Embedded Neural Networks in Resource-Constrained Hearing Instruments

Jelcicová, Zuzana

Publication date:
2022

Document Version
Publisher's PDF, also known as Version of record

Embedded Neural Networks in Resource-Constrained Hearing Instruments

Zuzana Jelčicová

Kongens Lyngby 2022
PhD

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, Building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk
PhD-2022
ISSN: 0000-0000

Abstract (English)

Deep neural networks have revolutionized many different areas, including speech enhancement, speech recognition, and speech separation that are relevant for hearing instrument users and professionals. At the same time, the state-of-the-art neural networks are huge models that require megabytes of storage and millions of operations for every input, which makes them extremely energy-intensive and thus difficult to deploy to resource-constrained devices such as hearing instruments. For these reasons, neural networks were previously only executed in a high-performance computing environment (cloud), and the results were afterwards sent back to IoT (edge) devices. However, real-time applications such as hearing instruments require low-latency connections and processing to not compromise sound quality. Moreover, in order to communicate with the cloud, an edge device must be connected constantly, which is unfeasible and quickly drains the battery of the device. Last but not least, sharing data with the cloud is not desirable due to security issues. Therefore, the focus in recent years has been on enabling the execution of neural networks directly in low-power devices.

To successfully accomplish this goal, it is imperative to develop computationally efficient hardware-aware deep learning algorithms that in turn should be executed on custom hardware accelerators optimized for neural network processing. Exploring the ways to achieve such algorithm-hardware co-optimization is the objective of this PhD thesis.

Therefore, this work firstly proposes two novel dynamic pruning algorithms, called *PeakRNN* and *StatsRNN*, for reducing the number of multiply-accumulates and memory accesses during inference. Since our focus is on audio (speech enhancement and speech recognition), we primarily explore Recurrent Neural Networks

(RNNs) and Transformer Neural Networks (TNNs) due to their capabilities to process temporal information. All our experiments demonstrate substantial reductions in computations while maintaining high performance in the evaluation metrics. *PeakRNN* is chosen for the next stage of the project as it prunes a layer by selecting a constant number of top elements every timestep, which offers, among others, determinism and worst-case execution time guarantees for the subsequent network operations.

The second part of the project focuses on efficient hardware support for neural networks. In total, three custom ASIC accelerators are presented. Firstly, a small-footprint low-power configurable accelerator for speech recognition is proposed. It implements a novel *deterministic two-step scaling* method for reducing the number of activation memory accesses at runtime. The accelerator is compared against a typical digital signal processor, and it considerably outperforms the processor in all aspects, including lower power consumption, a smaller area, and fewer memory accesses. Therefore, the accelerator can be easily used in hearing instruments. Secondly, an energy-efficient min-heap accelerator is designed to realize the selection of the top elements for *PeakRNN*. It is also a part of the third and final accelerator, called *PeakEngine*, that is capable of executing inference for both dense and pruned layers. *PeakEngine* is configurable, and it represents the first RNN ASIC accelerator for hearing-instrument-relevant use cases that applies dynamic pruning by selecting a constant number of elements to guarantee deterministic inference. The co-optimization between *PeakRNN* and *PeakEngine* results in a significant reduction of energy and latency of the original dense network, making the execution of big RNNs viable in hearing instruments within the imposed time and energy budget.

Resumé (Dansk)

Dybe neurale netværk har revolutioneret mange forskellige områder, herunder tale forbedring talegenkendelse og tale separation, som alle er relevante for høreapparat brugere og professionelle. Samtidig er state-of-the-art neurale netværk meget store modeller, der kræver mange MBytes lagerplads og millioner af operationer for hvert input. Dette gør dem ekstremt energi krævende og derfor svært tilgængelige for apparater med begrænsede resurser, som f.eks. høreapparater. Derfor har neurale netværk indtil videre udelukkende været anvendt i high-performance computer sammenhænge (Cloud computing). Eventuelle lokale klienter har sendt data til beregning i skyen, hvorefter resultaterne er sendt til klinten (f.eks. en IoT/Edge enhed). Realtids applikationer, som f.eks. høreapparater har behov for forbindelser og processering med lav latenstid for ikke at kompromittere lydkvaliteten. Derudover vil en konstant forbindelse til skyen medføre dræn af batteriet i IoT/Edge enheder. Endelig vil deling af data i skyen være et sikkerhedsproblem. Derfor har fokus de senere år været på at sikre udførsel af neurale netværk direkte i low-power enheder.

For at nå dette mål er det essentielt at udvikle beregnings effektive dybe lærings algoritmer, der kan eksekveres i tilpassede hardware acceleratorer, som er optimeret for neurale netværks beregninger. Udforskningen af området for algoritmer og hardware acceleratorer, samt optimeringen af begge, er formålet for denne Ph.D. afhandling.

Første del af dette arbejde foreslår to nye dynamiske reduktions algoritmer kaldet PeakRNN og StatsRNN, for at reducere antallet af henholdsvis multiplikationer-additioner og memory transaktioner under inference . Eftersom vores fokus er omkring audio (tale forbedring og talegenkendelse), så vil vi primært udforske

Recurrent Neural Networks (RNNs) and Transformer Neural Networks (TNNs) på grund af deres egenskaber med at behandle tidsrelateret information. Alle vores eksperimenter viser væsentlige reduktioner i beregninger samtidig med høj præcision i evalueringsmetrikkerne. PeakRNN er udvalgt til næste fase af projektet fordi den reducerer et lag ved at udvælge et konstant antal af elementer for hvert tidselement, hvilket blandt andet, betyder determinisme og worst-case eksekveringstidsgaranti for de efterfølgende netværks operationer.

Anden del af projektet har fokus på effektiv hardware understøttelse af neurale netværk. Der bliver præsenteret i alt tre ASIC acceleratorer. Først bliver en lille low-power konfigurerbar accelerator for talegenkendelse præsenteret. Den implementerer en ny deterministisk to-step skaleringsmetode for at reducere antallet af memory transaktioner under kørsel. Denne accelerator bliver sammenlignet med en typisk digital signal processor. Acceleratoren viser overlegne resultater i alle aspekter, herunder lavere effekt forbrug, lavere areal og væsentligt færre memory transaktioner. Denne accelerator vil derfor kunne anvendes i høreapparater. Dernæst bliver en energi effektiv min-heap accelerator designet til at udvælge top elementerne i en PeakRNN algoritme. Den er også del af den tredje og sidste accelerator, kaldet PeakEngine, som er i stand til at udføre inference for både tætte og reducere lag. PeakEngine er konfigurerbar og repræsenterer den første RNN ASIC accelerator til høreapparat brugs scenarier, som tilfører dynamisk reduktion ved at udvælge et konstant antal elementer for at garantere deterministisk inference. Den samtidige optimering mellem PeakRNN and PeakEngine resulterer i en signifikant reduktion af energi og latens i forhold til det oprindelige netværk. Dette gør det muligt at udføre store RNNs i høreapparater med deres begrænsede tids og energi budget.

Preface

The work presented in this thesis was conducted at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark (DTU) and Demant A/S in fulfillment of the requirements of the Industrial PhD program. The Industrial PhD program is a part of Innovation Fund Denmark, an independent fund at the Ministry of Higher Education and Science. The work was funded by Demant A/S and Innovation Fund Denmark.

This work was supervised by Professor Jens Sparsø as the principal supervisor on behalf of academia (DTU) and Evangelia Kasapaki as the principal supervisor on behalf of industry (Demant A/S), together with David Thorn Blix and Anders Hebsgaard as co-supervisors (Demant A/S). The thesis explores an efficient implementation of deep neural networks in resource-constrained devices, such as hearing instruments, with a focus on algorithm-hardware co-optimization.

The thesis is written as a collection of five papers and organized into nine chapters, starting with an introduction, background, and thesis contributions, followed by the five papers and a conclusion chapter.

Kongens Lyngby, 30-November-2022

Zuzana Jelčicová

Acknowledgements

First of all, I would like to express my deepest gratitude to my principal supervisors Professor Jens Sparsø and Evangelia Kasapaki for their guidance, commitment, and inspiring discussions. Likewise, I would like to thank my co-supervisors David Thorn Blix and Anders Hebsgaard for their support and dedication to the project.

Secondly, I would like to thank Oskar Andersson for all his invaluable help, feedback, and fruitful discussions.

I am also grateful to Professor Marian Verhelst for her dedication, amazing collaboration, and hospitality during my external research stay at KU Leuven in Belgium. Moreover, many thanks to Musa Aydogan, Man Shi, Jun Yin, Pouya Houshmand, Josse Van Delm, Sergio Massaioli, and others at MICAS for making my time in Leuven wonderful.

Furthermore, I would like to thank Rasmus Jones for his devotion to the research we worked on together, and for his patience and encouragement in every aspect of this unpredictable journey.

Next, I would like to thank Professor Jesper Jensen for all his priceless input and suggestions.

I am grateful to Axel Bogdan Bregnsbo, Vijay Kumar Bhat, Jesper Birch, Robert Rehr, Michael Syskind Pedersen, Asger Heidemann Andersen, Anders Brødløs Olsen, Mariaflavia Manigrasso, and many other people in the Silicon Engines team for always being there to help and discuss any issues or ideas.

Special thanks to my tea-and-chat buddies Adrian Mardari and Tobias Næblerød Jeppe.

Last but not least, I would like to thank my boyfriend Stanley Ondruš, family, and friends for their love and unconditional and endless support, especially during the moments of discouragement.

Contents

List of Acronyms

ASIC Application-Specific Integrated Circuit

CNNs Convolutional Neural Networks

DNNs Deep Neural Networks

DSP Digital Signal Processor

EEG Electroencephalography

EOG Electrooculography

FC Fully Connected

FCNNs Fully Connected Neural Networks

FNR False Negative Rate

FPR False Positive Rate

GRU Gated Recurrent Unit

GSCD Google Speech Commands Dataset

IBM Ideal Binary Mask

IRM Ideal Ratio Mask

KWS Keyword Spotting

LSTM Long Short-Term Memory

MAC Multiply-Accumulate

MFCCs Mel-Frequency Cepstral Coefficients

MHSA Multi-Head Self-Attention

ML Machine Learning

MSE Mean Squared Error

PESQ Perceptual Evaluation of Speech Quality

ReLU Rectified Linear Unit

RNNs Recurrent Neural Networks

ROC Receiver Operating Characteristic

SE Speech Enhancement

SNNs Spiking Neural Networks

SNRs Signal-to-Noise Ratios

STOI Short-Time Objective Intelligibility

tanh Hyperbolic Tangent

tinyML tiny Machine Learning

TNNs Transformer Neural Networks

TPR True Positive Rate

List of Publications

Journal Publications

[J1] Zuzana Jelčicová, Evangelia Kasapaki, Oskar Andersson, and Jens Sparsø. "PeakEngine: A Deterministic On-the-fly Pruning Neural Network Accelerator for Hearing Instruments". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (Submitted).

Conference Publications

[C1] Zuzana Jelčicová, Adrian Mardari, Oskar Andersson, Evangelia Kasapaki, and Jens Sparsø. "A Neural Network Engine for Resource Constrained Embedded Systems". In: *Proceedings of 54th Asilomar Conference on Signals, Systems, and Computers*. 2020, pp. 125–131.

[C2] Zuzana Jelčicová, Rasmus Jones, David Thorn Blix, Marian Verhelst, and Jens Sparsø. "PeakRNN and StatsRNN: Dynamic Pruning in Recurrent Neural Networks". In: *Proceedings of 29th European Signal Processing Conference (EUSIPCO)*. 2021, pp. 416–420.

[C3] Zuzana Jelčicová and Marian Verhelst. "Delta Keyword Transformer: Bringing Transformers to the Edge through Dynamically Pruned Multi-Head Self-Attention". In: *Proceedings of 2nd tinyML Research Symposium*. 2022. URL: https://arxiv.org/pdf/2204.03479.pdf.

[C4] Zuzana Jelčicová, Evangelia Kasapaki, Oskar Andersson, and Jens Sparsø. "A Min-Heap-based Accelerator for Deterministic On-the-fly Pruning in Neural Networks". In: *IEEE International Symposium on Circuits and Systems (ISCAS)*. (Submitted).

Other Works

[O1] Zuzana Jelčicová. *Winner of the Three Minutes Thesis contest (3MT) at EUSIPCO*. 2021. URL: https://eurasip.org/3mt-contest/.

[O2] Zuzana Jelčicová, Rasmus Jones, David Thorn Blix, Michael Syskind Pedersen, Jesper Jensen, and Asger Heidemann Andersen. "Hearing device comprising a recurrent neural network and a method of processing an audio signal". U.S. pat. US11330378B1. Oticon A/S. May 10, 2022. URL: https://patents.google.com/patent/US20220232331A1/en.

Chapter 1

Introduction

More than 1.5 billion people (nearly 20% of the world's population) live with hearing loss, out of which approximately 5% (430 millions) have *disabling hearing loss*. Disabling hearing loss refers to hearing loss greater than 35 decibels (dB) in the better hearing ear. Moreover, it is estimated that by 2050 one in every ten people (700 millions) will have disabling hearing loss [232]. However, the majority of adults who would benefit from a hearing instrument, do not use them, and many people given a hearing instrument do not wear it [65] saying *"It does not work for me"*.

The primary complaint of hearing-impaired people and hearing instrument users is that they cannot understand *speech-in-noise* [152]. Simple amplification does not help to solve this issue as it only makes sounds louder and not clearer. Understanding speech-in-noise is a well-known phenomenon, sometimes loosely referred to as a *"cocktail party problem"* [3], which describes one's ability to focus their auditory attention on a specific stimulus while filtering out other stimuli. This is a natural and subconscious process for people with normal hearing, but very challenging for those with hearing loss. A typical example of such a scenario is a dinner party, where many people talk at the same time. While it might require a bit of effort for a person with normal hearing to concentrate on the voice of a target speaker, selecting a single voice in a crowded room will be extremely difficult and exhausting for a hearing-impaired person. In fact, the

cocktail party problem is one of the biggest challenges in audio signal processing [64, 63, 15].

One of the areas that deals with addressing the cocktail party problem is *Speech Enhancement (SE)*. SE focuses on suppressing the noise in the noisy speech, and thus improving quality and/or intelligibility of the speech. Numerous advanced signal processing methods have been proposed throughout the years to achieve this goal. However, they still struggle in complex acoustic environments with low Signal-to-Noise Ratios (SNRs) and competing talkers. As a result, the ability to clearly understand speech-in-noise still remains very challenging, even with the latest modern hearing instruments.

Machine Learning (ML), and specifically *Deep Neural Networks (DNNs)*, introduce a completely new way of sound processing. DNNs have already demonstrated their powerful capabilities in many different areas such as natural language processing [49, 81, 74], image and video processing [132, 54, 98, 80], gaming [121, 66], robotics [140, 100], medicine [94, 77], and SE as well [112, 91, 99], surpassing even human performance in most fields. DNNs have also enabled demonstrations of other features that are relevant for hearing instrument users and professionals. These include *wake-up word detection* and *Keyword Spotting (KWS)* [255, 95, 70, 97, 169, 147, 216]. Such functionalities may allow hearing-impaired people to initiate actions on their hearing instrument using just their voice, e.g, *"Hey Demant, volume up/volume down"*. Moreover, with SE the noisy speech features would be cleaned before being passed to the KWS acoustic model.

Nonetheless, the success of DNNs comes at the cost of considerable computational and hardware resources. DNNs are generally large models that require millions of parameters, translating to tens of megabytes of storage, and hundreds of millions of computations per input [197]. They are, hence, usually executed on high performance devices (cloud), and the results are then wirelessly deployed to IoT (edge) devices. In fact, there already exist cloud-based DNN systems in hearing instruments [262, 264]. However, this setup has several drawbacks, including latency, connectivity, and security issues. All these problems would be eliminated with on-the-chip DNN technology that would also offer real-time sound processing. The first such an attempt is represented with *Oticon More* [225]. Nevertheless, the size of the models and the number of computations impose a challenge for deployment of DNNs directly to resource-constrained and battery powered wearable devices. For instance, hearing instruments are extremely limited in terms of area, available memory, throughput, and power budget that is in the few milliwatt (mW) range. Yet they need to operate reliably and efficiently and last around 16-18 hours every day [44].

Therefore, a considerable amount of effort has been invested into bringing the execution of DNNs into the edge devices over the past few years. This is

referred to as *tiny Machine Learning (tinyML)* [206], a fast growing field of ML technologies and applications including hardware, algorithms, and software capable of performing on-device sensor data analytics at extremely low power, typically in the mW range and below, and hence enabling a variety of always-on use-cases and targeting battery operated devices [265]. Therefore, in order to achieve low-power execution of DNNs, it is necessary to create computationally efficient deep learning algorithms that in turn will run on custom hardware optimized specifically for DNNs. An example of such hardware are *Application-Specific Integrated Circuit (ASIC)* designs that can heavily exploit the parallel nature of DNN computations.

This thesis focuses on the above challenges and targets the co-optimization of deep learning algorithms and hardware for SE and KWS use cases. The research is therefore twofold:

1. The first half of the project aims at developing deep learning algorithms that decrease the number of computations and hence energy to enable deployment of DNNs to resource-constrained hearing instruments. Our interest is in *event-driven/data-driven* architectures that reduce computations dynamically during inference based on input data. We propose a technique for skipping computations in a deterministic manner, demonstrated on the KWS and SE tasks with a wide variety of speakers and environments reflecting the most relevant acoustic situations that people are exposed to daily. Such *adaptive inference* is an attractive solution as it results in significant savings of multiplications and memory fetches while preserving the parameter space and representation power of DNNs. Since our target application is a hearing instrument, we primarily focus on Recurrent Neural Networks (RNNs) that are suited for processing data with temporal structures such as audio, and Transformer Neural Networks (TNNs) that combine the benefits of RNNs and Convolutional Neural Networks (CNNs). *Our proposed algorithms were successfully patented [O2].*
2. The second half of the project targets efficient designs and ASIC implementations of neural network accelerators that can support the execution of the DNN algorithms locally in a low-power hearing instrument. Three hardware accelerators that implement a number of optimization techniques are proposed and evaluated: i) a neural network accelerator for Fully Connected (FC) DNNs developed for KWS, ii) a min-heap-based accelerator for the selection of elements to be processed/skipped to support the algorithm from 1., demonstrated on SE, and iii) a neural network accelerator capable of executing a full neural network that supports the dynamic pruning algorithm proposed in part 1. The selection of elements to be processed/skipped is performed by utilizing the built-in min-heap-based

accelerator. The accelerator can run both FC and recurrent layers, and it is demonstrated on the SE use case.

All the algorithms and hardware designs are presented in the five attached papers that constitute the core of this PhD research. Chapter 2 therefore provides the necessary background for the topics of the papers and describes previous relevant research in the field. First, it gives an introduction to the SE and KWS problems and traditional approaches for solving them. Subsequently, an overview of the deep learning field, including a description of the fundamental types of DNNs as well as a training process, is provided. Thereafter, DNNs and their hardware challenges are described. This is followed by presenting various techniques for alleviating the challenges, which enables efficient deployment of DNNs into low-power embedded devices.

The remaining part of this thesis is structured as follows. Chapter 3 lists the contributions of the thesis, divided into algorithmic and hardware sections. Chapters 4-8 are the journal and four conference papers that form the core of the thesis, ordered according to the date of publication/submission. Last but not least, the conclusion can be found in Chapter 9. It summarizes the thesis contributions and addresses possible future work, including other methods that were tested and could be a promising direction to pursue.

Chapter 2

Background

This chapter provides the necessary background to cover the topics discussed in the five papers that can be found in Chapters 4-8. Firstly, the SE and KWS use cases are introduced (Sections 2.1-2.2). This also includes an overview of traditional methods for solving these tasks and evaluation criteria to assess the performance of the SE and KWS systems. Thereafter, deep learning is presented (Section 2.3), which consists of describing the fundamental types of DNNs, the training procedure, and specific details for deep learning-based SE and KWS. The next section, embedded neural networks (Section 2.4), serves as a transition to the hardware part. It discusses challenges of deploying DNNs into resource-constrained devices as well as multiple hardware techniques to alleviate these challenges and minimize energy dissipation.

2.1 Speech Enhancement

One of the major research areas that addresses the cocktail party problem is *SE*. The objective of SE is to attenuate noise in a noisy speech, and thus improve overall perceptual quality and/or intelligibility of a degraded speech signal. Noisy speech can be expressed with a linear model:

$$y(i) = s(i) + n(i), \tag{2.1}$$

where $y(i)$, $s(i)$, and $n(i)$ are the i^{th} samples of a noisy speech signal, a clean speech signal, and additive noise, respectively. The goal of SE is thus to obtain an estimate $\hat{s}(i)$ of $s(i)$ by observing only $y(i)$. Noise $n(i)$ can be represented with both speech, e.g., competing speakers, and non-speech sounds, e.g., background noise experienced in a car cabin.

Numerous techniques exist for estimating $s(i)$ and many of them use the gain-based approach [63, 64] as shown in Figure 2.1. Firstly, the noisy time-domain signal $y(i)$ is divided into a time-frequency representation $y\prime(k, m)$ for time-frame m and frequency index k, using, e.g., the Short-Time Fourier Transform. A *Gain Estimator* then estimates a scalar gain factor $g(k, m)$ that is applied to the magnitude spectrum of the noisy time-frequency tile $y\prime(k, m)$ to obtain an estimate of the clean speech magnitude spectrum. Finally, a synthesis stage reconstructs the time-domain signal $\hat{s}(i)$ by applying an inverse transform to the enhanced signal.

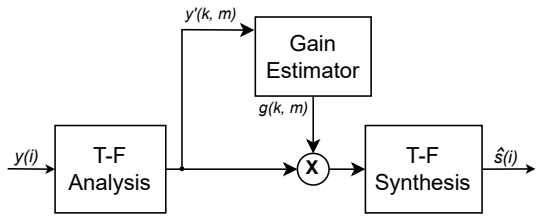


Figure 2.1: Typical gain-based SE system.

2.1.1 Traditional Speech Enhancement Algorithms

The gain value $g(k, m)$ is typically estimated using either *statistical methods* or *machine learning methods*. Statistical methods such as the Wiener filter [64], and Short-Time Spectral Amplitude Minimum Mean Square Error [10, 63] are based on an assumption that speech and noise are uncorrelated. Moreover, they require domain knowledge that is unknown in advance and hence must be estimated, such as a priori SNR, i.e., the ratio of the power of the clean signal and noise power. However, the only signal that can be observed in reality is the noisy signal. Furthermore, these methods are often based on assumptions that the noise statistics change more slowly across time than the speech statistics. However, this is not the case for many natural sources [166]. As a result, modern SE algorithms still struggle in complex acoustic environments with low signal-to-noise ratios or competing talkers.

Traditional ML methods used for SE such as Hidden Markov Models [11, 19], Support Vector Machines [18, 27, 55], and Gaussian Mixture Models [13, 42] make predictions and identify patterns based on their ability to learn from data. However, they often require a complex process of extracting features, where data analysis must be firstly performed, followed by a manual selection of the best features [240]. Moreover, classical ML algorithms can learn from relatively small datasets but perform poorly on a large amount of data.

On the other hand, *deep learning*, a subset of ML, uses *artificial neural networks* to mimic the learning process of the human brain. It tackles the SE task as a *supervised learning problem* [40], where an adaptive mathematical model, i.e., a neural network, is used instead of parametric statistical models or digital signal processing to design a gain estimator. In case of SE, during an initial training phase, i.e., before employment, a neural network is trained to enhance the noisy speech by observing a large number of examples of the clean speech along with their corresponding noisy speech. The parameters of the neural network are gradually tuned during the training process depending on how far its output (denoised speech) is from the desired output (clean speech). Compared to traditional ML methods, deep learning algorithms can automatically determine the important features that distinguish different categories of data, which significantly eliminates human intervention [261]. Hence, they scale effectively with a growing amount of data. Deep learning has demonstrated its superior performance over traditional ML methods in a huge variety of applications, including SE [62, 91, 42, 69]. Deep learning is presented in detail in Section 2.3.

Furthermore, SE algorithms can process sound signals captured by a single microphone or a microphone array. These are referred to as *single-microphone* systems and *multi-microphone* systems, respectively. The algorithms for single-microphone noise reduction do not rely on and cannot exploit the directional information of target and interference signals. They are useful in applications where microphone arrays cannot be used, e.g., for in-the-ear hearing aids that have space as well as power and hardware constraints. Moreover, single-microphone algorithms complement multi-microphone algorithms and can be used in a post-processing step after a beamforming stage [163]. The focus of this thesis is on *single-microphone SE*.

2.1.2 Evaluation Criteria

The best way to assess the quality and intelligibility of the enhanced speech is by performing listening tests with end users. However, such tests can become very expensive and time consuming, since they require careful planning and

execution, as well as numerous test subjects. Moreover, the results of listening tests might often be inconsistent due to various factors such as listener fatigue, or varying hearing ability among test subjects [163]. Therefore, much effort has been invested in designing objective measures that correlate highly with subjective rating scores [50]. Such measures are faster, cheaper, and produce the same result for the same testing condition. Although they cannot completely replace listening tests, they often provide a good estimate of listening-test results, see, e.g., [237] and [64]. Two of the most popular techniques to evaluate speech intelligibility and speech quality are *Short-Time Objective Intelligibility (STOI)* and *Perceptual Evaluation of Speech Quality (PESQ)*, respectively. These two techniques reflect different aspects of speech and are, thus, not equivalent. SNR is also included among the evaluation metrics.

2.1.2.1 Short-Time Objective Intelligibility

Speech intelligibility describes *what* a speaker says, i.e., the content of the spoken words. It is measured by presenting speech to a group of listeners and asking them to identify the words. Therefore, intelligibility can be quantified as a number between 0 and 1 that represents the percentage of words correctly understood [64, 41, 50]. *STOI* [47, 51] is the most widely used objective measure for estimating speech intelligibility. STOI is designed to produce a single scalar output in a similar range, with an output of 1 indicating fully intelligible speech [163]. STOI requires both the clean signal and the noise signal to estimate the intelligibility. It has proven to quite accurately predict the intelligibility of noisy speech in a wide range of acoustic scenarios [51, 93, 75].

2.1.2.2 Perceptual Evaluation of Speech Quality

Speech quality describes *how* a speaker produces an utterance. It is highly subjective and therefore very difficult to evaluate reliably, since people have different standards of what constitutes *good* or *poor* quality [41, 50]. The *PESQ* [26] is one of the most widely used objective measures for estimating speech quality [64], and it is used as the *International Telecommunications Union* recommendation P.862 [25]. It was originally designed for evaluating speech coding algorithms. PESQ results are designed to approximate *Mean Opinion Score*. Mean Opinion Score is a procedure where the test subjects listen to the test signal and rate the quality of the signal on a scale from 1 to 5 as shown in Table 2.1. The output of this method is the total quality (a single scalar) that is calculated as the average of the individual scores obtained from all test subjects - hence the name *Mean Opinion Score*. Nonetheless, the PESQ algorithm itself

is rather complex as it consists of multiple steps such as preprocessing, time alignment, accounting for masking effects, etc. [163]. It requires both the clean speech signal and the noisy signal to estimate the perceived quality of the noisy signal. Research has shown that PESQ is highly correlated with listening-test experiments based on Mean Opinion Score [26, 28].

Table 2.1: Mean Opinion Score rating scale that uses five discrete steps.

Rating	Speech Quality	Level of Distortion
5	Excellent	Imperceptible
4	Good	Just perceptible, but not annoying
3	Fair	Perceptible and slightly annoying
2	Poor	Annoying, but not objectionable
1	Bad	Very annoying and objectionable

2.1.2.3 Signal-to-Noise Ratio

The *SNR* is the most popular metric to compare the level of a desired signal to the level of background noise. The SNR is defined as the ratio of signal power to noise power, often expressed in dBs. Although SNR does not directly correlate with the speech intelligibility or quality [64], it is also an important metric for evaluating the efficiency of SE algorithms. For instance, a low SNR will decrease how accurately a system can recognize speech. Robust estimation of SNR can help guide the design of SE systems.

2.2 Keyword Spotting

The objective of *KWS* is to identify keywords in audio streams. Subsequently, different actions might be performed depending on the detected keyword (e.g., "lights on", "turn off the TV"). KWS has already become a ubiquitous and popular feature that is used in countless devices. A typical example are voice assistants such as Apple's Siri, Amazon's Alexa, Microsoft's Cortana, and Google's Assistant [159]. However, running KWS constantly would use a lot of computational resources and hence power. Therefore, KWS is firstly activated only when a *wake-up* word is detected, such as "Ok Google". Additionally, *voice activity* detection is often used to reduce power consumption even further, by recognizing whether speech is present in the signal or not. If speech is not present, all activities are in a sleep mode. Although the wake-up word and KWS tasks have different objectives, their underlying technology is very similar.

2.2.1 Traditional Keyword Spotting Algorithms

One of the first KWS attempts is based on *large-vocabulary continuous speech recognition* [17, 36]. In this method, the speech signal is decoded and the keyword is searched in the generated sequences of phonetic units [255]. Another approach is to use *Hidden Markov Models* with Viterbi decoding, where a keyword Hidden Markov Model and a filler Hidden Markov Model are trained to model keyword and non-keyword audio segments. When the likelihood ratio of the keyword model versus filler model is larger than a predefined threshold, the KWS system is triggered. As in the case of SE, DNNs outperform these techniques and are thus the preferred solution for KWS.

2.2.2 Evaluation Criteria

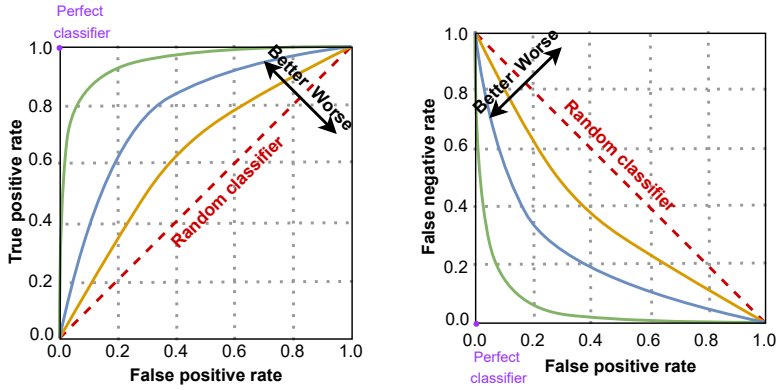
The best evaluation of a KWS system would, again, be performed with end users. Instead, objective metrics are used to substitute such costly and tedious tests. The following subsections present the most common metrics used for analyzing the predictive power of a KWS classifier. In this project, the focus is on *accuracy*. However, other commonly used metrics are stated for completeness.

2.2.2.1 Accuracy

Accuracy is the simplest metric that describes the ratio between the number of correctly classified keywords and the total number of words used in the test. Therefore, the resulting number is in the $[0, 1]$ range. Although simple, it is very important to use datasets with balanced class distributions, such as *Google Speech Commands Dataset (GSCD)* [178], to avoid biased evaluations [255]. Moreover, a *confusion matrix* is often used to summarize the performance of a classifier, where the rows of the matrix represent the instances of actual classes, while the columns represent the instances of predicted classes.

2.2.2.2 Receiver Operating Characteristic and Detection Error Trade-off Curves

The *Receiver Operating Characteristic (ROC)* curve [34] illustrates a plot of pairs of *True Positive Rate (TPR)* on the y-axis and *False Positive Rate (FPR)* on the x-axis, by varying the discrimination threshold (see Figure 2.2a). TPR,



(a) A *receiver operating characteristic* curve. **(b)** A *detection error trade-off* curve.

Figure 2.2: The *receiver operating characteristic* and *detection error trade-off* curves outlining space for a "better" and "worse" classifier. The purple dots correspond to a perfect classifier.

also called *recall* or *sensitivity*, is the fraction of correctly identified keywords. It is expressed as:

$$TPR = \frac{True\,Positive}{True\,Positive + False\,Negative} \tag{2.2}$$

FPR is the fraction of keywords that triggered a false alarm. It is expressed as:

$$FPR = \frac{False\,Positive}{False\,Positive + True\,Negative} \tag{2.3}$$

As it can be observed in the equations above, TPR and FPR are derived independently, which ensures that the class distribution in the dataset does not affect the ROC curve, unlike in the case of accuracy. A ROC curve of a good classifier is as close to the top left corner of the graph as possible (i.e., the green curve). A perfect classifier would correspond to FPR = 0 and TPR = 1, as shown with a purple circle in Figure 2.2a. If a system performs on the ROC space identity line, it randomly guesses the outputs.

The *detection error trade-off* [22] curve (see Figure 2.2), uses *False Negative Rate (FNR)* on the y-axis instead, and it is expressed as:

$$FNR = \frac{False\,Negative}{False\,Negative + True\,Positive} \tag{2.4}$$

Therefore, a perfect classifier is represented with FPR = 0 and FNR = 0. The *equal error rate* can thus be easily obtained as the intersection point between

FNR and FPR. The lower the equal error rate, the better the performance of the system. In real-world KWS applications, the cost of a false alarm is often significantly greater than a missed detection [68]. Therefore, FPR in the ROC and detection error trade-off curves is usually replaced with the number of false alarms per hour [176, 174, 230].

