approximate computing strategies for convolutions.

The algorithms presented by Manne et al. [30] approximate the multiplication of two equal-sized square matrices. The algorithms designed by the authors run in quadratic time, and the Frobenius norm of the error matrix can be made arbitrarily small. This is done by reducing the problem to that of solving a set of linear equations, for which approximations are known. However, this algorithm typically requires a large matrix multiplication to justify its overhead, whereas, convolutions in CNNs tend to use 3x3 and 5x5 kernels.

Jaderberg et al. [21] devised schemes to speed up the evaluation of a convolutional neural network by exploiting redundancies between different filters and feature channels. The authors provide two schemes to construct filters that are rank 1 in the spatial domain. Unlike the previous technique, this one works well for small filters. Combining the two schemes presented in their paper, the authors achieved up to 2.5x speedup with no loss in accuracy.

The approach by Baboulin et al. [9] shows that mixing 32-bit and 64-bit representations can accelerate scientific computations with no loss in accuracy. In particular, they solve linear systems for dense and sparse matrices using LU decomposition and Newton's algorithm, which can be extended to find eigenvalues.

The usage of memoization in convolution is explored by Khalvati et al. [23]. In their strategy, the results from the multiplication followed by a sum of the square sliding window (kernel) on a grayscale image are stored in memory, so they can be retrieved later if the window is slid over a similar (rather than identical) submatrix. The cache must be small enough to fit into the faster, lower levels of the physical cache, while also being large enough to provide enough hits. It is also worth noting that in order to increase the hit rate, the authors map the 256 levels of gray to only 16. This leads to the final result being approximated with a loss of accuracy of at most 2.2% in the 3 convolutional operators that were tested. The speedup varied from 1.30x to 6.27x, and was higher on images that had lower average local entropy (the detailedness metric used in the paper).

An efficient softmax approximation has been proposed by Grave et al. [14]. The authors attempt to model the computation time of matrix multiplication, acknowledging that it is not trivially linear in the dimensions of the matrices. Despite the results having shown the potential of approximate computing, the focus was on desktop GPU architectures rather than mobile ones, and the target to approximate were language models, rather than image classification.

Halide [33] [31] is a language and compiler that decouples the lower level implementation of image processing operations, which must be optimized, from the actual description of the algorithm, the processing pipeline. It makes it easier for the programmer to experiment with multithreading, tiling, locality, caching vs. recomputation, and other settings whose optimal values may differ according to the architecture and application. What we take away from Halide is that, whenever the choice space is large, exploring different settings should be as simple as possible, without needing to rewrite modules of the program that are not directly related to the knobs we want to turn.

XNOR-Net [34] explores Binary Weights Networks, in which 32-bit floating point weights are approximated to just one bit, resulting in 32x memory saving and roughly 2x speed-up, and XNOR Networks, in which both the filters and the input to convolutional layers are binary, reducing the number of high-precision operations by a factor of 58x. Binary networks had been experimented with before, but only on small datasets, such as CIFAR-10, MNIST, SVHN. XNOR-Net trains on the ImageNet dataset. At the time of its publication, running CNNs designed for ImageNet in real time was a task generally feasible only on GPUs, whereas XNOR-Net could run in real time on a desktop CPU. The first approach achieved up to 56.8% top-1 accuracy and 79.4% top-5 accuracy, and the second approach achieved up to 44.2% top-1 accuracy and 69.2% top-5 accuracy.

Denton et al. [11] attempt to optimize convolutional layers by applying approximations. Their strategies, by design, reduce the amount of computation by exploring the redundancy between channels and filters, using theorems from Linear Algebra. Note that the actual performance of the layer is not trivially linear on the amount of computation, so real world results may differ greatly from the analytical reduction in model size and computation. In our work, we select a few of these approximation methods to test their viability on modern CNNs running on mobile devices, comparing approximate networks to their original counterparts in accuracy, speed, and energy consumption.

Zhang et al. [40] employ the decomposition used by [21] with a different optimization strategy, while simultaneously implementing a strategy to decide by how much should the rank of the convolution be reduced to maintain good accuracy. To further contain the accuracy drop, the networks were re-trained using the approximate weights as initial values, and the authors observed that, with a small enough learning rate, the speed in which the training converges to a solution of a certain achievable level of accuracy is faster than with the original model being initialized to random values. This suggests that retraining an approximate network is faster than retraining a new network from scratch.

2.3 Hardware-level optimization

Gupta et al. [15] train deep neural networks with 16-bit fixed-point number representations that use stochastic rounding, and compare its performance with a standard IEEE 32-bit floating-point representation, with little to no degradation in accuracy. They also demonstrate the energy efficiency of a hardware accelerator that implements the former, with very promising results in savings and throughput.

Ristretto [16] is a 2016 open-source framework for CNN approximation that uses highly optimized routines that run on the GPU. It extends Caffe, and offers conversion of network data types from 32-bit float to quantized unsigned integer of 8 or 16 bits, and 6-, 8-, and 12-bit minifloat. These converted models are simulated, and their accuracy is evaluated. In some cases, especially when 12-bit minifloat or 16-bit quantized unsigned integer are being used, relative accuracy drops by no more than 1%. This work also provides insights on and means to determine how to quantize parameters, given a desired data size, allowing different operations to use different quantizations, focusing on minimizing accuracy loss. Its biggest flaw, in our opinion, is not comparing power usage, even though it claims to be one of its major concerns.

2.4 CNN performance analysis

DeepSense [29] is a framework designed specifically for mobile devices with a GPU. The authors explore the differences between server-class and mobile-class GPUs, and study the effectiveness of various optimization strategies (e.g. branch divergence elimination, memory coalescing, memory vectorization), and even approximation strategies, such as converting weights to 16-bit floating points number to double the bandwidth when offloading data to the GPU, resulting in decent performance gains for minimal accuracy trade-off.

NeuralPower [10] shows how polynomial regression can be used to model computational performance of a CNN by breaking it down into layers, and training a polynomial model to predict time, power, and energy consumption of the neural network. Such model defeated state-of-the-art performance predictors that were not specifically trained for CNN inference. While network performance prediction is out of scope for our project, we developed tools that allow models such as NeuralPower, which was only tested on a GTX 1070 (a desktop GPU), to use data collected from inferences ran on mobile devices. The authors report 88% accuracy in time and power prediction, and 97% accuracy in energy prediction.

Another study on energy efficiency of (Deep) Convolutional Neural Networks was performed by Li et al. [28]. Their work is interesting in that it compares several frameworks (e.g. TensorFlow, Caffe, Torch, Nervana), running inferences on CPU and on GPU, and with different settings enabled (e.g. Hyper-Threading, Error Correction Code in the RAM, Dynamic Voltage and Frequency Scaling). The authors present the average power, average completion time, and average energy consumed per image, and breaking it down into the operations of each layer, in a similar fashion to our analysis here. They found that convolutional layers consume about 87% of the energy required per inferece, fully connected layers need 10%, and activation layers less than 5%. However, their focus was on desktop setups, using an Intel Xeon CPU and an NVidia Titan X GPU, very powerful hardware that differ immensely from the mobile platforms that we target in our work.

2.5 Our project

In a similar direction to the aforementioned works, we intend to optimize the cost of running an inference on a convolutional neural network. Given that proposing new or modified hardware is beyond the scope of this project, we will focus on software optimizations. The ability to profile the energy consumption of inferences on TensorFlow Lite is also a contribution of this work. Our instrumentation can be used to provide energy-aware scheduling, or to assist in designing networks according with specific time and power constraints, which are out of scope for this project.

Chapter 3

Hypotheses

We hypothesize that a trained a network contains some amount redundancy in its filters, and that it may be possible to achieve better time and energy performance on mobile devices, without a significant loss of accuracy, by reducing such redundancies in certain layers that may contain them. We follow the strategies proposed in [11], and build upon TensorFlow Lite (TFL).

3.1 Approximation strategies

In the following subsections, we describe the strategies that we employed to approximate the convolution operator.

3.1.1 SVD factorization

This strategy is centered around the application, or rather, two successive applications, of the Singular Value Decomposition (SVD) to the weight tensors of certain convolution layers. The goal is to reduce the number of floating point operations performed by that single layer, and this strategy also shrinks the size of the layer, reducing the size of the final model file.

To understand how SVD can be applied to approximate an operator, consider the example of a convolution layer that takes as input a `35x35` matrix, 64 channels deep, and outputs another `35x35` matrix, this one being `96` channels deep. The tensor of weights, which we will call `W`, is a `96x3x3x64` matrix, where `3x3` is the shape of the window, `64` is the number of input channels, and `96` is the number of output channels, as previously described.

In summary, we perform a small number of pointwise convolutions with a certain, configurable number of filters each, which we call “phase C”. We then perform “phase Z”, a small set of `3x3` convolutions on the output of phase C, which contains less operations than the full `3x3` convolution of the original weight matrix `W`, and finally, we combine the outputs of phase Z through another small set of pointwise convolutions, which we call “phase F”.

The algorithm we used to decompose matrix `W` has, in total, 4 parameters, or "knobs", that can be adjusted according to the desired levels of accuracy and and computational

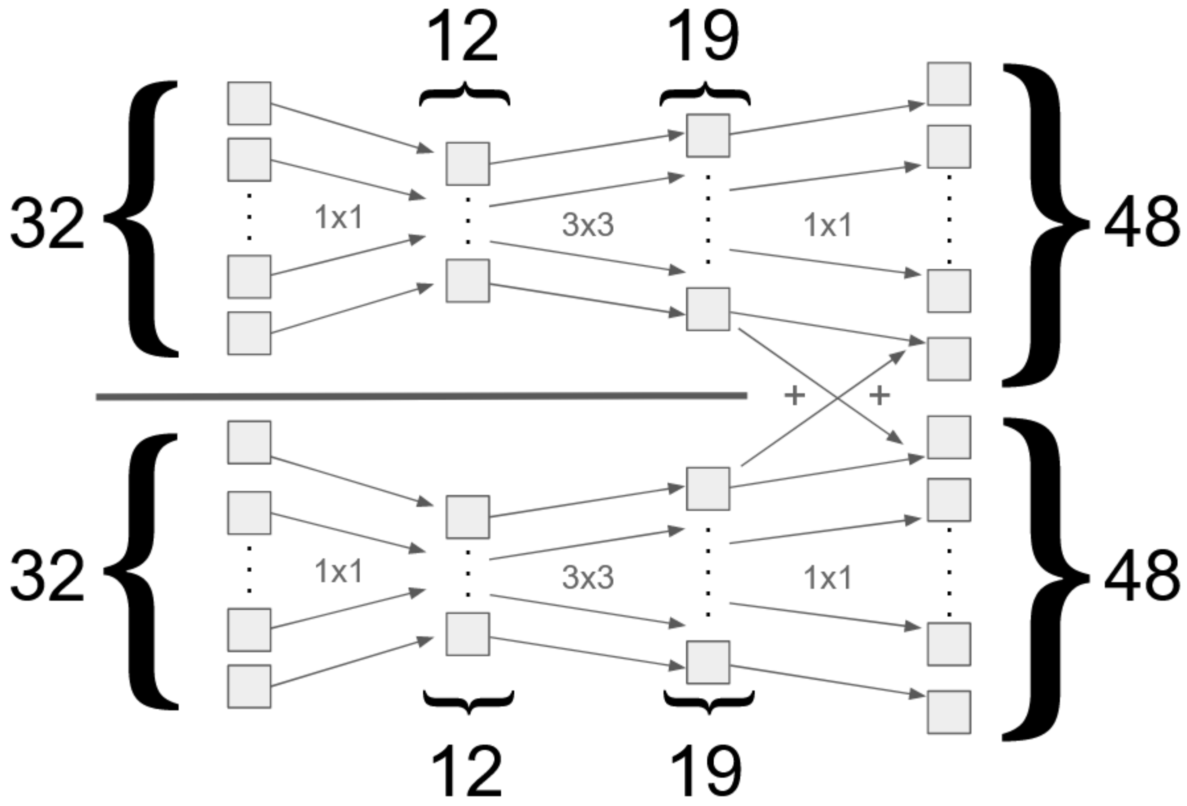

Figure 3.1: A visualization of how this approximation operates after being implemented. Note that this shows the behavior of the function during the inference, not how we compute its weights.

cost that are desired. Two knobs are `iclust` and `oclust`, which determine how many clusters of input channels and of output channels we will have. The two other knobs are `iratio` and `oratio`, which determine the factors (in the `[0.0, 1.0]` range) that we will multiply the rank of the input and the output clusters by, reducing the total computational cost of the convolutions.

As part of the example, assume the following settings: `iclust = 2; oclust = 2; iratio = 0.4; oratio = 0.4;` .

The first step is to take the 96 filters of the `W` matrix, and fold the other (`64x3x3`) dimensions, resulting in a matrix of shape `96x576`. We run a modified k-means clustering algorithm, which is identical to the well-known k-means, but with an added restriction of all clusters being of the same size. Note that the value for `iclust` must divide the number of input channels, and the value of `oclust` must divide the number of filters (channels in the output).

After clustering, we are left with `96 / 2 == 48` output filters in each cluster.

We perform the same procedure for the input channels, folding `W` into a matrix of shape `64x864`, clustering with the same implementation of k-means, resulting in `iclust` clusters with `64 / 2 == 32` input channels in each. Note that `iclust` must divide the number of input channels.

The second major step is where we apply SVD. In summary, after the input channels and output filters have been clustered, we apply SVD to each pair of input and output cluster, removing the singular vectors that correspond to the smallest singular values, such

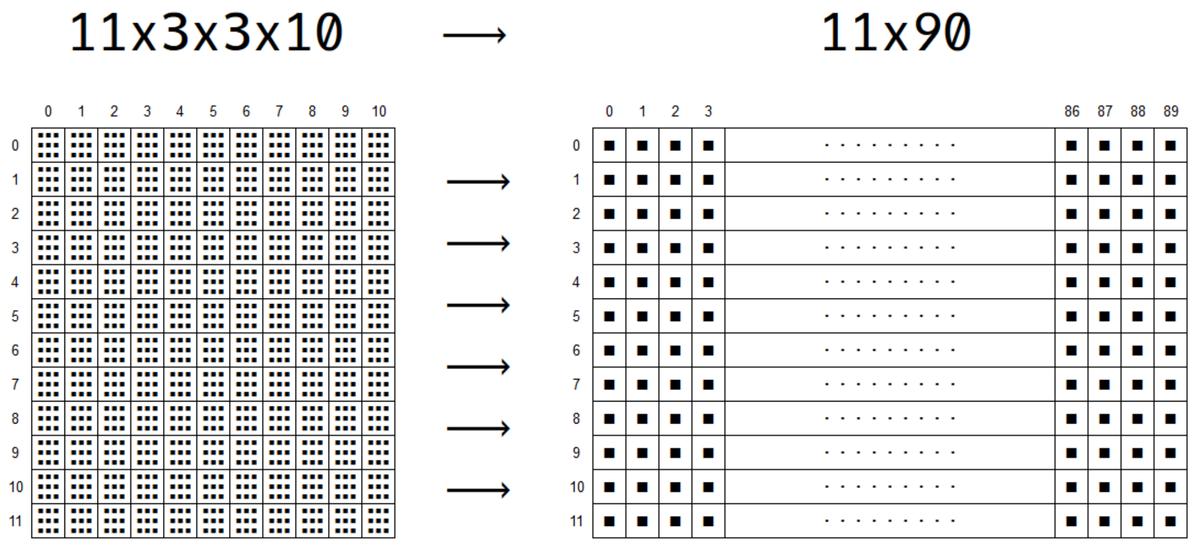

Figure 3.2: A visual representation of how folding works. Unfolding is the inverse of this reshaping operation.

that only `oratio * 48 == 19` values remain from each output cluster, and only `iratio * 32 == 12` remain from the input cluster.

To understand how this works, and how we can compute an approximation for the original convolution using matrices of lower ranks, consider three matrices, initialized to `0`, named after the three phases of computation, and the following shapes:

```
C - (32, 12, iclust, oclust) == 32x12x2x2
Z - (19, 3, 3, 12, iclust, oclust) == 19x3x3x12x2x2
F - (48, 19, iclust, oclust) == 48x19x2x2
```

We want a total of three stages, described below:

- Transform 1 (phase C) ⇒ a total of `iclust * oclust` (4) pointwise convolutions, each with an input of size `35x35x32`, where 32 is the number of input channels that it will take, according to the input cluster that it is derived from. The output of these 4 pointwise convolutions is 4 `35x35x12` matrices.
- Convolution (phase Z) ⇒ a total of `iclust * oclust` (4) convolutions with filters of the original height and width, `3x3` in our case, 12 input channels and 19 filters each. Note that this convolution, plus the two transforms in this list, are replacing the original `96x3x3x64` convolution with 6 pointwise convolutions, and 4 `19x3x3x19` convolutions.
- Transform 2 (phase F) ⇒ a total of `oclust` (2) pointwise convolutions, each with an input of size consisting of the concatenation (in the channel dimension) of the `iclust` (2) corresponding `35x35x19` matrices. The output of each of these convolutions is a `35x35x48` matrix, which are merged into a single `35x35x96` matrix, that approximates the result of the original `96x3x3x64` convolution.

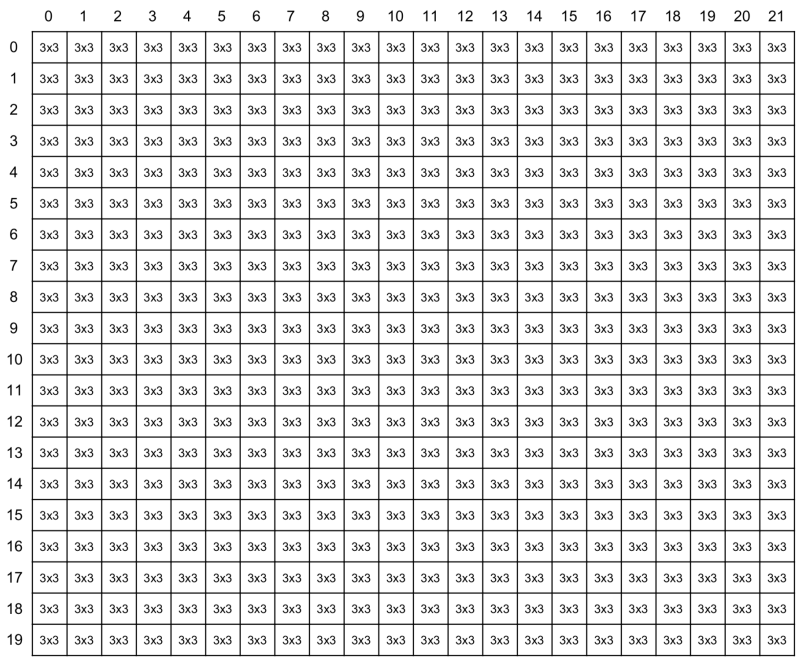

Figure 3.3: First step of taking a slice from two cluster indices: a 2-D array.

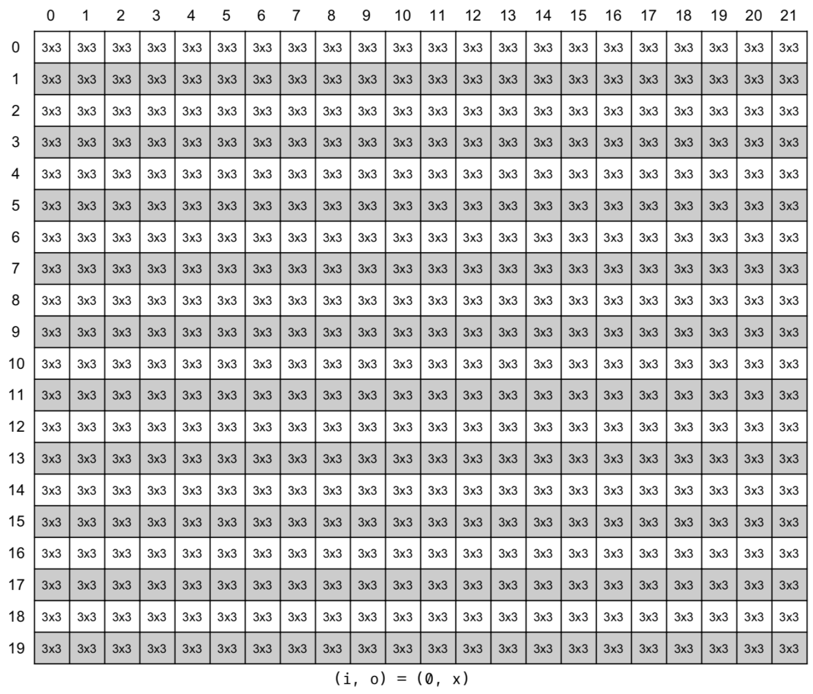

Figure 3.4: Second step: select every input channel that was assigned to input cluster `i == 0`, shown in white.

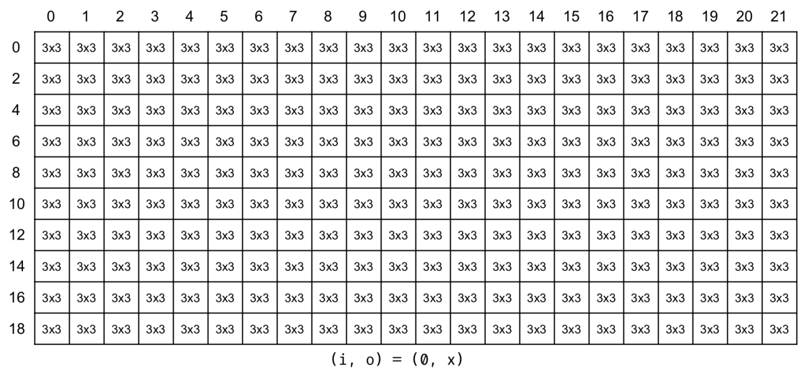

Figure 3.5: Third step: delete the remaining input channels, previously shown in gray.

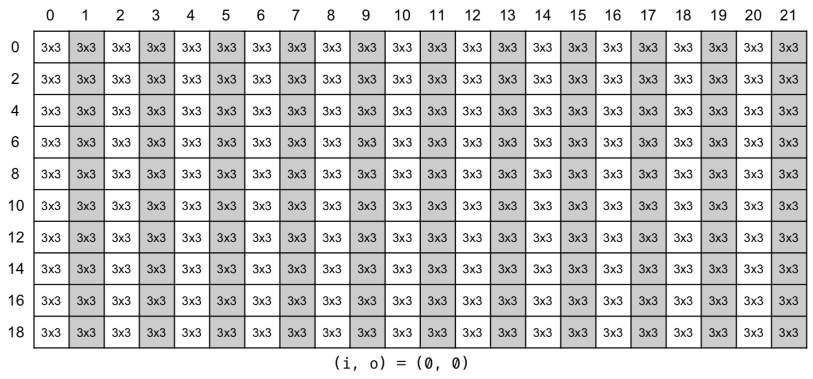

Figure 3.6: Fourth step: select every filter that was assigned to output cluster `o == 0`, shown in white.

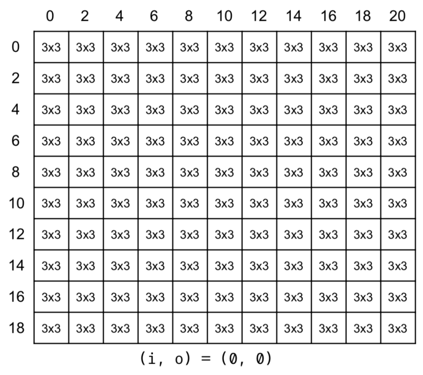

Figure 3.7: Fifth step: delete the remaining filters, previously shown in gray.

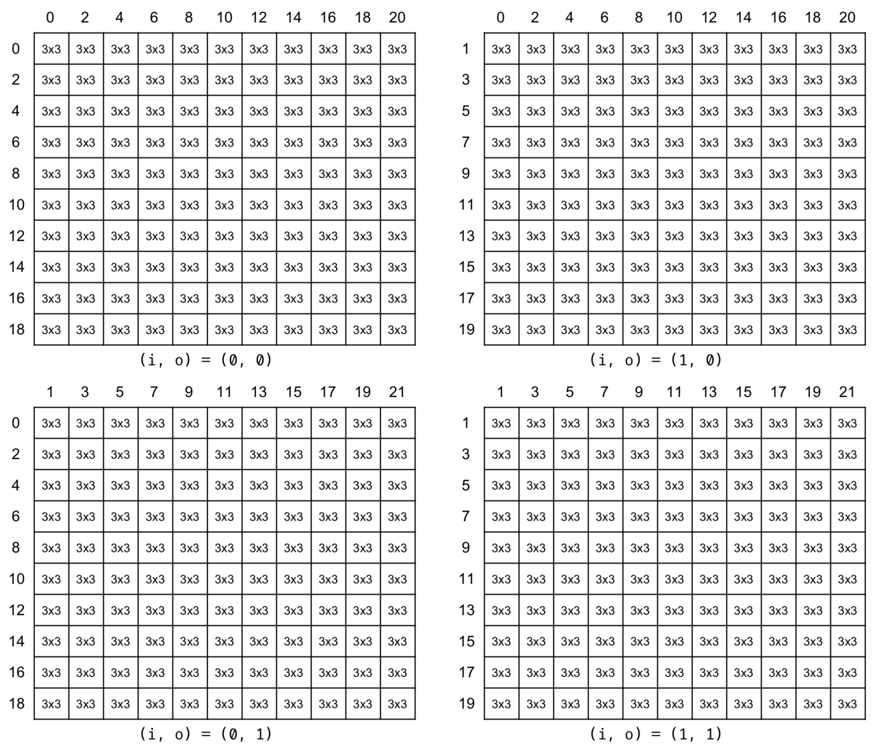

Figure 3.8: Repeat steps for all 4 possible combinations of `(i, o)`.

Consider the algorithm fresh out of the modified k-means clustering process. There, we can index the input clusters with integers from the `{0, ... iclust - 1}` range, and output clusters with the `{0, ... oclust - 1}` range. In our example, this leaves us with two input indexes, two output indexes, and a total of four possible pairs of indexes, `(0, 0), (0, 1), (1, 0), (1, 1)`. For each of these pairs, we perform two successive SVD decomposition, as explained in the following paragraphs.

Singular Value Decomposition, available in the NumPy module for Python3 as `numpy.linalg.svd`, takes as input an `MxN` matrix of real values that we will call `A`, and decomposes it into `U`, `S`, and `V`, where `U` is a square `MxM` matrix, V is a square `NxN` matrix, and `S` is a rectangular diagonal matrix with shape `MxN`. The non-zero elements of `S`, called "the singular values of `A`", are the square root of the non-zero eigenvalues of both the product of AA^T and the product of A^TA. Additionally, we have that $A = USV^T$.

For a formal and more detailed description of SVD and its early history, read [37].

Note that we can apply SVD recursively to V, further decomposing the original matrix, and for this algorithm, we do exactly that, after reducing the rank of the input.

Now we turn back to our example. For every pair of input cluster and output cluster `(i, o)`, we create a slice of the original weight matrix, with only the filters and input channels that were assigned to each cluster. In our example, this results in slices of shape `48x3x3x32`. We proceed to fold this slice into a `48x288` matrix. To the folded matrix, we apply SVD, obtaining `U`, `S`, and `V`, as described in the previous paragraphs. We delete matrix columns and rows from `S` to keep only 19 singular values in `S`, leaving it as a matrix with shape `19x19`, and also remove columns from `U` to make it `48x19`.

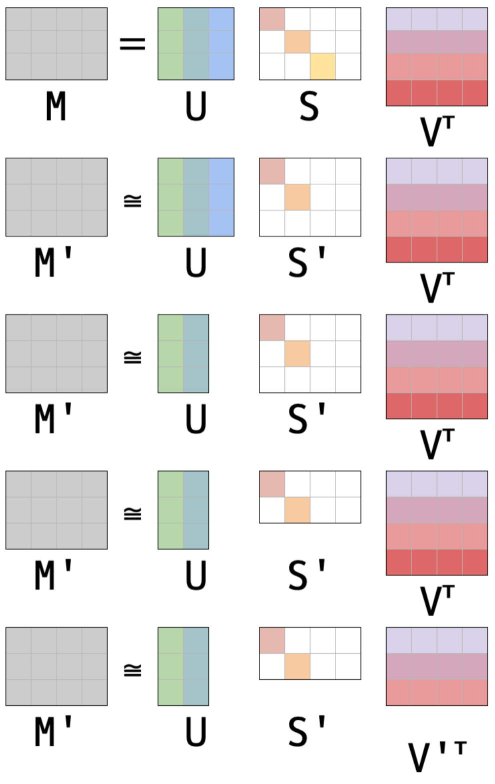

Figure 3.9: A visualization of how we apply the Eckart-Young theorem to approximate a matrix.

This is an application of the Eckart-Young theorem [12]. We can approximate matrix M with another matrix M', said to be "truncated". Which has a specific rank r. To minimize the Frobenius norm of the difference between M and M', we can compute the U, S, V matrices, and replace all except the r largest singular values in S with 0. The product USV^T now gives us M'. A visualization of this application of the theorem is represented in Figure 3.9.

Note that `U` and `S` can still be multiplied, resulting in a `48x19` matrix that we save into `F[:, :, i, o]`.

We turn back to matrix `V` that resulted from the first SVD decomposition of the `(i, o)` slice. We first delete columns from `V` such that the product SV^T is still valid and USV^T still results in a `48x288` matrix, but of lower rank than the original folded slice. We then unfold `V` into `19x3x3x32`, permute its shape into `32x19x3x3`, and fold it again, this time into shape `32x171`, so that SVD can be applied again.

Once again, we run SVD, which outputs factors `U`, `S`, and `V`. We repeat the steps of deleting entries to reduce the size of `U` and `S`, this time keeping 12 singular values in `S`, rather than 19. We store the product of US in `C[:, :, i, o]`.

From `V`, we once again remove columns, such that V is converted into a `171x12` matrix. We unfold V into a `19x3x3x12` matrix and save it into `Z[:, :, :, :, i, o]`. Note that V is a weight matrix for a 3x3 convolution with 12 input channels and 19 filters. Each `C[:, :, i, o]` and `F[:, :, i, o]` element can be unfolded into a `12x1x1x32` and a `48x1x1x19` weight matrix, respectively, adequate for a pointwise convolution.

The previous steps, which produce the matrices for the three phases, only needs to be computed once per layer of the model that we wish to apply the approximation to. Once we have `C`, `Z`, and `F`, we can compute an approximation to the output of this layer during inference as follows:

For every `(i, o)` pair of input, output clusters, take element `C[:, :, i, o]` of matrix `C` and unfold it into `32x1x1x12`. Perform a pointwise (`1x1`) convolution on the input that uses the unfolded element of C as kernel, and only the input channels that are in cluster `i` as input. Note that, due to there being `iclust * oclust` possible `(i, o)` pairs, of this process is repeated 4 times if both these knobs are set to 2.

Now for each result of the previous step, take element `Z[:, :, :, :, i, o]` and perform the corresponding 3x3 convolution.

Finally, for every output of the previous step, we take element `F[:, :, i, o]` of matrix `F`, unfold it into shape `48x1x1x19`, and perform pointwise convolutions. Note that of the 4 inputs to this phase, 2 are assigned to each input cluster, and 2 are assigned to each output cluster. This means, while each output cluster contains 48 channels, there are two intermediate results whose convolutions will produce values assigned to these channels. Therefore, we will accumulate the result of both `(i=0, o=0)` and `(i=1, o=0)` into the 48 channels of the output cluster of index 0, and also accumulate the result of both `(i=0, o=1)` and `(i=1, o=1)` into the 48 channels of the output cluster of index 1. This completes the main computation of the layer. If there are any biases to add, and activation functions to compute, we can perform them normally on the output of the approximate convolution.

The `C`, `Z`, and `F` matrices that we computed can then be used to compute an approximation to convolution layer of weights `W`. The previously described "phases" are how we can effectively reduce the number of computations required by this layer during the inference, but we can also construct a matrix `Wapprox` of the exact same shape as `W` that emulates the precision of an approximate operation that uses the 3 phases, so we can experiment with the accuracy drop without implementing the approximate operation itself into the accuracy benchmark.

To calculate `Wapprox`, we initialize a matrix with the same shape as `W` with all zero values. For every `(i, o)` pair, we take the corresponding slice from `C (32x12)`, `Z (19x3x3x12)`, and `F (48x19)`. We fold the `Z` slice into `171x12`, and multiply that and $\mathtt{C}^{\mathrm{T}}$ to obtain `ZC` of shape `171x32`. We unfold `ZC` into `19x3x3x32`, and fold it into `19x288`. Finally, we multiply the `F` slice and `ZC`, resulting in an array of shape `48x288`, which can be unfolded into `48x3x3x32`. Notice that 48 refers to the number of filters in an output cluster, and 32 refers to the number of channels in an input cluster. We add each `3x3` window to its corresponding position in the `Wapprox` matrix. At the end of the process, we have completed a weights matrix that emulates the behavior of the approximation on accuracy using the exact implementation.

3.1.2 Monochromatic approximation

This strategy replaces a normal 2D Convolution operation with 2 operations: the first being a pointwise convolution, that is, a linear combination of input channels, and the second being a depthwise convolution with high depth multiplier. This seeks to exploit similarities between different filters of the first convolutional layer, as evidenced by the clustering of output channels.

Assume, for the purpose of this explanation, that the number of input channels is 3 (RGB), the number of output channels (filters) is 32, the vertical and horizontal strides are both 2, and that the window of the convolution is `3x3`. This is the case for the first layer of MobileNet version 1.

The monochromatic approximation takes as input a weight matrix of shape `W` and the desired number of intermediate channels, which is a configurable knob called `num_colors`, and returns the following data:

Wapprox New weight array of the same shape as the input. By replacing the weights of the original layer with this, we can construct a model that emulates the impact that the approximation will have on the layer, while the computational cost remains unchanged. That is, we can determine if the approximation is viable from a network accuracy standpoint before we even begin to measure its benefits to execution time and energy consumption.