2.2.2.3 Precision-recall and F1-score Curves

Similarly, *precision-recall* curve plots pairs of TPR (recall) and precision values that are obtained by varying the discrimination threshold [33]. A perfect classifier has recall = 1 and precision = 1. Precision is defined as:

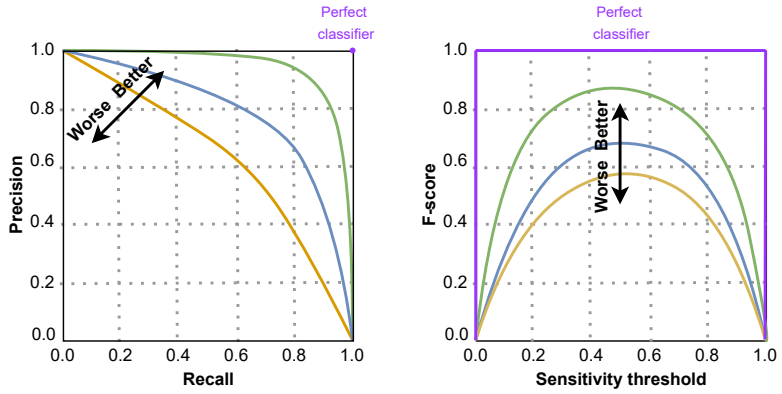
$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{2.5}$$

The precision-recall curve hence enables to focus on the minority positive class of interest.

The *F-score* metric, *F1*, is the harmonic mean of the precision and recall [7]:

$$F_1 = \frac{2}{Recall^{-1} + Precision^{-1}} = \frac{2\,True\ Positive}{2\,True\ Positive + False\ Positive + False\ Negative} \tag{2.6}$$

The value of F1 is again in range $[0, 1]$. The larger the F1, the better the performance.



(a) A precision-recall curve. **(b)** An F-score curve.

Figure 2.3: The precision-recall and F-score curves outlining space for a "better" and "worse" classifier. The purple dot and line for the precision-recall and F-score, respectively, corresponds to a perfect classifier.

2.3 Deep Learning

Deep Learning is a subset of ML, and it uses a hierarchy of data representations, where each of the levels transforms the input representation non-linearly into a higher, more abstract format. This hierarchical model is referred to as a *neural network* [2].

2.3.1 Neural Network Basics

Thanks to the hierarchical structure, neural networks are capable of capturing non-linear and complex relationships by learning from a vast amount of data. Each of the levels in a hierarchy corresponds to a *layer*, and each layer consists of a set of *neurons*. A neuron is connected to other neurons in the previous and/or subsequent layers using *weights*. The structure of neural networks shows crude similarities to a human brain that is composed of networks of billions of neurons.

Neural networks can solve a wide variety of tasks that can be broadly placed into two main categories. Either we need to predict i) discrete class labels (e.g., recognizing objects in pictures, spoken keywords), or ii) continuous values (e.g., stock prices, postfilter gain values). These are referred to as *classification* and *regression*, respectively. Moreover, different types of neural networks are suited for different problems and applications. The three most fundamental architectures are presented in Sections 2.3.2-2.3.4. Based on these, many other architectures have emerged, such as *TNNs* [146], *diffusion networks* [102], etc. TNNs are introduced in Section 2.3.5 as they are used in paper [C3].

2.3.2 Feed-Forward Neural Networks

Feed-forward neural networks, also known as dense or *Fully Connected Neural Networks (FCNNs)*, with K layers can be represented as a chain of non-linear functions:

$$f = f^K(f^{K-1}(...f^1)), \tag{2.7}$$

where f^1 and f^K correspond to the first (*input*) layer and and the last (*output*) layer, respectively. All the layers in between (f^2 to f^{K-1}) are called *hidden layers*. This flow from the input to the output layer is also called a *forward pass*. The number of layers, K, is referred to as the *depth* of the network, while the number of neurons per layer, N, is called the *width*. As shown in Figure 2.4, FCNNs have no internal feedback connections.

The most fundamental operation for all neural networks is a *dot product*. In case of a FCNNs, each neuron in a layer performs a dot product on an input feature (activation) vector X and its corresponding weights W, generating a single output feature (activation). Every multiplication of a single input and its respective weight, followed by an addition with the intermediate result, is known as a *Multiply-Accumulate (MAC)* operation. A *bias* value, b, which applies an affine transformation, is then added to the weighted sum. Finally, an activation

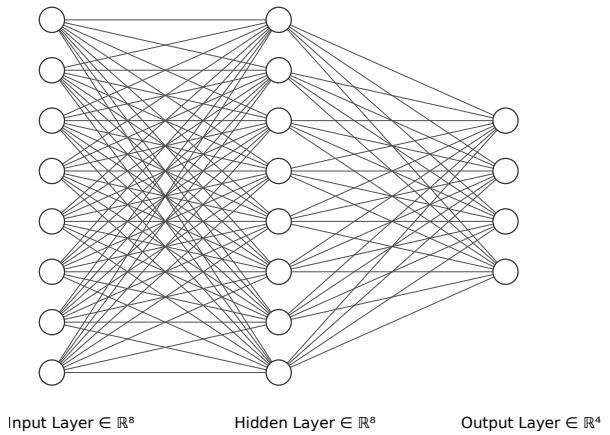


Figure 2.4: An example of a simple FCNNs with the *depth* $K = 3$ and the *width* $N = \{8, 8, 4\}$.

function, a, is used to model non-linearity. Various types and variations of activation functions exists. The most popular ones are illustrated in Figure 2.5 and described below:

- *Rectified Linear Unit (ReLU)* - outputs zero if the input is negative, otherwise, it outputs the input directly: $ReLU(x) = max(0, x)$. It has become the default activation function for many types of neural networks due to its simplicity and performance.

- *Sigmoid (σ)* - maps inputs into a range between 0 and 1: $\sigma = \frac{1}{1+e^{-x}}$. It is especially useful when a real number needs to be converted to a probability.

- *Hyperbolic Tangent (tanh)* - similarly to sigmoid, *tanh* maps inputs to a range between -1 and 1: $tanh = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

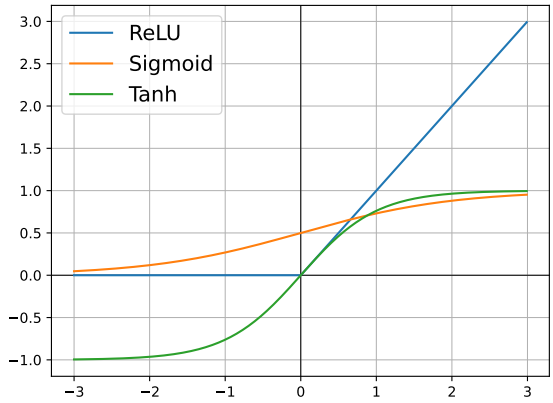


Figure 2.5: ReLU, sigmoid and tanh activation functions.

The functionality of a single neuron can thus be summarized with the following equation:

$$O = a(\sum_{i=1}^{N} W_i X_i + b) \tag{2.8}$$

The output O, called *activation*, will serve as an input for a subsequent layer. The simplest form of neural network consists of just three layers: input, hidden, and output layer, as visualized in Figure 2.4.

2.3.3 Convolutional Neural Networks

CNNs are a specialized type of a FCNN architecture that utilize *weight-sharing* [109]. Weight-sharing means that the model weights are shared for multiple inputs, which corresponds to the *convolution* between the input to the layer and a set of weights called *kernels* or *filters*. Such an approach differs from FCNNs where each input has its own fixed set of weights. Therefore, CNNs can be more parameter-efficient [109], especially for high-dimensional input data such as images. While CNNs can utilize small kernels, e.g., 3x3, in the image to extract different information like edges, a FCNN would require orders of magnitude more parameters to perform the same operation. CNNs are hence powerful for capturing *spatial* dependencies. However, as shown below, CNNs are at the same time computationally intensive.

The following description summarizes how a convolutional layer works. A convolutional layer receives an input image of dimensions $W \times H \times C$ and produces an output of *activation maps* $W\prime \times H\prime \times C\prime$, known as *feature maps*. W, H, and C are the width, height, and the number of channels in an *input feature map*, and $W\prime$, $H\prime$, and $C\prime$ in an output feature map. For instance, in a polychromatic image such as RGB, three channels are needed in order to represent the red, green, and blue channel. The filter for a convolution is hence a tensor of dimensions $K \times K \times C \times C'$, where K is the kernel size. In order to reduce the memory complexity, a step size (i.e., a stride) greater than one pixel is used when shifting the kernel across the image. As in case of a FCNN, a bias, b, can be added to the weighted sum, and the result will be passed through an activation function a. All these steps are illustrated in Figure 2.6 and described by the following equation:

$$O[c\prime][x][y] = a(\sum_{c=0}^{C}\sum_{i=0}^{K}\sum_{j=0}^{K} I[c][Sx+i][Sy+j] \times W[c\prime][c][i][j] + B[c\prime]), \quad (2.9)$$

where I and S represent an input feature map and a stride, respectively, and x, y are bounded by $x \in [0, ..., W]$ and $y \in [0, ..., H]$. As visualized in Figure 2.6, a CNN typically utilizes a FC layer in the end for classification.

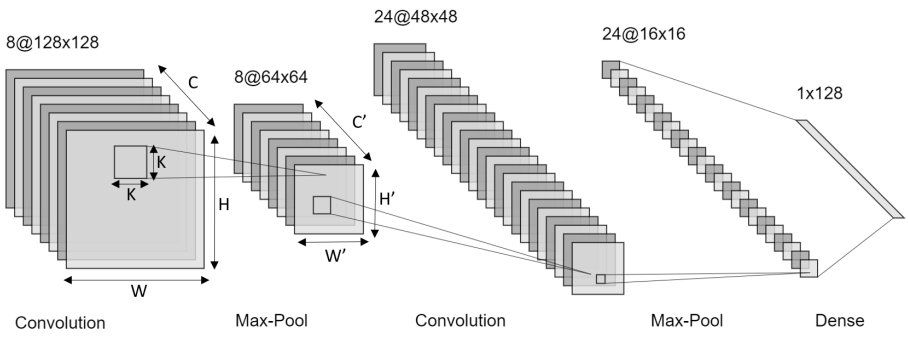


Figure 2.6: An example of a CNN with two convolutional and max-pooling layers, followed by a FC layer for classification.

Another layers used in CNNs are *pooling* layers, which i) by downsampling reduce the dimension of the feature map and thus memory requirements, and ii) add translational invariance to small shifts and distortions in the inputs. One of the typical pooling layers is *max-pooling*, which outputs only the maximum of a local patch in a feature map.

2.3.4 Recurrent Neural Networks

Traditional FCNNs struggle with modeling long-term time dependencies. *RNNs*, on the other hand, excel in this task. RNNs are feed-forward neural networks with a feedback connection that is shared between timesteps. This connection enables RNNs to retain information, which makes them powerful for processing data with *temporal* structures. They are, therefore, useful in a variety of applications such as, natural language processing, translations, speech recognition, and video processing. The simplest form of an RNN is illustrated in Figure 2.7, where a unit accepts input x_t and outputs h_t. The feedback loop allows information to be passed from one timestep to the next one. The loop can be unrolled in time, i.e., across different timesteps, and thus seen as several copies of the same unit. However, these *vanilla* RNNs suffer from short-term memory. If a sequence is

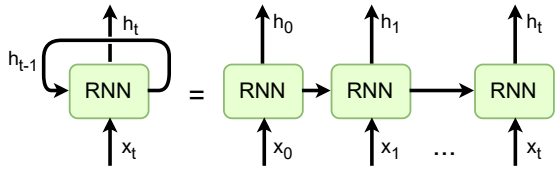


Figure 2.7: An unrolled RNN in time, where x and h correspond to inputs and outputs, respectively.

too long, they will have difficulties with forwarding information from previous timesteps to the later ones due to the *vanishing gradient problem* [109]. Gradients are used during training to update parameters of a neural network (see Section 2.3.6). If they become too small, the parameter updates will become insignificant and, as a consequence, no learning will take place. A *Long Short-Term Memory (LSTM)* [20] and a *Gated Recurrent Unit (GRU)* [71] were created as the solution to the short-term memory problem. They have internal mechanisms called *gates* that can learn which part of a sequence is important to keep, thus regulating the flow of information.

The key idea behind LSTMs is the *cell state* c_t, and three gates: *forget* f_t, *input* i_t, and *output gate* o_t, as illustrated in Figure 2.8. The cell state, c_t, runs through the entire chain, with minor linear interactions. Since it has neither an activation function nor weights, its value will remain constant during gradient updates, hence avoiding the vanishing gradient problem. Information can be added to or removed from the cell via gates, where each of the gates is a separate FCNN. The gates can therefore learn what information is relevant to keep or forget during training. All gates have a sigmoid function that maps values to the $[0, 1]$ interval. The closer the values are to 0, the less of the information will be propagated.

Similarly, the closer the values are to 1, the more of the information will be retained and passed to the next timestep. The following equations describe how an LSTM works:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{2.10}$$

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{2.11}$$

$$\tilde{c_t} = tanh(W_c[h_{t-1}, x_t] + b_c) \tag{2.12}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c_t} \tag{2.13}$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{2.14}$$

$$h_t = o_t \odot tanh(c_t) \tag{2.15}$$

$\tilde{c_t}$ represents new candidate values that are used to update old cell state, c_{t-1}, into the new cell state c_t. The GRU is a variation of the LSTM. The biggest

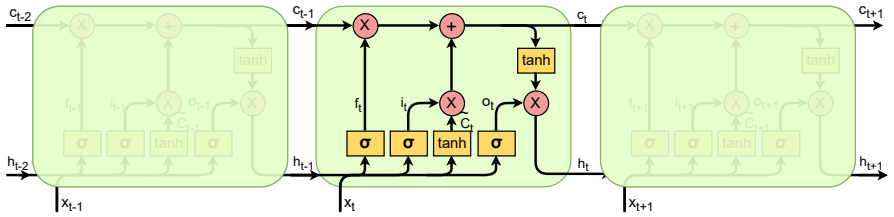


Figure 2.8: An LSTM unit unrolled in time across three timesteps.

differences between the two gated-RNNs is that the GRU i) uses two gates instead of three, called *update* and *reset* gate, and ii) merges the cell state and hidden state into a single hidden state h_t. This is illustrated in Figure 2.9. The GRU is hence less complex and contains fewer parameters than the LSTM. The GRU can be expressed as:

$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r) \tag{2.16}$$

$$u_t = \sigma(W_u[h_{t-1}, x_t] + b_u) \tag{2.17}$$

$$c_t = tanh(W_{xc}x_t + r_t \odot (W_{hc}h_{t-1}) + b_c) \tag{2.18}$$

$$h_t = u_t \odot h_{t-1} + (1 - u_t) \odot c_t \tag{2.19}$$

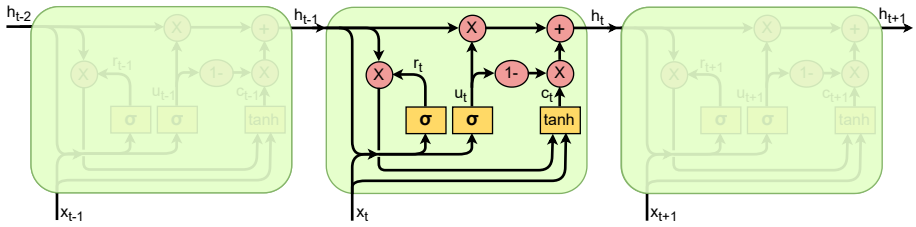


Figure 2.9: A GRU unrolled in time across three timesteps.

2.3.5 Transformer Neural Networks

As shown in Section 2.3.4, RNNs generate a sequence of hidden states $h(t)$ based on the previous hidden state $h(t-1)$ and input $x(t)$. This sequence dependence limits the amount of parallelism that can be exploited during computation. *TNNs*, firstly introduced in [146], eliminate this dependency by relying on a *self-attention mechanism* that enables them to attend to different parts of the inputs in parallel. Self-attention therefore combines the best of CNNs and RNNs, i.e., parallel computations and handling of long dependencies. Moreover, the significance of each input within a sequence is weighed differentially, as described below.

The attention mechanism (see Figure 2.10) is a function of three main components, namely *queries* Q, *keys* K, and *values* V, where the goal is to find a mapping between a query and a set of key-value pairs to an output. Q, K, and V are obtained by multiplying the input x with corresponding projection matrices:

$$Q = xW_Q \quad K = xW_K \quad V = xW_V \tag{2.20}$$

Typically, *Multi-Head Self-Attention (MHSA)* is applied afterwards, where the matrices are divided into k heads ($i = 1, .., k$) to execute the self-attention in parallel. Firstly, each query vector Q is mapped against a set of keys K to produce an *attention score* s. The $\sqrt{d}$, where d is the dimension of Q and K divided by k, is a scaling factor that normalizes the average dot-product to ensure more stable gradients on the softmax function introduced in the next step:

$$s_i = \frac{Q_i K_i^T}{\sqrt{d}} \tag{2.21}$$

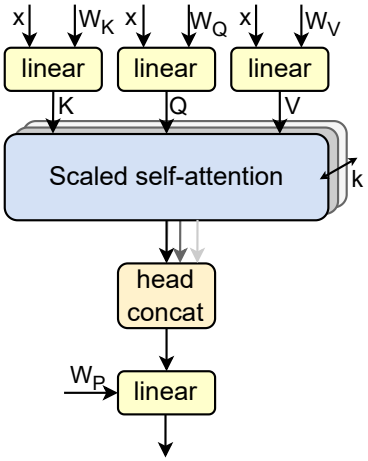


Figure 2.10: A Multi-Head Self-Attention principle based on the original [146].

Subsequently, a softmax function is applied on the scaled scores to generate the *attention weights* in a range of [0, 1]. The weights are then applied to V to compute the output:

$$head_i = attention(Q_i, K_i, V_i) = softmax\left(\frac{Q_i K_i^T}{\sqrt{d}}\right) V_i \tag{2.22}$$

The attention heads are in the end concatenated together and multiplied with a projection matrix W_P, producing the final MHSA output:

$$MHSA(Q, K, V) = concat(head_1, ..., head_1) W_P \tag{2.23}$$

TNNs are an emerging type of neural networks that have already demonstrated their powerful capabilities in many different areas, especially natural language processing [210, 186]. Natural language processing was also the objective of the first transformer [146], which defined the baseline TNN architecture as an encoder-decoder block. Both modules in the block consist of stacked self-attention and FC layers with a residual connection applied around each of the two layers, followed by a layer normalization. The decoder adds additional third layer, another MHSA, that implements masking to make the output dependent only on the preceding, i.e., known outputs. Our transformer in paper [C3] targets keyword spotting and hence utilizes only the encoder part.

2.3.6 Training Deep Neural Networks

The key factor that gives neural networks their power is the ability to learn from data, which is achieved through an iterative process called *training*. The core of training is a *backpropagation* algorithm. Backpropagation fine-tunes the parameters, i.e., weights and biases, of a neural network based on the error rate. The error rate is expressed with a *loss function*, which measures how much the predicted output $\hat{y}$ deviates from the ground truth y. Some of the commonly used loss functions are *cross-entropy* and *Mean Squared Error (MSE)*. Cross-entropy is used to optimize *classification* models. The cross-entropy loss function and its derivation are defined as:

$$L_{CE} = -\sum_{i=1}^{N} t_i log(p_i) \tag{2.24}$$

$$\frac{\partial L_{CE}}{\partial p_i} = \frac{-t_i}{p_i} \tag{2.25}$$

N, t_i, and p_i are the number of output neurons (classes), the truth label, and the softmax probability for the i^{th} class, respectively.

The MSE is a popular loss function for regression models. It is calculated as the average of the squared differences between predicted and expected target values. The closer a regression line is to a set of points, the lower the MSE and, consequently, the better the performance. The MSE loss function and its derivation are defined as:

$$L_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(t_i - \hat{t_i})^2 \tag{2.26}$$

$$\frac{\partial L_{MSE}}{\partial \hat{t_i}} = \frac{1}{N}(t_i - \hat{t_i}) \tag{2.27}$$

Minimizing a loss function requires to find a local minimum of a given differentiable function. This is achieved via a different variations of an optimisation algorithm called *gradient descent*. Gradient is a collection of all partial derivatives of the function with respect to its variables. It simply measures the change in all parameters with regard to the change in error. The partial derivatives of a loss function are obtained through the chain rule of differentiation, which corresponds to propagating the error from the output layer to the input layer. The chain rule is defined as:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \tag{2.28}$$

As it can be observed, a variable z depends on the variable y, which depends on the variable x. Hence, y and z are dependent variables, and z also depends on x via y. The chain rule is also visualized in Figure 2.11, with respect to the loss function L. Therefore, the partial derivatives of the loss function to any weight

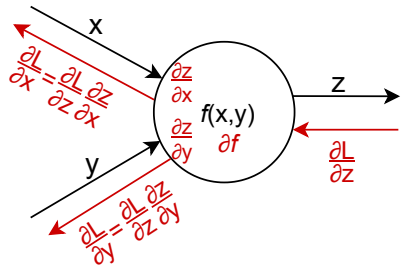


Figure 2.11: An illustration of the chain rule, where output z is calculated as a function of local inputs x and y, $f(x, y)$. The black and red arrows denote forward and backward pass, respectively.

w_{ij} can be expressed as:

$$\frac{\partial L}{\partial w_{ij}} = \sum_p [\frac{\partial L}{\partial \hat{t_p}} (\sum_k \frac{\partial \hat{t_p}}{\partial z_k} \frac{\partial z_k}{\partial w_{ij}})], \tag{2.29}$$

where $\sum_p$ is the summation over all output units and $\sum_k$ is the summation over all inputs contributing to t_p. The derivatives are then used to repeatedly adjust the network weights and biases to minimize the loss.

The whole process described above is performed iteratively on a large amount of data, where the data is usually represented in a 32-bit floating-point format. If the dataset is too small and/or not diverse enough, the network will not learn to generalize and it will start to *overfit* instead. Overfitting means that the model will learn the details and noise in the training dataset but will perform poorly on new, unseen data.

Once a good dataset has been collected, the data is typically divided into three main subsets: *training*, *validation*, and *test*, often in a 80:10:10 ratio. The training and validation subsets are used during *training*, while the test subset during *inference*. The train subset is split into groups of samples, referred to as *batches*, that will be propagated through the network. The network parameters are updated after every batch. When a forward and backward passes are completed for the entire dataset, called an *epoch*, the performance of the model is verified on the validation subset. The validation subset can be used to determine when to stop training (*early stopping*) in order to prevent the network from *overfitting*.

In the simplest case, training is stopped as soon as the performance (e.g., loss) on the validation subset in the current epoch is worse compared to the performance during the previous epoch.

Once training is completed, i.e., a specified number of epochs has been executed, the trained model with its fixed parameters can be evaluated on the test set during inference. Ideally, the test set is completely independent from the training set and validation set, i.e., the same data samples are not shared across any of the sets. This is necessary since the test set verifies how the model generalizes to unseen samples.

2.3.7 Deep Learning for Speech Enhancement

Recently, deep learning-based SE systems that operate directly on the noisy time-domain signal and output an enhanced time-domain signal, e.g., [249, 212], have been proposed. However, in this thesis we focus on methods that operate in a time-frequency domain, such as the Short-Time Fourier Transform domain, to comply with the hearing-instrument application.

In order to train a neural network for SE in the Short-Time Fourier Transform domain, it is necessary to define a suitable target. Training targets can be divided into two types: *signal approximation* and *mask approximation*. The goal of signal approximation is to train an estimator that minimizes the difference between the spectral magnitude of the clean speech and that of the estimated speech [82, 78, 99]. However, this thesis focuses on mask approximation, which describes the time-frequency relationships between the target speech and background noise [194]. The mask approximation computes a target mask and measures the error between the estimated mask and the target mask. The neural network will thus learn to estimate time-frequency masks from noisy acoustic features, guided by the MSE loss function of the estimated and target masks [194]. The most popular masks are *Ideal Binary Mask (IBM)* [43] and the *Ideal Ratio Mask (IRM)* [67]. IBM treats SE as a binary classification problem, as it classifies time-frequency units into speech-dominant and noise-dominant units. It is defined as:

$$IBM = \begin{cases} 1 & \text{if } \frac{|s(f,t)|}{|n(f,t)|} > T_{SNR}(f) \\ 0 & \text{otherwise,} \end{cases} \tag{2.30}$$

where the $s(f,t)$ and $n(f,t)$ represent the clean speech and noise, respectively, with frequency channel f and time frame t. $T_{SNR}(f)$ denotes a frequency-dependent tuning parameter [163]. However, IBM has difficulties capturing mask regions with low speech energy. This is due to the fact that time-frequency segmentation based on either speech or noise dominating is too coarse, since

speech and noise are likely to be present at the same time in the same time-frequency unit [153]. This problem is resolved by a continuous IRM that outputs a gain between zero and one. It is defined as:

$$IRM = \left(\frac{|s(f,t)|}{|s(f,t)| + |n(f,t)|} \right) \tag{2.31}$$

It has been shown that IRM outperforms IBM in SE tasks as well as in objective evaluation metrics [163].

2.3.8 Deep Learning for Keyword Spotting

The typical deep learning pipeline of a KWS task [255] is illustrated in Figure 2.12. It consists of i) a feature extractor, and ii) a neural network classifier. The speech feature extractor firstly converts the time-domain input speech signal of length L into overlapping frames of length l with a stride s, producing a total number of frames T equal to [147]:

$$T = \frac{L - l}{s} + 1 \tag{2.32}$$

Thereafter, F frequency-domain speech features are extracted from each frame, resulting in $T \times F$ features for the entire input speech signal. These features are then fed into a DNN acoustic model that outputs the likelihood of each keyword/filler class. In a real-world case, where keywords are recognized from a continuous audio stream, a posterior handling module would be used to average the output probabilities of each output class over a period of time, which would improve the overall confidence of the prediction [147].

Mel-Frequency Cepstral Coefficients (MFCCs) [8] are the most widely used speech features in KWS [183, 174, 230, 223, 147]. Another popular method [176, 170, 70, 97, 181] are *log-Mel filterbank energies* that are based on the Mel-scale filterbank, which tries to mimic the non-linear human ear perception of sound, by being more discriminative at lower frequencies and less discriminative at higher frequencies [1]. Both of these human-engineered types of speech features are commonly used in the state-of-the-art KWS systems. MFCC features are also used in some of our studies.

2.4 Embedded Neural Networks

As shown in the previous sections, DNNs are very powerful and capable of solving a vast variety of tasks. The state-of-the-art DNNs usually consist of

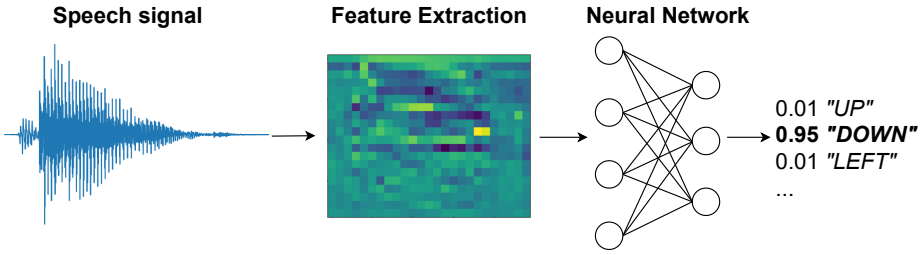


Figure 2.12: An example of a typical flow for KWS using DNNs. An input speech signal (here the word *"down"* from the GSCD [178]) is transformed into features (e.g., MFCC) that are subsequently passed to a DNN that produces posterior probabilities for each output class.

multiple hidden layers, each having hundreds to thousands of neurons. DNNs for SE and KWS achieve outstanding results. The existing systems are based on different network architectures, training and testing methods, speech corpora, noise databases and types, feature and target representations, the SNR, etc. What most of them, however, have in common is i) a large memory footprint in terms of network weights that must be stored, and ii) millions of computations per every input.

2.4.1 Computational and Memory Challenges

The current DNNs for SE and KWS require significant resources that hinder their execution in edge devices like hearing instruments. For instance, FCNNs for SE in [78, 99, 82, 91, 104] have an input dimension ranging from 1,230 to 1,845, one to five hidden layers of 1,024 - 2,048 neurons, and an output dimension of 64-320 neurons. The storage requirements are thus from 2 - 20.5 million weights. Assuming reduced 8-bit precision (see Section 2.4.3), $\sim$2 - 20 MB would be necessary to support these sizes. CNN models for SE in [165], [212], and [139] have 10 million, 33.5 million, and even 97.47 million parameters, respectively, requiring 10 - 91 MB of storage. Recent SE methods have utilized the power of RNNs [228, 179, 88]. In [228], a novel type of recurrent unit called Equilibrated RNN [190] is used. It has fewer parameters, yet, the biggest model still requires 1.05 million weights. LSTM networks in [88] and [179] result in 1 million to 65 million parameters (1 - 62 MB). Moreover, a combination of CNNs and RNNs has been proposed in several works such as [173] that uses a convolutional encoder-decoder structure. Similarly, TNNs are gaining more and more popularity [249].

However, these architectures still result in millions of parameters and even more computations.

The same issue can be observed in the existing KWS systems. Although many small-footprint neural networks have been designed [223, 230, 174, 183], the bigger and more powerful ones still require a significant amount of computations and storage. For instance, the model in [167] has "only" 251,000 parameters, but at the same time it performs 25.1 million MAC operations. Similarly, *Residual CNN res15* [174] has a relatively small number of parameters (238,000), yet it requires 894 million MACs. Plenty of other networks for KWS can be named, such as *Keyword Transformer KWT-3* [233] with 5.3 million parameters, *CENet-40* [185] with 61,000 parameters and 16.18 million MACs, or *Residual DS-CNN* [230] with 72,000 parameters and 285 million MACs.

Naturally, all of the presented models have different capabilities that have an impact on the objective measures. What can though be observed is that the model sizes in all of the proposed systems for either SE or KWS require from thousands to millions of both parameters and multiplications. Models with large enough capacity can capture the necessary relationships in order to work in real-life scenarios with, e.g., unseen environments, voices, and accents. For instance, it has been shown that small networks have difficulties learning the relationship between the noisy features and the target SNRs [31]. On the other hand, it is not currently possible to keep all the weights of such models in a small on-chip memory. Significant portion of the weights thus has to be stored in an expensive off-chip DRAM - if the amount of weights can be fitted.

The problem with the model size for battery-powered devices is, however, not only the memory requirements but also the number of *memory fetches* and *computations* as shown above. Executing a MAC operation requires fetching weights from the memory, which is yet another source of considerable power consumption. Both types of operations, particularly memory accesses, belong to the most costly ones [73]. Due to all these reasons, neural networks are typically realized on high-performance server devices, and the results are then deployed wirelessly to IoT devices. However, real-time applications require low latency connections. Moreover, sharing data with the cloud is not desirable due to security and privacy issues. Last but not least, wireless connections would quickly drain a battery of an edge device. Therefore, huge effort is being invested into moving the computations, especially *inference*, towards the edge [197]. *TinyML* pursues the goal of enabling ML applications on resource- and power-constrained devices. This effort is extraordinarily multidisciplinary, and requires optimizations on different levels, including hardware, software, data science, and ML. Therefore, in order to achieve this goal, it is crucial to look at deep learning algorithms and the hardware that runs them conjointly, and create innovations on both levels, i.e.:

1. Develop deep learning algorithms that help to reduce computations and power.
2. Design custom hardware that can support execution of these algorithms locally in a low-power device.

The following sections describe crucial aspects of the algorithm-hardware co-optimization based on the key intrinsic characteristics of neural networks.

2.4.2 Energy-Efficient Dataflow

Neural networks have a very particular dataflow that offers a large amount of potential parallelism and data reuse [197]. This can be heavily exploited by developing *customized hardware accelerators* that are designed specifically for the target algorithms. Such accelerators can maximize the parallel execution of the algorithm while minimizing the data movement, which will improve *energy-efficiency* and *throughput*, especially compared to using general purpose processors.

In neural networks, every input is multiplied with a set of weights. Each MAC thus requires three memory reads; one for an input, a weight, and a partial sum, and one memory write for the updated partial sum (see Figure 2.13). Therefore,

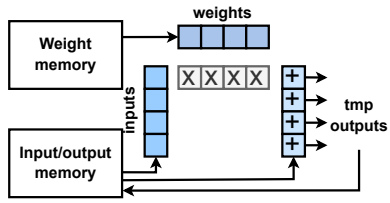


Figure 2.13: A visualization of the MAC operation and the necessary memory accesses that are denoted with arrows.

the bottleneck of the DNN execution lies in the memory access - one of the most expensive operations [73]. Hence, once the data is fetched, it should be reused as much as possible to reduce the number of subsequent memory accesses. Moreover, in the worst case data has to be fetched from a large off-chip memory, such as DRAM, as it is not always possible to fit the entire network model into an on-chip memory, as discussed in Section 2.4.1. As a result, data has to be moved constantly between the different memory levels. Such data movement is a

substantial issue since, e.g., DRAM consumes orders of magnitude higher energy per access than a small on-chip memory [73]. This problem again emphasizes the importance of reusing the data once it is fetched to reduce the number of expensive memory accesses. The characteristic operations in DNNs offer several opportunities for data reuse [143, 197], where the same data is reused *across multiple parallel execution units in the same clock cycle* (data parallelism), also visualized in Figure 2.14. Separate weight and input/output memories are assumed:

- *Input reuse* - many partial results are computed with one loaded input in one cycle. This approach is especially beneficial for FCNNs and RNNs.
- *Weight reuse* - many partial results are computed with one loaded weight in one cycle. This approach improves the weight memory bandwidth and is specifically suitable for CNNs.
- *Output reuse* - many partial results of one output are computed in one cycle.

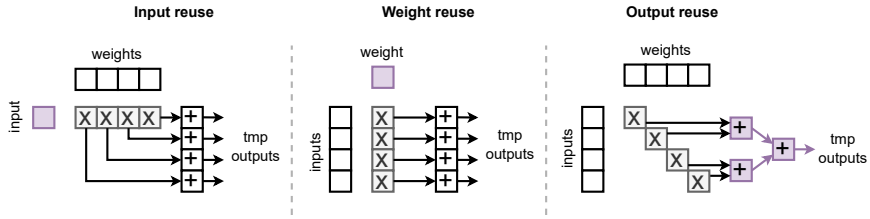


Figure 2.14: Three different data reuse techniques based on [197].