Wmono New weight array for the Depthwise Convolution. By applying one of the filters in Wmono to its respective intermediate channel, we obtain an approximation of an output channel.

colors An array of linear combinations that are used to compute the intermediate channels/colors. For instance, if we have num_colors == 4, the colors array will map each of the 4 intermediate channels to a triple (r, g, b). The pointwise convolution of an input image by a weight matrix of shape `1x1x3` yields one of the (4, in this example) intermediate channels.

perm A list that assigns each of the (32, in this example) output channels to an intermediate channel.

num_weights This is the number of numeric values required by the approximate computation. Comparing this to the size of the original `W` matrix provides the compression ratio.

The 'colors' array will be the core of the first sub-layer that approximates the target layer, and a combination of the 'perm' list and the 'Wmono' weight array will be the core of the second, more expensive sub-layer.

It should be noted that, ideally, we want an even clusterization, that is, we want to cluster the filters such that the number of filters assigned to each intermediate channel/color is identical – and therefore, equal to channel_outputs / num_colors. This allows the implementation of the operation to take more advantage of GPU and CPU parallelism. However, such constraint does not exist in kmeans, in which case two alternatives may be exploited: the quickest one, which is implementing an even clustering algorithm, with possible impact on accuracy, and the laborious one, which is implementing a custom operation that allows for an uneven depth multiplier, preventing the accuracy drop caused by introducing evenness as a constraint to the clusterization.

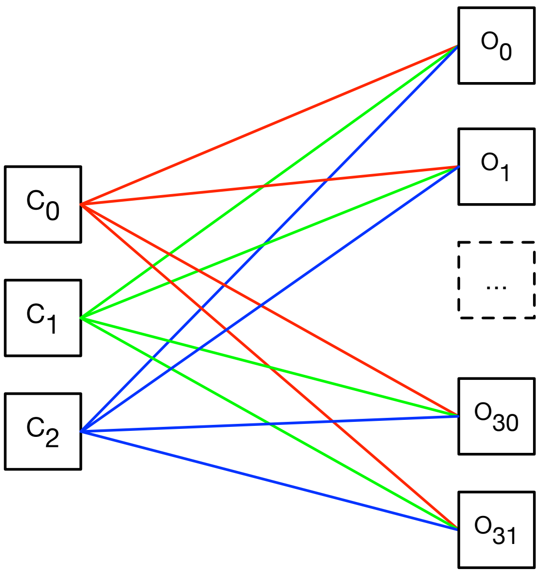

Figure 3.10: Original layer. Each output channel uses all input channels to process results.

The latter alternative would be to replace the depth_multiplier parameter, which defines how many output channels are calculated from each input channel, from being an integer to being an array of integers, each indicating how many filters will be applied to each input channel. Due to how vectorized instructions and memory/cache management work, it is possible that some clusterizations would be less efficient than the even variant.

Figures 3.10 and 3.11 represent the approximation process. In the original configuration, each output channel processes information from all input channels. In the approximate version, intermediate channels are generated from linear combinations of the input channels, and each output channel processes information from a single intermediate channel. Figure 3.10 output channels are generated, with each one processing information from 3 input channels (e.g. the RGB channels from the image in the first layer of the network). In the approximate version (Figure 3.11), the structure of the network is preserved, with 32 output channels being generated from 3 input channels. In the approximate version, however, 4 intermediate channels are generated as linear combinations of the original three input channels. Each output channel uses a single one of these intermediate channels to produce its results.

In order to calculate the `Wmono`, `colors`, and `perm` matrices, we perform the following steps:

We initialize two important matrices, `C`, of shape `(filters) x (channels)` (32x3 in our case), and `M`, of shape `(filters)x(height * width)` (32x9 in our case).

We start by transposing the input matrix of weights from its original `(filters) x (height) x (width) x (channels)` shape into `(filters) x (channels) x (height) x (width)`.

Then, we iterate over each filter from the resulting matrix, of shape `32x3x3x3`. and fold the last 3 dimensions, obtaining an array of shape `(channels) x (height) x (width)` (`3x9`). We perform an SVD factorization on this array, obtaining USV^T. We take the first colum of `U`, of shape `(channels)`, and assign it to its corresponding (according to

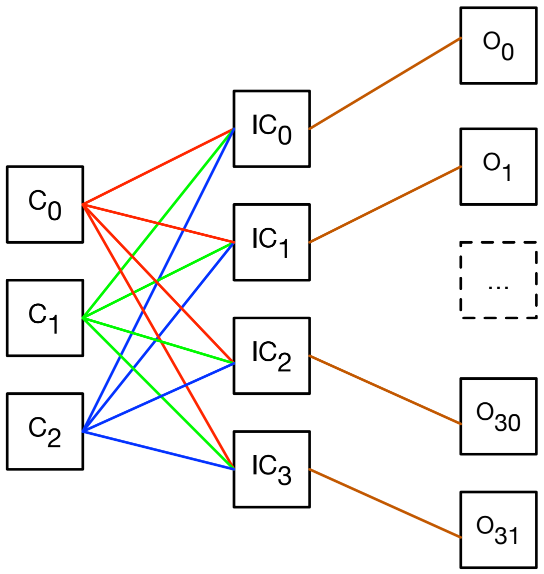

Figure 3.11: Approximate layer. Each output channel uses a single intermediate channel to process results.

iteration) line of `C`. Finally, we multiply the first eigenvalue in `S` with the first column of `V`, obtaining an array of shape `(height * width)`, (`9` in our case), and assign it to its corresponding line in `M`.

Once we have iterated over all filters, we transpose `C`, and feed it into a kmeans clustering algorithm. The result is an assignment of each filter to one of `num_colors` (`4`, in our case) intermediate channels, as well as a list of `num_colors` centroids, each of shape `(channels)` (`3`, in the case of RGB). This list can be reshaped into an array of shape `(num_colors) x 1 x 1 x (channels)` (`3x1x1x4`), which serves as a filter for a pointwise convolution that computes `num_colors` intermediate channels from an RGB input channel. This is our `colors` matrix. From this execution of kmeans, we also obtain a list that assigns each filter to a centroid. We use this same list to assign output channels to intermediate channels, returning it as the `perm` matrix.

The matrices `perm` and `colors` are not sufficient to compute an approximate convolution. Recall from when we performed the SVD factorization, and filled matrix `M` with the product of values from `S` and `V`^{T}. This matrix can be reshaped to `(filters) x (height) x (width) x 1` (`32x3x3x1`). According to the `perm` matrix, we can assign each of the slices shaped `1 x (height) x (width) x 1` to an intermediate channel, which will then be the weight matrix of a depthwise convolution that takes that intermediate channel as input, and produces one of the output channels, according to the number of filters that the original convolution had. We therefore perform this reshaping of `M` to return it as `Wmono`.

We are then ready to compute `num_weights`, which is the sum of the number of dimensions weights of the three matrices that are necessary to compute the approximate version of the convolution.

Finally, we can also provide `Wapprox`, which can simulate the effect of the approximation on accuracy without changing the operation used by the first layer. To compute it, we iterate over the lines in `M`. For every line index `f`, we compute the product colors[perm[f]] * M[f], that is, the product of the weights that take the input image's channels to the intermediate channel that corresponds to f by the weights in Wmono that will be applied to f. We unroll this product into 1 x (height) x (width) x (channels), which is 1x3x3x3 in our case, and assign it to a line of Wapprox. At the end of this process, Wapprox will be an tensor of shape (filters) x (height) x (width) x (channels) (32x3x3x3) that simulates the impact on accuracy of having each output channel (filter) approximated by this method.

3.2 Complexity analyses

In the following subsections, we describe the intuition behind why an approximate convolution operator, constructed as we described in the previous section, may be computationally cheaper than the exact convolution operator.

3.2.1 SVD factorization

Consider the three phases. We will count the number of floating-point multiply and accumulate (MAC) operations in each phase.

Additionally, consider the following values:

in_s the height and width of each input channel, which we also call a "pixel", 35 in our example;

out_s the height and width of each output channel, which we also call a "pixel", 35 in our example;

iclust the number of input clusters, 2 in our example;

iclust_sz the size of each input cluster, 32 in our example;

oclust the number of output clusters, 2 in our example;

oclust_sz the size of each output cluster, 48 in our example;

idegree the number of channels in each intermediate input cluster, after rank reduction, 12 in our example;

odegree the number of channels in each intermediate output cluster, after rank reduction, 19 in our example.

w_height and w_width, the height and width of the convolution window in phase Z, the only one wherein the convolutions are not pointwise, 3 in our example.

- Transform 1 (phase C) ⇒ For every input cluster and output cluster, we iterate over input pixels in the clustered input channels, applying idegree 1x1 filters. The cost of this phase is, therefore, iclust * oclust * in_s * in_s * iclust_sz * idegree.

- Convolution (phase Z) ⇒ For every input cluster and output cluster, we iterate over input pixels in the reduced number of output channels, applying odegree 3x3 filters. The cost of this phase is, therefore, iclust * oclust * out_s * out_s * idegree * odegree * w_height * w_width.

- Transform 2 (phase F) ⇒ For every input cluster and output cluster, we iterate over output pixels in reduced number of output channels, applying `oclust_sz 1x1` filters. The cost of this phase is, therefore, `iclust * oclust * out_s * out_s * odegree * oclust_sz`.

Additionally, as a model compression strategy, we replace an array of size `(filters) * (w_height) * (w_width) * (channels)` with an array that assigns input channels to clusters, one that assigns output channels to clusters, and a few arrays being used by convolutions. Namely:

- phase C ⇒ `iclust * oclust * iclust_size * idegree`.
- phase Z ⇒ `iclust * oclust * idegree * odegree * w_height * w_width`.
- phase F ⇒ `iclust * oclust * odegree * oclust_sz`.

3.2.2 Monochromatic approximation

Depending on the number of input channels generated, the approximate layer may require a number of operations significantly smaller than the original computation. The number of operations to produce a convolution layer C may be estimated as:

$$C = \frac{h}{s_h} \times \frac{w}{s_w} \times c \times k \times f \tag{3.1}$$

where h is input height, w is input width, s_h is vertical stride, s_w is horizontal stride, c is the number of input channels, k is the number of outputs (filters), and f is the size of each filter.

As an example, the first layer of MobileNet v1 uses three input channels, 32 output filters, each with size equal to 9, and a horizontal and vertical stride of 2, with the following complexity in the original network:

$$C_o = \frac{h}{2} \times \frac{w}{2} \times 3 \times 32 \times 9 = 216 \times h \times w \tag{3.2}$$

The approximate layer divides computation into two internal layers. The first layer uses a stride of one to generate a number of intermediate channels (four in our example), each using a filter size of one. Complexity for the first approximate layer may be estimated as:

$$C_{a1} = h \times w \times 3 \times 4 \times 1 = 12 \times h \times w \tag{3.3}$$

The output layers rely on a single intermediate channel, generating 32 output channels, each with a filter size of 9, and using stride 2.

$$C_{a2} = \frac{h}{2} \times \frac{w}{2} \times 1 \times 32 \times 9 = 72 \times h \times w \tag{3.4}$$

Total complexity for the approximate layer is:

$$C_a = C_{a1} + C_{a2} = (12 + 72) \times h \times w = 84 \times h \times w \tag{3.5}$$

Compared with the original first layer of MobileNet v1, the approximate layer using four intermediate channels reduces the total number of operations by a factor of ∼2.6x. Note that this first order estimation ignores any potential overheads and optimizations.

Additionally, as a model compression strategy, we replace an array of size `(filters) * (window_height) * (window_width) * (channels)` with an array that assigns output channels to intermediate channels, and a few arrays being used by convolutions. Namely:

- colors ⇒ `num_colors * channels`.
- Wmono ⇒ `num_colors * filters * w_height * w_width`.

Chapter 4

Methods

In this chapter, we introduce some of the elements that were chosen to test our hypotheses. First, we explain the design of our approximation pipeline from a dataflow perspective. Then, we list the CNNs that were selected to be subject to approximations, as well as contextualize the dataset that we used to test our approximations. Finally, we describe the software tools and hardware platforms that we selected for our work setup.

4.1 Approximation pipeline

We wanted to experiment with approximations while keeping the system modular and easy to develop and test. Each component can perform its task using the output of the previous component that is saved to a file. This made it quick to test new modifications, and different strategies and knobs for approximation. We present the dataflow chart in Figure 4.1, and we delve into the details of each application that comprises our system in Chapter 5.

4.2 Network models

We selected pre-trained models of popular architectures, such as **MobileNet** and **Inception**, and **ResNet** [17], already introduced in Section 2.1. MobileNet was chosen for its outstanding performance. However, due to its lack of convolution operations that are not pointwise nor depthwise, only the monochromatic approximation could be applied to it. In order to experiment with more generic approximations, we also selected a few versions of Inception, and one version of ResNet.

4.3 Image dataset

We use a subset of the ImageNet Large Scale Visual Recognition Competition 2011 (ILSVRC2011) [7], that year's iteration of the popular competition hosted ImageNet, a database of over 14 million annotated images. This dataset originally contained 150,000 photographs across 1,000 classes, but we limited our subset, due to time constraints of the experiments, to 94,609 images across 994 classes.

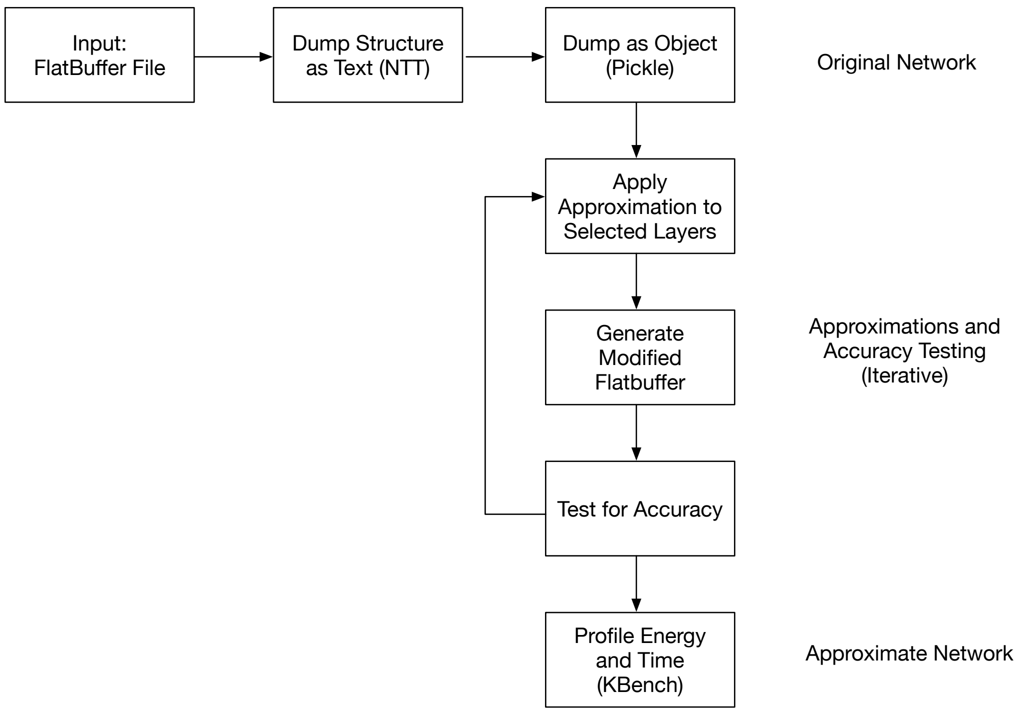
Input:
FlatBuffer File
Dump Structure as Text (NTT)
Dump as Object (Pickle)
Original Network
Apply Approximation to Selected Layers
Generate Modified Flatbuffer
Approximations and Accuracy Testing (Iterative)
Test for Accuracy
Profile Energy and Time (KBench)
Approximate Network

Figure 4.1: A dataflow perspective of the approximation pipeline.

4.4 Engine

We chose the mobile engine that is included in TensorFlow, which is one of the most popular machine learning frameworks, to provide the baseline for our exploration.

Our work builds upon TensorFlow Lite, compiled as a static library. To fully understand our project, some understanding of the TensorFlow framework is required.

4.4.1 TensorFlow

TensorFlow (TF) is library that allows users to define a data flow graph, in which each node is a mathematical operation, and let tensors, which are multidimensional arrays, flow through them. In particular, TensorFlow was developed, and is still widely used, for machine learning and deep neural network applications.

The API is fully available in Python, and available but not fully stable in C++, Java and Go. Figure 4.2 shows a simple TensorFlow model taken from [1]. If this example were to be trained to classify an input array, the inputs labeled as "Weight" would be adjusted by the learning algorithm, while parameters known as hyperparameters, which are set by the user, would not be overridden.

Examples of hyperparameters in convolutions include the **size of the stride**, that is, how many pixels will be skipped after each submatrix multiplication is calculated, resulting in a matrix that is approximately `stride` times smaller in each dimension, and the **padding method**, that is, how the window will operate over the edges of the matrix, or whether it will skip it, resulting in an output that is a few pixels smaller in each dimension than it would be if some padding was used.

Prime examples of parameters that can be learned are the entries of the 1x1xC, 3x3xC,

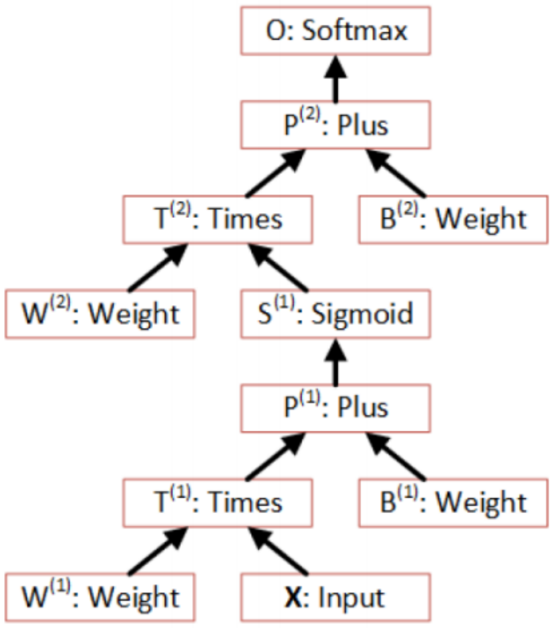

Figure 4.2: A simple dataflow graph that could be implemented in TensorFlow.

or 5x5xC filters used by the convolution kernel, where C is the number of input channels. Once the model has been trained for a particular application, it can be deployed as a component of a full product in the market. In this case, we may no longer need training, and can deploy the graph and model optimized for inference.

To build a network in TF, one of the methods that can be used, and in fact the one that we did use, is to iterate from the first operation to the last, declaring each new operation and its inputs according to the description of its architecture, or in our case, according to information extracted from a flatbuffer file of the model we wish to approximate. Normally, one would proceed to declaring learning parameters, before beginning to train the network, but since we are approximating existing, trained model, we proceed differently. Once our `flatbuffer_rebuilder` is done describing the network, we call a converter provided by TFL to convert it to a format that TFL understands.

4.4.2 TensorFlow Lite

TensorFlow Lite (TFL) offers a model conversion to a smaller binary, along with optimizations for mobile devices. Models are converted by mapping operations to TFL implementations, which can run on different hardware components, and utilize two different libraries, depending on the type of the operands. TFL currently only supports two types, `float32` and `quantized unsigned int8`, which rely on Eigen and gemmlowp, respectively. When using `uint8`, TFL will not only achieve a smaller size model, but also save memory bandwidth when operating with this type, although internal accumulators are still 32-bit wide.

Eigen [2] is a C++ library that offers its own implementation of highly optimized matrix operations. Being open-source and decoupled from system Basic Linear Algebra

Subprograms (BLAS), Google can tweak and extend functionality to provide the desired performance, and extend support for multi-core CPUs, GPUs and other hardware accelerators.

Gemmlowp [3] is a General Matrix Multiplication (GEMM) library owned by Google, released under Apache License 2.0, where *lowp* stands for *low precision.* The input and outputs are matrices of any size comprised of integers on at most 8 bits, but the internal accumulators use 32 bits, to avoid overflows. The resulting product has significant 8 bits extracted to fit into the output. This rounding is governed by the offset, multiplier and shift parameters that are passed by the caller, and takes place in a pipeline, to which the 32-bit result is fed. Gemmlowp's goals are to minimize latency, battery usage and, by consequence, memory bandwidth usage.

Because quantizing the weights involves setting additional parameters that depend on the expected minimum and maximum values that will result from Multiply and Accumulate (MAC) operations, and our strategies involve changing weights, we decided to focus only in floating point implementations, and leave quantized version for future extensions to the project. We expect that a more efficient usage of memory may lead to the bottleneck shifting away from memory access and into computation, and so we cannot predict how our approximations will behave under a quantized model.

4.5 Devices

We set up a Raspberry Pi (RPi) (Subsection 4.5.1) single-board computer to receive power and data from a SmartPower. The SmartPower (Subsection 4.5.2) is set to provide a direct current at 5 volts, while registering its amperage and sending it via USB to the RPi every millisecond. The relevant technical specifications for these models is detailed below.

4.5.1 Raspberry Pi 3 Model B

SoC	Broadcom BCM2837
CPU	Quad-core ARM Cortex-A53 @ 1.2GHz
RAM	1GB LPDDR2 (900 MHz)
y Storage	32 GB SanDisk Ultra microSD
Operating System	Raspbian Stretch 2018-04-18
Linux kernel version	4.14.79 armv7l

Table 4.1: Raspberry Pi specs [5]

4.5.2 HardKernel SmartPower 2

Microcontroller Unit	Espressif ESP8266
Digital Potentiometer	Microchip MCP4652
Current/Power Monitor	Texas Instrument INA231
10A 18V Stepdown	MPS MP8762
Maximum Output Current	5A

Table 4.2: SmartPower specs [4]

Additionally, we modified the kernel provided by HardKernel to message, via serial interface, 1000 measurements per second, in amperes, each as a string with 4 digits, falling into the 0.000-9.999 range.

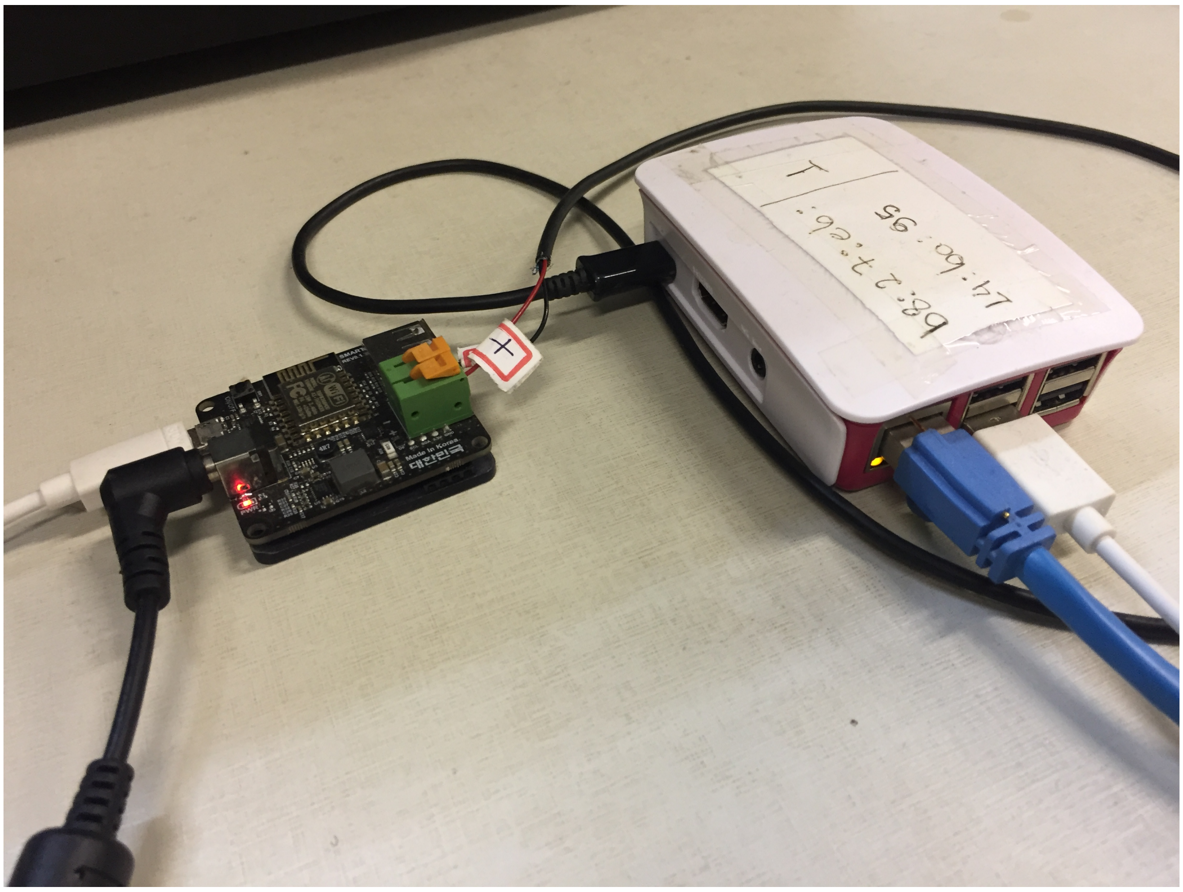

Figure 4.3: Our setup for power measurement. The dark cables and wires are used for powering the devices, and the white cable transfers data from the SmartPower to the Raspberry Pi.

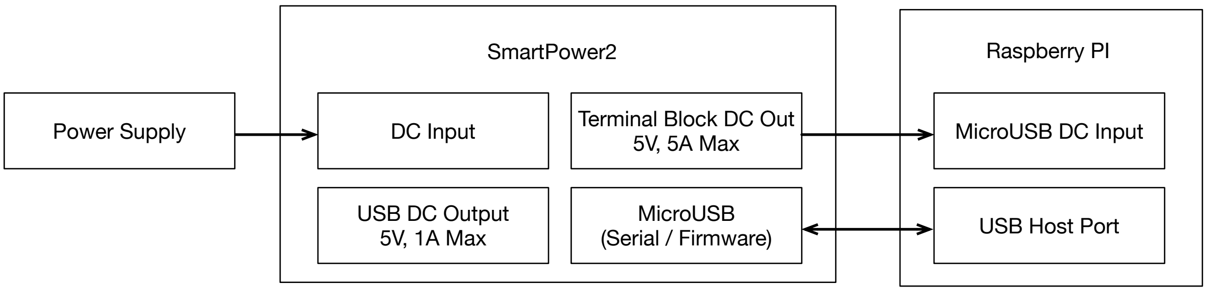

Figure 4.4: Diagram representing our setup for power measurement.

Chapter 5

System Components

In this chapter, we present software tools that we developed to assist us in this project. We also show how these tools come together to form an approximation and a profiling pipeline, that helps keeping development modular as well as enabling quicker testing of new features and components. In the Acknowledgements, we thank other researchers who contributed with the development of some of these software components.

5.1 Overview

Numerous tools were developed either from scratch, or by modifying existing tools originally written for other purposes. Figures 5.1 and 5.2 show an overview of the system components and how they interact with each other, while the following sections describe their operation in greater details. In the figures, the scroll icons represent Python3 scripts, and the pager icons represent compiled programs written in either C or C++.

The steps represented in Figure 5.1 are:

1. `nnt` receives the original flatbuffer file for a given model (e.g. `inception_v3`), and outputs a textual description of the network, including its weights and hyperparameters.

2. The textual output of `nnt` is passed to `dump_parser`, which converts it to an internal representation using Python classes, and saves it to a file using Pickle (e.g. `inception_v3.pkl`).

3. The Pickle file is loaded by the `flatbuffer_rebuilder` tool, which selects an appropriate layer to apply an approximation to.

4. `flatbuffer_rebuilder` calls the `protopyte` module, which interfaces with implementations of approximation algorithms, and passes the data from the layer or layers that are to be approximated. `prototype` returns a list of matrices that are used by to approximate the operation previously performed by the layer.

5. `flatbuffer_rebuilder` uses the matrices received in the previous step to construct a model via TensorFlow's Python interface, then converts it to a flatbuffer file (e.g.

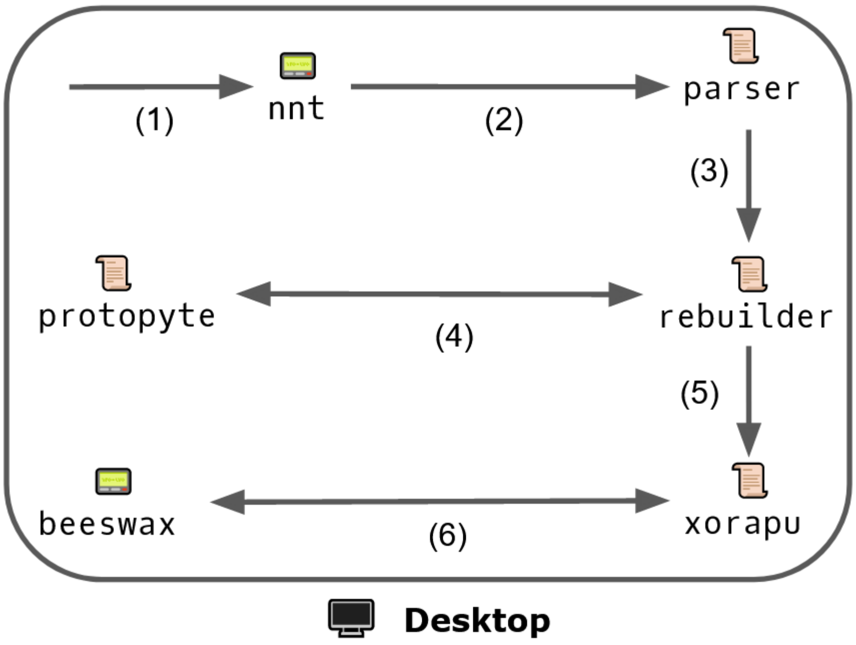

Figure 5.1: Approximation pipeline. Input is the original flatbuffer model. Output is a modified flatbuffer model, and its accuracy benchmark.

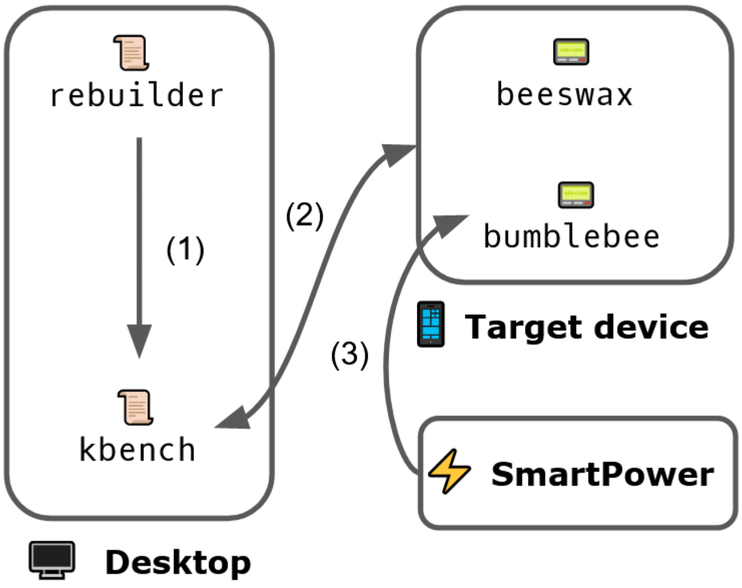

Figure 5.2: Energy pipeline. Input is a modified or original flatbuffer model. Output is time taken and energy consumed per operation on the target device.

reconstructed_inception_v3.tflite). Before terminating, it calls xorapu to measure the modified model's accuracy.

6. xorapu calls beeswax passing a list of files (we used subsets of ImageNet), and captures the modified model's Top 1 and Top 5 accuracy against those images. The results are reported to the user, who may choose to keep or discard the model.

The steps represented in Figure 5.2 are:

1. We provide the modified model produced by flatbuffer_rebuilder as input to kbench, as well as the network address of the target device.

2. kbench orchestrates the communication with the target device, signaling it to start collecting power measurements while running inferences. Once the device is done, kbench pulls the profiling files, parses them, and outputs per-operation power and time consumption in human-readable format.

3. bumblebee interfaces with an USB port to read power measurements that are sent by the SmartPower's kernel in regular intervals.

5.2 beeswax

This application is our modification of TFL's label-image. Originally, it was designed to perform inferences on a single image several times, which is only useful for profiling the performance of the model. We extended it to also take a list of image files as input, effectively testing the model's prediction accuracy over that set of inputs.

When we focus on benchmarking accuracy, we use the xorapu Python module (Section 5.9), which calls beeswax a single time for each input image in a dataset.

Combined with modifications in TFL's code, implemented so it will log the timestamps of when each kernel begins and ends its execution, we can profile the delay and energy consumption of not only the entire model, but of each operator that comprises the model. We rely on this to capture the efficiency in time and power of both builtin and custom implementations by running inference multiple times (for increased confidence), rather than a single time for each image.

5.3 bumblebee

This application, which relies on the termios.h library, listens to the USB port, collects the electrical current readings from our power measurement tool, and logs them to a file, along with the respective timestamps.

Given that the voltage of the power supply is constant and configurable, we set it according to the target device (e.g. 5 volts for the Raspberry Pi) and find out the power that is being drawn by the device during computation by simply multiplying the voltage by the electrical current reading.

5.4 kbench

`kbench` orchestrates the process of communicating with the remote device (e.g. Raspberry Pi), uploading the executables (`beeswax` and `bumblebee`), running the inferences, downloading the profiling data from the device back to the host, and matching the readings to execution timestamps, and parsing all the information for a human to read.