Data can be also reused *across multiple clock cycles on the same execution unit*, referred to as *data stationarity*. Similarly to the data reuse, three main options can be identified (visualized in Figure 2.15):

- *Input-stationary* - once the input data is fetched, it is stored in a register file and multiplied with the weights of neurons in a layer (multiple weight and output loads and stores).
- *Weight-stationary* - once the weight is fetched, it is stored in a register file and multiplied with layer inputs (multiple input and output loads and stores).

- *Output-stationary* - a very common approach that minimizes the energy consumption of reading and writing the partial sums. The partial sum of the same output activation is kept locally in a register file, and intermediate results are accumulated across different clock cycles (multiple input and weight loads). As shown with a blue arrow in Figure 2.15, the outputs are firstly stored in a memory once the final results are obtained.

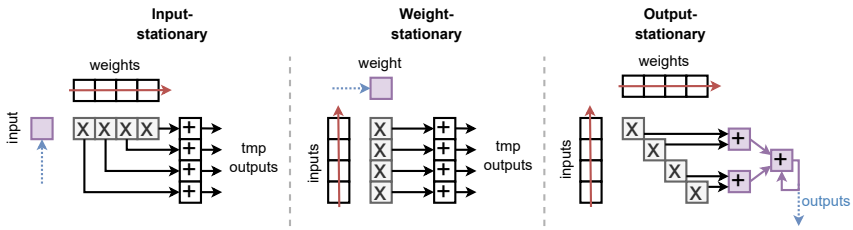


Figure 2.15: Three different data stationary techniques based on [197]. While the red arrows illustrate that new data is loaded in every cycle (parallelism), the blue arrows indicate the data that remains stationary across cycles.

The data parallel and stationary techniques are orthogonal and often combined in accelerators to implement a hybrid form, i.e., multi-dimensional parallelism and single-dimensional stationarity [142, 117, 137]. In our hardware accelerators [J1, C1], we utilize output-stationary and input-parallel (reuse) techniques.

Another emerging technique for removing the memory bottleneck is analog *in-memory computing*, where the MAC operation is executed directly in a memory array. This is done by using either *volatile memories* (SRAMs) or *non-volatile* memories (e.g., flash memories or resistive memory technologies [120]) as programmable resistive elements, called memristors [6]. This approach is the ultimate form of a weight stationary dataflow, as the weights are always held in place [143]. In-memory computing minimizes weight movement, which consequently reduces the energy used for transferring the data. Some of the disadvantages are reduced precision, ADC/DAC overhead, and wire energy dominating for large arrays [107].

2.4.3 Reduced Precision

Neural networks are, to a certain extent, robust to approximations or fault introductions [144]. Such a characteristic can be exploited by performing the

computations at reduced precision, i.e., with fewer bits, while still maintaining sufficient performance. Reducing the precision of the neural network parameters and activations is achieved by applying *quantization*, which maps typically 32-bit floating-point data to their quantized format. If this procedure is done after training, it is referred to as a *post-training quantization*. The performance of quantized neural network model can be further improved when the quantization process is introduced already during training [136, 131], referred to as *quantization-aware training*.

The energy and area of the memory scale approximately linearly with the number of bits [143]. Research has shown that for most applications, 8-bit or even lower fixed-point precision is sufficient and does not impact the performance significantly [238, 116]. Using an 8-bit fixed-point has the following impact on energy and area, as stated in [73, 143]:

- An 8-bit fixed-point addition consumes 3.3× less energy and 3.8× less area than a 32-bit fixed-point addition, and 30× less energy and 116× less area than a 32-bit floating-point addition. The energy and area of a fixed-point addition scale approximately linearly with the number of bits.
- An 8-bit fixed point multiplication consumes 15.5× less energy and 12.4× less area than a 32-bit fixed-point multiplication, and 18.5× less energy (27.5× less area) than a 32-bit floating point multiplication. The energy and area of a fixed-point multiplication scales approximately quadratically with the number of bits.

The benefits of quantization are smaller memory requirements, a reduced number of computations as well as costs for fetching network weights and intermediate results. These are crucial optimizations since memory accesses and data movement dominate energy dissipation, as mentioned previously.

Optimal bit precision and wordlength can vary across layers, weights, and activations within the same DNN [116]. *Dynamic fixed-point representation* enables to rescale, e.g., weights for each layer independently based on the layer's dynamic range, while *variable worldlength* supports computations with different wordlengths.

The most extreme case of quantization is represented with *binary nets*, which are neural networks trained to work with only 1-bit weights (i.e., -1 and 1) [119, 87] and activations [105]. In the first case, the MAC is reduced to additions only, and in the latter to an XNOR. The main disadvantage is a significant degradation in accuracy due to the limited wordlength. *Ternary nets* [114, 148] can be considered as a sparse version of binary nets as they allow weights to be

zero. This can subsequently be exploited to reduce computations and memory storage, thus compensating for the additional bit. Another category where extremely reduced precision is used are *Spiking Neural Networks (SNNs)*. In SNNs, neurons communicate using binary spike events. An example of such a network is TrueNorth chip [79] that utilizes binary activations and ternary weights. Other popular approaches that belong to a non-linear quantization group include logarithmic quantization [115] and weight sharing [85]. In logarithmic quantization, the quantization steps grow exponentially and are smaller in regions closer to the origin. If weights are quantized with base-2, the multiplication can be replaced with a bit shift [115, 257]. Weight sharing reduces the number of weights by forcing multiple weights to share the same value.

2.4.4 Reduced Computation and Model Size

Neural networks are typically highly *sparse* [188], i.e., after training many weights end up being zero or having a negligible value that does not contribute much to the final result. Moreover, after quantization to a reduced precision, numerous of these small, non-zero weights will be also set to zero. Likewise, lots of activations will become sparse during inference. A good example is the ReLU activation function that sets all negative values to zero. In hardware, all this knowledge can be exploited to reduce i) model size by compressing the weights, and ii) operations and memory accesses by skipping MACs with zero inputs, as well as fetches of the corresponding weights. The following sub-sections provide a brief overview of the typical methods for reducing the model size and computational complexity of neural networks, with a focus on *dynamic pruning*.

2.4.4.1 Techniques

The methods for reducing the model size and the number of computations may be roughly categorized into three main groups:

1. *Network Architecture Search* [187] - a procedure that programmatically discovers the best DNN architecture for a given task within a large predefined search space.
2. *Knowledge distillation* [32] - a process of transferring the knowledge from a large DNN (teacher) to a smaller DNN (student), where a student DNN is trained to imitate a teacher DNN.
3. *Network pruning* [209] - a technique that removes/skips unimportant connections and/or neurons in a network either statically or at runtime. It

offers a trade-off between efficiency and performance of the model, i.e., the higher the pruning ratio, the worse the performance.

Since our focus is on event-driven/data-driven architectures, i.e., conditioning computations based on inputs during inference while preserving the parameter space of a DNN, we concentrate on a specific subset of *pruning* that offers such possibilities.

The existing pruning techniques can be grouped based on different criteria [243] such as 1) unstructured and structured pruning (pruning individual elements vs larger blocks of elements) [256], 2) neuron and connection pruning [209], and 3) static and dynamic pruning [243, 253]. Our interest is in the last category, i.e., i) *static pruning*, which targets reduction of the model size, and, especially, ii) *dynamic pruning* that is data-driven. These approaches are further described below.

2.4.4.2 Static Pruning

The objective of *static pruning* is to produce a smaller network with performance comparable to the large, unpruned network. A trained neural network model is compressed by removing weights that contribute very little to the final model performance. This process is performed offline before inference and results in a smaller model size, bandwidth reduction, and faster inference due to the reduced computational complexity. Moreover, retraining may improve the performance of the pruned network, making it achieve comparable performance to the unpruned network. However, this may require significant offline computation time and energy [243]. Many methods produce pruned models that even outperform retraining from scratch with the same sparsity pattern and when holding the number of fine-tuning iterations constant [209]. The lottery ticket hypothesis [188] shows that it is important to preserve weight initialization as in the unpruned model.

Static pruning has been extensively researched throughout the years. Early approaches include magnitude-based pruning [12], and methods using the Hessian matrix of the loss function [14, 16]. Since then a multitude of techniques has been proposed, especially for CNNs. The *deep compression* method [110] identifies the important CNN weights that are subsequently quantized to enforce weight sharing, and encoded with Huffman encoding. A modified version called *soft weight-sharing* [145] performs both quantization and pruning in one (re-)training procedure, achieving comparable compression rates. Similarly, networks in [90] learn important connections that are pruned based on a threshold, and

retrained afterwards. Structured intra-kernel level pruning using strided sparsity is explored in [124]. Other methods that aim for achieving structural sparsity are, e.g., [113] that uses group sparsity constraints on the convolutional filters, [123] that applies both group sparsity and tensor low rank constraints [57] to remove neurons, and [122] that proposes *structured sparsity learning* to regularize different structures such as filters, channels, and layer depth. Structured pruning produces a full matrix with reduced dimensions, as opposed to unstructured pruning that outputs irregular sparse weight matrices that consequently require indices for locating weights stored in a compressed format. Such irregularity patterns cause workload imbalance in the system and reduce throughput. Structured pruning is therefore more suitable for hardware as it is compatible with the data-parallel architectures such as *single instruction, multiple data*. Sparse connectivity patterns are completely avoided in [133] by discarding whole filters together with their connecting feature maps instead of weights. Another example of a compression are *HashedNets* [85] that use a low-cost hash function to randomly group connection weights into hash buckets that share a single value. They are demonstrated on FCNNs but can be applied to CNNs as well.

However, most of the presented pruning methods cannot be directly applied to RNNs due to fundamental differences between RNN and CNN architectures. Moreover, static pruning of RNNs is more challenging as a recurrent unit is shared across all the timestep, which impacts all the steps in the sequence. *Relaxed pruning* is supported by an LSTM accelerator in [191] where a don't-care regions are allowed, i.e., connections can be either kept or discarded. LSTM weights in [177] are grouped and pruned based on their magnitude, leaving at most K non-zero values in a group. Similarly, a top-K technique for LSTMs called *compressed and balanced sparse row* is proposed in [168]. *VectorSparse* [208] partitions a weight matrix into several vectors and prunes each vector to the same sparsity, which results in a better workload balance and higher parallelism compared to the top-K pruned weight matrices. Likewise, [184] proposes a *bank-balanced sparsity* for sparse LSTM networks that divides rows of a weight matrix into banks for parallel computing, where each row has an equal number of non-zero values. *Load-balance-aware pruning* [129] assigns the same sparsity constraints to sub-matrices to achieve similar sparsity ratio for all the processing elements. It obtains the best performance with 90% sparsity in LSTM weight matrices for speech recognition.

Regardless of the type of the network, an assumption behind static pruning is that a compressed model can solve a given task equally well compared to the original, unpruned model. However, this might not always be the case for realistic DNNs that need to have the capacity to capture the necessary relationships in data to be useful in real-life scenarios. For instance, a trained DNN for SE should be able to apply optimal gain values to unseen environments and voices in order to benefit the hearing-impaired users in their daily lives. Static pruning

permanently destroys the original network structure which may lead to a decrease in model capability, representation power, and efficiency [243, 253]. Once the model is pruned and re-trained, the information is irreversibly gone.

Also, the number of weights alone does not always scale proportionally with the energy dissipation. For instance, FC layers in AlexNet [132] dominate the model size compared to the convolutional layers. However, the energy spent on convolutional layers is much higher than on FC layers [103]. Furthermore, research shows that the relative importance of neurons is heavily *input-dependent* [189].

2.4.4.3 Dynamic Pruning

Dynamic pruning is a *data-driven* approach, where a neural network structure is conditioned on the input during inference. It is commonly believed that different neurons represent different features, and thus do not need to be activated for every sample [253]. Therefore, dynamic architectures save computations for canonical ("easy") samples and also preserve their representation power when recognizing non-canonical ("hard") samples [253]. Such behavior provides remarkable advantages in efficiency compared to the acceleration techniques for static models [161, 111, 135], which handle both types of samples with identical computation, and fail to reduce intrinsic computational redundancy [253].

Dynamic pruning determines at runtime which, e.g., layers, channels, or neurons will be dropped from computations. For instance, pruning of activations can be exploited by skipping both memory fetches of the weights and the execution of MACs. Furthermore, the throughput could be increased by skipping the "idle" cycle. The activations can be made even more sparse by pruning low values [157], resulting in additional speedup and power reduction. Last but not least, the sparsity of activations can be exploited for energy and area savings using compression [103].

Moreover, dynamic pruning can also be applied on statically pruned models to further reduce computations and bandwidth requirements [243, 253]. A drawback of dynamic pruning is that it does not reduce the model size. Also, the criteria to determine which elements to prune must be computed at runtime. This introduces overhead in terms of additional computations, bandwidth, and power [243]. However, these downsides are counterbalanced with several other advantages that are absent in static models. These are presented below, as described in [253]:

1. *Efficiency* - computations are allocated on demand at runtime, by selectively activating model components (e.g. layers [160], channels [134] or sub-networks [141]) based on the input. Consequently, less computation is spent on samples that are relatively easy to recognize, or on less informative spatial/temporal locations of an input.

2. *Representation power* - due to data-dependency, these networks have significantly enlarged parameter space and improved representation power.

3. *Adaptiveness* - a desired trade-off between accuracy and efficiency for different computational budgets can be achieved on-the-fly. Therefore, dynamic networks are more adaptable to different hardware platforms and changing environments, compared to static models with a fixed computational cost.

4. *Compatibility* - the efficiency of dynamic networks can be further improved by other acceleration methods such as knowledge distillation [92].

5. *Generality* - many dynamic models can be applied seamlessly to a wide range of applications. For instance, the techniques developed for computer vision tasks are proven to transfer well to language models in natural language processing tasks [155, 213], and vice versa.

6. *Interpretability* - the research on dynamic networks may bridge the gap between the underlying mechanism of deep models and brains, as it is believed that the brains process information in a dynamic way [4, 24]. It is possible to analyze which components of a dynamic model are activated [231] when processing an input sample, and to observe which parts of the input are accountable for certain predictions [229].

A typical example of dynamic networks are hierarchical or staged networks [130], called *cascade* networks. Cascade networks have multiple intermediate classifiers to provide the ability of an *early exit* [243]. Only a few layers of the network are executed at each stage, and additional layers and classifiers are run only if the model does not have sufficiently distinct probabilities to make a decision. Another, finer-grained approach is conditional computing that only activates an optimal part of a network. For instance, in the case of skipping neurons [59], the non-activated neurons are perceived as pruned since they are not involved in computations. Conditional computing can be applied during both training and inference [72]. In [239], video streams are represented as a series of changes across frames and network activations, where computations only for the regions with significant changes are performed. A special category are SNNs [21, 29] that attempt to mimic the operation mechanism of the human brain more accurately than conventional DNNs. SNN neurons communicate by propagating pulses that are generated only if a specific threshold has been crossed, which gives them a potential to be extremely low-power. A notable RNN implementation of dynamic

pruning is presented in [138]. It exploits *temporal sparsity*, where a delta change between two adjacent time steps is computed for both input and hidden state vectors and compared against a threshold. To our best knowledge, such temporal sparsity of activations has been exploited in only two RNN accelerators from the same authors [157, 252], and a few CNN accelerators [239, 150, 211]. Our methods presented in the upcoming chapters build on the top of this [138] work.

2.5 Summary

This chapter provided a condensed overview of the deep learning field, starting with the most fundamental DNN algorithms, training procedure, and methods for building efficient deep learning-based SE and KWS systems relevant for our hearing-instrument application. It then continued by describing the hardware challenges, specifically for low-power devices, imposed by the size and computational complexity of DNNs. Several different optimizations were described to mitigate the challenges. These optimizations are adopted in most of the state-of-the-art hardware accelerators.

As the overview has shown, the field of DNNs is vast, and the focus has shifted from high-performance computers to portable and edge devices. While executing DNNs directly in the resource-constrained edge devices opens up a vast spectrum of options, it also demands innovations on both the hardware and algorithmic level to enable efficient and low-power inference. The next chapter describes the specific contributions of this thesis, which are represented with five publications that together focus on algorithm-hardware co-optimization to make a step closer towards inference in resource-limited hearing instruments while still delivering high-quality results.

Chapter 3

Thesis Contributions

The core of this thesis is composed of a collection of one journal and four conference papers (Chapters 4-8) that have contributed scientifically within the deep learning field. The relationship between these publications is illustrated in Figure 3.1. Each of the publications explores one of the two main areas and is demonstrated on either the *SE* or *KWS* task:

1. *Algorithms development* - developing novel hardware-aware deep learning algorithms that dynamically reduce, i.e., prune, the amount of computations and memory accesses by exploiting the temporal stability in data. Such algorithms may ease the deployment of otherwise computationally and memory-demanding neural networks directly into resource-constrained devices like hearing instruments. Papers [C2] and [C3] belong to this category, where the algorithms in [C2] were successfully patented [O2].

2. *Hardware design and implementation* - building efficient hardware accelerators that are optimized for neural network processing. These accelerators exploit the typical dataflow in DNNs by parallelizing operations and applying data reuse techniques while performing computations with reduced precision. Papers [C4], [J1], and [C1] belong to this category, where the first two support a dynamic pruning method from 1. All of these custom accelerators are more efficient in terms of power, area, and throughput

compared to general purpose processors and, specifically in this context, to a typical Digital Signal Processor (DSP).

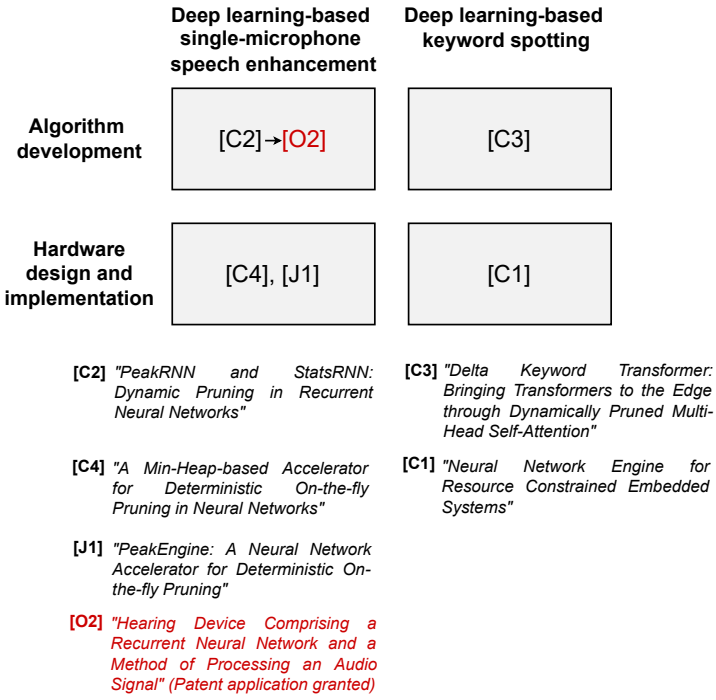


Figure 3.1: A block diagram illustrating the relationship between publications that constitute the core of this thesis. Paper [C2] targets the development of two novel deep learning algorithms for skipping operations. The algorithms were successfully patented [O2]. Papers [C4] and [J1] focus on efficient implementation of hardware accelerators to support the deep learning algorithms. All thee three works are demonstrated on SE. Similarly, paper [C3] targets the development of a deep learning algorithm for skipping operations, while paper [C1] presents an efficient neural network hardware accelerator. Both works are demonstrated on KWS.

3.1 Algorithmic Level

The focus of the first part of the research is on investigating efficient deep learning algorithms for reducing computations and hence the consumed power. The individual contributions of [C2] and [C3] are summarized in the paper excerpts below.

3.1.1 [C2] PeakRNN and StatsRNN: Dynamic Pruning in Recurrent Neural Networks

In this paper, we propose two novel pruning algorithms, *PeakRNN* and *StatsRNN* (later referred to as *PeakGRU* and *StatsGRU*), that save computations and power by dynamically transforming dense matrix-vector multiplications into highly-sparse matrix-vector multiplications in gated-RNNs during inference. We are motivated by i) the challenges of embedded neural networks presented in Section 2.4.1 that hinder the deployment to low-power edge devices, and ii) the potential of dynamic pruning stated in Section 2.4.4.3.

PeakRNN selects a fixed number of top K elements every timestep. Therefore, it offers robustness to the variations of input data (not threshold-based), determinism and bounded execution time for the subsequent neural network operations. *StatsRNN* tries to mimic this behavior by selecting the elements based on a priori and statistically derived thresholds. We evaluate both methods on the SE task (noise reduction) in a simulated hearing instrument setup, where we replace a typical noise reduction module with a DNN. The DNN contains a GRU hidden layer that is substituted with *PeakGRU* and *StatsGRU* layers during the experiments. We show that the two new pruning techniques can reduce the number of MACs and memory accesses by 70% compared to the baseline without compromising the SNR and PESQ objective measures. Moreover, savings of up to 88% can be achieved while maintaining sufficient quality in SNR and PESQ and outperforming the state-of-the-art DeltaGRU [138] method. In general, such savings open up the possibility for a significant reduction of energy dissipation and latency that might enable the execution of DNNs directly in hearing instruments when exploited by a custom hardware accelerator.

We also successfully patented the PeakRNN and StatsRNN algorithms [O2].

3.1.2 [C3] Delta Keyword Transformer: Bringing Transformers to the Edge through Dynamically Pruned Multi-Head Self-Attention

In this paper, we propose a simplified version of the DeltaRNN algorithm [138] and apply it in a TNN to address the challenge of quadratically growing complexity of the MHSA with respect to the input sequence length. Whilst the MHSA is the key component of TNNs, as described previously in Section 2.3.5, at the same time, it can easily become their bottleneck. Therefore, decreasing the computational complexity of TNNs is a step towards enabling their execution on low-power devices.

We first analyze the existing pre-trained Keyword Transformer model developed for the KWS task. We find significant correlation across data within each keyword, which motivates us to apply the delta pruning method to avoid computations and memory fetches and enable data compression. This objective is achieved by transforming dense matrix-vector multiplications into highly-sparse matrix-vector multiplications using a threshold. We demonstrate that the hardware overhead required for the simplified delta approach is negligible since no intermediate states and activations have to be stored for the FC-based attention layers, unlike in RNNs and CNNs. We also provide a breakdown of the maximum possible computational savings for each part of the attention layer where we apply our method. The results show that $\sim$80% and 87-94% of MAC operations in the MHSA can be skipped with no or only a 1-4% decrease in the original accuracy, respectively. These reductions correspond to a theoretical speedup factor of 4.2 $\times$ - 15.7 $\times$. Our proposed method thus helps to considerably decrease the computational complexity and enable significant data compression. Such potential could be exploited in an ultra-low power wake-up word detection front-end that triggers a more powerful detector once a keyword is recognized.

3.2 Hardware Level

The second part of the research targets development of efficient hardware accelerators optimized for neural network processing. An accelerator for small-footprint KWS [C1] is summarized in Section 3.2.1. The contributions of the two accelerators [C4, J1] supporting *PeakRNN* are listed in Sections 3.2.2 and 3.2.3.

3.2.1 [C1] A Neural Network Engine for Resource Constrained Embedded Systems

In this paper, we introduce a *dedicated neural network engine* for power-constrained embedded devices, such as hearing instruments, which is demonstrated on the KWS task. Enabling the KWS functionality directly in hearing instruments would make a hearing-impaired user able to interact with the device via speech commands. Therefore, our motivation is to create a small-footprint accelerator to enable such a feature in the current hearing instrument platform. Moreover, this process could be the second step in the KWS pipeline, triggered by a wake-up word detection mentioned in the algorithmic contribution in Section 3.1.2.

In order to achieve the goal of low power and area, we apply various orthogonal optimizations including data reuse techniques, parallelization of MAC operations, and data quantization. The introduction and motivation for these methods is provided in Sections 2.4.2 and 2.4.3. Furthermore, we develop a novel *dynamic two-step scaling* technique for quantizing the activations during inference without requiring a pre-computed scaling factor or expensive hardware. Moreover, two-step scaling does not increase the total latency and it makes the accelerator execute in a *deterministic* number of clock cycles. We evaluate our dedicated neural network accelerator against a DSP found in Demant's hearing instruments. We find that the accelerator clearly outperforms the DSP and results in significant reduction of power (5×), memory accesses (5.5×), memory requirements (3×), clock cycles (6×), and area (3.7×), while sacrificing only 1% of the original accuracy. Therefore, the accelerator could be used as a co-processor to Demant's DSP and take off the neural processing workload. In general, the accelerator offers a small area and low power, and it is suitable for resource-constrained embedded devices such as hearing instruments.

3.2.2 [C4] A Min-Heap-based Accelerator for Deterministic On-the-fly Pruning in Neural Networks

In this paper, we introduce a *min-heap-based accelerator* developed to support the selection of the top K elements for the previously described *PeakRNN* algorithm in Section 3.1.1. The focus is on efficiently realizing the pruning technique in hardware and hence bridging the algorithmic and hardware levels.

We firstly identify a min-heap data structure as an efficient solution for the top K selection due to its low computational complexity and memory requirements. As a result, we subsequently build a min-heap-based accelerator that uses small standard cell-based memories for storing the top K elements for both input

and hidden state sequences. We evaluate the performance of the accelerator on various number of K and compare the consequent savings of MACs and memory accesses against a dense network performing all the computations. In order to relate the post-synthesis results to our previous algorithmic study [C2], we use the same network and data for evaluation. The results show that the overhead of the accelerator to support the selection of 35 to 111 K elements (maximum acceptable degradation and no degradation of improvement in objective measures, respectively) out of N=512 is negligible compared to the estimated 78-93% savings of dynamic energy. The min-heap accelerator only dissipates $\sim$0.5% of the total energy and requires 4.2 - 33 kgates (without memories) and 14.8 - 37.2 kgates in total. We therefore demonstrate that the impact of our accelerator on the overall complexity is insignificant. Furthermore, the accelerator design is not limited to GRU layers or neural networks only. Its application is versatile and can be used in many other different areas.

3.2.3 [J1] PeakEngine: A Deterministic On-the-fly Pruning Neural Network Accelerator for Hearing Instruments

In this paper, we present the first RNN ASIC accelerator, called *PeakEngine*, for deterministic on-the-fly pruning targeting battery-powered wearable devices such as hearing instruments. We are motivated by the fact that the existing RNN accelerators for low-power devices i) mostly focus on small, statically pruned models with reduced representational power, and ii) do not consider the relative importance of neurons that is heavily input-dependent. We therefore build on the knowledge gained from our two previous works on the *PeakRNN* pruning algorithm [C2] and the *Min-heap engine* [C4] to design an optimized, energy-efficient custom hardware accelerator that supports inference of both dense and dynamically pruned GRU layers.

We build a configurable accelerator that can be used as a co-processor for typical digital signal processors found in hearing instruments. We perform a thorough evaluation of *PeakEngine* for different K values in terms of energy and latency. The experiments show that *PeakEngine* dissipates only 4.14-5.04 μJ per inference when running the dynamically pruned layers compared to 11.83 μJ for the baseline (unpruned) model that executes all computations. This yields energy savings of 2.35-2.86$\times$ with no to maximum acceptable degradation in improvement in audio quality and intelligibility for SE. The reduction of computational complexity also speeds up the inference 2.2-2.97$\times$. All of these results make the execution of even bigger RNNs feasible in a hearing instrument within the imposed time and energy budget. The accelerator is synthesized in a 22 nm CMOS process and occupies 0.053 mm^2 without the big weight memory and 2.95 mm^2 in total. Moreover, we also build a *software framework* for parameter space exploration of different

Q formats, wordlengths, and K values for GRU-based and FC DNNs executed on *PeakEngine*. The framework consists of identical modules as *PeakEngine* to mimic it *bit-accurately* during inference and hence also verify accelerator outputs. With this work, we complete our algorithm-hardware co-design effort and demonstrate the importance of developing deep learning algorithms and hardware platforms hand-in-hand.

Chapter 4

A Neural Network Engine for Resource Constrained Embedded Systems

By Zuzana Jelčicová, Adrian Mardari, Oskar Andersson, Evangelia Kasapaki, and Jens Sparsø [C1]

Abstract

This paper introduces a *dedicated neural network engine* developed for resource constrained embedded devices such as hearing aids. It implements a novel *dynamic two-step scaling* technique for quantizing the activations in order to minimize word size and thereby memory traffic. This technique requires neither computing a scaling factor during training nor expensive hardware for on-the-fly quantization. Memory traffic is further reduced by using a 12-element vectorized multiply-accumulate datapath that supports data-reuse. Using a keyword spotting neural network as benchmark, performance of the neural network engine is compared with an implementation on a typical audio digital signal processor used by Demant in some of its hearing instruments. In general, the neural network engine offers small area as well as low power. It outperforms the digital signal processor and results in significant reduction of, among others, power (5×), memory accesses (5.5×), and memory requirements (3×).

Furthermore, the *two-step scaling* ensures that the engine always executes in a deterministic number of clock cycles for a given neural network.

4.1 Introduction

Deep neural networks (DNNs) are ubiquitous, finding their application in various areas such as image and video processing [96], robotics [100], medicine [126], games [121] as well as audiology [60]. They are typically executed in the cloud due to their computational complexity and size. The results are then deployed wirelessly to power-constrained edge devices such as hearing instruments. However, sharing data with the cloud is not desirable due to issues such as security, privacy, latency, and connectivity [197]. On the other hand, embedding NNs directly in always-on devices that are extremely limited in area, memory, power budget, and throughput, is a challenging task.

To minimize power consumption, hearing instruments are typically implemented using heterogeneous platforms that include specialized accelerators and DSPs. These DSPs often contain vector datapaths that support 16 or 24-bit fixed-point vector elements matching the accuracy of the audio samples. [56, 38, 203].

Vector operations are also attractive for NNs, especially for performing multiply-accumulate (MAC) operations. Many works have demonstrated that a wordlength of 8 bits is sufficient for inference, having no or insignificant impact on accuracy [52, 151, 162]. The benefits are again reduced computational complexity, reduced memory requirements, and – if a vectorized datapath is used – processing of more vector elements per instruction. On the other hand, it increases the risk of overflows when the final MAC product in a wide accumulator needs to be stored back to memory in a reduced format.

A common approach to handle this issue is *quantization*. One of the most widely used quantization techniques is a *static-precision quantization*, where the scale factor is determined for the entire NN [84, 86, 89]. Opposed to it, a *dynamic-precision quantization* firstly proposed by [118] enables varying multi-precision fixed-point for every layer.

Although DSPs support MAC operations, they are generally not able to exploit input sharing, a fundamental *data reuse optimization* for NNs. This can be solved by moving away from DSPs and developing customized hardware accelerators specifically for NNs. Such accelerators exploit characteristic dataflows found in NNs to enhance parallelism and reduce data movement [197].

This paper targets the issues discussed above and introduces the following contributions:

1. *A dedicated NN engine (NNE)* used as a co-processor to Demant's DSP (*xDSP*). The NNE is optimized by applying a set of mutually dependent techniques with a novel *dynamic two-step scaling*.

2. *A dynamic two-step scaling* mechanism capable of fitting MAC products into the required wordlength at runtime without analysing the data ranges during training and adding expensive hardware for quantization on-the-fly. This method ensures that the NNE always executes in a deterministic number of clock cycles for any arbitrary NN.

The NNE and two-step scaling technique further incorporate the following methods: i) *wordlength reduction* from 24 to 8 bits for all NN parameters ii) *12 optimized MACs in parallel*, and iii) *input* and *output stationary* techniques [143] to significantly reduce memory accesses. A keyword spotting (KWS) NN [147] with a Google Speech Command Dataset (GSCD) [178] is used as benchmark and implemented on both the NNE and the xDSP. The NNE outperforms the xDSP in all aspects (significantly smaller power consumption, memory requirements, and number of memory accesses) thanks to the set of the proposed techniques.

The rest of the paper is structured as follows: Section 4.2 presents related work. Section 4.3 provides background on the xDSP and the KWS NN. Section 4.4 describes the proposed optimization techniques along with the NNE design. Section 4.5 discusses the results, and finally Section 4.6 concludes the paper.

4.2 Related Work

Lower precision introduces higher risk of overflows when the activations need to be stored back to memory in a reduced wordlength. This issue is usually solved by dynamically quantizing activations during inference, or defining a quantization factor during training that is fixed once the model is deployed. Since weights and biases are fixed once training is completed, quantizing them can be done statically.

Authors in [154] propose an overflow management scheme which accumulates partial INT32 results (INT16 inputs) into FP32, along with trading off input precision with length of accumulate chain to gain performance.

In [162], MAC products in INT32 accumulator are firstly downscaled using a multiplier, then cast down to uint8 (with saturation), and finally run through an activation function to produce the final 8-bit output.

The authors of [89] convert the final accumulated value by either clipping it to the predefined limits set by integer and fractional length (wordlength) or rounding to fractional length bits using a specific rounding mode. The wordlength for the fixed-point representation is set to 16 bits.

Using the static approaches in the works above might result in encountering values outside the observed ranges at runtime that must be clipped, likely causing additional loss of accuracy.

A dynamic, multi-precision per-layer data quantization flow is introduced in [118] and adopted in [147]. The weight quantization phase analyses the dynamic ranges of weights in each layer to find the optimal fractional length per layer. Fractional length is then initialized to avoid data overflow. The intermediate data of the fixed-point CNN model and the floating-point CNN model are compared layer by layer using a greedy algorithm to reduce the accuracy loss.

All the above techniques require very deep and detailed analysis of the used NN during training or additional computation resources that are usually not an option for resource-limited devices such as hearing instruments.

Our *dynamic two-step scaling* method that handles overflows on-the-fly does not require to compute a scaling-factor in advance, and it does not introduce any instruction overhead. Furthermore, it works on a per-layer basis, the HW implementation is cheap (implemented as an arithmetic shift operation), and it makes the NNE execute in a deterministic number of cycles.