It supports communication via `ssh`, `adb` (for Android devices), and `sdb` (for Samsung devices running Tizen).

5.5 NNT

The Neural Network Transpiler was designed to read TFL models in the flatbuffer format, and output human-readable information about them. We modified it to output all the information that is necessary to rebuild the network, including operators, input and output indices, shapes, hyperparameters, and weights.

5.6 dump_parser

This program reads and parses the human-readable output of NNT, instantiates Python3 objects that hold the same information, and saves them into a file using the Pickle module.

This Pickle file allows us to load and modify the networks using other Python3 scripts without having to re-parse the output of NNT each time, reducing the overhead of working with neural networks from several minutes to just a few seconds, or a fraction of a second, depending on the size of the model.

5.7 flatbuffer_rebuilder

Loads the Pickle file that is produced by `dump_parser`, builds a new TensorFlow model, and calls TOCO to convert it to a flatbuffer, which is then saved as a `.tflite` file.

What this component effectively accomplishes is allowing us to modify a network, either (1) by simply changing the weights of a specific layer (useful when a weight matrix that emulates an approximation is available), (2) by replacing a layer with a set of layers that implement an approximation using pre-existing operations from TensorFlow Lite, or (3) by swapping a layer with a new custom operation that computes an approximation of the result. (1) was used early in the project to easily benchmark for accuracy of the approximate operation, (2) was used to validate the sequence of operations that we would have to implement later on, and (3) was used to effectively perform (2) with less overhead, allowing for the final implementation to be tested for time and energy efficiency.

5.8 protopyte

This module interfaces with the implementation of the approximation, which is called by `flatbuffer_rebuilder`, passing the weights of an arbitrary layer to it, as well as the desired approximation strategy, and the parameters for the configurable knobs, to which `protopyte` will return a weight matrix of the same dimensions for early emulations, as well as the smaller weight matrices used for the proper implementation of the accuracy.

This module can also be called separately to approximate a layer passed by a Pickle file, allowing for faster debugging of different approximation strategies.

5.9 xorapu

A simplified `kbench` (Section 5.4). `xorapu` runs our benchmark application (Section 5.2) on the host machine. This allows us to compare the **accuracy** of different models on large data sets. Because neither repeatedly running inference for a given input image, nor running it on different hardware will change the model's top guesses, we run the inference a single time on each input image, and on a much faster hardware than our remote mobile devices. When running `xorapu`, we are only interested in the model's answers to a dataset. This tools allows us to test different settings for the knobs, and to discover which layers can be approximated without destroying the model's accuracy, and which layers cannot.

Chapter 6

Experimental Results

Our experiments compare the baseline accuracy and performance of the `CNNs` that we selected, as well as the pros and cons of approximate models. We use the top 1 and top 5 accuracy ratings of the models to determine how well they performed after receiving approximations, and we show the time and energy consumption of each layer before and after. Unless otherwise stated, time is being measured in milliseconds (ms), and energy is being measured in joules (J).

6.1 Baseline

Table 6.1 shows the CNNs that we selected, as well as their accuracy, performance, and, in the final two columns, how many convolutional layers they have, and how many of those feature a window of size larger than 1 in both dimensions (e.g. 3x3, 5x5). Note that this is usually the case for the first layer, to which we can apply the monochromatic strategy.

Table 6.2 shows how much of the execution time and energy consumption is taken up by convolutions in the select CNNs.

6.2 Monochromatic approximation

Tables 6.3 and 6.4 show the accuracy that we could achieve with the monochromatic strategy, using different values for the number of intermediate channels, and either fea-

CNN	Top 1 %	Top 5 %	Time	Energy	Conv2D	(>1)x(>1) %
inception_v3	75.20	83.80	5350	17.1	95	21.05
inception_v4	76.60	83.70	10106	32.0	149	16.11
squeezenet	25.25	44.75	844	2.63	26	34.62
mobilenet_v1	67.25	82.00	568	1.64	15	6.67
mobilenet_v2	69.50	82.00	467	1.29	36	2.78
inception_resnet_v2	72.00	82.30	8816	28.9	41	21.81
resnet_v2	71.70	82.80	11127	35.9	36	32.38

Table 6.1: Original CNN accuracy and performance.

CNN	T. %	E.%	(>1)x(>1) T. %	(>1)x(>1) E. %
inception_v3	94.8	97.5	51.9	52.6
inception_v4	95.0	97.4	36.9	37.3
squeezenet	91.9	96.4	48.1	49.7
mobilenet_v1	76.8	83.6	0.0	0.0
mobilenet_v2	64.5	77.1	0.0	0.0
inception_resnet_v2	97.2	99.5	40.7	40.8
resnet_v2	81.6	84.7	43.5	45.0

Table 6.2: Percentage of time and energy in each model that is taken by convolution, both the general kind, and the kind featuring a large convolution window, as described in Section 6.1

, excluding the first layer.

CNN	2 inter. ch.		4 inter. ch.		8 inter. ch.	
	Top 1 %	Top 5 %	Top 1 %	Top 5 %	Top 1 %	Top 5 %
inception_v3	70.08	81.33	68.00	80.55	69.90	81.75
inception_v4	71.05	81.58	64.22	78.50	71.33	81.62
squeezenet	12.30	27.02	18.32	34.67	21.40	40.08
mobilenet_v1	18.52	34.77	30.30	50.70	52.90	71.30
mobilenet_v2	50.22	69.85	57.55	75.10	59.40	76.62
inception_resnet_v2	66.95	80.10	61.32	77.58	70.70	81.83
resnet_v2	57.93	74.88	38.35	56.47	67.25	80.83

Table 6.3: Accuracy of the monochromatic approximation, with different numbers of intermediate channels, and equal-sized clusters.

turing or not featuring the restriction of even-sized clusters.

There are a few things to note. First, for inception_v3, the best result was without the constraint, with a 4.37% drop to Top 1 accuracy and a 2.05% drop to Top 5 accuracy, but for inception_v4, the best result was also without the constraint, with a 1.80% drop to Top 1 accuracy and a 0.85% drop to Top 5 accuracy. This shows two similar networks performing quite differently under the same approximation strategy. Additionally, mobilenet_v1 achieved an accuracy drop of 0.48% drop to Top 1, whereas the similar mobilenet_v2 had a 6.52% drop to the same metric. We observe that the monochromatic approximation only retains accuracy with more than 4 intermediate channels, at which point performance might degrade.

6.3 SVD factorization

We implemented this strategy into TFL as '**Dragunov**', a custom operation that is divided into 6 steps:

1. **Slicing** – divide the input matrix into several smaller matrices.
2. **Phase C** – perform pointwise convolutions in the matrices that resulted from the previous step.

CNN	2 inter. ch.		4 inter. ch.		8 inter. ch.	
	Top 1 %	Top 5 %	Top 1 %	Top 5 %	Top 1 %	Top 5 %
inception_v3	70.08	81.33	69.05	81.30	70.83	81.88
inception_v4	72.95	82.12	73.06	82.38	74.80	82.85
squeezenet	19.73	36.75	27.82	47.35	26.67	46.27
mobilenet_v1	45.27	65.80	60.32	76.22	66.77	79.77
mobilenet_v2	59.40	75.90	61.95	77.38	62.98	77.80
inception_resnet_v2	68.95	81.05	70.75	81.58	70.58	81.65
resnet_v2	67.40	80.50	69.60	81.58	70.12	81.92

Table 6.4: Accuracy of the monochromatic approximation, with different numbers of intermediate channels, without the restriction on equal-sized clusters.

3. **Phase Z** – perform convolutions with windows larger than 1x1.
4. **Phase F** – perform pointwise convolutions.
5. **Sum** – perform additions to combine the outputs of the previous step into a single output matrix.
6. **Bias+ReLU** – add the bias array to the output and calculate the activation function.

The bulk of the computation is in the 3 phases that have convolutions in them, plus the bias+ReLU step. The other two steps are mostly copying data around. The 4 main operations are using optimized implementations, but the 2 copy steps are not. Ideally, we would like to implement faster versions of them, or to build their actions into the the optimized implementations of the other steps. Specifically, by modifying the previous layer to output a tensor in the shape that the approximate operation requires, we can improve the locality of the first step, `Slicing`. By implementing a matrix multiplication function that accumulates on the output matrix rather than re-initializing it to a 0-filled array, and by modifying the following layer to take a permuted input channels, the penultimate step, called `Sum`, can be removed.

We therefore acknowledge that our current implementation of 'Dragunov' is not optimal, and so we break the profiling information into parts, such that the parts that can be optimized away are separated, and the main part serves as an upper bound for how fast it could be. It should be noted, however, that even without these optimizations in place, we can still see improvements in performance from applying the approximations.

Tables 6.5, 6.6, 6.7, 6.8, and 6.9 show the accuracy metrics after the approximation is applied to certain sets of layers, as well as the change in execution time and energy consumption, according to the experimentally determined performance. Figure 6.1 summarizes some of the best results we achieved.

Figures 6.2, 6.3, 6.4, 6.5, and 6.6 visually represent the difference in energy consumption achieved by the strategy, both including and excluding the overheads described in the beginning of Section 6.3. It is interesting to note that, for cheap convolutions, the approximation tends to produce nearly identical or even worse results (in energy performance) than the original implementation. For more expensive convolutions, the savings

Conv Layers	Top 1 %	Top 5 %	Δ T.	Δ E.
–	75.20	83.80		
16,29,71,75	72.20	82.15	19	0.06
16,29,75	72.47	82.25	11	0.04
16,24,29,75	71.30	81.95	21	0.08

Table 6.5: Variation in accuracy of the inception_v3 model, with the SVD factorization strategy applied to different subsets of its convolutional layers. Higher values represent lower loss.

Conv Layers	Top 1 %	Top 5 %	Δ T.	Δ E.
–	76.80	83.70		
13,23	74.20	82.75	17	0.08
5,13,23	61.45	76.38	44	0.25
16,22,27,34,36,39,42	73.25	82.45	125	0.52
16,22,27,34,36,39,41,42	72.42	82.12	236	0.95
16,22,34,37,39,42,114	73.05	82.38	120	0.50
16,22,27,34,36,39,42,114	72.45	82.25	127	0.52
16,22,34,37,39,42,114,118	71.40	81.62	138	0.56
13,16,22,23,27,34,36,39,42,114	68.80	80.73	144	0.60

Table 6.6: Variation in accuracy of the inception_v4 model, with the SVD factorization strategy applied to different subsets of its convolutional layers. Higher values represent lower loss.

are more consistent, but they do not increase proportionally to the cost of the original implementation.

Tables 6.10, 6.12, 6.14, 6.16, and 6.18 show the performance of the approximable layer of each model, when subject to the current implementation of the approximate convolution, including the operations required to reshape the data in memory.

Tables 6.11, 6.13, 6.15, 6.17, and 6.19 show the performance of the approximable layers of each model disregarding the overheads, as previously explained in this section. These values serve as an upper bound on the gains that can be achieved using the currently available convolution implementation on the platform. We can see that, especially on the more expensive convolutions, the savings go up by about 20-25%.

Conv Layers	Top 1 %	Top 5 %	Δ T.	Δ E.
–	25.25	44.75		
18	18.07	34.85	3	0.01
21	14.40	29.73	13	0.06
15	16.75	33.67	3	0.01
24	4.88	12.55	3	0.01
15,18	12.20	25.70	6	0.03
18,21	9.07	21.35	16	0.07
15,18,21	6.60	16.53	19	0.08
15,18,21,24	1.25	5.22	22	0.09

Table 6.7: Variation in accuracy of the squeezenet model, with the SVD factorization strategy applied to different subsets of its convolutional layers. Higher values represent lower loss.

Conv Layers	Top 1 %	Top 5 %	Δ T.	Δ E.
–	72.00	82.30		
16	71.38	81.90	-3	-0.01
70	71.50	81.88	-1	-0.01
16,28	71.40	81.92	-5	-0.02
28,70	71.43	81.90	-3	-0.02
16,28,70	71.25	82.00	-6	-0.03
28,70,73	71.15	81.80	-4	-0.01
28,63,70	71.43	81.92	-5	-0.03
28,63,65,70	70.97	81.85	-8	-0.04
28,63,70,84	70.88	81.85	159	0.65
28,63,65,70,84	71.00	81.83	156	0.64
65,84	71.05	81.73	161	0.67
31,52	71.20	81.83	-2	0.01
17,52	71.15	81.80	-1	0.01
38,80	71.00	81.62	-2	0.01
187	21.77	47.95	31	0.07
38,65,80	70.75	81.67	-5	0.00
38,65,80,82	70.65	81.65	50	0.23

Table 6.8: Variation in accuracy of the inception_resnet_v2 model, with the SVD factorization strategy applied to different subsets of its convolutional layers. Higher values represent lower loss.

Conv Layers	Top 1 %	Top 5 %	Δ T.	Δ E.
–	71.70	82.80		
53	71.60	81.97	59	0.21
29,50,86	71.20	81.90	177	0.64
6,29,50,86	67.03	80.27	214	0.80
29,50,68,86	70.93	81.80	235	0.85
29,50,53,68,86	70.95	81.83	294	1.06
29,32,50,53,68,86	71.20	81.85	353	1.28
32,50,53,68,86	70.85	81.73	294	1.06
29,32,44,50,53,68,86	70.83	81.90	412	1.49
47,50,53,56,59,62	70.88	81.73	353	1.28
29,32,44,47,50,53,56,59,62,68,86	70.83	81.62	647	2.34
44,47,50,53,56,59,62,68,86	70.60	81.62	529	1.92
22,29,32,44,47,50,53,56,59,62,68,86	68.08	80.73	649	2.35

Table 6.9: Variation in accuracy of the resnet_v2 model, with the SVD factorization strategy applied to different subsets of its convolutional layers. Higher values represent lower loss.

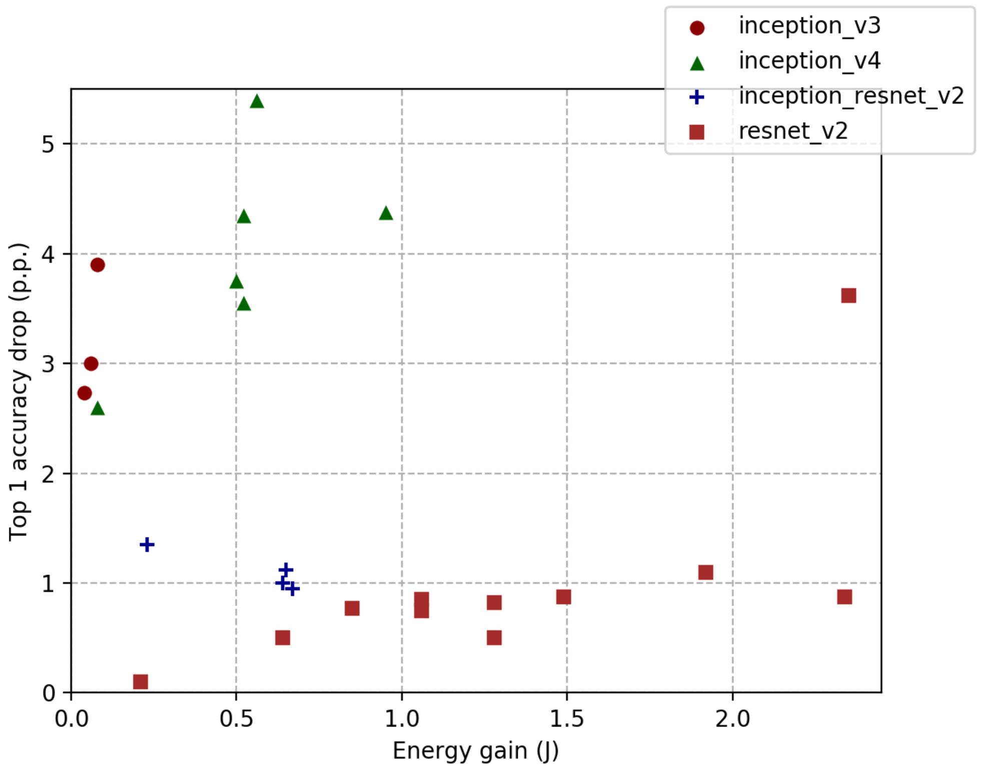

Figure 6.1: Scatter plot comparing the gains and losses of some of the best approximations that we performed.