4.3 Background

4.3.1 Keyword Spotting (KWS) Neural Network

KWS systems in edge devices have limited power budget since they must be always-on and operate real-time, while still delivering high accuracy [147]. These requirements are even more strict for extremely resource-constrained devices like hearing instruments. The pretrained KWS model [147] used as a benchmark in this paper is a fully-connected feed-forward NN with a 250x144x144x144x12 configuration trained for 32-bit floating-point (FP32). The input to the network

is a flattened feature matrix where a one-second audio recording is divided into $40ms$ frames with a stride of $40ms$, producing 25 frames analysed in 10 frequency bins (250 inputs). Each hidden layer consists of 144 neurons. The output layer has 12 neurons, each representing one category. The first two neurons correspond to *"silence"* (no speech present in the recording) and *"unknown"* (NN is unable to classify the word). The remaining ten neurons represent the following keywords: *"yes"*, *"no"*, *"up"*, *"down"*, *"left"*, *"right"*, *"on"*, *"off"*, *"stop"*, *"go"*. The dataset [178] has more than 65,000 one-second recordings of 30 short words. The final test set consists of 4,890 audio files.

Both inputs and all parameters (weights and biases) were statically quantized to an 8-bit integer representation without any re-training (post-training quantization) before running the inference. Two quantization modes were tested, namely *asymmetric* and *symmetric*, where the latter one proved to work better for our use case. This technique resulted in a small accuracy loss when going from FP32 to 8-bits. Further fine-tuning could improve the accuracy even more. The activations are quantized dynamically to an 8-bit fixed-point using a *dynamic two-step scaling* technique.

4.3.2 Digital Signal Processor (xDSP)

The xDSP is a processor optimized for DSP applications with multiple datapaths, register files, and custom functional units. It has a 96-bit vector datapath that supports 4× 24-bit fixed-point elements in a Q5.19 format. This format is used for all NN parameters (weights, biases) and activations. The MAC unit can multiply four elements at a time and store the intermediate results in a single accumulator. This design has limitations since neurons can only be calculated sequentially.

Weights and inputs/activations are fetched from memory as vectors while biases as 24-bit scalars since only one neuron is processed at a time. The 24-bit datapath is also important for overflow management. Scaling out-of-range values involves a global decision across a layer of neurons to keep the same ratio among the outputs. Scaling and storing outputs of a layer individually requires fetching the previously computed outputs whenever a new largest overflow occurs. If, e.g. the first result in a layer needs one shift, but the second one requires two shifts to fit into 8 bits, the first neuron has to be re-fetched from memory and shifted by the missing number of positions.

4.4 The NNE Accelerator

Significant improvements can be gained by using the dedicated NNE instead of DSPs, to improve resource utilization during inference by exploiting a set of the proposed optimization techniques.

4.4.1 Reduced Wordlength

The wordlength and representation of the NN model parameters has a big impact on the inference performance. Reduced parameter wordlength enables reading more elements through the same memory interface in a single cycle, and lower precision multipliers require less silicon area and power.

The original accuracy using FP32 on the KWS task is 81.77%. Using 24, 12, 8, and 6-bit representation for all NN parameters results in 81.24%, 80.36%, 80.28%, and 76.13% accuracy, respectively. The FP32 and 24-bit xDSP implementations achieve comparable accuracy, and a negligible drop is also observed using 12 and 8 bits. Further reduction to 6 bits results in a significant drop, and 8 bits are therefore selected as a convenient trade-off between wordlength and accuracy, decreasing memory requirements up to 3×.

4.4.2 Parallel MACs

The MAC unit in the xDSP can only process one neuron at a time. Considering a 96-bit memory interface and an 8-bit wordlength, the MAC is designed such that it can compute 12 intermediate results in parallel (see Figure 4.1). The intermediate values are accumulated in accumulators with larger precision to avoid overflows and loss of precision. Additionally, a feature for preloading the biases in the accumulators is included. This step saves one addition for each neuron, and serves as reset for the accumulators. The theoretical minimum number of MAC operations required for the inference of an arbitrary network using our NNE is given by:

$$MAC_{op} = \left(\sum_{i=1}^{N} A_i \times O_i \right) / V, \tag{4.1}$$

where MAC_{op} is the number of vectorized MAC operations, A is the number of activations/inputs to a given layer, O is the number of outputs in the layer, N is the number of layers (excluding input layer), and V is the number of elements

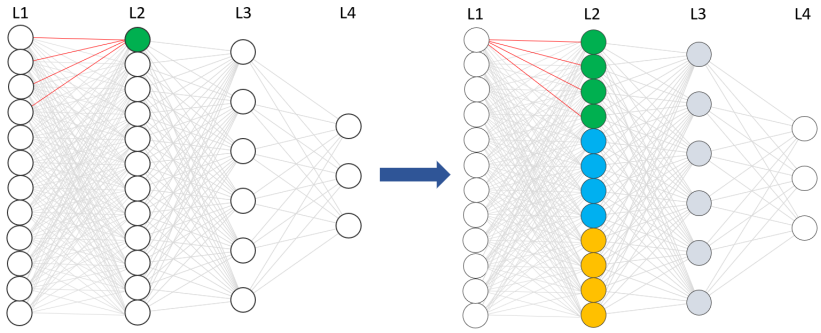


Figure 4.1: xDSP (left) vs NNE (right) MAC unit. The xDSP is able to process one neuron at a time using four 24-bit inputs. The NNE processes vectors of twelve 8-bit neurons in parallel (vectors of four are shown for simplicity). The two-step scaling firstly identifies scale factors for vectors in L2, and then performs additional shifts when these vectors are loaded as inputs for L3.

in a vector. The minimum number of MAC operations required for the KWS application is therefore 6600 per inference in the NNE.

4.4.3 Data Reuse Techniques

Input stationary and *output stationary* dataflows [143] are used in our design to minimize the data movement between the NNE and memory. An *Input stationary* dataflow reduces the power by multiplying the input with weights of 12 different neurons in a layer, and an *output stationary* dataflow does so by accumulating 12 intermediate results. The final results are transferred to memory as a 12× 8-bit vector. By combining all the optimization techniques introduced previously, memory accesses are reduced by at least 5.5× as shown in Table 4.1.

In the xDSP, only four elements are fetched in a vector. We thus need 63 vectors to represent 250 inputs, and 36 vectors to represent 144 inputs. All these vectors are loaded for each neuron in a corresponding layer since parallel processing is not possible in the xDSP.

In contrast, the NNE efficiently handles 12 elements at a time (both inputs and outputs), resulting in 21 vectors for the layer with 250 neurons, and 12 input vectors for the layers with 144 neurons. These are then multiplied with vectors

Table 4.1: Number of input and weight vectors to load in the xDSP vs NNE.

		xDSP	**NNE**
Layer 1 (144)	**Inputs**	63 x 144	21 x 12
	Weights	63 x 144	21 x 144
Layer 2 (144)	**Inputs**	36 x 144	12 x 12
	Weights	36 x 144	12 x 144
Layer 3 (144)	**Inputs**	36 x 144	12 x 12
	Weights	36 x 144	12 x 144
Layer 4 (12)	**Inputs**	36 x 12	12 x 1
	Weights	36 x 12	12 x 12
Total		**39,744**	**7,176**

of 12 neurons instead of a single one. This gives further reduction from 144 to 12 vectors, and 12 to one vector. The number of memory reads (inputs and weights) is, thus, considerably reduced from 39,744 to only 7,176. The number of loads for biases is negligible, and is therefore omitted from the calculations. Moreover, the total number of memory accesses might grow even more for the xDSP, depending on how often the overflows occur.

4.4.4 Two-Step Scaling

As mentioned in section 4.3.2 and 4.4.3, additional memory accesses are necessary in the xDSP to reload and scale already computed outputs in a layer if an overflow occurs. This increases the power consumption, and makes the cycle count non-deterministic which may compromise real-time processing. These issues are handled in the NNE by introducing a *dynamic two-step scaling* method. It is divided in the following parts:

1. **Within a vector (when writing results to the memory)** - this step handles overflows when writing accumulator values back to memory. The MAC products that are stored in accumulator registers are scaled down. This is done in vectors of 12 neurons (see Figure 4.1) that are computed simultaneously.

   Once the computations per vector are finished, the biggest positive number among the 12 results is found. Negative numbers are excluded from the calculations since the ReLU activation function zeros them out afterwards.

Table 4.2: Two-step scaling, part 1 - finding the biggest scale factor per vector of 12 neurons as well as per layer (red number).

Group	Number of shifts per vector	Additional shifts
Green	2	?*
Blue	1	?*
Orange	**3**	?*

* Additional shifts are not known at this point

Table 4.3: Two-step scaling, part 2 - calculating the missing number of shifts for each input vector.

Group	Number of shifts per vector	Additional shifts
Green	2	**3** - 2 = **1**
Blue	1	**3** - 1 = **2**
Orange	**3**	**3** - 3 = **0**

The biggest positive value determines the scale factor per vector, i.e. the number of shifts necessary to fit the result into 8 bits. This is shown in Table 4.2. The scaled vector elements are stored in memory and the corresponding per-vector scaling factors (green 2, blue 1, orange 3) in a special register in the register file.

When all the vectors of neurons have been calculated (the layer has been completed), the biggest number of shifts among all the vectors is determined (3 in this example).

2. **Across layer (when reading results from the memory in a subsequent layer)** - this step maintains the ratio across the entire layer. In the previous step, a scaling factor for each vector of 12 neurons and the biggest scaling factor across a layer were found. When computing a new layer L3 as shown in Figure 4.1, the previously scaled outputs are retrieved as inputs.

   The biggest number of shifts determined in the previous step is used to specify missing shifts for each vector in order to scale the inputs correctly. Therefore, the biggest number of shifts per vector needs to be subtracted from the biggest number of shifts in the previous layer (see Table 4.3). The additional scaling is performed when the vector is read from memory in the next layer.

Using this *dynamic two-step scaling* never requires to retrieve and process already computed results to perform additional scaling as is the case when using the xDSP. Moreover, the number of memory accesses to execute is always calculable using the *dynamic two-step scaling*. It also makes the NNE always execute in a deterministic number of cycles without adding any additional overhead cycles related to activation scaling. Furthermore, quantizing the fixed-point by powers of two is considerably cheaper than other presented approaches, since it only requires arithmetic shift operations. Finally, the following equation yields the total number of cycles needed for inference of an arbitrary network:

$$2N + \sum_{i=1}^{N} \left\lceil \frac{O_i}{12} \right\rceil \times \left(3 + 13 \left\lceil \frac{A_i}{12} \right\rceil \right) + \left\lceil \frac{O_i}{12} \right\rceil , \tag{4.2}$$

where N is the number of layers excluding the input layer; O is the number of output neurons in the current layer; and A is the number of inputs/activations to the layer.

4.4.5 The NNE Design

Figure 4.2 illustrates the NNE datapath which implements the optimizations explained above. The control path that consists of address generation modules (reading/writing data from/to memory), a configuration module (registers storing parameters such as the number of layers, starting read/write address etc.), and a finite state machine (handling the entire NNE flow) is not shown for clarity.

The NNE `MEMORY` consists of seven memory instances. Smaller memories are preferred over a single, big memory to decrease dynamic power for a read operation that dominates the inference. The increased leakage is solved by switching on/off the individual memory blocks when necessary. The memory instances (see Figure 4.2) are split into three address blocks: `block_0`, `block_1`, and `read_weights`, where the first two blocks switch the *read inputs/write results* role after every layer. The memory block used for reading the inputs in one layer will become a block for storing the results in the next layer and vice versa. The weight memory block (`read_weights`) does not change, and the weight address is incremented after each retrieved weight vector. Biases are stored in the same memory block. The first address in the `read_weights` block contains biases for the vector of neurons to be processed. The following addresses represent their weights. Summing all the required vectors together for the KWS task results in 6,694 96-bit vectors (∼78.45kB) that can be represented with a 13-bit address range (`addr`).

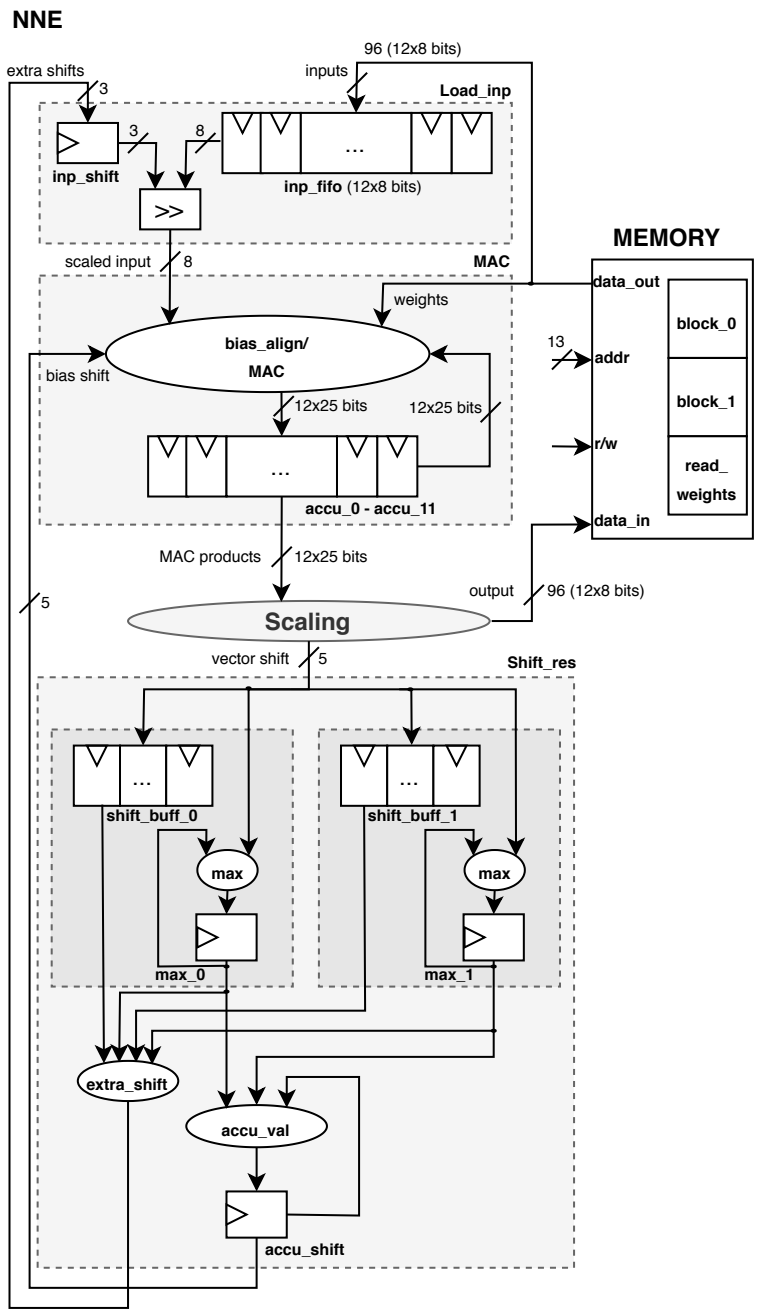


Figure 4.2: A high-level overview of the NNE datapath that implements the four optimization techniques.

The NNE performs the computations on a per-layer basis. When the NN inference begins, the bias values are preloaded in the accumulator registers (`bias_align/MAC` and `accu_0-accu_11` in the `MAC` unit). This step saves one addition and also resets the accumulators.

The first 96-bit vector of 12 inputs is then loaded into the `inp_fifo` in the `Load_inp` unit, along with a 3-bit value (`inp_shift`) for additional realignment (two-step scaling, 2nd part).

The next step are 12 parallel MAC operations (`bias_align/MAC` in the `MAC` unit). A 96-bit input vector consisting of 12 weights for 12 different output neurons is loaded. Intermediate results are stored in twelve 25-bit accumulator registers to avoid overflows and any loss of precision.

When all input vectors have been loaded, and all MACs have been performed for the current vector of 12 neurons, a local scaling factor is found (two-step scaling, 1st part). The `Scaling` logic applies ReLU on 12 accumulated values, and outputs both twelve 8-bit results as a 96-bit vector, and a 5-bit output representing a scaling factor per vector.

The scaling factor is found among positive inputs by counting leading zeros. Once it is determined, the number of shifts for each input is obtained by subtracting the computed minimum leading zero count and final result word length (8 bits) from the accumulator word length (25 bits). If the incoming 25-bit input is negative, the 8-bit result is directly set to zero (ReLU). Else the input is shifted by the calculated value. The 96-bit result will be stored in the memory in the next clock cycle. The 5-bit shift value is mapped as an input for the `shift_buff`s in the `Shift_res` unit.

The `shift_buff` keeps track of the arithmetic shift count performed for each vector of activations, such that each vector is appropriately aligned when loaded into the `inp_fifo`. Two `shift_buff`s are necessary (`shift_buff_0` and `shift_buff_1`). One `shift_buff` is used for storing 5-bit unsigned shift values for the vectors in the layer currently being computed. The second `shift_buff` contains 5-bit unsigned shift values from the previous layer that will be used to calculate the scaling factors for the current inputs (two-step scaling, 2nd part, `extra_shift`). The `Shift_res` unit decides which `shift_buff` is for reading and writing the shift values. The roles of the `shift_buff`s are swapped at the end of every layer. The `accu_val` logic populates the `accu_shift` register with the accumulated number of shifts from previous layers in order to correctly align biases in the accumulator.

Table 4.4: 24-bit DSP vs 8-bit NNE. Both designs were synthesized with 28nm CMOS technology at 0.7V and 2MHz.

	xDSP (24-bit)	**NNE (8-bit)**
Memory accesses (SIMD4, 96 bits)	39,744	7,250
Memory accesses (scalar, 24 bits)	888*	-
Bits transferred (kB)	468.35	84.97
Memory occupied (kB)	235.37	78.45
Power (μW)	43.3	9
Clock cycles	~45,000	7,332
Area (mm^2)	0.71	0.19
Accuracy (%)	81.24	80.28

* This number can grow depending on how often overflows occur

4.5 Results and Discussion

The performance of the NNE and the xDSP is compared in terms of memory traffic, memory capacity, power, area, and accuracy. Similar comparisons with other works are provided as well.

Both the xDSP and the NNE design were synthesized with 28nm CMOS technology. Power simulations for the KWS application were run at 0.7V and 2MHz. The results per inference, i.e. processing of a one-second audio file, are shown in Table 4.4. Accuracy is given as an average over 4,890 audio recordings.

The total memory capacity required for storing all parameters along with results and inputs is 235.37kB and 78.45kB for the xDSP and NNE, respectively. The decrease in memory capacity requirements of more than 3× for the NNE is mostly thanks to the reduced wordlength from 24 to 8-bit for all parameters. The second contributing factor is an optimized approach of reusing the memory space as described in Section 4.4.5. The xDSP memory has limited capacity. Therefore, a KWS NN with 4× fewer neurons in each layer (63x36x36x36x3) was executed instead, and the memory leakage was scaled as well as the execution time such that it corresponds to the processing time of one frame in order to match the NNE.

The full xDSP implementation of the KWS application requires ~5.5x more vector memory fetches than the NNE. Moreover, it also needs a baseline of 888 scalar memory accesses to retrieve biases (444) and store the results (444), while the NNE only works with vector operations. The number of scalar memory fetches can grow depending on how often overflows occur. Whenever an overflow is encountered, previously computed results in the same layer must be reloaded

Table 4.5: Comparison of the NNE with other works.

	ISSCC 2017[125]	VLSI 2018[180]	JSSC 2019[216]	ESSCIRC 2018[158]	ISSCC 2020[226]	This work*
Tech (nm)	40	28	65	65	28	28
Algorithm	DNN	CNN	LSTM	LSTM	DSCNN	DNN
Voltage (V)	0.65	0.57	0.6	0.575	0.41	0.7
Memory (kB)	270	52	65	32	2	79
Area (mm^2)	7.1	1.29	2.56	1.04	0.23	0.19
Freq (MHz)	3.9	2.5	0.250	0.250	0.040	2.0
Latency (ms)	7	0.5-25	16	16	64	40
Keywords	10	11	10	4	1∼2	10
Power (μW)	288	141	10.6	5	0.51	9
Dataset	NA	TIDIGITS	GSCD	TIMIT	GSCD	GSCD
Accuracy (%)	NA	96.11	90.87	91.8	98@1 word 94.6@2 words	80.28

* Synthesis results.

from the memory. All these additional accesses significantly contribute to the total of ∼45k clock cycles for the xDSP, while the NNE reduces the clock cycles down to 7,332 (6×), which is very close to the theoretical MAC-based minimum.

Reducing wordlength and the number of memory operations had therefore a significant impact on memory traffic. Only ∼85kB are transferred in the NNE instead of original ∼469kB in the xDSP, resulting in reduction of 5.5×. This assumes the best-case scenario for the xDSP, i.e. excluding additional operations due to overflows.

Average power consumption during inference for the xDSP and the NNE is 43.3μW and 9μW, respectively, making the accelerator ∼5× more power efficient. As expected, most of the NNE power is used on the memory accesses that dominate the inference. The NN memory is accessed in 98.88% of the clock cycles with 7,213 vector load operations and 37 vector store operations. Memory operations consume almost 91% of the total power, while for the xDSP it is approximately 80%. Moreover, the NNE area (0.19mm^2) is 3.7× smaller than the xDSP area (0.71mm^2).

All the above improvements compensate for a negligible 1% loss of accuracy from 81.24% (xDSP) to 80.28% (NNE).

Table 4.5 is included for completeness and shows comparisons with prior works. As observed in the table, comparing all the works on equal terms is a difficult task since each varies from the rest in many different perspectives. The NNE

accuracy is lowest from all the referred works. However, this is mainly due to the selected network topology, which has an accuracy of 81.77% in FP32 precision. The 8-bit implementation results in an almost negligible drop of 1.5% unit using post-training quantization. Retraining the network could bring the accuracy closer to the FP32 result. However, a different algorithm or topology would be required for a more significant improvement. Therefore, the main takeaway is that the NNE offers small area and low power consumption with decent accuracy for a given NN. It outperforms the xDSP significantly and can efficiently execute NN inference in low-power embedded devices such as hearing instruments.

4.6 Conclusion

This paper presented a dedicated NNE for hearing instruments that implements a cheap, novel dynamic two-step scaling technique for fitting the extended MAC products back to memory in a reduced format. It is implemented as a shift operation that scales the fixed-point by powers of two within i) vectors of neurons, and ii) across all neurons in a layer. The two-step scaling makes the NNE always execute in a deterministic number of cycles. This number of cycles is close to the number of vectorized parameters that need to be retrieved from memory. The NNE also implements other complementary methods to further improve its performance. These are: reducing wordlength from 24 to 8 bits, executing 12 MAC units in parallel, and exploiting input and output stationary techniques. The combination of all of these approaches makes the NNE outperform a typical audio DSP in all aspects. The two implementations were tested using a benchmark KWS NN. In comparison to the xDSP, the NNE reduced power 5×, memory accesses 5.5×, clock cycles 6×, memory requirements 3×, and area 3.7×, while the accuracy dropped only by less than 1%. In general, the NNE offers small area and low power, and it can be easily used in resource constrained embedded devices such as hearing instruments.

Chapter 5

PeakRNN and StatsRNN: Dynamic Pruning in Recurrent Neural Networks

By Zuzana Jelčicová, Rasmus Jones, David Thorn Blix, Marian Verhelst, and Jens Sparsø [C2]

Abstract

This paper introduces two dynamic real-time pruning techniques *PeakRNN* and *StatsRNN* for reducing costly multiplications and memory accesses in recurrent neural networks. The methods are demonstrated on a gated recurrent unit in a multi-layer network, solving a single-channel speech enhancement task with a wide variety of real-world acoustic environments and speakers. The performance is compared against the baseline gated recurrent unit and the DeltaRNN method. Compared to the unprocessed speech, the SNR and Perceptual Evaluation of Speech Quality were on average improved by 8.11 dB and 0.43 MOS-LQO, respectively. Additionally, the two proposed methods outperformed DeltaRNN by 0.7 dB and 0.11 MOS-LQO in the two objective measures, while using the same computational budget per timestep and reducing the original operations by 88%. Furthermore, *PeakRNN* is fully deterministic, i.e. it is always known in advance how many computations will

be executed. Such worst-case guarantees are crucial for real-time acoustics applications.

Index Terms— RNN, determinism, statistics, peaks, threshold, single-channel speech enhancement, hearing instruments

5.1 Introduction

Speech enhancement (SE) is a classical problem in signal processing that focuses on attenuating background noise from a speech signal. Traditionally, statistical methods such as the Wiener filter [64], non-Negative Matrix Factorization [82], and Short-Time Spectral Amplitude [10] have been used to solve the single-channel SE task. Applications such as hearing instruments (HIs), (wireless) headsets, and mobile communications require algorithms that are able to handle a wide range of noise environments and speakers to be useful in real-life situations. However, such signals do not follow normal distributions and are often non-stationary [166], i.e. the statistical structure of the signal changes over time, which imposes a challenge for the traditional methods [53]. Deep neural networks (DNNs) are able to capture non-linear and complex relationships, and have proved successful in SE applications [82, 99], outperforming classical signal processing methods.

SE is a challenging problem that is usually solved by complex DNN models using several hidden layers consisting of hundreds to thousands of neurons, resulting in a model with millions of parameters. Often such models can, however, be pruned, leading to computational savings that are crucial for low-power edge devices such as HIs. Static pruning [214] results in a smaller dense model where, however, the capabilities of the pruned neurons and weights are irreversibly gone. Moreover, static pruning cannot capture the importance of neurons and weights that are highly input-dependent. Dynamic approaches [134], on the other hand, enable to use parts of the NN relevant for the current input. Yet these methods are often complex, and the deployment hence still remains challenging.

In this work, we propose two dynamic pruning techniques, called *PeakRNN* and *StatsRNN*, which during inference reduce the number of memory accesses (MAs) and multiply-accumulates (MACs) dynamically in a data-driven way. The reduction is demonstrated on a SE task using a gated recurrent unit (GRU) hidden layer in a three-layer network. The evaluations are based on the computational costs and objective measures such as SNR and Perceptual Evaluation of Speech Quality (PESQ). While *PeakRNN* offers determinism and robustness without any

prior data analysis, *StatsRNN* approaches pruning by exploring the underlying statistical properties of the data. These two pruning techniques can be used to find an optimal model and they outperform the current state-of-the-art DeltaRNN [138] technique.

5.2 Related Work

Recurrent neural networks (RNNs) and their variants, such as long short-term memory (LSTM) units [20] and GRU [71], are suited for time-series tasks since they are able to model complex temporal structures. GRUs are computationally more efficient and thus preferred over LSTMs in real-time SE tasks [175]. However, they still require many matrix-vector multiplications and memory accesses. Works targeting single-channel SE primarily focus on a high-level exploration, i.e. comparing different objective measures for speech quality and intelligibility without considering the complexity of the proposed algorithms.

In [164] the authors introduce FastGRNNs that use a scalar weighted residual connection for each coordinate of the hidden state h. They have lower training times and prediction costs, as well as 2-4x fewer parameters than LSTMs and GRUs, while matching the state-of-the-art prediction accuracies. However, this static method does not consider temporal dependencies in data. DeltaRNN [138] addresses the computational issues by exploiting the temporal stability of inputs and activations, i.e. by caching neuron activations, operations can be skipped where no significant changes occur from the previous update. This data-driven approach saves fetches of entire columns of weight matrices, leading to substantial speedups of 5.7-100x for a RNN on classification tasks with negligible accuracy loss.

Our work builds on and outperforms the idea of DeltaRNNs with novel PeakRNN and StatsRNN techniques. The algorithms are demonstrated on the single-channel SE regression task.

5.3 Peak RNN Algorithm

Similar to DeltaRNN, the objective of PeakRNN is to transform a dense matrix-vector multiplication into a highly-sparse matrix-vector multiplication to save both MAC operations and, above all, MAs. These two methods therefore share the underlying computations targeting GRU, equations (5.1)-(5.12), that are

detailed in [138]. The actual difference arises in selecting the elements for computations (5.1)-(5.4). DeltaRNN applies a single threshold θ on both the input vector x and the activation vector h, resulting in a variable number of required computations. In contrast, PeakRNN selects a desired number of *peak* elements N_p from both vectors individually in every timestep. Hence, unlike DeltaRNN where the number of computations is non-deterministic, PeakRNN saves computations in a *deterministic manner* as it is always known in advance how many operations will be executed. Therefore, it provides *worst-case execution guarantees*, which is a crucial aspect in real-time low-power devices, and not ensured by DeltaRNN.

$$\hat{x}(t) = \begin{cases} x(t) & \text{if } |x(t) - \hat{x}(t-1)| \text{ among } N_p \\ \hat{x}(t-1) & \text{otherwise} \end{cases} \tag{5.1}$$

$$\hat{h}(t-1) = \begin{cases} h(t-1) & \text{if } |h(t-1) - \hat{h}(t-2)| \text{ among } N_p \\ \hat{h}(t-2) & \text{otherwise} \end{cases} \tag{5.2}$$

$$\Delta x(t) = \begin{cases} x(t) - \hat{x}(t-1) & \text{if } |x(t) - \hat{x}(t-1)| \text{ among } N_p \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

$$\Delta h(t-1) = \begin{cases} h(t-1) - \hat{h}(t-2) & \text{if } |h(t-1) - \hat{h}(t-2)| \text{ among } N_p \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

$$M_r(t) = W_{xr}\Delta x(t) + W_{hr}\Delta h(t-1) + M_r(t-1) \tag{5.5}$$

$$M_u(t) = W_{xu}\Delta x(t) + W_{hu}\Delta h(t-1) + M_u(t-1) \tag{5.6}$$

$$M_{xc}(t) = W_{xc}\Delta x(t) + M_{xc}(t-1) \tag{5.7}$$

$$M_{hc}(t) = W_{hc}\Delta h(t-1) + M_{hc}(t-1) \tag{5.8}$$

$$r(t) = \sigma[M_r(t)] \tag{5.9}$$

$$u(t) = \sigma[M_u(t)] \tag{5.10}$$

$$c(t) = tanh[M_{xc}(t) + r(t) \odot M_{hc}(t)] \tag{5.11}$$

$$h(t) = u(t) \odot h(t-1) + (1 - u(t)) \odot c(t) \tag{5.12}$$

Moreover, PeakRNN is robust to the variations of the input data as the algorithm selects the top elements regardless of the threshold. Additionally, the number of peaks for the x and h vectors can either be equal or different. In our experiments, we used the same number of peaks for both vectors, but this combination can be optimized depending on a given task. The top N_p elements might be selected using sorting, but an alternative solution is to approximate N_p elements

Table 5.1: Theoretical cost calculations of MACs and MAs.

	GRU	**DeltaRNN/PeakRNN**	**Equations**
MACs (x + h)	$3(N_xN_h)$ + $3(N_hN_h)$	$o_x[3(N_xN_h)]$ + $o_h[3(N_hN_h)]$	(5) - (8)
MACs (pointwise)	$3N_h$	$3N_h$	(11) - (12)
MAs (x+h **weights**)	$3(N_xN_h)$ + $3(N_hN_h)$	$o_x[3(N_xN_h)]$ + $o_h[3(N_hN_h)]$	(5) - (8)
MAs (x+h **read**)	N_x + N_h	$2N_x$ + $2N_h$	(1) - (4)
MAs (h **write**)	N_h	N_h	(12)
MAs (M **states read**)	-	$4N_h$	(5) - (8)
MAs (M **states write**)	-	$4N_h$	(5) - (8)
MAs ($\hat{x}$ + $\hat{h}$ **write**)	-	o_xN_x +o_hN_h	(1) - (2)

by exploring the underlying statistical properties of the data as introduced in Section 5.4.

Table 5.1 provides an overview of the theoretical estimations of MACs and MAs required for a GRU every timestep as these, particularly MAs, are among the most costly operations [73]. These estimations enable us to compare DeltaRNN and PeakRNN on equal terms. The calculations are derived from equations (5.1) - (5.12). N_x and N_h refer to the dimensionality of an input vector x and activation vector h, respectively. o_x and o_h, called *occupancy* [138], correspond to the fraction of non-zero values of an input vector x and activation vector h, respectively. These fractions define how many operations will be executed. As it can be seen in Table 5.1, there is MA overhead compared to GRU due to keeping track of additional states. However, these excess operations are negligible for huge RNNs as the MAs for weights scale approximately quadratically. Therefore, significant savings and increased speedup can be achieved with a sparse occupancy. This is pronounced even more for MACs, and the results are presented in Section 5.6.