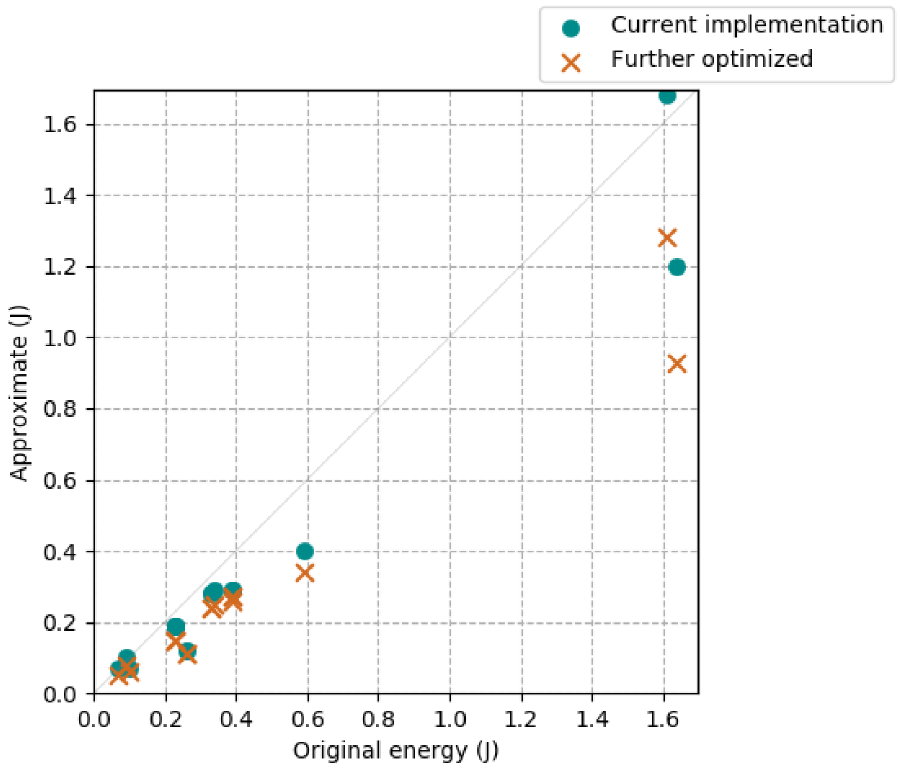

Figure 6.2: Comparison between the original energy consumption within each layer of inception_v3, and the energy consumption within those same layers when approximated. Also shown is the consumption of a 'perfectly optimized' implementation, as described in the beginning of Section 6.3.

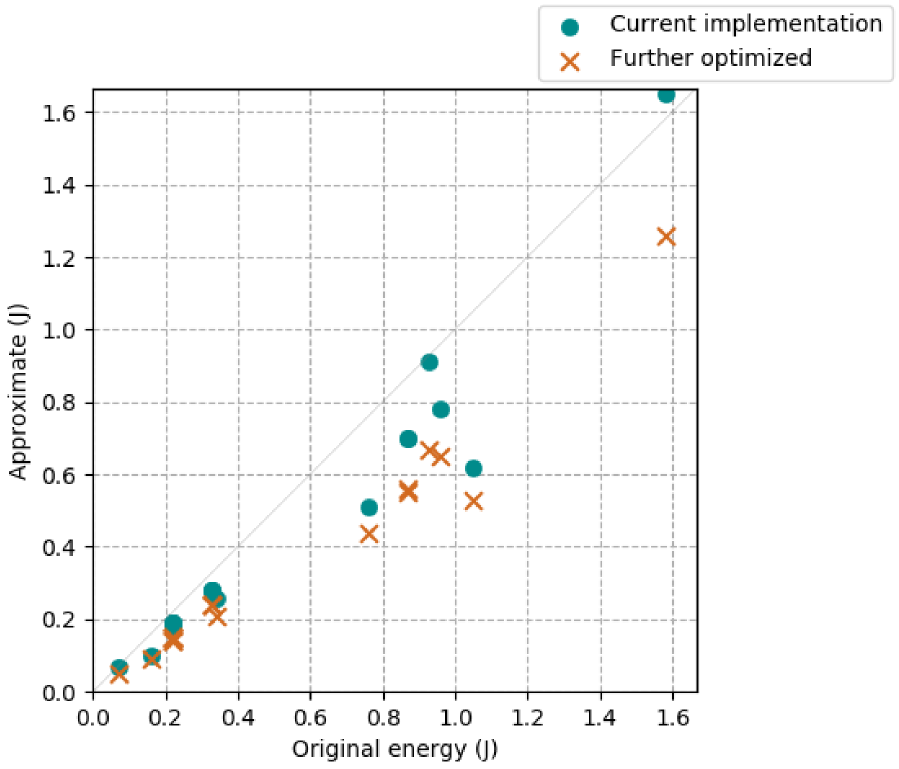

Figure 6.3: Comparison between the original energy consumption within each layer of inception_v4, and the energy consumption within those same layers when approximated. Also shown is the consumption of a ‘perfectly optimized’ implementation, as described in the beginning of Section 6.3.

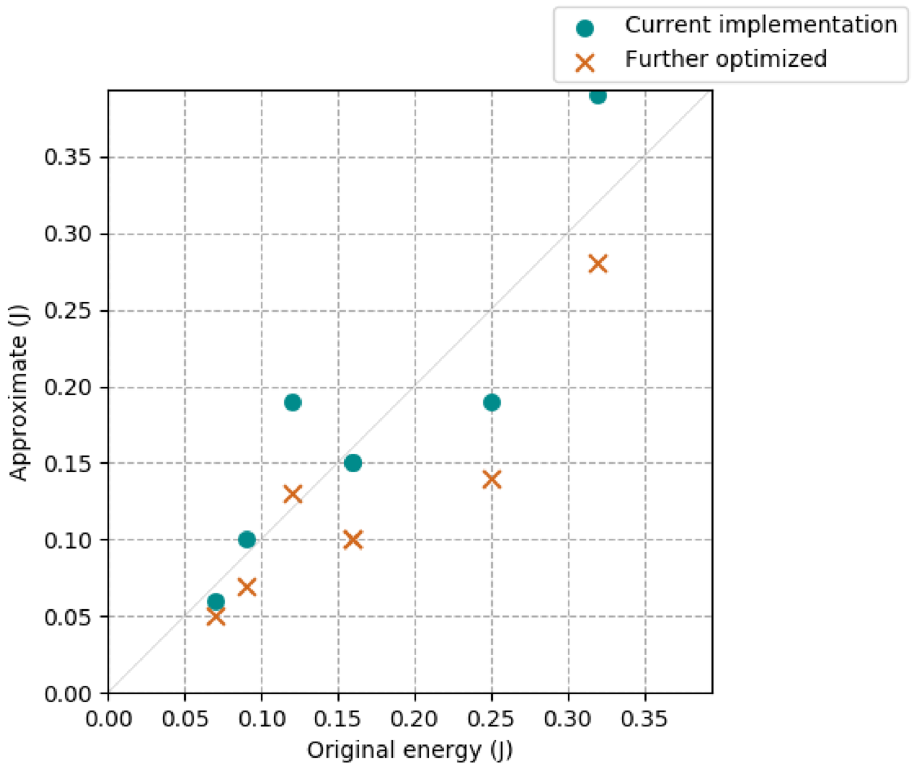

Figure 6.4: Comparison between the original energy consumption within each layer of squeezenet, and the energy consumption within those same layers when approximated. Also shown is the consumption of a ‘perfectly optimized’ implementation, as described in the beginning of Section 6.3.

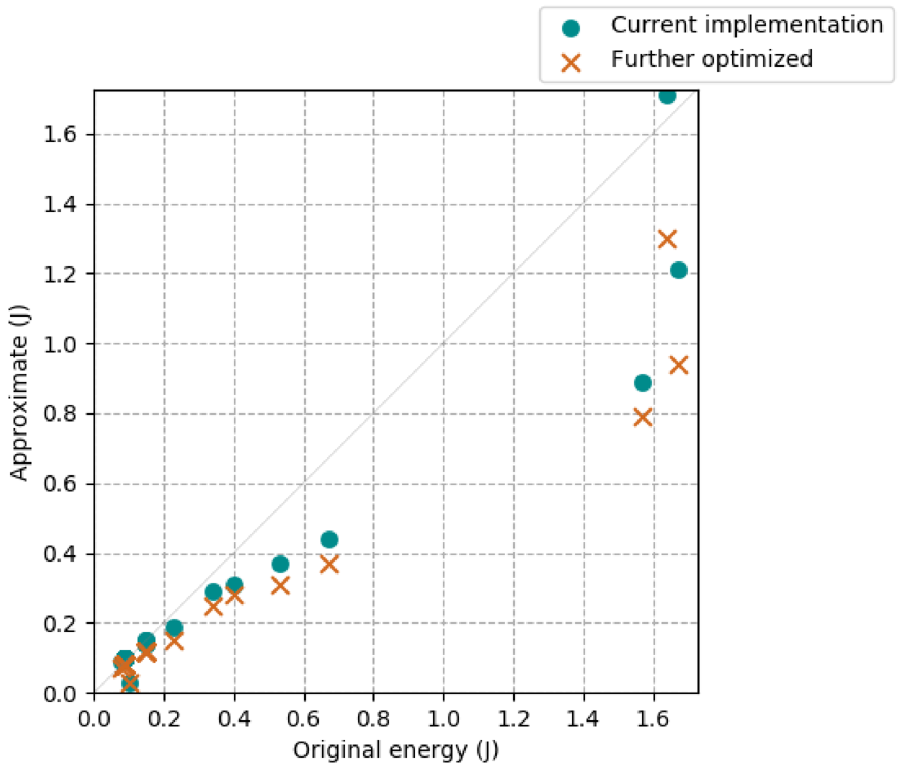

Figure 6.5: Comparison between the original energy consumption within each layer of inception_resnet_v2, and the energy consumption within those same layers when approximated. Also shown is the consumption of a 'perfectly optimized' implementation, as described in the beginning of Section 6.3.

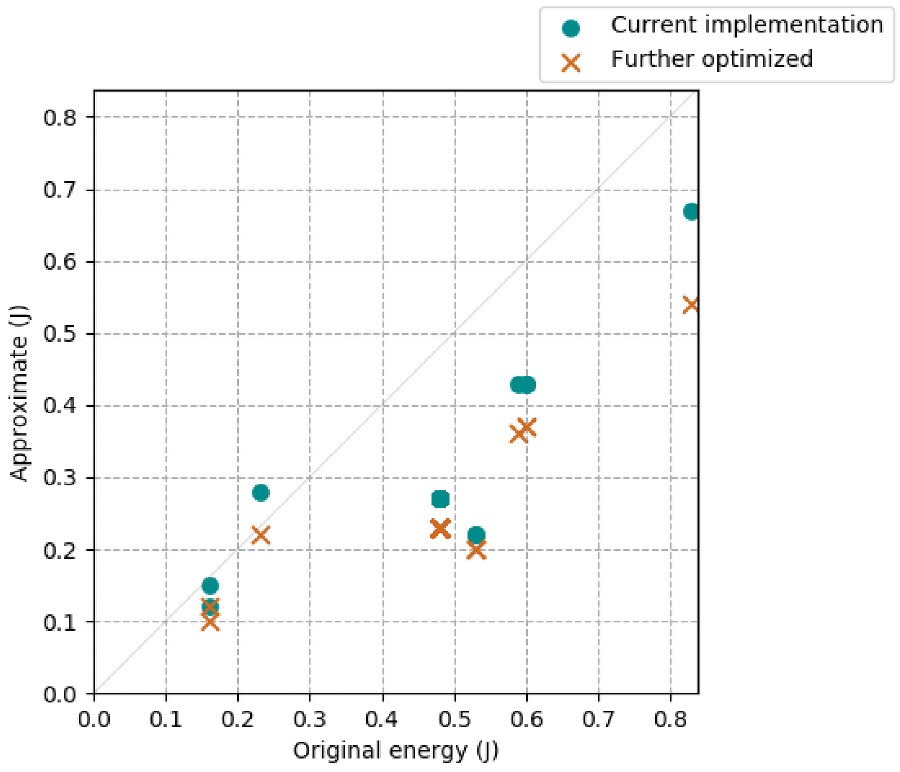

Figure 6.6: Comparison between the original energy consumption within each layer of resnet_v2, and the energy consumption within those same layers when approximated. Also shown is the consumption of a ‘perfectly optimized’ implementation, as described in the beginning of Section 6.3.

Conv layer	O. time	O. energy	Time Δ	Energy Δ	Time %	Energy %
01	415 ± 2	1.28 ± 0.01	-61 ± 8	-0.10 ± 0.02	-14.8	-7.9
02	502 ± 2	1.61 ± 0.02	-80 ± 13	-0.07 ± 0.03	-16.0	-4.2
04	480 ± 3	1.64 ± 0.01	77 ± 11	0.44 ± 0.02	16.1	27.0
07	124 ± 2	0.39 ± 0.00	20 ± 5	0.10 ± 0.02	15.8	24.2
09	71 ± 1	0.23 ± 0.00	11 ± 4	0.04 ± 0.01	15.4	19.0
10	105 ± 1	0.34 ± 0.00	10 ± 3	0.05 ± 0.01	9.6	15.1
14	125 ± 2	0.39 ± 0.01	22 ± 2	0.10 ± 0.01	17.6	24.8
16	71 ± 2	0.23 ± 0.00	11 ± 4	0.04 ± 0.01	15.1	18.2
17	104 ± 2	0.33 ± 0.00	10 ± 3	0.05 ± 0.01	9.7	15.1
21	124 ± 2	0.39 ± 0.00	21 ± 3	0.10 ± 0.01	17.3	24.6
23	71 ± 2	0.23 ± 0.00	11 ± 5	0.04 ± 0.02	14.9	18.3
24	104 ± 2	0.33 ± 0.00	10 ± 3	0.05 ± 0.01	9.3	14.6
26	173 ± 11	0.59 ± 0.03	46 ± 6	0.19 ± 0.02	26.6	32.4
28	72 ± 2	0.23 ± 0.00	10 ± 5	0.04 ± 0.01	14.0	17.3
29	27 ± 3	0.09 ± 0.01	-2 ± 2	-0.01 ± 0.01	-8.1	-11.3
71	31 ± 3	0.10 ± 0.01	8 ± 2	0.03 ± 0.01	25.9	24.7
75	21 ± 2	0.07 ± 0.00	2 ± 1	0.00 ± 0.01	8.8	4.9
81	79 ± 15	0.26 ± 0.04	43 ± 2	0.14 ± 0.01	54.9	53.9
90	78 ± 16	0.26 ± 0.04	43 ± 2	0.14 ± 0.01	54.7	54.0

Table 6.10: Performance of the convolutional layers of inception_v3, before the application of the SVD factorization strategy, and the gains from after its application. Positive values show improvement.

Conv layer	O. time	O. energy	Time Δ	Energy Δ	Time %	Energy %
01	415 ± 2	1.28 ± 0.01	34 ± 7	0.13 ± 0.03	8.0	10.4
02	502 ± 2	1.61 ± 0.02	81 ± 9	0.33 ± 0.03	16.0	20.3
04	480 ± 3	1.64 ± 0.01	187 ± 9	0.71 ± 0.02	39.0	43.4
07	124 ± 2	0.39 ± 0.00	29 ± 6	0.12 ± 0.02	24.0	31.5
09	71 ± 1	0.23 ± 0.00	25 ± 5	0.08 ± 0.01	35.0	36.7
10	105 ± 1	0.34 ± 0.00	26 ± 4	0.09 ± 0.01	24.0	28.3
14	125 ± 2	0.39 ± 0.01	32 ± 4	0.13 ± 0.01	25.0	32.1
16	71 ± 2	0.23 ± 0.00	25 ± 5	0.08 ± 0.01	35.0	35.9
17	104 ± 2	0.33 ± 0.00	26 ± 3	0.09 ± 0.01	25.0	28.3
21	124 ± 2	0.39 ± 0.00	31 ± 4	0.12 ± 0.01	25.0	32.0
23	71 ± 2	0.23 ± 0.00	25 ± 5	0.08 ± 0.02	35.0	36.3
24	104 ± 2	0.33 ± 0.00	25 ± 4	0.09 ± 0.01	24.0	27.8
26	173 ± 11	0.59 ± 0.03	68 ± 16	0.25 ± 0.04	40.0	42.8
28	72 ± 2	0.23 ± 0.00	24 ± 6	0.08 ± 0.02	34.0	35.6
29	27 ± 3	0.09 ± 0.01	4 ± 4	0.01 ± 0.01	16.0	12.0
71	31 ± 3	0.10 ± 0.01	12 ± 5	0.04 ± 0.02	38.0	38.1
75	21 ± 2	0.07 ± 0.00	5 ± 2	0.02 ± 0.01	23.0	22.2
81	79 ± 15	0.26 ± 0.04	46 ± 17	0.15 ± 0.05	59.0	59.2
90	78 ± 16	0.26 ± 0.04	46 ± 18	0.15 ± 0.05	59.0	59.3

Table 6.11: Upper bound on the performance of the convolutional layers of inception_v3 with the SVD factorization strategy, assuming only the time and energy spent on convolutions (Phases C, Z, and F), bias addition, and activation. Positive values show improvement.