5.4 Statistical RNN Algorithm

StatsRNN finds the top N_p elements by exploiting the statistical properties of the $|x(t)-\hat{x}(t-1)|$ and $|h(t-1)-\hat{h}(t-2)|$ computations that determine whether the elements should be zeroed out. We hypothesized that our training dataset is representative of our SE application. Consequently, we can assume that the underlying statistical distributions within the network are a good approximation of distributions in the application. We created an average histogram with 256

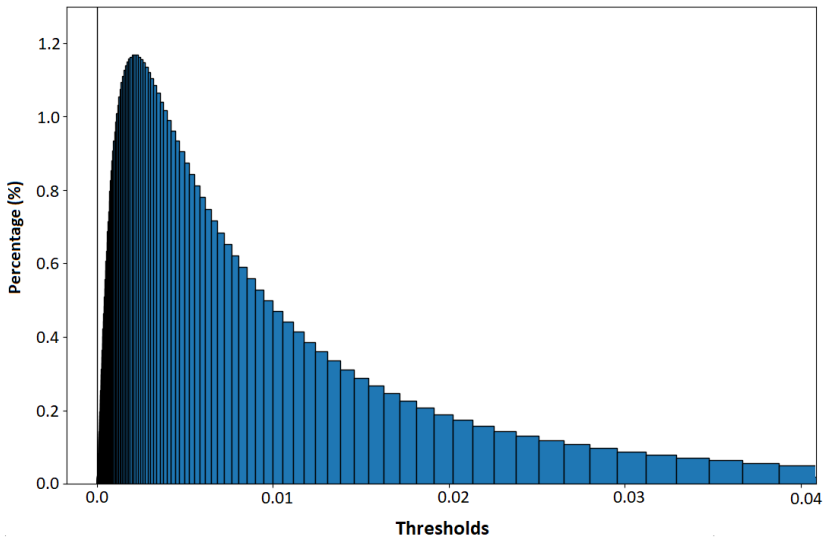


Figure 5.1: A zoomed view on a part of the histogram with logarithmic bins for the delta calculation $|x(t) - \hat{x}(t-1)|$ from the training dataset. The x-axis represents bin boundaries that are used to determine x and h thresholds. The thin black vertical line to the very left corresponds to the first bin that contains ∼31% of zeros.

logarithmic bins for the x and h computations separately, based on the entire training dataset. Figure 5.1 shows a histogram example for $|x(t) - \hat{x}(t-1)|$ ($|h(t-1) - \hat{h}(t-2)|$ has a similar distribution) where it can be observed that the data is correlated. Also, due to ReLU in the first fully connected (FC) layer, a lot of values are zero, ∼31% and 21% for x and h, respectively, which enables to exploit sparsity to a high degree. After training, the threshold can be statistically determined for x and h separately using the bin boundaries, i.e. selecting a boundary where all the bins to its right (greater values) represent the percentage of elements (o_x and o_h) equivalent to the number of peak elements (PeakRNN) that should be processed. This approach preserves the idea of PeakRNN and eliminates sorting. In the presented experiments, the same percentage of top elements to extract was set for x and h.

The StatsRNN approach was applied to the entire test dataset, trying to obtain various percentages of x and h elements individually, varying from 50% down to 1%. Due to a big variety of scenes, some environments naturally deviate from the required percentage more while others less. The smallest difference between the expected and the actual obtained percentage was only 0.05% (Quiet Street),

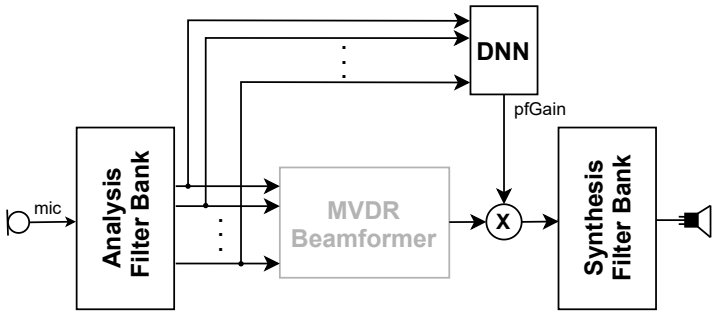


Figure 5.2: Full system overview representing simplified internals of a HI. The highlighted parts (black) are used for the single-channel SE experiments.

where, on average, 40.05% of the h elements were processed instead of 40%. The biggest outliers were Pink and, in particular, White noise, where ∼31.65% of the h vector elements was processed instead of 50%. All the details about the datasets are described in Section 5.5.3. StatsRNN analytically defines a threshold for x and h individually. This is a very important property since the vectors have different sparsity as also shown in [138], and their individual handling might contribute to additional improvements. Therefore, exploiting a priori statistical knowledge about the data will lead to a better and more deterministic algorithm with consistent performance compared to DeltaRNN. Furthermore, the estimations provided in Table 5.1 can be directly used for StatsRNN as well. Instead of N_p in equations (5.1)-(5.4), two statistically derived thresholds Θ_x and Θ_h will be applied for StatsRNN (a single threshold Θ in original DeltaRNN [138]).

5.5 Experimental Setup

This section describes the entire system setup used for performing the experiments.

5.5.1 Hearing-Instrument Application

Figure 5.2 illustrates a simplified system extracted from a real HI setup where the DNN replaces a typical noise reduction module for obtaining postfilter

gain. Firstly, the *Analysis Filter Bank* applies a 1024-point FFT and a square-root Hanning window on the 20 kHz microphone signal (*mic*), resulting in 512 frequency sub-bands and downsampling to 40 Hz (a new frame every 25 ms, no overlapping). The output is then passed to the *DNN* that will learn to estimate a postfilter gain ($pfGain$) to be applied on the original signal. Finally, the *Synthesis Filter Bank* reconstructs a wideband signal and passes it to the speaker. The proposed techniques are demonstrated on a single microphone but they can be also applied in a setup with a microphone array (e.g. HIs with two microphones). In such case, the *Minimum Variance Distortionless Response (MVDR) Beamformer* could be used for multi-channel processing where interferences from undesired directions would be attenuated.

5.5.2 DNN Architecture

The DNN architecture used in the experiments consists of three layers: FC-GRU-FC, each having 512 output neurons, with 512 inputs to the first FC layer. The first FC layer is followed by ReLU activation function, while the GRU layer uses tanh and sigmoid activation functions. The final output of the network are 512 $pfGain$ values that are applied on the original signal. The GRU component is replaced with DeltaRNN, PeakRNN, and StatsRNN during the experiments.

5.5.3 Dataset

The DNN input is a mixed signal y, i.e. clean speech corrupted with noise, constructed by adding 30-second segments of noise n and clean speech s together ($y = s + n$). The 30-second segments contain one to three speakers with a maximum gap of 300 ms and up to 30% overlap, creating seemingly natural flow and tempo of a conversation. The speech was obtained from the *VCTK Corpus* [204] and *Akustiske Database for Dansk* [48]. Fifteen different types of audio environments (referred to as background noise) were used, reflecting the most relevant acoustic situations that people are exposed to in the real world in order to obtain the required variations in SNR estimates. Eight scenes (Beach, Busy Street, Park, Pedestrian Zone, Quiet Street, Shopping Centre, Train Station, Woodland) were obtained from [128], five scenes are a part of the internal database of the Demant company (Bar, Cafe, Canteen, Car, Office), and Pink and White stationary noises were simulated. The entire dataset contains ∼25 hours (left and right channels together) of mixed signal, divided into training (∼19.5 h), test, and validation (each ∼2.7 h) subsets. Each of the three subsets is unique, i.e. the speakers and the background noise sections are not shared across the subsets.

The unprocessed speech has SNR and PESQ of 4.39 dB and 1.85 MOS-LQO (Mean Opinion Score - Listening Quality Objective) [25], respectively. The starting SNRs for the scenes in the test dataset vary between -12.7 dB (e.g. White noise) to 14.4 dB (e.g. Park), with most acoustic scenes having SNR up to 8 dB (80%). The high SNRs are present in only a few cases to cover the necessary variations as mentioned before.

5.5.4 Training

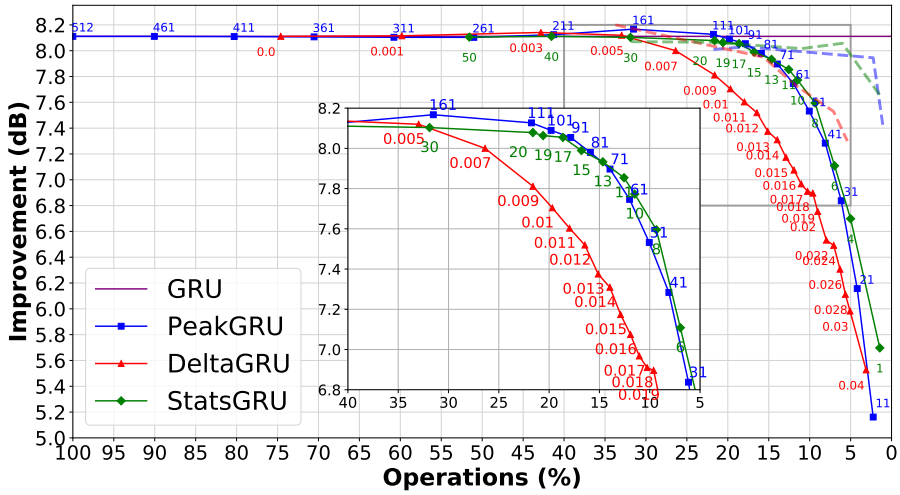
The DNN is trained on a linear ideal ratio mask (IRM) where the mask value is a continuous gain between zero and one. The IRM is defined as follows:

$$IRM = \left(\frac{|s(t,f)|}{|s(t,f)| + |n(t,f)|} \right) \tag{5.13}$$

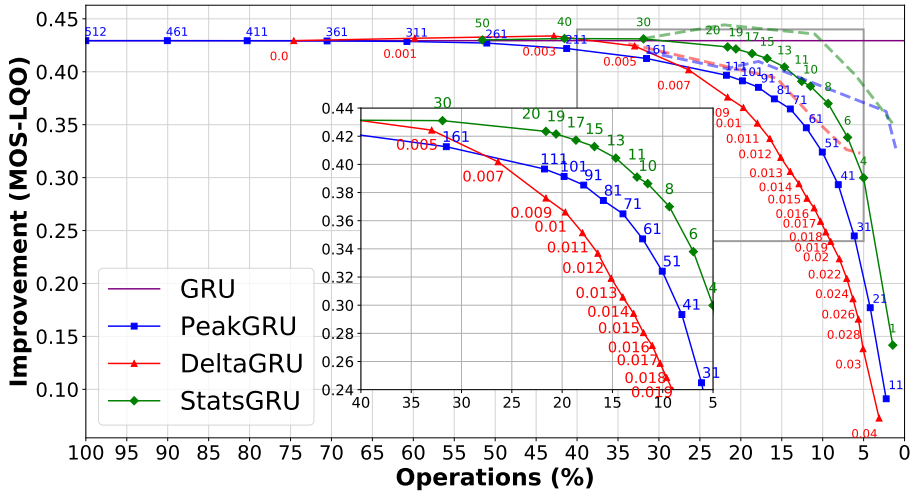
The $s(t,f)$ and $n(t,f)$ represent the clean speech and noise, respectively, with time frame t and frequency channel f. The mean squared error between the target IRM and the denoised speech is used as the loss function. The final DNN was trained using a batch size of 128 and a sequence length of 100 samples, which corresponds to 2.5 seconds. The DNN with a GRU layer was trained in Tensorflow using 32-bit floating-point, and the weights and biases were transferred to DeltaRNN, PeakRNN, and StatsRNN (further referred to as DeltaGRU, PeakGRU, and StatsGRU) for inference. The results are presented in Section 5.6. Transferring the learned parameters replaces computationally expensive and time demanding training from scratch for each configuration, and enables to compare all the methods in a fair manner. The DeltaGRU, PeakGRU, and StatsGRU networks were finally also retrained using transfer learning, where the transferred GRU model served as a starting point. This approach is discussed at the end of Section 5.6.

5.6 Results and Discussion

Figure 5.3 shows the improvement of the different GRU implementations to unprocessed speech in SNR and PESQ for decreasing percentage of MAC operations per timestep. It also provides a zoomed view on a part of the plot discussed in this section. The annotations on the PeakGRU, DeltaGRU, and StatsGRU curves describe the tested number of processed peaks (512-11), thresholds (0.0-0.04) and the desired percentages of elements to extract (50-1), respectively. The GRU performance of 8.11 dB and 0.43 MOS-LQO in (a) and (b), respectively, is shown as baseline with 100% MAC operations, corresponding to 1.5744 MOps. The same number of operations is also required for MAs. Only a plot for MAC



(a) SNR improvement vs MACs (unprocessed speech = 4.39 dB).



(b) PESQ improvement vs MACs (unprocessed speech = 1.85 MOS-LQO).

Figure 5.3: Plots (a) and (b) show the percentage of executed MACs with respect to improvement in SNR/PESQ (including a zoomed part of the plot). 100% of MACs corresponds to the baseline GRU illustrated as a constant horizontal line, i.e. the x-axis does not apply to it. The data labels annotated on the lines represent the number of processed peaks, thresholds, and the desired % of elements to extract for PeakGRU, DeltaGRU, and StatsGRU, respectively. The dashed lines show the benefit of retraining that further optimizes the performance.

results is presented since MA reduction is done in the same linear manner and its diagram would, therefore, look almost the same (max ∼0.4% more MAs from the original number of operations). Due to the MA overhead estimated in Table 5.1, all the three modified GRU algorithms have a total of ∼1.581 MOps when processing all features. The computations are based on an *overall* scene, i.e. an average across all the acoustic environments and x and h subsets. However, a similar trend can be also observed for the individual scenes, i.e. PeakGRU and StatsGRU outperforming DeltaGRU. The only scene where DeltaGRU has subtly better performance is White noise, and it is on par with PeakGRU for Pink noise and Busy Street scenes. The performance of StatsGRU is very similar to PeakGRU as it approximates the number of top elements to be processed. The largest SNR improvement on average was obtained for the White noise environment (∼15.72 dB), while the smallest one for the Office scene (∼4.26 dB).

As it can be observed in Figure 5.3(a), all the methods have equal and no loss in performance down to ∼30%, corresponding to ∼0.4723 MOps and thus reducing the computations by 70%. DeltaGRU, and likewise StatsGRU, already saves 25% of computations with Θ=0.0 without any SNR/PESQ degradation due to the ReLU in the first FC layer that produces a sparse input to the GRU. However, with decreasing % of computations, the objective metrics start to noticeably degrade especially for DeltaGRU, while PeakGRU and StatsGRU have a less steep decrease. StatsGRU and PeakGRU are aligned regarding SNR, and StatsGRU has slightly better PESQ. This behavior can be explained by the fact that StatsGRU selects a specific percentage of x and h separately across the *entire dataset*, while PeakGRU does so *every timestep*, and thus sometimes chooses less significant values, which is subtly reflected in speech quality. At ∼7.8 dB SNR improvement, the two new methods reduce the computations down to ∼12%, which is a reduction by almost a factor of 2 compared to ∼21% for DeltaGRU. Even at 7 dB, which represents an acceptable loss of 1 dB, PeakGRU and StatsGRU execute on average 7% of operations while DeltaGRU 11%. Similarly, a better PESQ performance of the two proposed techniques can be seen in Figure 5.3(b). On the other hand, if the number of operations is fixed at 12%, PeakGRU and StatsGRU outperform DeltaGRU by 0.7 dB on average. Figure 5.4 further presents the results per scene, where the SNR improvements for each of the methods are plotted as overlapping bars.

Retraining the models further optimizes the performance, especially for the configurations with high thresholds/many peaks skipped. This is illustrated with dashed lines in Figure 5.3, where all the three methods benefit comparably from transfer learning. Therefore, the same relative gains for DeltaRNN, PeakRNN, and StatsRNN remain valid.

Future work further explores the most optimal ratio between x and h sparsities, since the best model might not necessarily use the same threshold/number of

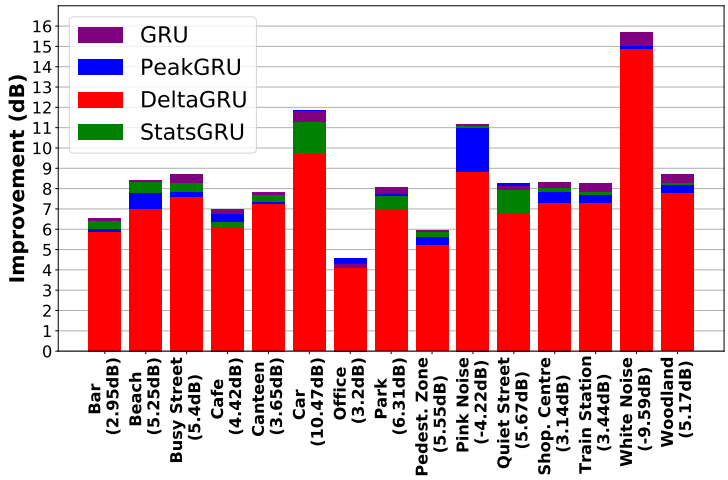


Figure 5.4: SNR improvement per scene under the fixed number of operations per timestep (12%) plotted as overlapping bars, with PeakGRU and StatsGRU outperforming DeltaGRU by 0.7 dB on average. The configurations use 61 peaks, Θ=0.016, and top 10% of elements for PeakGRU, DeltaGRU, and StatsGRU, respectively. The x-axis denotes each tested scene along with its initial average SNR. PeakGRU is on par with GRU, and StatsGRU with DeltaGRU for the Car and Office scene, respectively.

peaks for both. This could be done by making the sparsity ratio parameter differentiable, such that it can be trained together with the network itself.

5.7 Conclusion

This paper introduced two new pruning techniques demonstrated on a single-channel SE task using complex environments and big variety of speakers. The results proved that computations in a RNN can be efficiently reduced nearly $2x$ compared to the state-of-the-art DeltaRNN while maintaining sufficient quality of objective measures. In addition, reducing the SNR quality by only $\sim$0.3 dB saves 88% of operations in PeakRNN and StatsRNN, while the same reduction in DeltaRNN is achieved by degrading SNR by 1 dB. Furthermore, PeakRNN is deterministic and thus provides worst-case execution guarantees required by real-time applications. If the deterministic aspect can be slightly relaxed, further

savings can be achieved by StatsRNN using a statistical approximation. Overall, both algorithms are hence suitable for resource-constrained embedded devices such as HIs.

Acknowledgment

We would like to thank Asger Heidemann Andersen and Robert Rehr for all their advice, help, and knowledge they provided.

Chapter 6

Delta Keyword Transformer: Bringing Transformers to the Edge through Dynamically Pruned Multi-Head Self-Attention

By Zuzana Jelčicová and Marian Verhelst [C3]

Abstract

Multi-head self-attention forms the core of Transformer networks. However, their quadratically growing complexity with respect to the input sequence length impedes their deployment on resource-constrained edge devices. We address this challenge by proposing a dynamic pruning method, which exploits the temporal stability of data across tokens to reduce inference cost. The threshold-based method only retains significant differences between the subsequent tokens, effectively reducing the number of multiply-accumulates, as well as the internal tensor data sizes. The approach is evaluated on the Google Speech Commands Dataset for keyword spotting, and the performance is compared against the baseline Keyword Transformer.

Our experiments show that we can reduce $\sim 80\%$ of operations while maintaining the original 98.4% accuracy. Moreover, a reduction of $\sim 87 - 94\%$ operations can be achieved when only degrading the accuracy by 1-4%, speeding up the multi-head self-attention inference by a factor of $\sim 7.5 - 16$.

Index Terms— Transformers, delta computations, pruning, compression, keyword spotting, edge devices

6.1 Introduction

The Transformer architecture [146] is an emerging type of neural networks that has already proven to be successful in many different areas such as natural language processing [186, 193, 210, 198], computer vision [236, 247, 250, 246], and speech recognition [219, 235, 234, 244]. Its success lies in the multi-head self-attention (MHSA), which is a collection of attention mechanisms executed in parallel. Although Transformers achieve state-of-the-art results, deployment to resource-constrained devices is challenging due to their large size and computational complexity that grows quadratically with respect to the sequence length. Hence, self-attention, despite being extremely efficient and powerful, can easily become a bottleneck in these models. A widely used compression technique to reduce the size and computations of DNNs is pruning, that has been extensively researched throughout the years [90, 110, 188, 124]. An increasing number of works focusing on MHSA pruning recently emerge. These mainly aim for reducing the number of attention heads in each Transformer layer [196, 205, 195], and token pruning [254, 218, 241, 248]. Eliminating attention heads completely to speed up the processing might significantly impact accuracy. Therefore, token (a vector in the sequence) pruning represents a more suitable approach, where attention heads are preserved and only unnecessary tokens within the individual heads are removed. However, most of the methods above i) require demanding training procedures that hinder utilizing a single method across various models and applications without unnecessary overhead, and ii) focus on coarse-grained pruning.

In this work, we further push pruning to finer granularity, where individual features within tokens are discarded at runtime using a threshold in the MHSA pipeline. The reduction is based on the comparison of similarities between corresponding features of subsequent tokens, where only the above-threshold delta differences are stored and used for performing the multiplications (MACs). This technique significantly reduces computational complexity during inference

and offers intermediate data compression opportunities. Our method does not require any training and can, therefore, be used directly in the existing pretrained Transformer models. Moreover, no special and expensive hardware has to be developed as only comparisons are used in the algorithm. The evaluation is done on a pretrained Keyword Transformer model (KWT) [233] using the Google Speech Commands Dataset (GSCD) [178] with the focus on the accuracy-complexity trade-off. The results show that the number of computations can be reduced by $4.2x$ without losing any accuracy, and $7.5x$ while sacrificing 1% of the baseline accuracy. Furthermore, the processing of the original MHSA block can be sped up by a factor of ~ 16 while still achieving high accuracy of $\sim 95\%$. Therefore, this work represents the next step to enable efficient inference of Transformers in low-power edge devices with the tinyML constraints.

6.2 Related Work

Different approaches have been used to reduce the computational complexity of the MHSA, such as cross-layer parameter sharing [222], trimming individual weights [217] or removing encoders by distillation [199, 200, 192, 227, 201]. Recent research [196, 242, 205, 195] demonstrates that some attention heads can be eliminated without degrading the performance significantly. However, in order to obtain substantial computational savings and thus inference time gains, a considerable portion of heads would have to be discarded, inevitably leading to noticeable accuracy drops.

Other works focus on token pruning instead of removing redundant parameters. In [218], redundant word-vectors are eliminated, outperforming previous distillation [199, 200] and head-pruning methods [196]. However, it requires training of a separate model for each efficiency constraint. This issue is resolved in [241] by adopting one-shot training that can be used for various inference scenarios, but the training process is complicated and involves multiple steps. Cascade pruning on both the tokens and heads is applied in [248], i.e., once a token and/or head is pruned, it is removed in all following layers. Nonetheless, this approach requires sorting of tokens and heads depending on their importance dynamically to select the top-k candidates, which needs specialized hardware. Similar to our work, recently published [254] also adopts a threshold-based pruning approach, which removes unimportant tokens as the input passes through the Transformer layers. However, this method requires a three-step training procedure to obtain a per-layer learned threshold, which again prevents to easily deploy the technique across a wide range of pre-trained networks. Most of the previous methods, moreover, only focus on optimizing Transformers for the natural language processing task.

The idea of threshold-based pruning using delta values for performing computations has already been explored for other types of DNNs, such as recurrent [138] and convolutional [239] neural networks. However, incorporating a delta threshold in these networks results in significant memory overhead, as it requires storing intermediate states and activations. This issue is eliminated in our Delta Transformer, where almost no additional resources are required.

6.3 The Keyword Transformer

The typical Transformer encoder [146] adopted in KWT consists of a stack of several identical Transformer blocks. Each Transformer block comprises of Multi-Head Self-Attention (MHSA), Multi-Layer Perceptron (MLP), layer normalizations, and residual connections as illustrated in Figure 6.1. The key component in Transformers is the MHSA containing several attention mechanisms (heads) that can attend to different parts of the inputs in parallel. We base our explanation on the KWT, proposed in [233]. This model takes as an input the MFCC spectrogram of T non-overlapping patches $X_{MFCC} \in R^{T\,x\,F}$, with $t = 1, ..., T$ and $f = 1, ..., F$ corresponding to time windows and frequencies, respectively. This input is first mapped to a higher dimension d using a linear projection matrix $W_0 \in R^{F\,x\,d}$ along the frequency dimension, resulting in T tokens of dimension d. These are then concatenated with a learnable class embedding token $X_{CE} \in R^{1\,x\,d}$ representing a global feature for the spectrogram. Subsequently, a learnable positional embedding $X_{PE} \in R^{(T+1)\,x\,d}$ is added to form a final input to the Transformer encoder:

$$X = [X_{CE}; X_{MFCC}W_0] + X_{PE} \tag{6.1}$$

The Transformer encoder multiplies the input X with the projection matrices W_Q, W_K, $W_V \in R^{d\,x\,d}$, producing Query (Q), Key (K), and Value (V) input embedding matrices:

$$Q = XW_Q; \qquad K = XW_K; \qquad V = XW_V \tag{6.2}$$

The matrices are then divided into k attention heads to perform the self-attention computations in parallel, where each of the heads $i = 1, 2, .., k$ is given by:

$$head_i = attention(Q_i, K_i, V_i) = softmax\left(\frac{Q_i(K_i)^T}{\sqrt{d_h}}\right)V_i \tag{6.3}$$

The MHSA is defined as a concatenation of the attention heads, weighted by a projection matrix $W_P \in R^{kd_h\,x\,d}$, where $d_h = d/k$:

$$X_{MHSA}(Q, K, V) = [head_1, head_2, ..., head_k]W_P \tag{6.4}$$

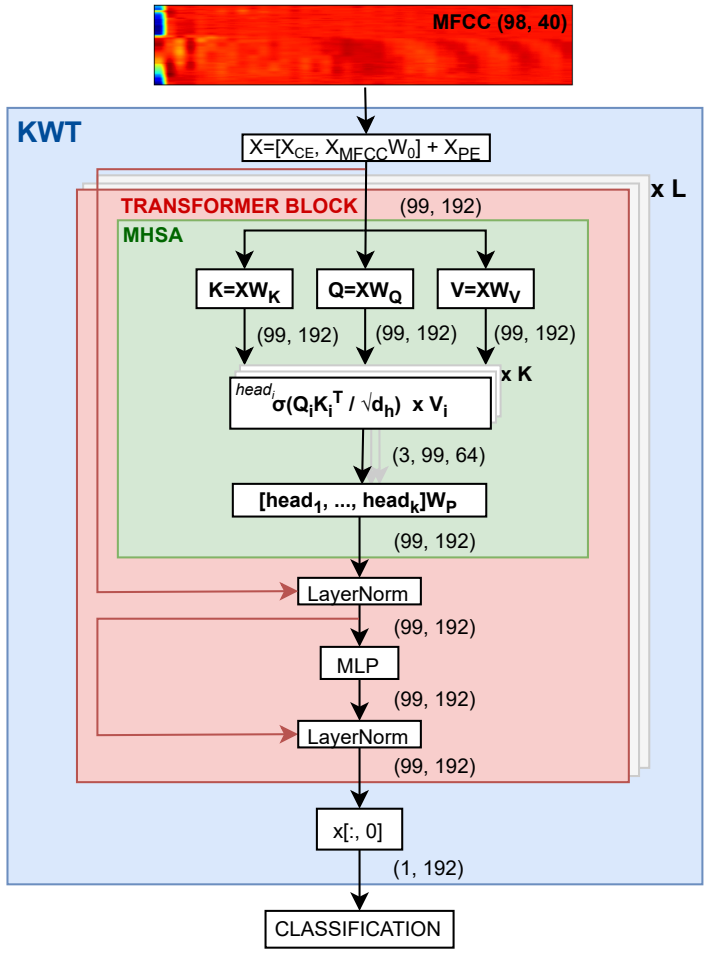


Figure 6.1: A high-level overview of the KWT model along with its dimensions. Red lines denote the residual connections.

The MHSA output is then added to the input X with a residual connection and passed though the first layer normalization and the MLP block, followed by another addition of a residual input and second normalization:

$$X_{LN1} = LN(X_{MHSA} + X); \qquad X_{LN2} = LN(X_{MLP} + X_{LN1}) \tag{6.5}$$

This structure is repeated L times, denoting layers, to create an architecture of stacked Transformer layers.

Table 6.1: Configuration of the KWT-3 architecture

Model	dim d	dim d_{MLP}	heads k	layers L	#params
KWT-3	192	768	3	12	5,361k

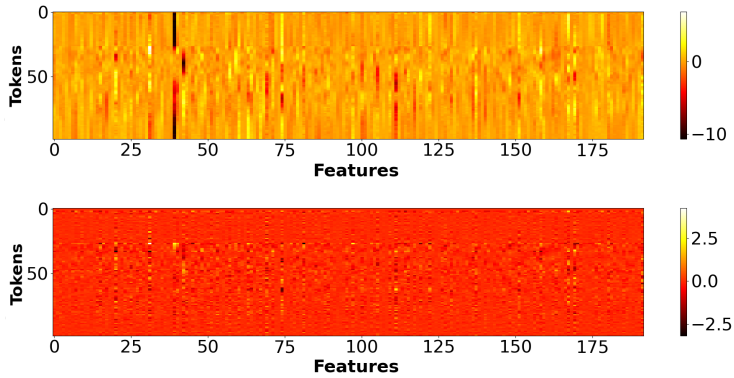


Figure 6.2: Input data to the 7th Transformer layer at the top along with its delta version at the bottom for keyword right.

In the KWT model, the MLP block is a two-layer feed-forward neural network using a GELU activation function after the first layer. The class embedding vector is extracted from the output of the last Transformer block to perform classification.

Three KWT models are proposed in the original work: KWT-1 (607k parameters, $97.72\% \pm 0.01$ accuracy), KWT-2 (2,394k parameters, $98.21\% \pm 0.06$ accuracy), and KWT-3 (5,361k parameters, $98.54\% \pm 0.17$ accuracy). We selected KWT-3 for our experiments, as it poses the biggest challenge as well as potential for compressing and reducing the computational complexity. The KWT-3 configuration is listed in Table 6.1.

6.4 KWT Model Analysis

The attention mechanism involves MACs of two matrices, resulting in $O(n^2)$ time and space complexity. However, as all tokens attend to each other, a certain level of redundancy is expected to be found in the system due to diffusion of information. Therefore, we analyze the KWT model on the GSCD to observe the degree of change across the tokens as they pass though the MHSA. We feed multiple different keywords through the 12-layer KWT and inspect the MHSA

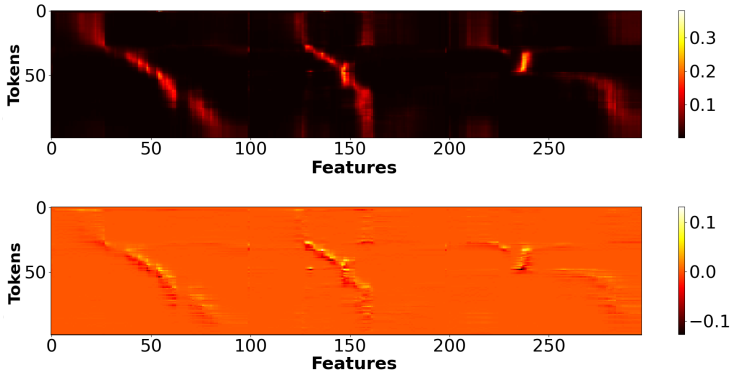


Figure 6.3: Softmax output of the 7th Transformer layer at the top along with its delta version at the bottom for the keyword **right**. The figure illustrates three attention heads.

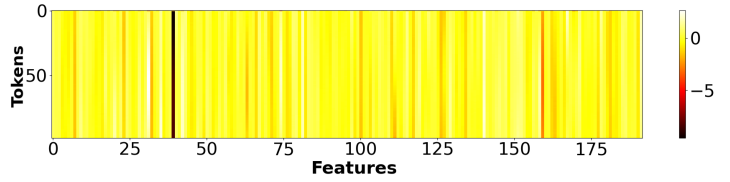


Figure 6.4: Input data to the 7th Transformer layer for **_silence_**.

inputs as well as intermediate results within the block. While considerable correlation across the tokens is expected for the initial input and intermediate results in the first layer, it is noteworthy to observe such behavior also in the MHSA of deeper layers, which is in line with cosine similarity measurements on word-vectors performed in [218]. Correlation is illustrated in Figure 6.2 showing the input X (top) together with the difference between subsequent rows of this tensor (bottom), for the 7th layer of a keyword *right*. Figure 6.3 repeats the same analysis for the softmax output of layer 7. It is clear that there is a significant amount of correlation between consecutive tokens, which opens up opportunities for data compression and/or computational data reuse. For example, $\sim 84\%$ of the differences between corresponding features of subsequent tokens in X are smaller than 1% of the dynamic range of X (7th layer). Such a tendency was observed for all voice-containing input sequences.

Moreover, when analyzing intermediate tensors from inputs of the *_silence_* class, even larger data redundancy can be observed (Figure 6.4). It is clear that fully computing every single token would be a waste of computational and memory resources.

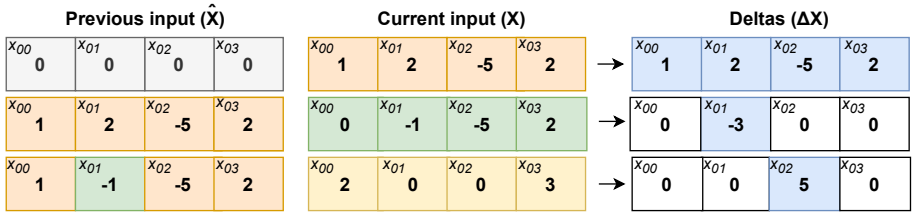


Figure 6.5: Delta algorithm example across three tokens with threshold $\theta = \mathbf{1.0}$. The top row corresponds to the first input vector that is always left untouched (no threshold).

All these observations demonstrate that the amount of a significant change across the tokens constitutes only a small portion of the whole. Hence, introducing a threshold for recomputing could drastically decrease the computational load and inference time. Furthermore, exploiting sparsity across the tokens can also offer data compression. Therefore, we propose a delta algorithm that utilizes a threshold to discard insignificant values, further described in Section 6.5.

6.5 Delta Algorithm

The objective of the delta algorithm is to transform a dense matrix-vector multiplication into a highly-sparse matrix-vector multiplication to reduce computational complexity and enable data compression, where only non-zero deltas are stored and used for computations.

The input X always starts with the class embedding vector, followed by the first input vector. These two vectors (rows of the tensors) will always be left untouched throughout the complete MHSA pipeline. Every subsequent token after these will be represented by its delta value. This delta change $\Delta X(t)$ is calculated as the difference between the current input $X(t)$ and reference vector $\hat{X}(t-1)$. Only delta differences larger than a threshold θ are retained and used to update the reference vector $\hat{X}(t)$:

$$\Delta X(t) = \begin{cases} X(t) - \hat{X}(t-1) & \text{if } |X(t) - \hat{X}(t-1)| > \theta \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

$$\hat{X}(t) = \begin{cases} X(t) & \text{if } |X(t) - \hat{X}(t-1)| > \theta \\ \hat{X}(t-1) & \text{otherwise} \end{cases} \tag{6.7}$$

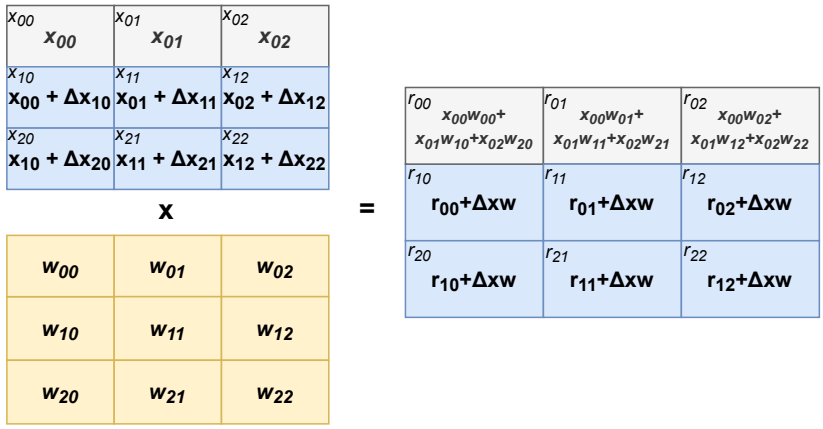


Figure 6.6: Baseline delta algorithm.

Where the $\hat{X}$ vector is initialized to 0s and updated once the first token arrives. Figure 6.5 visualizes this encoding over three tokens with $\theta = 1.0$. The top row represents the first input vector that is left untouched (no delta algorithm applied). The orange and green colors in $\hat{X}$ show which values from the current input X are propagated for the next token. White ΔX positions denote values of which magnitude equals to/is below θ and thus are skipped.

We apply the delta encoding of data at six different places in the MHSA: layer input X, matrices K and Q, scaled QK^T, softmax output, and the attention head output. While the computations of delta values are the same everywhere, the subsequent operations with these deltas differ depending on whether i) a delta-encoded matrix is multiplied with a regular matrix, ii) two delta-encoded matrices are multiplied together, or iii) a non-linear $softmax$ function is applied. These three versions are described in the next subsections.

6.5.1 Delta-Regular Matrix Multiplication

Thanks to the delta representation, only non-zero ΔX are stored and used for multiplications as visualized in Figure 6.6. A weight matrix is denoted as W, and indices for Δxw in the result matrix R are excluded for clarity.

The output $R(t)$ of the tensor operation can hence be computed by accumulating the result of the previous reference token $R(t-1)$ with the multiplication results of the weights with the delta values only. The updated $R(t)$ will then be the

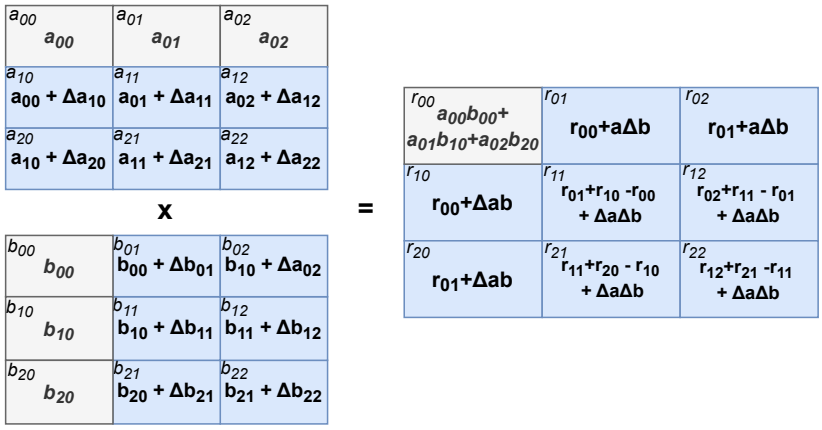


Figure 6.7: Delta algorithm for $\boldsymbol{QK^T}$ represented with matrices $\boldsymbol{A}$ and $\boldsymbol{B}$.

new baseline for the upcoming token:

$$R(t) = \Delta X(t)W + R(t-1) \tag{6.8}$$

With $R(0)$ initialized to 0. These delta multiplications are used in XW_Q, XW_K, XW_V, $softmaxV$ and $[head_1, head_2, head_3]W_P$.

6.5.2 Delta-Delta Matrix Multiplication

As a result of the delta encoding, both Q and K will be expressed in their delta versions, and the multiplications will thus be slightly modified. This is described below and illustrated in Figure 6.7 in a general form, with matrices A and B representing Q and K^T, respectively.

The multiplication of the first A row with the first B column is done as usually without using deltas:

$$r_{00} = a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} \tag{6.9}$$

Then, the multiplication of the first A row and second B column exploits the delta approach in horizontal direction, where the $a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20}$ expression can be replaced with r_{00} from eq. 6.9 (marked with red):

$$\begin{aligned} r_{01} &= a_{00}(b_{00} + \Delta b_{01}) + a_{01}(b_{10} + \Delta b_{11}) + a_{02}(b_{20} + \Delta b_{21}) \\ &= a_{00}b_{00} + a_{00}\Delta b_{01} + a_{01}b_{10} + a_{01}\Delta b_{11} + a_{02}b_{20} + a_{02}\Delta b_{21} \\ &= r_{00} + a_{00}\Delta b_{01} + a_{01}\Delta b_{11} + a_{02}\Delta b_{21} \end{aligned} \tag{6.10}$$

Similarly, calculating results in the vertical direction for the rows of A and first column of B is given by:

$$\begin{aligned} r_{10} &= b_{00}(a_{00} + \Delta a_{10}) + b_{10}(a_{01} + \Delta a_{11}) + b_{20}(a_{02} + \Delta a_{12}) \\ &= b_{00}a_{00} + b_{00}\Delta a_{10} + b_{10}a_{01} + b_{10}\Delta a_{11} + b_{20}a_{02} + b_{20}\Delta a_{12} \\ &= r_{00} + \Delta a_{10}b_{00} + \Delta a_{11}b_{10} + \Delta a_{12}b_{20} \end{aligned} \tag{6.11}$$

An approach for multiplications for all the other positions is demonstrated on the second A row and second B column:

$$\begin{aligned} r_{11} &= (a_{00} + \Delta a_{10})(b_{00} + \Delta b_{01}) + (a_{01} + \Delta a_{11})(b_{10} + \Delta b_{11}) \\ &+ (a_{02} + \Delta a_{12})(b_{20} + \Delta b_{21}) \\ &= a_{00}b_{00} + a_{00}\Delta b_{01} + \Delta a_{10}b_{00} + \Delta a_{10}\Delta b_{01} \\ &+ a_{01}b_{10} + a_{01}\Delta b_{11} + \Delta a_{11}b_{10} + \Delta a_{11}\Delta b_{11} \\ &+ a_{02}b_{20} + a_{02}\Delta b_{21} + \Delta a_{12}b_{20} + \Delta a_{12}\Delta b_{21} \\ &= r_{01} + r_{10} - r_{00} + \Delta a_{10}\Delta b_{01} + \Delta a_{11}\Delta b_{11} + \Delta a_{12}\Delta b_{21} \end{aligned} \tag{6.12}$$

Where different colors mark each of the three multiplications. Simplifying parenthesis shows that the expressions not involving any deltas can be substituted with r_{00}. Next, the terms with Δb are replaced with r_{01}, while those containing Δa with r_{10}. Since r_{00}, r_{01}, and r_{10} have already been computed in previous timesteps, we only need to do the (sparse) delta multiplications themselves and subtract the r_{00} result as it is present in both r_{01} and r_{10}. These steps are then applied to all the other slots as shown in Figure 6.7.

6.5.3 Delta for Softmax

Delta algorithm cannot be directly applied for softmax as this function introduces a non-linearity to the system:

$$softmax(r)_i = \frac{exp(r_i)}{\sum_j exp(r_j)} \tag{6.13}$$

We will have to introduce a scaling factor to correct the softmax computations. As done earlier, we will again start by performing unaltered processing of the initial row $r_0 = [r_{00}\, r_{01}\, r_{02}]$ (class embedding excluded for clarity) with a regular softmax function:

$$softmax(r)_0 = \frac{[exp(r_{00})\, exp(r_{01})\, exp(r_{02})]}{\sum[exp(r_{00})\, exp(r_{01})\, exp(r_{02})]} \tag{6.14}$$

The next row of the scaled input QK^T is already expressed with deltas:

$$r_1 = [\Delta r_{10}\, \Delta r_{11}\, \Delta r_{12}] \tag{6.15}$$

The r_1 nominator NOM_{r_1} for softmax is thus given by:

$$NOM_{r_1} = [exp(\Delta r_{10})\, exp(\Delta r_{11})\, exp(\Delta r_{12})] \tag{6.16}$$

While the denominator $DENOM_{r_1}$ as:

$$DENOM_{r_1} = \frac{\sum[exp(r_{00} + \Delta r_{10})\, exp(r_{01} + \Delta r_{11})\, exp(r_{02} + \Delta r_{12})]}{\sum[exp(r_{00})\, exp(r_{01})\, exp(r_{02})]} \tag{6.17}$$

Finally, a scaling factor for each of the values to correct the softmax result is:

$$SF_{r_1} = softmax(r)_1 \frac{NOM_{r_1}}{DENOM_{r_1}} \tag{6.18}$$

6.5.4 Computational Savings

To assess the potential computational savings for the Delta KWT, we differentiate between the two main sublayers: i) MHSA, and ii) MLP. The MLP block consists of two fully connected layers with weight matrices of dimensions (192,768) and (768,192), respectively. Without any delta modification, ∼39% of the multiplication of the original KWT can be found in the MHSA and ∼61% in the MLP. Although MLP is the prevailing module in this specific scenario, its complexity does not grow quadratically with the input sequence length. Moreover, there are many well-established compression techniques available, some of them presented in Section 6.2. Hence, pruning of the MLP is out of the scope of our work, and it is only stated for completeness. The MHSA multiplication operations can be further split into XW_K, XW_Q, XW_V (∼59.63%), QK^T (∼10.25%), $softmax(QK^T)V$ (∼10.25%), and final projection with attention heads $[head_1, head_2, head_3]W_P$ (∼19.88%). The KWT model offers an optimization in the last layer. As shown in Figure 6.1, only the class embedding token is used for the final prediction, making the rest of the tokens within the sequence unused. This dependency can be tracked up to QK^T. The MAC savings in last layer are thus worth 59.64%, always making the total savings at least 4.97% for the whole KWT without losing any accuracy.

Maximum possible computational savings, i.e., cases when only the class embedding and first vector are computed since all deltas are 0, are stated below for each of the MHSA parts. For simplicity, all the terms use matrices A and B, and *row* and *col* for dimensions.

Savings for XW_K, XW_Q, and XW_V for each of the first 11 layers are:

$$l_{0-10} = 1 - \frac{(colA\,x\,2)\,x\,colB\,x\,3}{(colA\,x\,colB\,x\,rowA)\,x\,3} =\sim 97.98\% \tag{6.19}$$

Where $A = (99, 192)$ and $B = (192, 192)$. Computations for XW_Q in the last layer are expressed as:

$$l_{11} = 1 - \frac{(colA\,x\,2)\,x\,colB\,x\,2 + colA\,x\,colB}{(colA\,x\,colB\,x\,rowA)\,x\,3} =\sim 98.32\% \tag{6.20}$$

Savings for QK^T:

$$l_{0-10} = 1 - \frac{(colA\,x\,2\,x\,2)\,x\,heads}{(colA\,x\,colB\,x\,rowA)\,x\,heads} =\sim 99.96\% \tag{6.21}$$

$$l_{11} = 1 - \frac{(colA\,x\,2)\,x\,heads}{(colA\,x\,colB\,x\,rowA)\,x\,heads} =\sim 99.98\% \tag{6.22}$$

Where $A = (99, 64)$ and $B = (64, 99)$. Savings for $softmax(QK^T)V$:

$$l_{0-10} = 1 - \frac{(colA\,x\,2\,x\,colB)\,x\,heads}{(colA\,x\,colB\,x\,rowA)\,x\,heads} =\sim 97.98\% \tag{6.23}$$

$$l_{11} = 1 - \frac{(colA\,x\,colB)\,x\,heads}{(colA\,x\,colB\,x\,rowA)\,x\,heads} =\sim 98.99\% \tag{6.24}$$

Where $A = (99, 99)$ and $B = (99, 64)$. Finally, the projection with attention heads:

$$l_{0-10} = 1 - \frac{(colA\,x\,2)\,x\,colB}{colA\,x\,colB\,x\,rowA} =\sim 97.98\% \tag{6.25}$$

$$l_{11} = 1 - \frac{colA\,x\,colB}{colA\,x\,colB\,x\,rowA} =\sim 98.99\% \tag{6.26}$$

Where $A = (99, 192)$ and $B = (192, 192)$.

Of course, the savings estimated above only hold for the extreme case, which means that either a) all tokens are perfectly correlated, or b) very large thresholds are used, resulting in significant accuracy degradation. Section 6.7 will therefore analyze the complete accuracy-complexity trade-off for real data sequences.

6.5.5 Resources

The proposed delta approach neither requires expensive hardware nor comes with a large memory overhead. Only a single token has to be stored as a reference

whenever the delta method is used. The softmax delta version additionally needs to keep the sum of exp from one timestep to another. In terms of computations, an additional division is needed when calculating scaling factors, along with multiplications with scaling factors for features within a token.

The downside of our method is compute and data irregularity due to the algorithm's *unstructured* pruning. However, there are many techniques proposed in literature such as [207] on how to handle this challenge.

6.6 Experimental Setup

The GSCD v2 [178] is used to evaluate our method as well as the original KWT performance. The dataset contains 105,000 1-second audio snippets of 35 different words sampled at 16 kHz. The model classifies 4,800 keywords from a test set into one of the 12 categories: "up", "down", "left", "right", "yes", "no", "on", "off", "go", and "stop", "_silence_" and "_unknown_".

To assess the impact of the thresholds for the different parts of the MHSA on accuracy and model complexity, we executed threshold sweeps on a subset of 100 keywords (6-12 words from each category). While the thresholds might be different for each delta encoding within the MHSA block, they are the same across every Transformer layer. This means that MHSA in the first layer uses the same thresholds as MHSAs in other layers. From these sweeps, the thresholds leading to a Pareto-optimal accuracy-computations trade-off are used in a full run with all 4,800 keywords. We focused on those configurations that yielded at least 94% accuracy. Since the thresholds are first determined on a subset of the complete dataset, it was expected to obtain variations in the results when performing the test on the full dataset. Additional finetuning, i.e., threshold adjusting, was done and the results are presented and discussed in Section 6.7.

6.7 Results and Discussion

The Pareto-optimal results evaluated on all 4,800 audio files are shown in Figure 6.8, where the delta configurations are provided in the legend. The x- and the left y-axis show a percentage of executed MACs averaged across the layers and achieved accuracy, respectively. The second y-axis represents a speedup factor derived from the amount of MACs. The blue circle corresponds to the original KWT-3 model that achieves $\sim 98.4\%$ accuracy with 100% MACs.

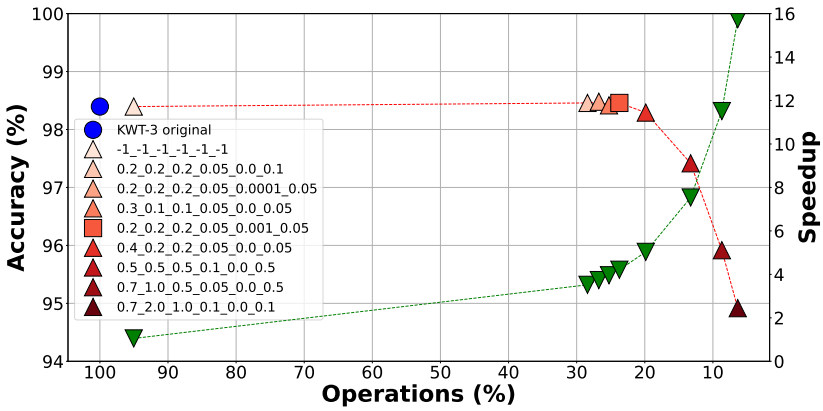
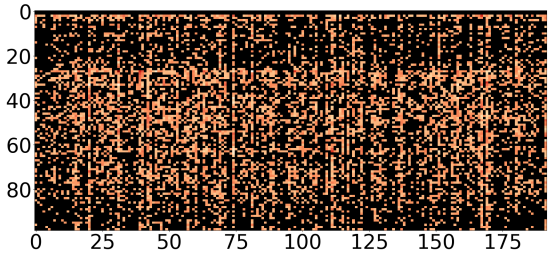


Figure 6.8: The results of running the original and the delta version of the KWT model. X-axis represents MACs, while the left and right y-axis correspond to the accuracy and speedup, respectively. Each of the red-shaded triangles (and a square) in the legend is annotated with thresholds used during the experiment in order: $\theta_{\mathbf{X}}$, $\theta_{\mathbf{Q}}$, $\theta_{\mathbf{K}}$, $\theta_{\mathbf{QK^T}}$, $\theta_{\mathbf{softmax}}$, and $\theta_{\mathbf{head_{1..k}}}$

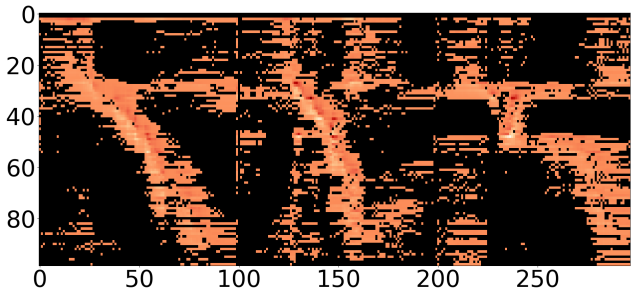
The red and green triangles represent our delta KWT-3 model with regard to accuracy and speedup, respectively. The inference time gains for the MHSA range from $\sim 1.05x$ to $\sim 16x$, and there is no accuracy degradation down to $\sim 23.7\%$ MACs (4.2x speedup). Moreover, some of the configurations even slightly outperform the original KWT-3 (98.46%, 98.48%, and 98.42%). Decreasing the accuracy by only 0.1% results in further speedup of $5x$. Moreover, if the accuracy requirements can be relaxed by 1-4%, the MHSA inference becomes faster by $7.5 - 15.7x$, which translates to 86.73-93.65% of skipped MACs. Table 6.2 shows the % of executed MHSA operations for one instance of each keyword category, averaged across the layers. The configuration (0.2_0.2_0.2_0.05_0.001_0.05) used to obtain the results is represented with a square in Figure 6.8. Although the MAC percentage naturally fluctuates for keywords within the same group, the objective of the table is to provide a general overview of how much operations are approximately performed in each of the parts. We can observe that $\sim 60 - 70\%$ of $XW_{Q,K,V}$, $90 - 95\%$ of QK^T, $73 - 79\%$ of $softmaxV$, and $83 - 87\%$ of $head_{1..k}W_P$ are discarded, which sums up to $70 - 77\%$ of skipped operations for the entire model. To visualize the savings, Figure 6.9 shows the delta values of the input data X and the softmax output of the 7th layer of a keyword *right* (same instance as used in Table 6.2). One special case are the instances from the *_silence_* class, that have the amount of discarded computations very close to the theoretical maximum defined in Section 6.5.4. Figure 6.10 shows the *_silence_*

Table 6.2: Percentage of executed MACs averaged across the layers for one instance of each keyword category. The configuration is: $\theta_{\mathbf{X}} = \mathbf{0.2}$, $\theta_{\mathbf{Q}} = \mathbf{0.2}$, $\theta_{\mathbf{K}} = \mathbf{0.2}$, $\theta_{\mathbf{QK^T}} = \mathbf{0.05}$, $\theta_{\mathbf{softmax}} = \mathbf{0.001}$, and $\theta_{\mathbf{head_{1..k}}} = \mathbf{0.05}$.

Keyword	$\mathbf{XW_{Q,K,V}}$	$\mathbf{QK^T}$	softmaxV	$\mathbf{head_{1..k}W_P}$	Total
silence	3.02	0.08	2.4	2.02	2.46
unknown	35.97	7.68	26.95	16.93	28.36
yes	31.82	6.26	24.06	16.31	25.32
no	36.98	8.93	25.42	14.27	28.41
up	33.48	6.28	23.64	13.32	25.68
down	29.88	5.08	22.38	14.24	23.46
left	38.73	10.09	26.95	16.97	30.26
right	33.52	6.68	25.65	16.55	26.59
on	31.13	5.64	21.21	13.02	23.9
off	39.5	10.17	26.68	15.11	30.33
stop	32.39	6.15	23.36	13.72	25.06
go	33.37	6.57	23.26	14.65	25.87



(a) Delta inputs



(b) Delta softmax outputs (three attention heads)

Figure 6.9: Deltas for a) inputs and b) softmax outputs for the 7th Transformer layer of the keyword **right**. Black color marks 0s.

input, for which only a small fraction of the deltas are non-zero, resulting in $97 - 99.9\%$ of skipped operations.

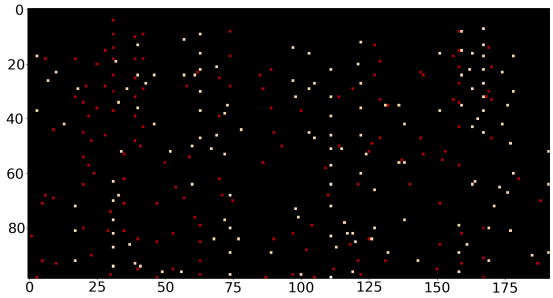


Figure 6.10: Deltas for inputs to the 7th Transformer layer for _**silence**_. Black color marks 0s.

A potential future improvement involves applying deltas on the input embedding matrix V. Although these cannot be exploited in multiplications with the softmax output due to the direction of computations (softmax output compensates for it), it would still contribute to V's data compression. Future work also explores the most optimal thresholds for each of the layers individually. This might further optimize the point where the accuracy starts dropping since a varying number of MACs is executed within each of the 12 layers.

6.8 Conclusion

This paper introduced a dynamic threshold-based pruning technique that drastically reduces MAC operations during inference. It was demonstrated on a keyword spotting task on the GSCD, where $\sim 80\%$ of operations in the MHSA can be discarded without degrading the accuracy. If the accuracy requirements can be slightly relaxed, a speedup factor of $\sim 5 - 16x$ is achieved. Our method thus helps to considerably decrease the computational complexity and enable significant data compression. The proposed technique can be exploited to enable an ultra-low power wake-up word detection front-end, that triggers a more powerful detector once a keyword is recognized. More generally, this work represents a stepping stone towards enabling the execution of Transformers on low-power devices.

Chapter 7

A Min-Heap-based Accelerator for Deterministic On-the-fly Pruning in Neural Networks

By Zuzana Jelčicová, Evangelia Kasapaki, Oskar Andersson, and Jens Sparsø [C4]

Abstract

This paper addresses the design of an area and energy efficient hardware accelerator that supports on-the-fly pruning in neural networks. In a layer of N neurons, the accelerator selects the top K neurons in every timestep. As K is fixed, the runtime of the pruned network is deterministic, which is an important property in real-time systems such as hearing aids. As a first contribution, we propose to use a min-heap for the top K selection due to its efficient data structure and low time complexity. As a second contribution, we design and implement a hardware accelerator for dynamic pruning that is based on the min-heap algorithm. The heap memory storing the top K neurons and their index is realized as a 3-port standard cell-based memory implemented with latches. As a third contribution, we evaluate the energy savings from pruning of a gated recurrent unit used in a neural network for speech enhancement. Our experiments demonstrate energy savings of $\sim$78% without degrading the SNR

improvement, and up to ∼93% while reducing the SNR improvement by 0.1 - 1.11 dB. Moreover, the overhead of the hardware accelerator constitutes negligible ∼0.5% of the total energy. The accelerator is implemented in a 22 nm CMOS process.

Index Terms— Min-heap, top K elements, determinism, neural networks, pruning, hearing aids

7.1 Introduction

Neural networks are powerful in solving various tasks, but they are also computationally expensive, requiring millions of operations per input. This is particularly challenging for always-on real-time embedded devices such as hearing aids that have tight constraints on area, memory, and power (a few mW). However, in deep neural networks a considerable portion of data processing might be omitted without degrading the overall performance [188]. Therefore, pruning can substantially reduce the amount of computations as well as memory accesses, and thus enable inference at the edge. Moreover, *dynamic* pruning enables networks to use parameters relevant for the current inputs, and to allocate a computational budget at runtime, as opposed to a *static* approach. Nonetheless, it often results in a varying number of computations from one timestep to another, making the execution time unpredictable. This challenge is addressed in [C2] that proposes a pruning algorithm based on a selection of the top K elements (neurons) out of N, which ensures determinism and worst-case execution time guarantees that are essential for real-time system. The work reports theoretical savings of up to ∼70% of computations and memory accesses for a pruned gated recurrent unit (GRU) [71] layer, without degrading the quality of the objective measures. Such savings also lead to a substantial speedup of the inference. However, the study is purely algorithmic and does not consider a hardware implementation including how the top K elements are identified, and hence no assessment of energy savings. These aspects are the topics addressed in this paper.

A straightforward solution to extract the top K elements is to sort the N elements first and select a K subset afterwards. Nevertheless, sorting performs more work than necessary and thus results in computational overhead as the top K elements do not have to be in an order. Instead, an efficient data structure commonly used to obtain a subset of the top K unordered elements is a binary heap [5] that can be implemented with minimal storage requirements, $O(K)$, using an array (Figure 7.1), with the worst-case time complexity of $O(N \log K)$.

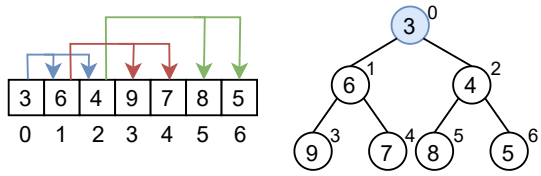


Figure 7.1: An example of a *min-heap* with size K=7 and a corresponding array representation.

In this work, we present an efficient low power implementation of a *min-heap*-based hardware accelerator to support data-driven deterministic pruning of neural networks on-the-fly. Our contributions are:

1) Applying a *min-heap* algorithm to select the top K neurons in every timestep. Several other works use a heap such as [220, 149], but to our knowledge, none of them for dynamic pruning of neurons during inference.

2) Designing and implementing a *min-heap*-based hardware accelerator, tested on heap sizes ranging from K=35 to K=111, with N=512. The heap memory is implemented as a 3-port (2R1W) standard cell-based memory (SCM) using latches.

3) Evaluating the energy savings of the pruning on a GRU layer in a three-layer network trained for speech enhancement (noise reduction). In order to relate our post-synthesis results to the algorithmic study, we use the same network and data and estimate the saved energy for a varying number of the top K elements linked to different SNR improvements in [C2].

The experiments demonstrate that the impact of the *min-heap*-based accelerator on the overall complexity is insignificant compared to the savings that can be achieved. Furthermore, our heap implementation for obtaining the top K elements can be used in any type of neural networks, such as convolutional networks and Transformers [146].

7.2 PeakGRU Pruning Algorithm

PeakGRU turns dense matrix-vector multiplications into highly-sparse matrix-vector multiplications, hence saving both multiply-accumulates (MACs) and memory accesses [C2]. This is accomplished by selecting the top K elements

out of N from the input vectors in every timestep. The current input vector $x(t)$, $h(t-1)$ and a vector of previously cached input values $\hat{x}(t-1)$, $\hat{h}(t-2)$ are subtracted, see (7.1) - (7.4), producing delta changes. The *hat states*, i.e., $\hat{x}(t-1)$, $\hat{h}(t-2)$, are initially 0. The magnitude of the delta change must be among the top K to be included in MACs, and the hat states are updated only for the top K elements. The MACs are then executed with the top K delta values, see (7.5) - (7.8), followed by generating the hidden state $h(t)$ as in the original GRU. One difference is a need for extra *delta memory vectors* M to keep track of the delta changes. Therefore, the hat and memory states along with the heap impose additional memory and computational overhead. However, this is counterbalanced with significant savings of MACs and memory fetches in bigger networks as shown in Section 7.6.

$$\hat{x}(t) = \begin{cases} x(t) & \text{if } |x(t) - \hat{x}(t-1)| \text{ among } K \\ \hat{x}(t-1) & \text{otherwise} \end{cases} \tag{7.1}$$

$$\hat{h}(t-1) = \begin{cases} h(t-1) & \text{if } |h(t-1) - \hat{h}(t-2)| \text{ among } K \\ \hat{h}(t-2) & \text{otherwise} \end{cases} \tag{7.2}$$

$$\Delta x(t) = \begin{cases} x(t) - \hat{x}(t-1) & \text{if } |x(t) - \hat{x}(t-1)| \text{ among } K \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

$$\Delta h(t-1) = \begin{cases} h(t-1) - \hat{h}(t-2) & \text{if } |h(t-1) - \hat{h}(t-2)| \text{ among } K \\ 0 & \text{otherwise} \end{cases} \tag{7.4}$$

$$M_r(t) = W_{xr}\Delta x(t) + W_{hr}\Delta h(t-1) + M_r(t-1) \tag{7.5}$$

$$M_u(t) = W_{xu}\Delta x(t) + W_{hu}\Delta h(t-1) + M_u(t-1) \tag{7.6}$$

$$M_{xc}(t) = W_{xc}\Delta x(t) + M_{xc}(t-1) \tag{7.7}$$

$$M_{hc}(t) = W_{hc}\Delta h(t-1) + M_{hc}(t-1) \tag{7.8}$$

$$r(t) = \sigma[M_r(t)] \tag{7.9}$$

$$eu(t) = \sigma[M_u(t)] \tag{7.10}$$

$$c(t) = tanh[M_{xc}(t) + r(t) \odot M_{hc}(t)] \tag{7.11}$$

$$h(t) = u(t) \odot h(t-1) + (1 - u(t)) \odot c(t) \tag{7.12}$$

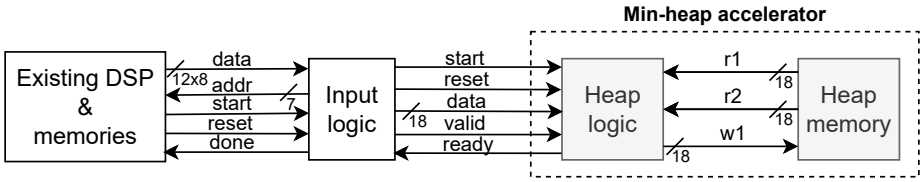


Figure 7.2: A high-level system overview. The *min-heap* accelerator is marked with a dashed rectangle, and only these modules are synthesized.

7.3 Binary Heap

A binary heap [5] is a binary tree structure where every level, except possibly the lowest one, is completely filled (Figure 7.1). The insertion of elements is done from left to right and level by level, ensuring a height-balanced tree. We employ a *min-heap* where each node is numerically smaller than or equal to its child nodes. If not, the elements must be swapped. The parent, left child, and right child nodes in an array implementation are always stored at indices $\lfloor (i-1)/2 \rfloor$, $(2*i)+1$, and $(2*i)+2$, respectively, where i corresponds to the index of the current node under processing. The leaf elements, i.e., nodes without children, have indices starting from and including $\lfloor K/2 \rfloor$. A heap with K nodes has $\lceil \log_2(K+1) \rceil$ levels and $O(\log K)$ complexity of the insert operation as there is at most one swap per level. There are two main heap functions that we will refer to as i) *heapify up* when a heap is being built and hence is not filled yet (bottom-up traversal), and ii) *heapify down* when a heap is already full (top-down traversal). *Heapify up* swaps elements until i) new element's parent is smaller than or equal to the element, or ii) the element reaches the root. The starting position of a new element is always at the next available leaf node. Once the *min-heap* is filled, *heapify down* is used to insert new elements from the top. Each new element is compared against the root that represents the smallest element in a heap. If the new element is smaller than or equal to the root, it is dropped, and no more action is required. If the new element is greater, it replaces the root and begins its traversal down the heap until i) its child is bigger than or equal to the element, or ii) it reaches the bottom and becomes a leaf node.

7.4 System Overview

A simplified overview of the whole system can be seen in Figure 7.2. The *min-heap* accelerator performs pruning only; the computations are performed

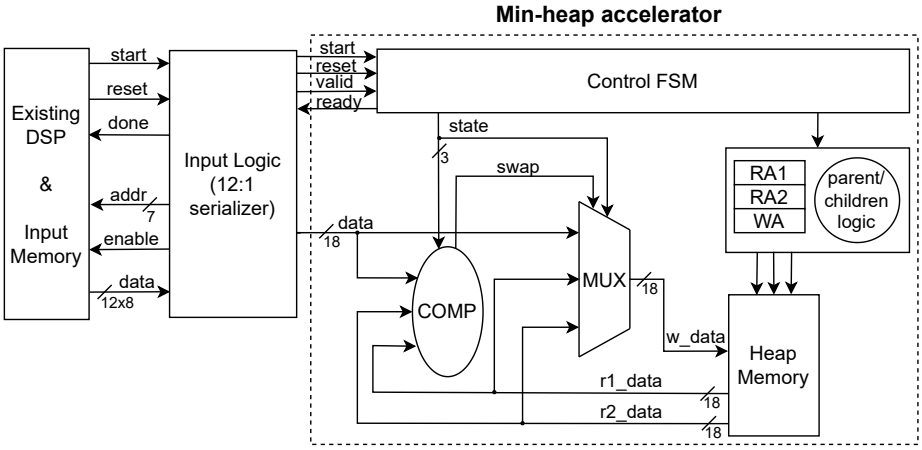


Figure 7.3: An overview of the control FSM and datapath of the *heap_logic* interacting with the *heap_memory* and the rest of the system. The control path is simplified for clarity.

by an existing digital signal processor (DSP) that contains a memory with a 96-bit vector interface [C1]. Further description of each logic block and the *heap_memory* is provided in the subsections below.