Conv layer	O. time	O. energy	Time Δ	Energy Δ	Time %	Energy %
01	415 ± 3	1.26 ± 0.01	-62 ± 9	-0.10 ± 0.02	-14.9	-8.4
02	503 ± 2	1.58 ± 0.01	-81 ± 14	-0.07 ± 0.03	-16.1	-4.4
03	292 ± 4	0.93 ± 0.01	-25 ± 11	0.02 ± 0.02	-8.6	1.9
05	272 ± 2	0.87 ± 0.01	27 ± 6	0.17 ± 0.02	10.1	19.3
09	272 ± 2	0.87 ± 0.01	28 ± 5	0.17 ± 0.01	10.4	19.1
10	291 ± 4	0.96 ± 0.01	28 ± 9	0.18 ± 0.02	9.7	18.4
13	71 ± 2	0.22 ± 0.00	7 ± 6	0.03 ± 0.02	10.1	14.8
15	71 ± 1	0.22 ± 0.00	8 ± 4	0.04 ± 0.01	11.2	16.0
16	105 ± 2	0.33 ± 0.01	10 ± 3	0.05 ± 0.01	9.5	14.8
20	71 ± 1	0.22 ± 0.00	8 ± 4	0.03 ± 0.01	11.2	15.5
22	71 ± 1	0.22 ± 0.00	7 ± 6	0.03 ± 0.02	9.8	14.9
23	104 ± 2	0.33 ± 0.00	10 ± 3	0.05 ± 0.01	9.2	14.5
27	71 ± 1	0.22 ± 0.00	8 ± 4	0.03 ± 0.01	11.3	15.2
29	71 ± 2	0.22 ± 0.00	9 ± 4	0.04 ± 0.01	12.0	15.8
30	104 ± 2	0.33 ± 0.00	9 ± 3	0.05 ± 0.01	9.1	14.5
34	72 ± 2	0.22 ± 0.01	9 ± 4	0.04 ± 0.01	12.3	15.9
36	71 ± 1	0.22 ± 0.00	8 ± 5	0.03 ± 0.01	11.1	15.2
37	104 ± 2	0.33 ± 0.00	9 ± 3	0.05 ± 0.01	9.1	14.4
39	228 ± 15	0.76 ± 0.04	63 ± 10	0.25 ± 0.02	27.5	32.9
41	316 ± 3	1.05 ± 0.01	111 ± 5	0.43 ± 0.01	35.0	41.4
42	104 ± 7	0.34 ± 0.02	20 ± 5	0.08 ± 0.01	19.6	24.1
114	21 ± 2	0.07 ± 0.00	2 ± 1	0.00 ± 0.01	11.2	5.4
118	49 ± 8	0.16 ± 0.02	18 ± 1	0.06 ± 0.01	36.1	34.6

Table 6.12: Performance of the convolutional layers of inception_v4, before the application of the SVD factorization strategy, and the gains from after its application. Positive values show improvement.

Conv layer	O. time	O. energy	Time Δ	Energy Δ	Time %	Energy %
01	415 ± 3	1.26 ± 0.01	32 ± 8	0.13 ± 0.03	8.0	10.1
02	503 ± 2	1.58 ± 0.01	82 ± 8	0.32 ± 0.03	16.0	20.4
03	292 ± 4	0.93 ± 0.01	71 ± 10	0.26 ± 0.02	24.0	27.3
05	272 ± 2	0.87 ± 0.01	87 ± 4	0.32 ± 0.02	32.0	36.4
09	272 ± 2	0.87 ± 0.01	88 ± 4	0.31 ± 0.02	32.0	36.2
10	291 ± 4	0.96 ± 0.01	82 ± 10	0.31 ± 0.02	28.0	32.6
13	71 ± 2	0.22 ± 0.00	21 ± 6	0.07 ± 0.02	30.0	32.9
15	71 ± 1	0.22 ± 0.00	22 ± 5	0.08 ± 0.01	31.0	33.9
16	105 ± 2	0.33 ± 0.01	25 ± 4	0.09 ± 0.02	24.0	28.0
20	71 ± 1	0.22 ± 0.00	22 ± 5	0.07 ± 0.02	31.0	33.6
22	71 ± 1	0.22 ± 0.00	21 ± 5	0.07 ± 0.02	30.0	33.0
23	104 ± 2	0.33 ± 0.00	25 ± 4	0.09 ± 0.01	24.0	27.8
27	71 ± 1	0.22 ± 0.00	22 ± 4	0.07 ± 0.01	31.0	33.5
29	71 ± 2	0.22 ± 0.00	23 ± 5	0.08 ± 0.02	32.0	33.9
30	104 ± 2	0.33 ± 0.00	25 ± 3	0.09 ± 0.01	24.0	27.7
34	72 ± 2	0.22 ± 0.01	23 ± 5	0.08 ± 0.02	32.0	34.0
36	71 ± 1	0.22 ± 0.00	22 ± 5	0.07 ± 0.01	31.0	33.2
37	104 ± 2	0.33 ± 0.00	25 ± 3	0.09 ± 0.01	24.0	27.7
39	228 ± 15	0.76 ± 0.04	89 ± 24	0.32 ± 0.06	39.0	42.0
41	316 ± 3	1.05 ± 0.01	146 ± 7	0.52 ± 0.02	46.0	50.0
42	104 ± 7	0.34 ± 0.02	36 ± 10	0.13 ± 0.03	35.0	37.4
114	21 ± 2	0.07 ± 0.00	5 ± 2	0.02 ± 0.01	25.0	22.8
118	49 ± 8	0.16 ± 0.02	23 ± 8	0.07 ± 0.03	46.0	44.8

Table 6.13: Upper bound on the performance of the convolutional layers of inception_v4 with the SVD factorization strategy, assuming only the time and energy spent on convolutions (Phases C, Z, and F), bias addition, and activation. Positive values show improvement.

Conv layer	O. time	O. energy	Time Δ	Energy Δ	Time %	Energy %
03	40 ± 1	0.12 ± 0.00	-26 ± 5	-0.07 ± 0.01	-64.3	-57.2
06	40 ± 1	0.12 ± 0.00	-25 ± 4	-0.07 ± 0.01	-62.2	-56.0
09	98 ± 2	0.32 ± 0.00	-40 ± 3	-0.07 ± 0.01	-41.2	-21.0
12	28 ± 1	0.09 ± 0.00	-1 ± 2	-0.01 ± 0.01	-5.2	-7.3
15	51 ± 2	0.16 ± 0.00	3 ± 4	0.01 ± 0.01	5.8	8.0
18	51 ± 2	0.16 ± 0.00	3 ± 4	0.01 ± 0.01	6.1	8.2
21	77 ± 2	0.25 ± 0.00	13 ± 5	0.06 ± 0.01	16.9	21.8
24	20 ± 2	0.07 ± 0.00	3 ± 1	0.01 ± 0.01	13.1	10.7

Table 6.14: Performance of the convolutional layers of squeezenet, before the application of the SVD factorization strategy, and the gains from after its application. Positive values show improvement.

Conv layer	O. time	O. energy	Time Δ	Energy Δ	Time %	Energy %
03	40 ± 1	0.12 ± 0.00	-4 ± 5	-0.01 ± 0.01	-11.0	-10.4
06	40 ± 1	0.12 ± 0.00	-4 ± 4	-0.01 ± 0.01	-10.0	-9.4
09	98 ± 2	0.32 ± 0.00	1 ± 2	0.04 ± 0.01	1.0	12.3
12	28 ± 1	0.09 ± 0.00	9 ± 2	0.02 ± 0.01	31.0	27.1
15	51 ± 2	0.16 ± 0.00	18 ± 5	0.06 ± 0.01	36.0	35.0
18	51 ± 2	0.16 ± 0.00	19 ± 4	0.06 ± 0.01	36.0	35.4
21	77 ± 2	0.25 ± 0.00	33 ± 5	0.11 ± 0.02	43.0	43.9
24	20 ± 2	0.07 ± 0.00	7 ± 2	0.02 ± 0.01	36.0	33.9

Table 6.15: Upper bound on the performance of the convolutional layers of squeezenet with the SVD factorization strategy, assuming only the time and energy spent on convolutions (Phases C, Z, and F), bias addition, and activation. Positive values show improvement.

Conv layer	O. time	O. energy	Time Δ	Energy Δ	Time %	Energy %
01	416 ± 2	1.30 ± 0.01	-63 ± 9	-0.11 ± 0.03	-15.2	-8.3
02	504 ± 4	1.64 ± 0.02	-81 ± 13	-0.07 ± 0.03	-16.1	-4.0
04	481 ± 3	1.67 ± 0.02	79 ± 11	0.46 ± 0.03	16.4	27.4
07	124 ± 2	0.40 ± 0.01	19 ± 5	0.09 ± 0.02	15.1	23.6
09	71 ± 1	0.23 ± 0.00	9 ± 3	0.04 ± 0.01	12.5	16.7
10	105 ± 2	0.34 ± 0.01	10 ± 3	0.05 ± 0.01	9.7	14.9
14	27 ± 1	0.09 ± 0.00	-2 ± 3	-0.01 ± 0.01	-7.3	-11.4
16	30 ± 1	0.09 ± 0.00	-3 ± 3	-0.01 ± 0.01	-10.9	-10.1
17	48 ± 2	0.15 ± 0.00	-1 ± 6	0.01 ± 0.02	-1.5	3.8
21	27 ± 1	0.09 ± 0.00	-2 ± 3	-0.01 ± 0.01	-7.6	-11.1
23	30 ± 1	0.09 ± 0.00	-3 ± 3	-0.01 ± 0.01	-10.9	-10.0
24	48 ± 1	0.15 ± 0.00	-1 ± 6	0.01 ± 0.02	-2.2	3.4
28	27 ± 1	0.09 ± 0.00	-2 ± 3	-0.01 ± 0.01	-5.6	-9.8
30	30 ± 2	0.09 ± 0.00	-3 ± 3	-0.01 ± 0.01	-10.5	-10.0
31	48 ± 2	0.15 ± 0.00	-2 ± 6	0.00 ± 0.02	-3.1	2.6
35	27 ± 1	0.09 ± 0.00	-2 ± 3	-0.01 ± 0.01	-6.2	-10.4
37	30 ± 2	0.09 ± 0.00	-3 ± 3	-0.01 ± 0.01	-10.4	-9.9
38	48 ± 1	0.15 ± 0.00	-1 ± 6	0.01 ± 0.02	-1.5	3.7
42	27 ± 1	0.09 ± 0.00	-1 ± 3	-0.01 ± 0.01	-5.1	-9.0
44	30 ± 1	0.09 ± 0.00	-3 ± 3	-0.01 ± 0.01	-10.8	-10.3
45	48 ± 1	0.15 ± 0.00	-1 ± 5	0.01 ± 0.02	-1.3	3.8
49	27 ± 1	0.09 ± 0.00	-2 ± 3	-0.01 ± 0.01	-5.7	-9.3
51	30 ± 1	0.09 ± 0.00	-3 ± 3	-0.01 ± 0.01	-10.8	-9.7
52	48 ± 2	0.15 ± 0.00	0 ± 5	0.01 ± 0.02	-0.6	3.9
56	28 ± 1	0.09 ± 0.00	-1 ± 3	-0.01 ± 0.01	-5.4	-10.1
58	30 ± 2	0.09 ± 0.00	-4 ± 4	-0.01 ± 0.01	-12.6	-10.9
59	48 ± 1	0.15 ± 0.00	-1 ± 6	0.00 ± 0.02	-2.5	2.9
63	27 ± 1	0.09 ± 0.00	-2 ± 3	-0.01 ± 0.01	-6.3	-10.2
65	30 ± 1	0.09 ± 0.00	-3 ± 4	-0.01 ± 0.01	-11.1	-10.6
66	48 ± 1	0.15 ± 0.00	-1 ± 5	0.01 ± 0.02	-1.3	3.9
70	28 ± 2	0.09 ± 0.00	-1 ± 2	-0.01 ± 0.01	-4.9	-9.5
72	30 ± 1	0.09 ± 0.00	-3 ± 3	-0.01 ± 0.01	-11.4	-10.3
73	48 ± 2	0.15 ± 0.00	-1 ± 6	0.01 ± 0.02	-1.1	3.6
77	27 ± 1	0.08 ± 0.00	-2 ± 3	-0.01 ± 0.01	-6.1	-10.5
79	30 ± 2	0.09 ± 0.00	-3 ± 3	-0.01 ± 0.01	-10.9	-10.2
80	48 ± 2	0.15 ± 0.00	-1 ± 6	0.00 ± 0.02	-1.6	3.2
82	192 ± 9	0.67 ± 0.02	55 ± 7	0.23 ± 0.02	28.7	34.4
84	456 ± 5	1.57 ± 0.02	164 ± 5	0.68 ± 0.02	36.0	43.1
85	155 ± 8	0.53 ± 0.02	37 ± 7	0.16 ± 0.02	24.0	30.5
187	40 ± 18	0.10 ± 0.05	31 ± 0	0.07 ± 0.01	76.0	69.0

Table 6.16: Performance of the convolutional layers of inception_resnet_v2, before the application of the SVD factorization strategy, and the gains from after its application. Positive values show improvement.

Conv layer	O. time	O. energy	Time Δ	Energy Δ	Time %	Energy %
01	416 ± 2	1.30 ± 0.01	32 ± 7	0.13 ± 0.03	8.0	10.1
02	504 ± 4	1.64 ± 0.02	81 ± 10	0.34 ± 0.04	16.0	20.6
04	481 ± 3	1.67 ± 0.02	188 ± 9	0.73 ± 0.03	39.0	43.9
07	124 ± 2	0.40 ± 0.01	28 ± 6	0.12 ± 0.02	23.0	30.8
09	71 ± 1	0.23 ± 0.00	23 ± 4	0.08 ± 0.01	32.0	34.5
10	105 ± 2	0.34 ± 0.01	26 ± 3	0.09 ± 0.01	25.0	28.1
14	27 ± 1	0.09 ± 0.00	3 ± 3	0.01 ± 0.01	12.0	10.7
16	30 ± 1	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	13.0	14.4
17	48 ± 2	0.15 ± 0.00	9 ± 7	0.03 ± 0.02	18.0	22.3
21	27 ± 1	0.09 ± 0.00	3 ± 4	0.01 ± 0.01	12.0	10.6
23	30 ± 1	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	13.0	14.3
24	48 ± 1	0.15 ± 0.00	9 ± 6	0.03 ± 0.02	18.0	21.9
28	27 ± 1	0.09 ± 0.00	4 ± 3	0.01 ± 0.01	14.0	12.0
30	30 ± 2	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	14.0	14.1
31	48 ± 2	0.15 ± 0.00	8 ± 7	0.03 ± 0.02	17.0	20.9
35	27 ± 1	0.09 ± 0.00	4 ± 3	0.01 ± 0.01	13.0	11.3
37	30 ± 2	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	14.0	14.5
38	48 ± 1	0.15 ± 0.00	9 ± 6	0.03 ± 0.02	19.0	22.2
42	27 ± 1	0.09 ± 0.00	4 ± 3	0.01 ± 0.01	14.0	12.7
44	30 ± 1	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	13.0	14.4
45	48 ± 1	0.15 ± 0.00	9 ± 5	0.03 ± 0.02	19.0	22.5
49	27 ± 1	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	14.0	12.6
51	30 ± 1	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	13.0	14.7
52	48 ± 2	0.15 ± 0.00	9 ± 6	0.03 ± 0.02	19.0	22.2
56	28 ± 1	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	14.0	11.5
58	30 ± 2	0.09 ± 0.00	3 ± 5	0.01 ± 0.01	12.0	13.6
59	48 ± 1	0.15 ± 0.00	8 ± 6	0.03 ± 0.02	18.0	21.3
63	27 ± 1	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	13.0	11.5
65	30 ± 1	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	13.0	14.0
66	48 ± 1	0.15 ± 0.00	9 ± 5	0.03 ± 0.02	19.0	22.3
70	28 ± 2	0.09 ± 0.00	4 ± 3	0.01 ± 0.01	14.0	12.1
72	30 ± 1	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	13.0	14.1
73	48 ± 2	0.15 ± 0.00	9 ± 6	0.03 ± 0.02	19.0	22.2
77	27 ± 1	0.08 ± 0.00	4 ± 4	0.01 ± 0.01	13.0	11.5
79	30 ± 2	0.09 ± 0.00	4 ± 4	0.01 ± 0.01	13.0	14.3
80	48 ± 2	0.15 ± 0.00	9 ± 7	0.03 ± 0.02	18.0	21.5
82	192 ± 9	0.67 ± 0.02	78 ± 14	0.30 ± 0.04	41.0	44.0
84	456 ± 5	1.57 ± 0.02	204 ± 8	0.78 ± 0.03	45.0	49.9
85	155 ± 8	0.53 ± 0.02	58 ± 14	0.22 ± 0.04	37.0	41.5
187	40 ± 18	0.10 ± 0.05	31 ± 18	0.07 ± 0.06	76.0	70.3

Table 6.17: Upper bound on the performance of the convolutional layers of inception_resnet_v2 with the SVD factorization strategy, assuming only the time and energy spent on convolutions (Phases C, Z, and F), bias addition, and activation. Positive values show improvement.

Conv layer	O. time	O. energy	Time Δ	Energy Δ	Time %	Energy %
03	261 ± 2	0.83 ± 0.01	37 ± 6	0.17 ± 0.02	14.0	19.8
06	262 ± 2	0.83 ± 0.01	37 ± 7	0.16 ± 0.02	14.0	19.7
09	73 ± 4	0.23 ± 0.01	-25 ± 3	-0.05 ± 0.01	-33.8	-21.0
13	180 ± 3	0.59 ± 0.01	38 ± 5	0.16 ± 0.01	21.3	27.7
16	180 ± 3	0.60 ± 0.01	38 ± 5	0.17 ± 0.01	21.2	27.9
19	180 ± 3	0.60 ± 0.01	38 ± 4	0.17 ± 0.01	21.3	28.0
22	51 ± 4	0.16 ± 0.01	2 ± 3	0.01 ± 0.01	4.1	5.1
26	142 ± 6	0.48 ± 0.01	58 ± 5	0.21 ± 0.02	41.0	44.5
29	142 ± 6	0.48 ± 0.01	59 ± 4	0.21 ± 0.01	41.6	44.4
32	142 ± 6	0.48 ± 0.01	59 ± 5	0.21 ± 0.02	41.3	44.4
35	142 ± 6	0.48 ± 0.01	59 ± 4	0.21 ± 0.01	41.6	44.3
38	143 ± 7	0.48 ± 0.02	60 ± 4	0.21 ± 0.01	41.7	44.6
41	142 ± 6	0.48 ± 0.01	59 ± 4	0.21 ± 0.01	41.3	44.0
44	142 ± 6	0.48 ± 0.02	59 ± 4	0.21 ± 0.01	41.3	44.4
47	142 ± 6	0.48 ± 0.01	59 ± 3	0.21 ± 0.01	41.4	44.2
50	143 ± 6	0.48 ± 0.01	59 ± 4	0.21 ± 0.01	41.4	44.4
53	142 ± 6	0.48 ± 0.01	59 ± 3	0.21 ± 0.01	41.4	44.5
56	142 ± 6	0.48 ± 0.01	58 ± 5	0.21 ± 0.01	41.1	44.3
59	142 ± 6	0.48 ± 0.01	59 ± 3	0.21 ± 0.01	41.5	44.6
62	142 ± 6	0.48 ± 0.01	59 ± 4	0.21 ± 0.01	41.4	44.1
65	142 ± 6	0.48 ± 0.01	58 ± 5	0.21 ± 0.02	41.1	44.2
68	142 ± 6	0.48 ± 0.01	58 ± 5	0.21 ± 0.01	41.1	44.3
71	142 ± 6	0.48 ± 0.01	59 ± 4	0.21 ± 0.01	41.5	44.4
74	142 ± 6	0.48 ± 0.01	59 ± 4	0.21 ± 0.01	41.3	44.2
77	142 ± 6	0.48 ± 0.01	59 ± 4	0.21 ± 0.01	41.4	44.1
80	142 ± 6	0.48 ± 0.01	59 ± 4	0.21 ± 0.01	41.4	44.0
83	142 ± 6	0.48 ± 0.01	59 ± 4	0.21 ± 0.01	41.4	44.1
86	142 ± 6	0.48 ± 0.02	59 ± 4	0.21 ± 0.01	41.4	44.0
89	142 ± 6	0.48 ± 0.01	59 ± 4	0.21 ± 0.01	41.4	44.0
92	49 ± 5	0.16 ± 0.01	12 ± 2	0.04 ± 0.01	24.2	24.2
96	157 ± 19	0.53 ± 0.05	90 ± 2	0.31 ± 0.01	57.2	58.3
99	156 ± 18	0.53 ± 0.05	89 ± 2	0.31 ± 0.01	57.0	58.1
102	156 ± 19	0.53 ± 0.05	89 ± 2	0.31 ± 0.01	56.9	57.9

Table 6.18: Performance of the convolutional layers of resnet_v2, before the application of the SVD factorization strategy, and the gains from after its application. Positive values show improvement.

Conv layer	O. time	O. energy	Time Δ	Energy Δ	Time %	Energy %
03	261 ± 2	0.83 ± 0.01	85 ± 6	0.29 ± 0.02	32.0	34.9
06	262 ± 2	0.83 ± 0.01	85 ± 6	0.29 ± 0.02	32.0	34.8
09	73 ± 4	0.23 ± 0.01	-3 ± 5	0.01 ± 0.01	-4.0	4.6
13	180 ± 3	0.59 ± 0.01	63 ± 6	0.23 ± 0.02	35.0	39.0
16	180 ± 3	0.60 ± 0.01	63 ± 6	0.23 ± 0.02	35.0	39.2
19	180 ± 3	0.60 ± 0.01	63 ± 6	0.23 ± 0.02	35.0	39.3
22	51 ± 4	0.16 ± 0.01	13 ± 6	0.04 ± 0.02	26.0	25.2
26	142 ± 6	0.48 ± 0.01	70 ± 10	0.25 ± 0.03	50.0	52.0
29	142 ± 6	0.48 ± 0.01	71 ± 9	0.25 ± 0.03	50.0	52.0
32	142 ± 6	0.48 ± 0.01	71 ± 10	0.25 ± 0.03	50.0	51.9
35	142 ± 6	0.48 ± 0.01	71 ± 9	0.25 ± 0.03	50.0	51.8
38	143 ± 7	0.48 ± 0.02	72 ± 10	0.25 ± 0.03	50.0	52.1
41	142 ± 6	0.48 ± 0.01	70 ± 8	0.25 ± 0.02	50.0	51.5
44	142 ± 6	0.48 ± 0.02	71 ± 10	0.25 ± 0.03	50.0	51.9
47	142 ± 6	0.48 ± 0.01	71 ± 8	0.25 ± 0.02	50.0	51.7
50	143 ± 6	0.48 ± 0.01	71 ± 10	0.25 ± 0.03	50.0	51.8
53	142 ± 6	0.48 ± 0.01	71 ± 8	0.25 ± 0.02	50.0	51.9
56	142 ± 6	0.48 ± 0.01	70 ± 10	0.25 ± 0.03	50.0	51.7
59	142 ± 6	0.48 ± 0.01	71 ± 8	0.25 ± 0.02	50.0	52.1
62	142 ± 6	0.48 ± 0.01	71 ± 9	0.25 ± 0.02	50.0	51.6
65	142 ± 6	0.48 ± 0.01	70 ± 10	0.25 ± 0.03	49.0	51.6
68	142 ± 6	0.48 ± 0.01	70 ± 10	0.25 ± 0.03	49.0	51.7
71	142 ± 6	0.48 ± 0.01	71 ± 9	0.25 ± 0.03	50.0	51.9
74	142 ± 6	0.48 ± 0.01	71 ± 9	0.25 ± 0.03	50.0	51.7
77	142 ± 6	0.48 ± 0.01	71 ± 9	0.25 ± 0.03	50.0	51.6
80	142 ± 6	0.48 ± 0.01	71 ± 9	0.25 ± 0.02	50.0	51.5
83	142 ± 6	0.48 ± 0.01	71 ± 9	0.25 ± 0.03	50.0	51.6
86	142 ± 6	0.48 ± 0.02	71 ± 9	0.25 ± 0.03	50.0	51.5
89	142 ± 6	0.48 ± 0.01	71 ± 8	0.25 ± 0.02	50.0	51.5
92	49 ± 5	0.16 ± 0.01	18 ± 7	0.06 ± 0.02	36.0	36.6
96	157 ± 19	0.53 ± 0.05	96 ± 20	0.33 ± 0.06	61.0	62.5
99	156 ± 18	0.53 ± 0.05	96 ± 20	0.33 ± 0.06	61.0	62.3
102	156 ± 19	0.53 ± 0.05	96 ± 20	0.33 ± 0.06	61.0	62.1

Table 6.19: Upper bound on the performance of the convolutional layers of resnet_v2 with the SVD factorization strategy, assuming only the time and energy spent on convolutions (Phases C, Z, and F), bias addition, and activation. Positive values show improvement.

Chapter 7

Conclusion & Future Work

In this work, we began by arguing that optimizing convolution is imperative to improving the performance of CNNs on mobile devices, even more so if the goal is to reduce energy consumption over inference latency. We hypothesized that we could exploit the redundancy within the weight matrices of the convolutional layers to improve the time and energy performance of an inference framework on mobile devices.

We were able to modify an existing TFL model, replacing existing operations with approximate variants using tools that we developed (Chapter 5). Our tools allow this process to be reproduced by anyone with trivial knowledge of CNN architectures, on a commonly available desktop without any particularly fast arithmetic processing unit, such as discrete graphics cards. We offer the possibility of reducing the accuracy of an already trained model in exchange for better performance, should the timing requirements and energy budget of an application or platform not be met by the current neural network.

We also developed a testbed for the accuracy drop post transformations. Our preliminary results show that not many layers can be approximated before the accuracy drop reaches the point by which it is best to simply replace the entire model with a different architecture that is faster by design, assuming one is already available. However, one possibility that we have not explored is to retrain the network to mitigate the accuracy loss. Because the approximate model still contains many values in common with the original one, the retraining process can reuse that data as a starting point [40].

Finally, we modified TensorFlow Lite and wrote tools that parse profiling data, allowing users to, in addition to measuring time taken per operation, also measure energy consumption of each layer of the CNN. Combining our profiling tools to our accuracy testbed, we found that, depending on the model, up to 5% of the energy can be saved, with less than 1% in Top 1 accuracy drop. The tools we developed and used are available at `https://github.com/SusinMat/Convolution-Profiling-and-Approximation`

There are a few possible directions to improve these numbers. One may retrain the network to adapt to the accuracy loss, or further refine the implementation of the approximate operation within the engine (Section 6.3), or even automate the search space exploration, which could also lead to better sets of layers to approximate being found quicker. The software and experimental setup designed by this project can be reused to implement and test different approximation strategies, to try out different parameters, and to analyze the viability of approximating CNNs for other applications.

Ultimately, all the aforementioned ideas combined can lead into a network designer assistant, informing neural network architects of the cost of adding each layer at the same time as they are choosing the building blocks, and suggesting approximations that may help them fit their models within the appropriate time and energy constraints.

Glossary

approximate computing The science and theory of replacing one implementation of an algorithm with another that is expected to perform better under some performance metric (time, power, energy), at the expense of the accuracy of the result.

engine The library that provides the implementation necessary for executing a neural network. In our work, we are only concerned with the implementation of the operations that are ran during inference, not with training.

entry Any numeric, real or integer, value that is stored at a position of an n-dimensional array.

flatbuffer The filetype used by TensorFlow Lite to store trained models.

float16 The 16-bit floating point type, also known as the half-precision floating-point format, as defined by the IEEE 754-2008 standard.

float32 The 32-bit floating point type, normally as defined by the IEEE 754 standard, but compliance can be laxed in exchange for performance, at virtually no loss in accuracy.

framework (machine learning) A framework, in the context of machine learning, is a module that provides the building blocks for neural network designers, including the tools needed to set training parameters, connect operation outputs to inputs, etc.. Examples include TensorFlow, PyTorch.

hyperparameter Parameters of an operation that are not learned such as stride and padding, in contrast with filter weights and biases. In the flatbuffer format, the activation function is also stored as a hyperparameter..

kernel The implementation of an operation.

layer In the context of neural networks, a *layer* usually refers to a short sequence of operations, such as convolution operation, followed by one bias add operation, and ending with one activation function operation. It can also be used as a synonym to *module*, that is, an element defined by the architecture designer, that typically includes several convolutions and one pooling operation.

matrix Used interchangeably with *tensor* to mean an n-dimensional array.

operator An operation is a function that takes one or more tensors as input, and outputs one or more tensors. Examples of operations include addition, matrix multiplication, convolution.

operator Used interchangeably with *operation*.

pointwise convolution A NxM depthwise convolution can be seen as a convolution in which each output channel is the result of an NxM convolution of one corresponding input channel, i.e., the values in each entry of an output channel are the linear combination of entries in an input channel, rather than a linear combination of entries in all input channels of the input tensor. Typically, the number of output channels is the same as the number of input channels, but it is also possible to have a depthwise convolution in which the number of output channels is the number of input channels multiplied by a positive integer factor.

pointwise convolution A convolution whose kernel is shaped Fx1x1xC, where F is the number of filters (equal to the number of output channels) and C is the depth (equal to the number of input channels).

quint8 A type that represents a real number, quantized into an unsigned 8-bit integer type.

tensor An n-dimensional array used in machine learning.

Top 1 accuracy The percentage of input images in the dataset for which the network's top guess was the correct answer in the classification problem.

Top 5 accuracy The percentage of input images in the dataset for which the network's top 5 guesses included the correct answer in the classification problem.

uint8 The unsigned 8-bit integer type.

weights The tensor containing values that were learned by a network. Contrasting with hyperparameters, these are not set by the human user, but it is possible to initialize them to values that the user knows to work well for that model in particular.

window In the case of a convolution, the window is the shape of each of its filters, disregarding the number of channels. A 3x7 convolution, that is, a convolution with a window height of 3 and a window width of 7, will have its weights shaped Fx3x7xC, where F is the number of filters and C the number of input channels.

Bibliography

[1] `https://github.com/zer0n/deepframeworks/blob/master/README.md`, Retrieved on 2018-03-10.

[2] `http://eigen.tuxfamily.org/index.php?title=Main_Page`, Retrieved on 2018-03-13.

[3] `https://github.com/google/gemmlowp`, Retrieved on 2018-03-13.

[4] `https://web.archive.org/web/20190708183221/https://odroid.com/dokuwiki/doku.php?id=en:acc:smartpower2`, Retrieved on 2019-08-01.

[5] `https://web.archive.org/web/20190622081810/https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/`, Retrieved on 2019-08-02.

[6] `http://web.archive.org/web/20190805033607/http://www.image-net.org/challenges/LSVRC/2012/`, Retrieved on 2019-08-05.

[7] `http://web.archive.org/web/20190201044550/http://image-net.org/challenges/LSVRC/2011/`, Retrieved on 2019-08-05.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[9] Marc Baboulin, Alfredo Buttari, Jack J. Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating scientific computations with mixed precision algorithms. *CoRR*, abs/0808.2794, 2008.

[10] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. Neuralpower: Predict and deploy energy-efficient convolutional neural networks. *CoRR*, abs/1710.05420, 2017.

[11] Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. *CoRR*, abs/1404.0736, 2014.

[12] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, Sep 1936.

[13] Fernando Gama, Antonio Garcia Marques, Geert Leus, and Alejandro R. Ribeiro. Convolutional neural network architectures for signals supported on graphs. *IEEE Transactions on Signal Processing*, PP:1–1, 12 2018.

[14] Edouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou. Efficient softmax approximation for GPUs. *CoRR*, abs/1609.04309, 2016.

[15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[16] Philipp Gysel. Ristretto: Hardware-oriented approximation of convolutional neural networks. *CoRR*, abs/1605.06402, 2016.

[17] 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)*, pages 770–778, June 2016.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[19] 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.

[20] 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.

[21] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *CoRR*, abs/1405.3866, 2014.

[22] Hokuto Kagaya, Kiyoharu Aizawa, and Makoto Ogawa. Food detection and recognition using convolutional neural network. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 1085–1088, New York, NY, USA, 2014. ACM.

[23] F. Khalvati, M. D. Aagaard, and H. R. Tizhoosh. Accelerating image processing algorithms based on the reuse of spatial patterns. In *2007 Canadian Conference on Electrical and Computer Engineering*, pages 172–175, April 2007.

[24] S. Kiranyaz, T. Ince, O. Abdeljaber, O. Avci, and M. Gabbouj. 1-D convolutional neural networks for signal processing applications. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8360–8364, May 2019.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[26] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. 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)*, pages 1–12, April 2016.

[27] Carsten Lecon. Motion sickness in VR learning environments. In *14th International Conference on Information Technology & Computer Science*, 08 2018.

[28] D. Li, X. Chen, M. Becchi, and Z. Zong. Evaluating the energy efficiency of deep convolutional neural networks on CPUs and GPUs. In *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, pages 477–484, Oct 2016.

[29] Huynh Nguyen Loc, Rajesh Krishna Balan, and Youngki Lee. DeepSense: A GPU-based deep convolutional neural network framework on commodity mobile devices. In *WearSys'16: Proceedings of the 2016 Workshop on Wearable Systems and Applications: June 30, 2016, Singapore*, pages 25–30, 2016.

[30] Shiva Manne and Manjish Pal. Fast approximate matrix multiplication by solving linear systems. *CoRR*, abs/1408.4230, 2014.

[31] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016.

[32] nagadomi. Image super-resolution for anime-style art using deep convolutional neural networks. `https://github.com/nagadomi/waifu2x`.

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013.

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

[35] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[36] I. Song, H. J. Kim, and P. B. Jeon. Deep learning for real-time robust facial expression recognition on a smartphone. In *2014 IEEE International Conference on Consumer Electronics (ICCE)*, pages 564–567, Jan 2014.

[37] G. W. Stewart. On the early history of the singular value decomposition. *SIAM Review*, 35(4):551–566, 1993.

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[39] TensorFlow. Introduction to tensorflow lite. `https://www.tensorflow.org/mobile/tflite/`, Retrieved on 2018-02-01.

[40] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutional networks for classification and detection. *CoRR*, abs/1505.06798, 2015.