7.4.1 Input Logic

A more detailed view of the system is shown in Figure 7.3. The execution of the system is triggered with a *start* signal coming from the DSP. The *input_logic* has two main functions: 1) Working as a 12:1 serializer. The *input_logic* reads 96-bit vectors consisting of twelve 8-bit keyword elements from the *input_memory*. 2) Creating element-index pairs. Each element from the serialized input vector is concatenated with an index. The index is generated by a local counter and it represents the element's position among all N inputs ($[0, N-1]$). The index is needed for fetching corresponding weights during MAC operations. Once the first element-index pair is ready for processing, the *input_logic* triggers the *heap_logic* with another *start* signal.

The *input_logic* has a local counter to keep track of the number of inserted elements. When all N elements have been processed, a *done* signal is set high to notify the DSP that the selection of the top K elements has been completed.

7.4.2 Heap Logic

The communication between the *input_logic* and *heap_logic* is via an AXI-style ready/valid streaming interface, where a high *valid* signal indicates that an element-index pair is available for processing, and a high *ready* signal informs about the ability to accept a next pair. Once a new element-index pair is received, it is passed to the comparison (COMP) logic where the element is compared against other elements already stored in the heap.

When the *heap_logic* executes the *heapify up* function for the first element, the element is directly inserted into the heap as the root. The subsequent elements are then always compared against their parent node. Once the heap is filled, the execution switches to the *heapify down* function. The first comparison is always done against the root element. We use an extra register for storing the root locally in order to avoid memory fetches, as the root is the most accessed element in the heap. If the new element is smaller than the root, the element will be immediately dropped and no more processing will be performed with the element. The addresses (RA1, RA2, WA), i.e., heap locations, for reading and writing data are generated based on the equations for parent and child indices described in Section 7.3. Moreover, in our implementation a new element is stored in a memory only when its correct location is found in order to reduce memory accesses.

7.4.3 Heap Memory

A binary heap can be implemented with two major options for on-chip memories: 1) an addressable register array of flip-flops or latches (referred to as SCMs), and 2) an SRAM macro. SRAM macros usually require a full-custom implementation, and the macros have to be developed for each technology using a dedicated memory compiler [46]. The SRAM peripheral circuitry causes overhead for smaller memories (< 8 kb) [127, 58] that are used in this work (< 2 kb). Moreover, using SCMs enables operation at supply voltages in the near/sub-threshold region where a standard 6T-SRAM does not operate reliably [58]. In addition, SCMs can be easily ported to other technologies and merged with logic blocks, realized with any number of read/write ports, read/write-logic style, word count, and wordlength based on requirements. SCMs are therefore selected for our accelerator. Our 3-port SCM design is based on [101]. It is realized with latches as a storage element since latches use less area and power compared to flip-flops. Read and write in our design have delay of one clock cycle.

7.5 Experimental Setup

The three-layer neural network model used in our evaluations is the same as in [C2], where it is a part of a setup that simulates internals of a hearing aid. The network comprises of FC-GRU-FC layers, each having 512 output neurons. The first FC layer uses a ReLU activation function on its output. The 512 outputs of the last FC layer correspond to postfilter gains that are in the end applied on the original signal. During pruning, a GRU layer is substituted with PeakGRU.

The input data is obtained by applying a 1024-point FFT on a 20 kHz noisy speech y, i.e., clean speech and noise added together ($y = s + n$). The 512 resulting frequency sub-bands then serve as an input to the network, with a new frame being provided every 25 ms. In this paper, the speech was obtained from the *VCTK Corpus* [204] and *Akustiske Database for Dansk* [48], and the noise from [128] (*BusyStreet* background). A frame from approximately the middle of a 30-second long noisy speech was used for the experiments. The data is statically quantized to 8 bits using a symmetric mode, i.e., the quantization range is set as an absolute value between min/max. Since N=512 inputs and outputs are used for a GRU, 9 bits are needed for indexing, making the final element-index wordlength 17 bits.

The heap memory is implemented as a latch-based SCM with three ports to reduce the number of cycles spent on the *heapify down* process since both child nodes can be fetched simultaneously. The min-heap accelerator is synthesized in 22 nm CMOS technology using four different heap sizes (K): 111, 83, 65, and 35, corresponding to 21.68%, 16.21%, 12.7%, and 6.84% of the original N=512 elements. The sizes were determined based on the acceptable SNR degradation due to pruning [C2] that are further described in Section 7.6.2. It is also important to emphasize that our focus is not high performance but very low-power inference at edge devices instead. Therefore, power simulations were run at 2 MHz and 0.6 V.

7.6 Results and Discussion

Two types of results are presented in this section: i) heap results in terms of total area (kgates), latency (clock cycles), and dissipated energy (nJ), and ii) a comparison of dissipated energy between the full and the pruned *GRU* hidden layer.

Table 7.1: Results of synthesis and power simulations. The area is the same for both heaps as K_x==K_h.

			Area (kgates)	Energy X (nJ)	Energy H (nJ)	Cycle count X	Cycle count H
Heap size K	35	Logic	2.1	0.16	0.17	1,007	982
		Memory	5.3	0.18	0.15		
	65	Logic	2.1	0.20	0.20	1,257	1,261
		Memory	9.8	0.45	0.44		
	83	Logic	2.1	0.25	0.23	1,400	1,365
		Memory	12.4	0.56	0.53		
	111	Logic	2.1	0.29	0.28	1,556	1,573
		Memory	16.5	0.82	0.79		

7.6.1 Heap Results

Table 7.1 shows the results of synthesis and power simulations for the x and the h heaps, where *Logic* stands for *heap_logic* and *Memory* for *heap_memory*. While the area of the *heap_logic* is 2.1 kgates, the storage requirements range from 5.3 - 16.5 kgates, making the area of the *heap_logic* negligible for all the configurations explored. The total energy per run is obtained by multiplying power and latency. Both energy and cycle count are almost the same for both heaps.

7.6.2 Energy Savings in a GRU Layer

The unprocessed speech has an SNR of 4.39 dB. The original network with a full GRU layer results in an SNR improvement of 8.11 dB relative to unprocessed speech. When applying the top K selection (where K_x==K_h), the same performance is maintained down to K=111 elements. Processing fewer elements, e.g., 83, 65, 35, reduces SNR improvement to 8.0 dB, 7.8 dB, and 7.0 dB, respectively [C2]. These SNR improvement values become our energy evaluation points (see Figure 7.4).

To put the results in perspective, we relate the cost of the accelerator to the savings of MACs and memory accesses, executed by the DSP. The savings are obtained by only processing the top K elements. To evaluate the benefit of using the *min-heap* accelerator, we therefore need to determine the energy dissipation of the DSP and memory during execution. To err on the safe side and obtain results that are not biased from the use of a specific DSP, we consider only the energy for performing MAC operations and memory accesses. All data is based on the same 22 nm CMOS technology.

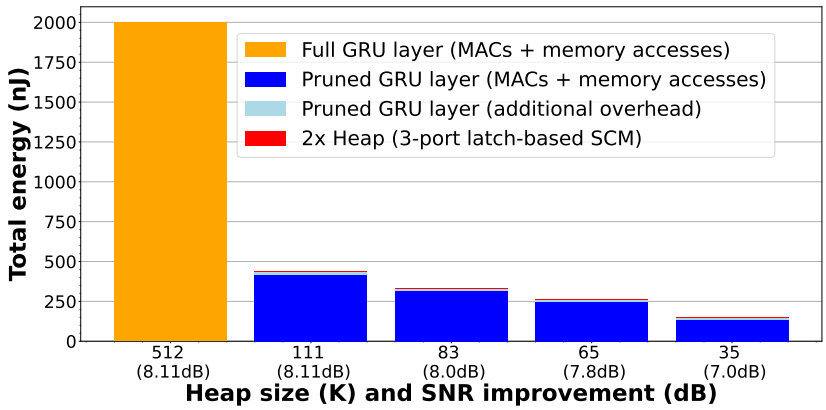


Figure 7.4: A comparison of dissipated energy for the full and pruned GRU layer with different heap sizes using a 3-port latch-based SCM.

Memory accesses cover fetching 8-bit weights (SRAM), 8-bit inputs for the GRU layer (SRAM), 17-bit inputs (8-bit elements, 9-bit indices) for the pruned layer (SCM), and storing 512 8-bit results (SRAM). The overhead introduced by fetching/storing the $\hat{x}$, $\hat{h}$, and M states (SRAM) necessary for pruning is also considered (light blue bars in Figure 7.4). The wordlength of the intermediate ($\hat{x}$, $\hat{h}$) and memory states (M) is assumed to be 8 and 24 bits, respectively. The overhead also contains initial fetching of 512 8-bit x and h elements to perform the top K selection for both inputs and hidden states.

The total energy (excluding biases) required for executing a single GRU timestep is thus $\sim 2\,\mu J$, as shown with a yellow bar in Figure 7.4. There is no degradation for the pruned network down to 111 elements, translating to $\sim 436.94\,\mathrm{n}J$, which is only $\sim 21.87\,\%$ of the original GRU energy. For 83, 65, and 35 elements, only $330.25\,\mathrm{n}J$, $261.77\,\mathrm{n}J$, and $147.48\,\mathrm{n}J$ are required, respectively. This corresponds to an energy reduction of 6.05-13.55×. The energy dissipated on the two heap setups together is $2.17\,\mathrm{n}J$, $1.57\,\mathrm{n}J$, $1.29\,\mathrm{n}J$, and $0.66\,\mathrm{n}J$ (red bars), while $\sim 14.19\,\mathrm{n}J$, $14.08\,\mathrm{n}J$, $14.01\,\mathrm{n}J$, and $13.9\,\mathrm{n}J$ on the *PeakGRU* algorithm overhead (light blue bars). These energy numbers are very small and thus practically invisible in Figure 7.4. Therefore, the accelerator overhead is negligible, and the energy savings are thus proportional to the amount of the skipped computations. Such savings can enable a standard DSP to perform neural network inference, which might otherwise not be an option due to the computational load. Moreover, if the DSP has the necessary capacity, the energy dissipation reduced to 22%, corresponding to no SNR degradation, offers a possibility to execute either i) more complex networks, or ii) four equally complex networks. Our technique

could also be used in systems where big hardware accelerators for dedicated neural network processing are not an affordable solution.

7.7 Related Work

A hybrid heap priority queue architecture [76] developed for an FPGA stores elements of each level of the heap in a separate on-chip BRAM. This method enables parallel access to the elements, resulting in an insert operation in $O(1)$ time. Nevertheless, the heap needs to precalculate a path from the leaf node to the root and fetch all the elements in the path from corresponding BRAMs. This approach increases the number of memory accesses that are among the most costly operations [73]. In [35] the authors propose heap management algorithms that can be pipelined to achieve high throughput. The downside of this approach is increased hardware complexity. A bitonic sort-based algorithm called bitonic-top-k is introduced in [171]. However, this massively parallel algorithm targets GPUs (and GPU-based frameworks like TensorFlow) where its complexity $O(N \log^2 K)$ can be hidden. The closest to our use case is [248] that prunes data in Transformer neural networks on-the-fly based on importance scores. It consists of two parallel top-K engines with a quick-select module that randomly selects a pivot to partition the input array, running iteratively until the Kth largest element is found. Although the average time complexity of quick-select is $O(N)$, it has considerably higher memory requirements $O(N)$ than a heap $O(K)$, with a worst-case time complexity of $O(N^2)$ compared to heap's $O(N \log K)$. All the presented designs offer a solution for the top K selection, however, their algorithmic/hardware overhead is too high for our budget and application.

7.8 Conclusion

The paper presented a hardware accelerator for on-the-fly pruning in neural networks. The accelerator selects the top K neurons within a layer of N neurons. A constant K value results in a deterministic and bounded execution time, an important property in real-time systems such as hearing aids. Firstly, we identified a min-heap-based algorithm as an attractive solution for this task due to its efficient space and time complexity. Secondly, we designed an accelerator using a 3-port latch-based SCM to store the K elements. Finally, we demonstrated that the energy dissipated by the accelerator is negligible ($\sim 0.5\%$ of the total energy) in comparison to the energy saved by pruning (up to 93%), while the

corresponding degradation of the SNR improvement ranges from 0 - 1.11 dB. We evaluated the min-heap accelerator on a GRU, but it can be applied in other types of neural networks. The accelerator is implemented in a 22 nm CMOS process.

Chapter 8

PeakEngine: A Deterministic On-the-fly Pruning Neural Network Accelerator for Hearing Instruments

By Zuzana Jelčicová, Evangelia Kasapaki, Oskar Andersson, and Jens Sparsø [J1]

Abstract

Recurrent neural networks are well-suited for sequential tasks such as speech enhancement. However, their performance comes with high computational complexity and latency. This impedes their deployment to battery-powered and resource-constrained hearing instruments that need to operate for 16-18 hours daily at only a few milliwatts. In this paper, we introduce *PeakEngine*, a configurable ASIC accelerator that decreases the amount of computations and memory accesses, and thus latency, in a gated recurrent unit by means of adaptive inference. The reduction is achieved by on-the-fly pruning that selects the top K elements based on magnitudes of delta changes across timesteps from both input and hidden state sequences. Since K is constant, it results in a deterministic execution time. The experiments show that *PeakEngine* dissipates 11.83 μJ per

inference for the baseline (unpruned) network and only 4.14-5.04 μJ for the pruned networks, with maximum acceptable degradation to no degradation in the improvement in audio quality and intelligibility. Moreover, the inference is on average sped up 2.2-2.97×, hence meeting the real-time requirements imposed by a hearing instrument application. To the best of our knowledge, *PeakEngine* is the first ASIC accelerator for deterministic and dynamic pruning in recurrent neural networks targeting hearing instruments and speech enhancement.

Index Terms— PeakEngine, speech enhancement, hearing instruments, RNNs, min-heap, top K, dynamic pruning, determinism.

8.1 Introduction

Understanding speech-in-noise is critically important yet one of the biggest problems for hearing instrument (HI) users [152, 23, 83]. *Speech enhancement (SE)* addresses this issue by attenuating the background noise, thus improving speech quality and/or intelligibility. Traditional methods for SE are very advanced but most of them heavily rely on *domain knowledge* [163], such as requiring *unknown* a priori Signal-to-Noise Ratio (SNR), or assuming that speech and noise are uncorrelated. This is not the case for deep neural networks (DNNs) that learn directly from the data and have already demonstrated their superior performance for SE over the traditional methods [9, 64, 63, 42].

Recurrent neural networks (RNNs) are an attractive solution for SE due to their powerful capabilities of processing sequential data. This is possible thanks to their feedback connection that is shared between timesteps. The feedback connection enables RNNs to retain information, making them superior for applications that use data with temporal structures, such as video processing [108], natural language processing [81], translations [106], and speech recognition [61]. The two most typical variants of RNNs are a Long Short-Term Memory (LSTM) [20] and a Gated Recurrent Unit (GRU) [71].

At the same time, RNNs suffer from high computational complexity and latency. Since the sequences are dependent on each other, they cannot be processed simultaneously, which prevents exploiting parallelism. Moreover, the dominant operation in RNNs is a matrix-vector multiplication that grows quadratically with the number of hidden units, which consequently increases the amount of memory accesses, latency, and power consumption. As shown in [73], memory

accesses dominate over arithmetic operations in terms of energy dissipation, and thus become the biggest bottleneck in RNNs. All these issues impede the deployment and execution of RNNs in low-power and resource-constrained HIs that operate with a few milliwatts (mW) and require audio latency below 30 ms [30, 259] to ensure optimal sound quality and comfort. Moreover, they need to last around 16-18 hours every day [44] on a miniature-sized battery. Therefore, in order to enable RNN inference in HIs, it is crucial to reduce multiply-accumulates (MACs) and corresponding memory accesses that are the main sources of power consumption and latency.

A typical approach to decrease the number of memory accesses (and consequently MACs) is *static pruning*. Static pruning compresses the size of the original model by permanently removing weights that contribute very little to the final outcome. A myriad of pruning methods have been proposed throughout the years, as surveyed in [209, 243]. Some of the RNN approaches include magnitude-based and load-balance-aware pruning of weights using an empirical threshold [129], structured pruning with a learnable threshold [214], and iterative compression of randomly selected weight blocks [221]. Static pruning of RNNs is generally challenging as a recurrent unit is shared across all the timesteps. Compressing the unit thus impacts all the steps in the sequence. Permanently discarding weights destroys the original network structure which may lead to decreased capability, representation power, and efficiency of a model [243, 253]. It has also been shown that small SE networks have difficulties learning the necessary relationship between the noisy features and the target SNRs [31].

Dynamic pruning, on the other hand, is a data-driven approach, where computations are conditioned on the input at runtime. Dynamic pruning on its own does not reduce the model size. However, it counterbalances this issue with several other advantages [253] that are missing in static models such as *1) Efficiency* - allocating computations on demand at runtime, *2) Representation power* - identifying "easy" and "hard" samples and hence computational redundancy, *3) Adaptiveness* - achieving a desired trade-off between accuracy and efficiency at runtime, and *4) Generality* - seamlessly adapting to a wide range of applications.

An example of dynamic pruning includes an accelerator in [248] that prunes attention layers in transformer neural networks using the top cumulative importance scores. Furthermore, *temporal sparsity* based on a threshold is exploited in several works. These include [202] that skips state updates using also a skip-criterion, [182] that prunes hidden state vectors in LSTMs, and [138] implemented as GRU [215] and LSTM [252] accelerators, where dense state vectors are substituted with their sparse delta versions obtained as a temporal difference across two adjacent timesteps. However, the actual computation time in all of these state-of-the-art threshold-based dynamic pruning approaches is *unpredictable*, which is an issue for real-time systems like HIs.

To the best of our knowledge, none of the existing accelerators supports dynamic pruning of RNNs that results in deterministic inference, targeting HIs and relevant use cases such as SE. Our accelerator fills this gap and offers adaptive yet computationally predictable inference that is absent in the state-of-the-art RNN accelerators. The main contributions of this work are:

1. *Min-heap engine*, a low-area and energy hardware unit that selects the top K elements in a stream of N elements, where $N > K$. The engine is used to support our deterministic *PeakRNN* [C2] pruning algorithm. Moreover, its application is versatile and not limited to neural networks only.
2. *PeakEngine*, the first ASIC accelerator for HIs that enables deterministic on-the-fly pruning of RNNs. *PeakEngine* is *configurable* and *portable* thanks to its standard interfaces, and it encompasses the *Min-heap engine*.
3. A thorough investigation of *PeakEngine* for different K values with regard to saved energy and reduced latency.
4. A bit-accurate *software framework* for parameter space exploration of different Q formats, wordlengths, and K values for GRU-based and fully connected neural networks.

The rest of the paper is structured as follows. Section 8.2 provides the necessary background for the *PeakRNN* algorithm, while Section 8.3 describes the algorithm itself. Section 8.4 introduces an emulated HI setup used for experiments with SE. Information about the DNN architecture, datasets, and training is stated in Section 8.5. Section 8.6 details the *PeakEngine* design and Section 8.7 talks about the software framework. The experimental setup is described in Section 8.8. Section 8.9 presents and discusses the results as well as comparisons to state-of-the-art works. Finally, Section 8.10 concludes the paper.

8.2 Background

This section presents the original GRU [71] and DeltaGRU [138] algorithms on top of which our modified version called PeakGRU [C2] is built.

8.2.1 GRU Algorithm

A GRU (see Figure 8.1) has a feedback connection called a *hidden state* $h(t)$ that maintains both short- and long-term dependencies. It is controlled by two

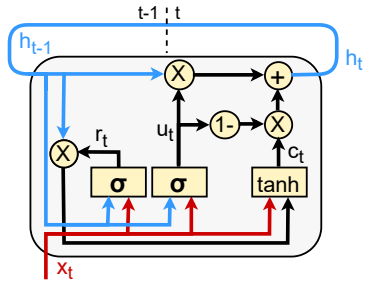


Figure 8.1: An illustration of a GRU. The multiplications with weight matrices and summations of the partial dot products are excluded for clarity.

internal *gate* mechanisms, *reset* $r(t)$ and *update* $u(t)$ gate, that regulate the flow of information in the unit using sigmoid (σ) activation function, see (8.1)-(8.2). A GRU processes two types of inputs: i) an input sequence $x(t)$ for the current timestep, and ii) a sequence of previous hidden states $h(t-1)$. The sequences are multiplied with their respective *weight matrices* W and summed with *bias values* b. The reset gate then determines how much of the past information should be forgotten. The relevant past information, on the other hand, is stored in a *candidate state* $c(t)$ that applies hyperbolic tangent ($tanh$) activation function, see (8.3). The update gate decides on the importance of both the past information, $h(t-1)$, and the new information, $c(t)$, as shown in (8.4). Finally, a new hidden state $h(t)$ that also serves as an output for the current timestep is generated.

$$r(t) = \sigma(W_{xr}x(t) + W_{hr}h(t-1) + b_r) \tag{8.1}$$

$$u(t) = \sigma(W_{xu}x(t) + W_{hu}h(t-1) + b_u) \tag{8.2}$$

$$c(t) = tanh(W_{xc}x(t) + r(t) \odot (W_{hc}h(t-1)) + b_c) \tag{8.3}$$

$$h(t) = u(t) \odot h(t-1) + (1 - u(t)) \odot c(t) \tag{8.4}$$

Based on (8.1)-(8.3), the number of matrix-vector multiplications (and corresponding memory fetches) in a GRU can be expressed as

$$3 \times (XH + H^2), \tag{8.5}$$

where X and H correspond to the dimensions of input sequences $x(t)$ and $h(t-1)$, respectively. The computational complexity grows quadratically with respect to the $h(t-1)$ that consequently increases the number of memory accesses and hence power.

8.2.2 DeltaGRU Algorithm

DeltaGRU, also called *Delta Networks* [138], dynamically reduces the number of MAC operations and memory fetches by transforming a dense matrix-vector multiplication into a highly-sparse matrix-vector multiplication in every timestep, as shown in (8.6)-(8.17).

$$\hat{x}(t) = \begin{cases} x(t) & \text{if } |x(t) - \hat{x}(t-1)| > \Theta \\ \hat{x}(t-1) & \text{otherwise} \end{cases} \tag{8.6}$$

$$\hat{h}(t-1) = \begin{cases} h(t-1) & \text{if } |h(t-1) - \hat{h}(t-2)| > \Theta \\ \hat{h}(t-2) & \text{otherwise} \end{cases} \tag{8.7}$$

$$\Delta x(t) = \begin{cases} x(t) - \hat{x}(t-1) & \text{if } |x(t) - \hat{x}(t-1)| > \Theta \\ 0 & \text{otherwise} \end{cases} \tag{8.8}$$

$$\Delta h(t-1) = \begin{cases} h(t-1) - \hat{h}(t-2) & \text{if } |h(t-1) - \hat{h}(t-2)| > \Theta \\ 0 & \text{otherwise} \end{cases} \tag{8.9}$$

$$M_r(t) = W_{xr}\Delta x(t) + W_{hr}\Delta h(t-1) + M_r(t-1) \tag{8.10}$$

$$M_u(t) = W_{xu}\Delta x(t) + W_{hu}\Delta h(t-1) + M_u(t-1) \tag{8.11}$$

$$M_{xc}(t) = W_{xc}\Delta x(t) + M_{xc}(t-1) \tag{8.12}$$

$$M_{hc}(t) = W_{hc}\Delta h(t-1) + M_{hc}(t-1) \tag{8.13}$$

$$r(t) = \sigma[M_r(t)] \tag{8.14}$$

$$u(t) = \sigma[M_u(t)] \tag{8.15}$$

$$c(t) = tanh[M_{xc}(t) + r(t) \odot M_{hc}(t)] \tag{8.16}$$

$$h(t) = u(t) \odot h(t-1) + (1 - u(t)) \odot c(t) \tag{8.17}$$

The dense-to-sparse transformation is achieved by applying a threshold Θ on the magnitude of input change across adjacent timesteps, see (8.6)-(8.9). The magnitude of change, i.e., the absolute value of the *delta change*, is calculated by subtracting previously cached inputs, $\hat{x}(t-1)$ and $\hat{h}(t-2)$, from the current inputs, $x(t)$ and $h(t-1)$. The initial value of *hat states* $\hat{x}(t-1)$ and $\hat{h}(t-2)$ is 0. The magnitudes above Θ are then selected, and their actual subtraction results are used in MACs as $\Delta x(t)$ and $\Delta h(t-1)$, as shown in (8.10)-(8.13). The hat state updates are subsequently only performed for the magnitudes above Θ. *DeltaGRU* needs additional *delta memory states* M to track the delta changes across timesteps. Finally, a new hidden state $h(t)$, see (8.17), for the current timestep is generated the same way as shown in (8.4) in Section 8.2.1.

The hat and memory states inflict memory and computational overhead. However, this is compensated with substantial reduction of MACs and memory fetches as demonstrated with *PeakGRU* in Section 8.9.

8.3 PeakGRU Algorithm

Our *PeakGRU* algorithm builds on the top of *DeltaGRU*. This section describes the differences between the two approaches, and explains how *PeakGRU* is efficiently realized in hardware.

8.3.1 Top-K Magnitudes

PeakGRU and *DeltaGRU* share the underlying computations in (8.6)-(8.17). The difference between the two algorithms is in the method of how the Δ elements are obtained in (8.8)-(8.9). While *DeltaGRU* uses a threshold-based approach, *PeakGRU* selects the top K_x and the top K_h subsets of elements from $x(t)$ and $h(t-1)$ sequences, respectively, as shown in (8.18)-(8.21).

$$\hat{x}(t) = \begin{cases} x(t) & \text{if } |x(t) - \hat{x}(t-1)| \text{ among } K_x \\ \hat{x}(t-1) & \text{otherwise} \end{cases} \tag{8.18}$$

$$\hat{h}(t-1) = \begin{cases} h(t-1) & \text{if } |h(t-1) - \hat{h}(t-2)| \text{ among } K_h \\ \hat{h}(t-2) & \text{otherwise} \end{cases} \tag{8.19}$$

$$\Delta x(t) = \begin{cases} x(t) - \hat{x}(t-1) & \text{if } |x(t) - \hat{x}(t-1)| \text{ among } K_x \\ 0 & \text{otherwise} \end{cases} \tag{8.20}$$

$$\Delta h(t-1) = \begin{cases} h(t-1) - \hat{h}(t-2) & \text{if } |h(t-1) - \hat{h}(t-2)| \text{ among } K_h \\ 0 & \text{otherwise} \end{cases} \tag{8.21}$$

This modification is illustrated in Figure 8.2, where $\Delta x(t)$ is calculated across three timesteps, along with the propagation of $\hat{x}(t-1)$. The matching colors in $x(t)$ and $\hat{x}(t)$ vectors denote the updates of $\hat{x}(t-1)$ with $x(t)$. The same visualization applies to $\Delta h(t-1)$ and $\hat{h}(t-2)$.

Since the number of elements to process is known in advance, the actual computation time in PeakGRU is *deterministic*. This is an important characteristic for

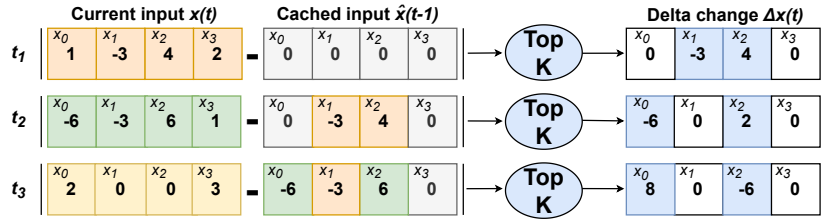


Figure 8.2: An example of calculating a *sparse delta vector* along with the hat states across three timesteps (t_1-t_3) for $x(t)$ using *PeakGRU*. X=4, K_x=2, and the black vertical lines around the subtraction represent absolute value.

real-time and resource-constrained embedded devices, where the worst-case execution guarantees are imperative. Moreover, since the algorithm selects elements based on the range of inputs and not a threshold, it is robust to the variations in data compared to the threshold-based approaches such as *DeltaGRU* [138].

8.3.2 Top-K Selection

Processing only the top K elements offers numerous advantages but it also imposes a challenge of how the elements should be selected. For instance, sorting the N elements first and selecting a top K subset afterwards would cause unnecessary computational and memory overhead since the order of elements is irrelevant. Instead, a *binary heap* [5] can be used for efficient selection, imposing i) minimal storage requirements $O(K)$ when implemented as a simple array, and ii) low worst-time computational complexity $O(N \log K)$.

A binary heap is a *complete* binary tree, where all the levels ($\lceil \log_2(K+1) \rceil$) are fully filled, except possibly the deepest one, as shown in Figure 8.3. The elements are inserted into the heap from left to right and level by level, with the worst-time complexity of $O(\log K)$ per element, which corresponds to a swap across all the heap levels. The nodes in the deepest level (leaves) start at index $\lfloor K/2 \rfloor$. To support PeakGRU, we employ a *min-heap* binary tree, where all the nodes within a level are numerically greater than or equal to their parent nodes in the level above. The parent, left, and right child nodes in an array implementation have indices $\lfloor (i-1)/2 \rfloor$, $(2*i)+1$, and $(2*i)+2$, respectively, where i is the index of the current node.

In our design, the very first data is directly inserted into the min-heap. Subsequent data becomes a leaf and is compared against its parent node. It is traversed up

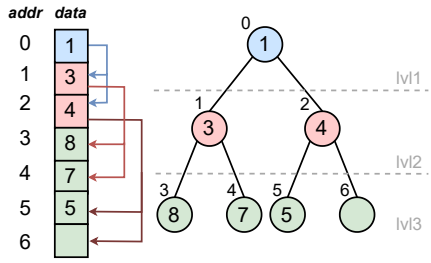


Figure 8.3: An example of a *min-heap* with K=7 and its memory implementation as a simple array, where the last level is not fully filled (index 6).

the tree (swapped) until numerically smaller than the parent, or the *root* - the smallest element in the min-heap - is reached. Once the min-heap is full, the data is inserted from the top. The first comparison is hence always done against the *root*. If the new data is smaller than/equal to the *root*, it is immediately skipped, and no more comparisons are needed. Otherwise, it replaces the *root* and is swapped with one of its child nodes until it is greater or becomes a leaf node.

All these operations are executed by the proposed *Min-heap engine* that is used to support the PeakGRU algorithm. However, the engine design is not tied to the algorithm or neural networks, and it could be applied in many other contexts such as priority queues [76] and data compression [37].

8.3.3 Top-K Storage

An on-chip implementation of the min-heap memories can be realized with either 1) a standard-cell based memory (SCM), i.e., an addressable array of flip-flops or latches, or 2) an SRAM macro. Since the memory requirements for our min-heaps are small (3.25 kb each) and below the break-even point with SRAMs [127, 58], SCMs (with latches) are selected for storing the top K_x and K_h values. Furthermore, we use 3-port (2R1W) SCMs to parallelize min-heap computations (fetching of the child elements). Our SCM design is based on [101].

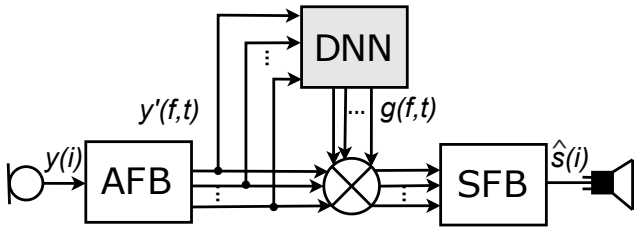


Figure 8.4: The overview of the gain-based SE system in a HI used for the experiments, where the DNN generates postfilter gains $g(f,t)$.

8.4 Hearing Instrument Application

Deterministic on-the-fly pruning supported in *PeakEngine* is demonstrated in a SE task. The subsections below present details about the emulated HI setup and objective metrics used for evaluating the performance of the system. The setup is based on our previous work on the PeakGRU [C2] algorithm.

8.4.1 Speech Enhancement System

A DNN architecture described in Section 8.5.1 is used in a *gain-based single-microphone SE* system that simulates the inner parts of a HI. The system is illustrated in Figure 8.4, where a DNN substitutes a traditional signal processing-based module for generating postfilter gain values $g(f,t)$, with f and t corresponding to a frequency bin and time-frame, respectively. The input for a DNN is generated by preprocessing samples (i) of a noisy 20 kHz single-microphone signal $y(i)$ in an Analysis Filter Bank (AFB), where the noisy signal is clean speech $s(i)$ corrupted with noise $n(i)$. The AFB applies a 1024-point FFT and a square-root Hanning window, downsampling the time-domain microphone signal to 40 Hz and producing a time-frequency representation $y\prime(f,t)$. In our specific scenario, the result is a new frame of 512 frequency bin values every 25 ms without overlapping. The $y\prime(f,t)$ values are passed to a DNN that generates 512 postfilter gain values $g(f,t)$. The gain values are applied on the noisy signal $y(i)$ to obtain an estimate $\hat{s}(i)$ of the clean speech magnitude spectrum $s(i)$. Finally, the Synthesis Filter Bank (SFB) reconstructs the time-domain signal and forwards the result to the speaker.

8.4.2 Objective Measures

To evaluate the performance of a DNN under a varying number of the top K elements and full- and reduced-precision, the HI system in Figure 8.4 is extended with *postprocessing*. The postprocessing phase saves the enhanced signals from the SFB along with their corresponding clean speech and noise that are used for calculating three objective metrics:

- *Perceptual Evaluation of Speech Quality (PESQ)* - evaluates the *quality* of noisy speech by estimating *Mean Opinion Score (MOS)*. MOS is judged using a discrete scale of 1 (bad) to 5 (excellent). The result is an average of these ratings. *Mean Opinion Score - Listening Quality Objective (MOS-LQO)* [25] is used as a unit.
- *Speech Test of Objective Intelligibility (STOI)* - evaluates the *intelligibility* of noisy speech (ability to recognize word, syllables, etc.), and it thus produces a scalar value in a range of 0 to 1, where 1 corresponds to fully intelligible speech.
- *Signal-to-Noise Ratio (SNR)* - compares the level of a desired signal to the level of background noise, expressed in *dB*s.

PESQ and STOI approximate human ranking and thus replace time-consuming and expensive listening tests [50].

8.5 DNN for Speech Enhancement

After introducing the HI and SE setup, we can now focus on the DNN itself, the selected architecture, datasets, as well as training procedure. The architecture and datasets were also used in our original algorithmic study [C2].

8.5.1 DNN Architecture

The neural network used in the experiments consists of three layers. The first and the last layer are fully connected (FC), and the hidden layer is a GRU. Each of the layers has 512 output neurons. A non-linearity is introduced after the first FC layer with the ReLU activation function. The GRU layer is swapped with a PeakGRU layer to evaluate and compare the performance and computational

savings of the two methods against each other. A GRU accounts for a significant part (75%) of the vectorized operations, i.e., MACs and memory accesses, while the remaining 25% accounts for both of the two FC layers.

8.5.2 Dataset

A noisy DNN input, y, is created by combining 30-second segments of clean speech, s, with noise, n, i.e., $y = s+n$. The resulting noisy speech has up to three speakers with a silence gap of approximately 300 ms and up to 30% overlap to mimic a regular conversation. The speech dataset is composed of VCTK Corpus [204] and Akustiske Database for Dansk [48]. Thirteen noise environments were selected to represent a wide variety of the typical daily situations. These were obtained from EigenScape [128] (Beach, Busy Street, Park, Pedestrian Zone, Quiet Street, Shopping Centre, Train Station, Woodland), and Demant's database (Bar, Cafe, Canteen, Car, Office). Additionally, two stationary types of noise, pink and white, were simulated. The 25-hour noisy speech consisting of both left and right channel data is divided into training (19.5 h), validation (2.7 h), and test (2.7 h) subsets.

8.5.3 Training Target and Hardware-Aware Training

The DNN learns to match its output against a linear *Ideal Ratio Mask (IRM)* target. IRM represents an ideal scenario, i.e., when speech and noise are perfectly separated. The IRM outputs a continuous gain value between $[0, 1]$ that is computed as a ratio between the magnitude of a clean speech signal and a sum of the magnitudes of the clean and noise signal:

$$IRM = \left(\frac{|s(f,t)|}{|s(f,t)| + |n(f,t)|} \right), \tag{8.22}$$

where $s(f,t)$ and $n(f,t)$ are the time-frequency representations of the clean speech and noise, respectively. The error between the IRM and the postfilter gain values estimated by the DNN is obtained with the mean squared error loss function. The baseline DNN with a GRU layer, i.e., when all computations are performed, was trained in TensorFlow using 32-bit floating-point with a batch size of 128, where each input sequence was 2.5 seconds long (100 samples). The trained parameters were then transferred to the PeakGRU network for inference to i) replace computationally demanding and tedious training from scratch, ii) enable a fair comparison of both methods, and last but not least to iii) simulate a real-world scenario, where new weights would not be transferred

to a HI whenever a different number of K values should be processed. Instead, a single, robust DNN model capable of executing both a full and a pruned model should be used while still delivering sufficient performance in terms of objective measures. Naturally, retraining the model for a specific number of K values further improves its performance, as also demonstrated in our previous work [C2].

In order to prepare a DNN for inference in a hardware environment with limited precision and computational resources, several hardware-aware constraints were incorporated into the training. Firstly, the weights are limited to the [-1, 1] range. Secondly, a challenge is represented by the activation functions. For instance, the output of *ReLU* can theoretically be within a range of $[0, +\infty]$. While a 32-bit floating point is sufficient to handle large numbers, a fixed-point representation needs too many bits on the integer part. Therefore, we use *Capped ReLU* that applies a maximum upper bound (in our case 6). Furthermore, computing sigmoid (σ) and hyperbolic tangent ($tanh$) activation functions would be expensive due to their exponential terms:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{8.23}$$

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{8.24}$$

Typical methods for computing these functions are based on piecewise linear/non-linear approximations [39], lookup tables (LUTs) [158], and hybrid techniques [45]. Although LUTs outperform the other methods in terms of speed as they require the fewest computations, they require additional area, which, depending on the necessary precision, might grow into a significant overhead. Therefore, we train the DNN and run inference with approximated versions called *hard sigmoid* and *hard tanh* that are expressed as:

$$\sigma(x) = \begin{cases} 0, & \text{if } x <= -2.5 \\ 1, & \text{if } x >= 2.5 \\ 0.2 \times x + 0.5, & \text{otherwise} \end{cases} \tag{8.25}$$

$$tanh(x) = \begin{cases} -1, & \text{if } x <= -1.25 \\ 1, & \text{if } x >= 1.25 \\ 0.75 \times x, & \text{otherwise} \end{cases} \tag{8.26}$$

In terms of hardware, each of these approximations requires only one multiplier and two comparators (and an adder for $\sigma(x)$). Moreover, whenever the input value exceeds the upper and lower bounds, the output values are immediately saturated without performing any further computations.

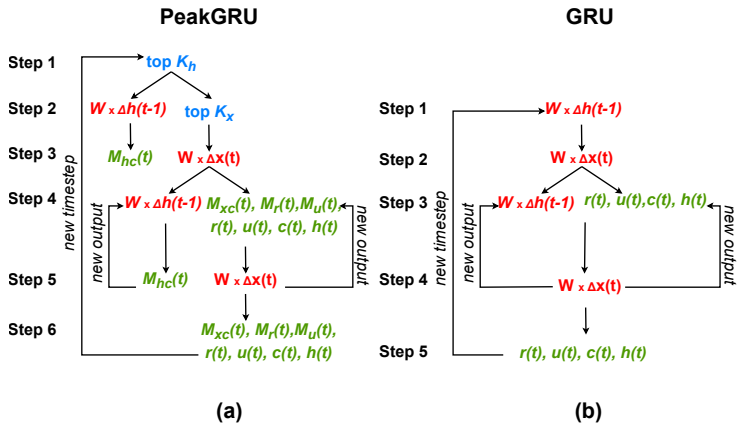


Figure 8.5: Potential concurrency that can be exploited in (a) PeakGRU and (b) GRU, where blue, red, and green correspond to operations in the *Peak Unit*, the *Mac Unit*, and the *Activation Unit*, respectively.

8.6 PeakEngine Design

This section presents architectural design choices (Section 8.6.1) and describes the design of *PeakEngine* (Section 8.6.2) along with implementation details (Section 8.6.3).

8.6.1 Architectural Design Choices

PeakEngine is designed to support both dense and dynamically pruned GRU layers with a focus on low energy, resource sharing, and configurability to make DNN inference viable in HIs. Each of these points is further described below.

Opportunities for parallelism

The architecture of *PeakEngine* is tailored for the needs of GRU and PeakGRU. It is derived from (8.10)-(8.17) and (8.18)-(8.21) that can be grouped based on functionality, suggesting three core units in the system: *1) Peak Unit - selecting the top K elements* (8.18)-(8.21), *2) Mac Unit - computing dot products* (8.10)-(8.13), and *3) Activation Unit - producing M states and output activations* (8.14)-(8.17). Further analysis of the equations shows intrinsic parallelism and dependencies between these units, also illustrated in Figure 8.5(a).

PeakGRU computation begins with finding the K_h subset (Step 1). Thereafter, a dot product with $\Delta h(t-1)$ (and weights) is obtained, while the selection of the top K_x is executed in parallel (Step 2). Once dot products with all $\Delta h(t-1)$ for the first neuron are produced, the operations for an output activation start (loading previous memory states $M_{hc}(t-1)$). Computations of dot products with $\Delta x(t)$ commence as well - assuming that the K_x subset is available (Step 3). Completing a dot product with all $\Delta x(t)$ enables computation of $M_r(t)$, $M_u(t)$, and $M_{xc}(t)$ memory states, and consequently $r(t)$, $u(t)$, and $c(t)$ terms, along with the first output activation $h(t)$. Concurrently, $\Delta h(t-1)$ dot product computations for the next neuron begin (Step 4). Steps 4-5 are repeated until all output activations have been computed (Step 6). This means that the current timestep for PeakGRU is completed, and the system will start again from Step 1 in the next timestep. The steps are almost identical for the baseline GRU in Figure 8.5(b). The main difference is the absence of the K subsets, which reduces the flow by one step. Also, no M states are produced.

The computations in the *Peak Unit* and the *Mac Unit* can be further refined. While the insertion into the min-heap is carried out, delta changes for the subsequent neurons are calculated. Such optimization minimizes idle time and maximizes hardware utilization. Similarly, when the K subset stored in the min-heap is used in the *Mac Unit*, the fetched data (specifically an index of each K, i.e., a neuron index) is simultaneously used to update $\hat{x}(t)$ and $\hat{h}(t-1)$ in (8.18)-(8.19). This avoids expensive re-fetching of the same data. The hat update operations can thus be coupled with the *Mac Unit*, producing an *UpdateMac Unit* in *PeakEngine*.

This analysis leads to a design of five hierarchically interacting units (FSMDs). Such a co-operation results in optimized execution of GRU-based layers and reduced processing time at negligible hardware costs. The FSMDs are orchestrated by the *Main FSM* that handles a coarse-grain top control, as shown in Figure 8.6(a). The final *PeakEngine* architecture can be seen in Figure 8.6(b).

Minimizing energy

High-speed processing is not a driving factor in low-power devices such as HIs. Instead, some HIs may operate at lower clock frequencies and need to finish computations just *"in time"*. When the units complete the execution before the timestep is over, they idle and still consume a certain amount of dynamic and static (leakage) power. Moreover, memories - that DNNs heavily rely on - are often the major source of leakage. Such factors have a significant impact on battery-powered wearables like HIs. Therefore, we employ *clock-gating* and *memory retention* techniques to prevent additional leakage and energy dissipation when *PeakEngine* idles.

Resource-sharing and reuse

To accomplish the objective of minimizing hardware resources yet completing the execution of DNNs in time, it is necessary to perform additional optimizations on multiple levels. Firstly, we only use single-port SRAMs to minimize area. Secondly, we employ multiple smaller memories instead of a single shared one, which enables the proposed units to operate concurrently. Most of the memories are shared among multiple units. The scheduling of memory accesses in the five co-operating FSMDs minimizes memory access conflicts and stalling. The memory accesses are time-multiplexed via memory management units (MMUs) to guarantee that only one unit accesses any memory at any given point in time. Finally, our small *Mac* unit represents a trade-off between area and computations with a focus on minimum hardware resources, while still delivering parallelized dot product computations.

Configurability

It is essential to enable the execution of different DNN configurations (type of layers/activation functions, number of neurons, etc.) to provide the necessary flexibility. Therefore, *PeakEngine* is *configurable* (see Section 8.6.3.1) and supports, among others, three layer types (FC, GRU, PeakGRU) and four activation functions (ReLU, Capped ReLU, hard sigmoid, and hard tanh).

8.6.2 Top-Level Architecture

Figure 8.6(b) shows a high-level architecture of *PeakEngine* with the arrows representing a simplified dataflow. Besides the previously mentioned MMUs, SRAMs, and the three main units (*Peak*, *UpdateMac*, *Activation*), the top level consists of a configuration module (*Config*), a clock management unit (*CMU*) for coarse-level clock-gating, and a power management unit *(PMU)* for the retention of SRAMs. Model parameters are stored in a big *SRAM WB* outside *PeakEngine*. All these modules are further described in Section 8.6.3.

The accelerator communicates with the *Master Processor* via a 24-bit *advanced peripheral bus (APB)* interface that is used for writing and reading configuration registers in the *Config* module. This includes writing the *start_nne* register to trigger the execution of *PeakEngine* and reading the *nne_done* register to check whether the accelerator has completed inference for the current timestep. *PeakEngine* has two other interfaces i) a 32-bit *memory interface (MEM)* for writing inputs and reading the final results for the current timestep stored in *SRAM X* and *SRAM H*, and reading weights and biases from *SRAM WB*, and ii) a clock and reset interface (*CLK/RST*). Thanks to these three generic interfaces,

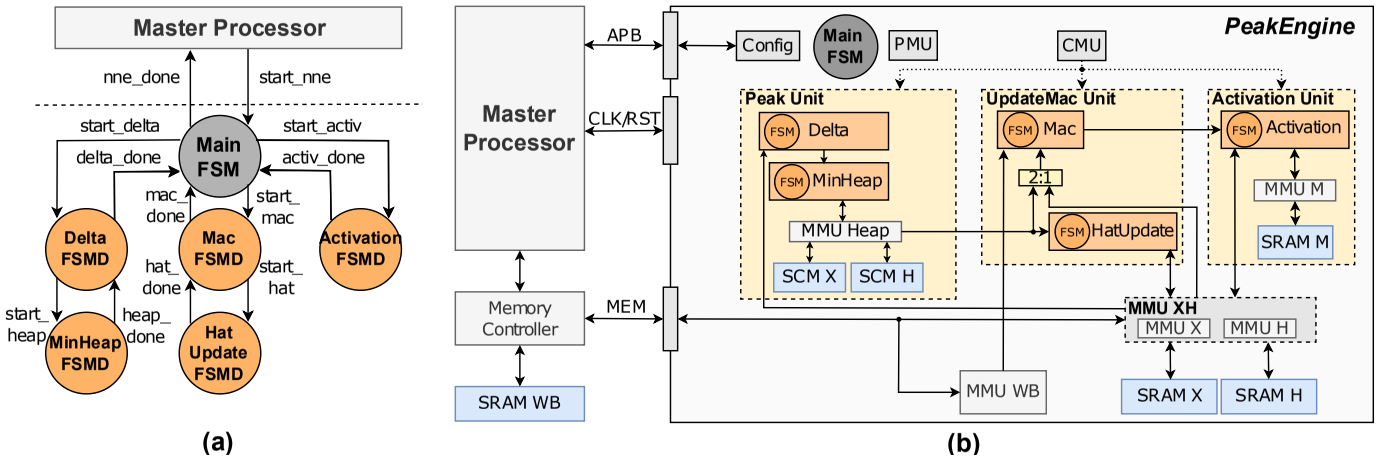


Figure 8.6: (a) A hierarchical relationship among the *Main FSM* and the FSMDs in the system. (b) A top-level overview of *PeakEngine* with a simplified dataflow indicated by the arrows.

PeakEngine is easily configurable and can be ported to any system that supports such communication.

8.6.3 Implementation

This subsection describes details about the main modules and submodules in the system.

8.6.3.1 Config

PeakEngine is *configurable*, i.e., all network parameters can be specified at runtime by writing configuration registers via the *APB* interface. These parameters are: the number of inputs, layers, and neurons per each layer, layer and activation type, starting weight address for each layer, K values to process in a PeakGRU layer, input and result locations (*SRAM X*, *SRAM H*), and a fixed-point format per layer, along with the previously mentioned *start_nne* and *nne_done* registers.

8.6.3.2 SRAMs and MMUs

SRAM X and *SRAM H* are 1,536-word memories that store 16-bit inputs, outputs, and hat states in separate memory blocks. Corresponding *MMU X* and *MMU H* track which of the blocks should be used for reading and writing in different layers. If neural network inference is not needed, the memories can be reused to store other data for a HI.

SRAM WB stores up to ~ 2.07 MB of 96-bit vectors composed of 12×8-bit model parameters (see Table 8.1). When smaller networks are executed, the unused part of the memory can be utilized for another purpose. *SRAM WB* is implemented as eleven memory banks of 16,384 words each. Unused memory banks are put into a retention mode during inference. For GRU and PeakGRU, each bias and weight vector is a concatenation of four r, u, and c terms, which enables four new $h(t)$ states to be calculated simultaneously.

SRAMs are supplied by the foundry and operated on two supply voltages: 0.6 V (logic) and 0.8 V (bit cells). All memories in the system (including SCMs) have a delay of one clock cycle for both read and write.

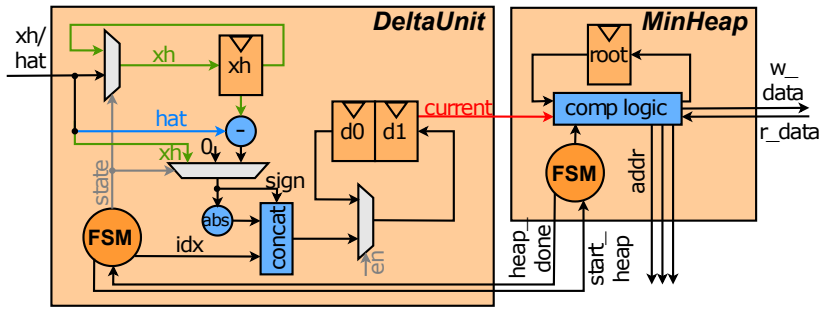


Figure 8.7: A simplified overview of the *Delta* and *MinHeap* submodules.

8.6.3.3 Peak Unit

The *Peak Unit* consists of two main submodules, *Delta* and *MinHeap* (see Figure 8.7), *MMU Heap*, and two SCMs for storing the top K_x and K_h elements. If a PeakGRU layer is not used in a neural network, the entire module is clock-gated by *CMU*.

PeakEngine executes inference on a per-layer basis and performs the top K selection in parallel with computations in other layers. For instance, while the first FC layer in the demonstrated DNN performs MACs, the *Peak Unit* meanwhile selects the top K_h values for $\Delta h(t-1)$. Similarly, when MAC operations with $\Delta h(t-1)$ values are performed in the *PeakGRU* layer, the selection of K_x for $\Delta x(t)$ begins. Hence, the latency of selecting the top K elements in both cases is hidden. We also optimize the algorithm by skipping the selection of K_h in the first timestep since both the hidden and hat states are 0.

Delta

This unit performs the subtraction, absolute value, and comparison operations in (8.18)-(8.19). It reads $x(t)$, $\hat{x}(t-1)$, $h(t-1)$ and $\hat{h}(t-2)$ from *SRAM X* and *SRAM H*. The unit has registers for storing a tuple composed of *sign-magnitude-index* data, where the sign of the subtraction is necessary for decoding the magnitudes later during the MAC operations. Furthermore, only *non-zero* data is passed to *MinHeap*. We optimize the delta algorithm by skipping subtractions in the first timesteps when $\hat{x}(t-1)$ and $\hat{h}(t-2)$ are still 0.

MinHeap

MinHeap selects the top K elements by utilizing a *min-heap* data structure (see Section 8.3.2). The K elements are stored as 26-bit tuples (sign-magnitude-index) in the 3-port latch-based *SCM X* and *SCM H* that consist of 128 words each. Since the *root* is the most accessed element in the min-heap, we store it locally to avoid unnecessary memory fetches. To further reduce memory accesses, we write a new element to a memory only when its correct position in the heap is found.

Mac and *MinHeap* alternate their access between the two SCMs. While the *Mac* module reads data from *SCM X*, *MinHeap* uses *SCM H* for the top K_h selection, and vice versa. Hence, both SCMs are fully utilized. To optimize subsequent multiplications, the amount of inserted non-zero K_x and K_h elements is stored locally as they can be fewer than K specified in the *Config*.

8.6.3.4 UpdateMac Unit

It receives input data from either i) *SRAM X* or *SRAM H* for GRU and FC layers, or ii) *SCM X* or *SCM H* for a PeakGRU layer. In the latter case, it checks the sign of the sign-magnitude-index input tuple to extract the original value stored as a magnitude. This value is needed for the multiplications in the *Mac* unit.

Mac

The *Mac* submodule loads biases and performs vectorized multiplications between inputs and weights. It utilizes i) *output stationary technique*, where the twelve intermediate dot products are kept in local registers until the final result has been computed, which is then passed to the *Activation Unit*, and ii) *input parallelism*, where the input is used for twelve multiplications at a time.

For GRU and PeakGRU layers, the accumulators store four r, u, and c dot products. While the accumulators for r and u hold the final weighted sum, the accumulators for c always keep the results of multiplications with either the input or hidden state vector at a time since these are not directly summed together, as shown in (8.3) and (8.16). The four partial c weighted sum are passed to the *Activation Unit*. The weights for PeakGRU are fetched based on the extracted tuple index.

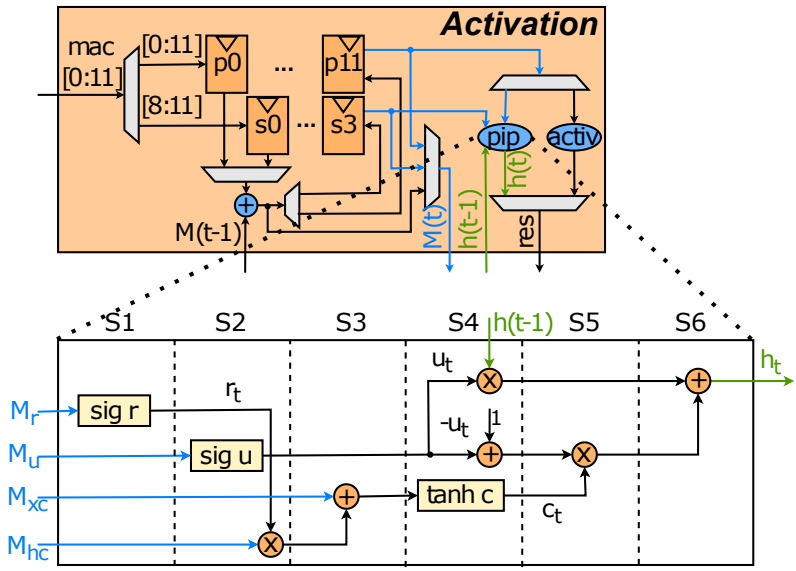


Figure 8.8: A simplified overview of *Activation* along with its 6-stage pipeline for the GRU and PeakGRU layers.

HatUpdate

This small submodule updates $\hat{x}(t)$ and $\hat{h}(t-1)$, see (8.18)-(8.19). It is triggered only when a new K input, for which a hat update has not been performed yet, has been fetched. If a PeakGRU layer is not used, *HatUpdate* is clock-gated.

When a new element is fetched from one of the SCMs for a MAC operation, its index is stored in a small local buffer. The hat update process requires two cycles to update a single hat value. Firstly, it fetches $x(t)$ or $h(t-1)$ based on the stored index. Then a write is initiated in the next clock cycle to update $\hat{x}(t)$ or $\hat{h}(t-1)$ with the previously fetched value, where the address is generated using the same index.

8.6.3.5 Activation Unit

It consists of *Activation* (see Figure 8.8), *MMU M*, and *SRAM M*. When a DNN does not contain a PeakGRU layer, *SRAM M* is powered down.

Activation

This unit uses 16 registers for storing the MAC results; twelve primary for all the layers, and four secondary specifically for the GRU and PeakGRU layers to temporarily store the c dot products as described in the *Mac* submodule.

When a FC layer is processed, the MAC results are extracted and stored in the primary registers. The twelve output activations are calculated and written one by one to a memory (*SRAM X* or *SRAM H*) defined in the *Config* module. If no activation function is specified, the final result will be extracted directly from the weighted sum.

For GRU and PeakGRU, output activations are computed in a 6-stage *pipeline* illustrated in Figure 8.8. As shown in (8.13) and in Figure 8.5(a) for PeakGRU, previous 24-bit M_{hc} delta memory values are fetched from 2,048-word *SRAM M* and added to the dot product with $\Delta h(t-1)$. Once the multiplications with $\Delta x(t)$ are completed, the results are written to the primary registers. Thereafter, previous M_r, M_u, and M_{xc} states are fetched from *SRAM M*, updated, and stored back while the pipeline runs. We optimize the memory accesses and computations by skipping zero M states in the first timesteps.

8.7 Parameter Space Exploration Framework

A *bit-accurate* in-house software framework supporting FC, GRU, and PeakGRU layers was developed to find the most optimal wordlengths, Q formats, and K values for DNNs executed on *PeakEngine*. It was also used to verify the accelerator outputs. The framework contains identical modules as *PeakEngine* to mimic it bit-accurately during inference. It supports both floating-point and custom fixed-point formats, where the latter is based on the library in [263]. The framework contains a configuration file where wordlengths and Q formats can be defined independently for the network inputs, weights (and biases), outputs, accumulators, and M states. The formats of the remaining intermediate terms are derived automatically based on the other wordlengths specified. The framework supports the creation of a neural network with any number of layers, where each layer is defined as: layer type, number of outputs, activation function, and K value for a PeakGRU layer. The framework performs quantization of network inputs and model parameters, and stores the DNN outputs for the subsequent postprocessing phase. During inference, it exploits multiprocessing to run several noisy speech recordings concurrently and hence speeds up the fixed-point execution.

Table 8.1: Final wordlength and Q format values used in the presented experiments.

	Wordlength	Q format
DNN inputs	16	Q1.15
PeakGRU magnitudes	16	Q3.13
Hat states	16	Q2.14
Input/Output Activations	16	Q2.14
Weights/biases	8	Q2.6
Accumulators	26	flexible
M states	24	Q8.16

Table 8.1 shows the selected configuration where almost no drop in the objective measures (SNR, PESQ, and STOI) compared to its hardware-aware model (see Section 8.5.3) was observed. The framework was validated against TensorFlow with a maximum error of 10^{-6} (floating-point).

8.8 Experimental Setup

The *PeakEngine* accelerator is evaluated by executing two main DNN architectures:

1. The baseline model with a GRU layer that performs all computations every timestep.
2. The PeakGRU-based model that performs a subset of computations every timestep based on the top K values. Various K values were tested.

The objective of the PeakGRU is, besides reducing computations and power consumption, to decrease latency and hence make real-time inference of big DNNs possible. At the same time, it is important to ensure that the impact of the reduced computations on the objective measures is within acceptable boundaries. Therefore, the selection of the top K values was guided by both i) the need to fit within a specified time window, and ii) the amount of degradation in the improvement of the objective measures. The following four top K configurations were selected for the final hardware tests: $\{128, 96, 64, 48\}$. K=128 represents a setup with no performance degradation compared to the baseline GRU, while K=48 corresponds to improvements in SNR and PESQ, but no improvement in STOI compared to the unprocessed speech. Lower top K values than 48 result in worse STOI than the unprocessed speech. The same number of K values is used for both input and hidden state sequences, i.e., K_x==K_h.

Table 8.2: Required memory and obtained improvements (Δ) in objective measures compared to the unprocessed speech for: 1) 32-bit floating-point model *(FP)* [C2], and 2) fixed-point model *(FX)* running on *PeakEngine*. The (*) denotes approximated values based on [C2].

	GRU 512		**Peak K=128**		**Peak K=96**		**Peak K=64**		**Peak K=48**	
Representation	**FP**	**FX**	**FP***	**FX**	**FP***	**FX**	**FP***	**FX**	**FP***	**FX**
#Params	2,099,712									
Memory (MB)	8.01	2	8.01	2	8.01	2	8.01	2	8.01	2
ΔSNR (dB)	8.11	7.81	8.14	7.89	8.07	7.78	7.8	7.46	7.48	7.09
ΔPESQ (MOS-LQO)	0.43	0.42	0.41	0.41	0.39	0.39	0.36	0.36	0.32	0.32
ΔSTOI	0.039	0.037	0.032	0.031	0.026	0.024	0.013	0.007	0.003	0.000

When *PeakEngine* completes inference, it idles until the 25 ms have elapsed and the *Master Processor* writes new data to *SRAM X* or *SRAM H*. Our objective is low-power inference and completing computations *"in time"* instead of achieving extremely low latency and idling for the remainder of the timestep. Therefore, the design operates at a low clock frequency to utilize the majority of the 25 ms window. Several timesteps of a 30-second noisy speech with *Busy Street* background noise are used for the demonstration of *PeakEngine* for all the setups. The results are presented in Section 8.9.

8.9 Results and Discussion

The *PeakEngine* design (including *SRAM WB*) was synthesized in a 22 nm ultra low leakage CMOS process for a 4 MHz clock frequency. The entire system uses ultra high density and ultra low leakage SRAMs that are supplied by the foundry and operated on two supply voltages: 0.6 V (logic) and 0.8 V (bit cells).

This section presents results in terms of area, energy, latency, memory requirements, and objective measures for the tested configurations. They are divided into three main subsections: A. comparison between the algorithmic study [C2] and the hardware implementation of GRU and PeakGRU, B. comparison when *PeakEngine* executes GRU- and PeakGRU-based DNNs, and C. comparison of *PeakEngine* against previous works.

8.9.1 Algorithmic vs Hardware Implementation

Table 8.2 compares 32-bit floating-point DNN models *(FP)* from the algorithmic study [C2] and our fixed-point DNN models *(FX)* executed by *PeakEngine*

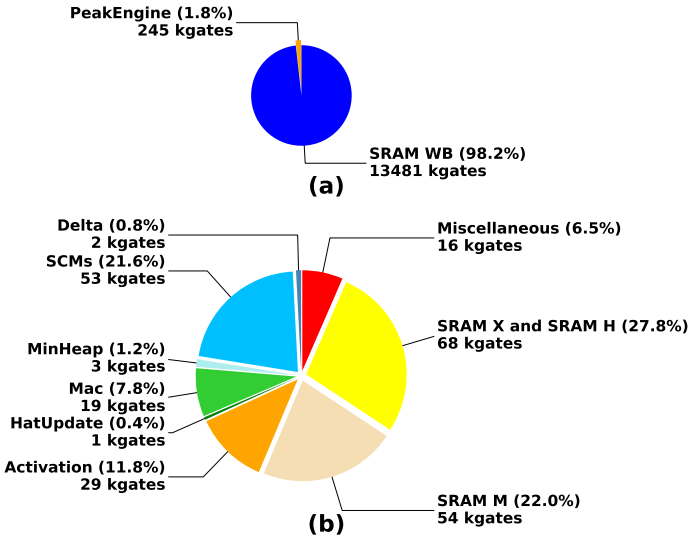


Figure 8.9: Area breakdown of (a) the entire system and (b) *PeakEngine* only.

in terms of memory requirements and relative improvements in the objective measures to the unprocessed speech. The *FX* models use wordlengths from Table 8.1. The objective measures for most *FP* models are derived from the plots in [C2] (marked with *), since those experiments were executed for a different number of K than our final experiments. The unprocessed speech has starting values of 4.39 dB (SNR), 1.85 MOS-LQO (PESQ), and 0.83 (STOI).

As shown in Table 8.2, the *FX* models have almost the same performance as the *FP* DNNs, while being better suited for hardware inference due to their fixed-point nature. The differences between the two corresponding models cannot be perceived in audio recordings. The knee point in [C2], i.e., when the PeakGRU configurations show the first decrease in performance, is around K=111 for SNR. Until then the measures are unchanged. Considering the reported PESQ, the actual knee point is already around K=128, matching the *FX* model. The *FP* and *FX* results are therefore comparable, where 4× less memory is needed for 8-bit FX models, corresponding to 6 MB saved.

8.9.2 PeakGRU vs GRU

Area

Figure 8.9 shows the area breakdown of *PeakEngine* (a) with *SRAM WB* and (b) without *SRAM WB* in kgates. As expected, the majority, i.e, 98.22% of

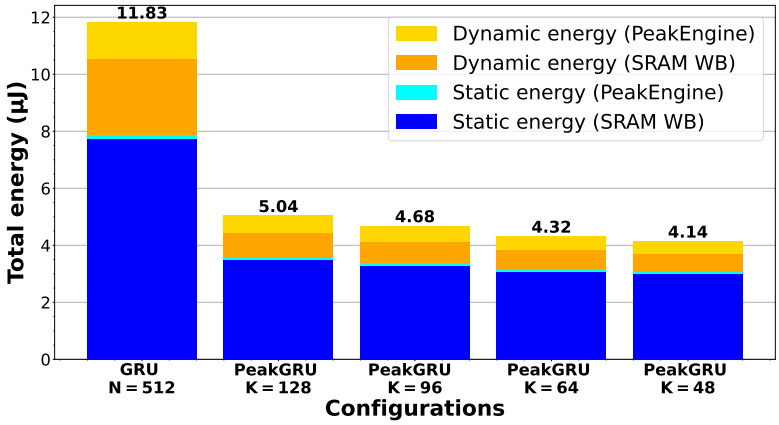


Figure 8.10: Comparison of total energy for GRU and four PeakGRU configurations per inference, further divided into static and dynamic.

the whole area (13,726 kgates, 2.95 mm^2) is dominated by the large *SRAM WB* (13,481 kgates, 2.9 mm^2), while the accelerator itself only represents the remaining 1.78% (245 kgates, 0.053 mm^2). Almost 30% of the *PeakEngine* area (Figure 8.9(b)) is used by *SRAM X* and *SRAM H* for storing inputs, results, and hat states. The *Activation* and *Mac* modules used for all layer types occupy $\sim$20%. The PeakGRU-related modules, i.e., the *Peak Unit* (*Delta*, *SCMs*, *MinHeap*), *HatUpdate*, and *SRAM M*, comprise 46.1% of the area, where the SCMs and SRAMs clearly dominate. The total PeakGRU overhead area is in general insignificant (113 kgates), especially when compared to the obtained energy savings described next. Small modules such as *MMUs*, *Config*, and *CMU* along with the logic in the top level are summed together and represented as *Miscellaneous*. They account for only 6.5% of the area.

Energy

Our energy evaluations are divided into two parts: i) comparing the total savings in terms of static and dynamic energy to see the overall affect of pruning (Figure 8.10), and ii) comparing the savings of dynamic energy for all the configurations (Figure 8.11).

Figure 8.10 shows the total energy dissipation (y-axis) of *PeakEngine* together with *SRAM WB* for different configurations (x-axis) as stacked bars. In total, the baseline GRU DNN dissipates 11.83 μJ per inference compared to only 5.04-4.14 μJ for the PeakGRU configurations. The energy is further divided into *static* and *dynamic*, where the static energy clearly dominates for all five

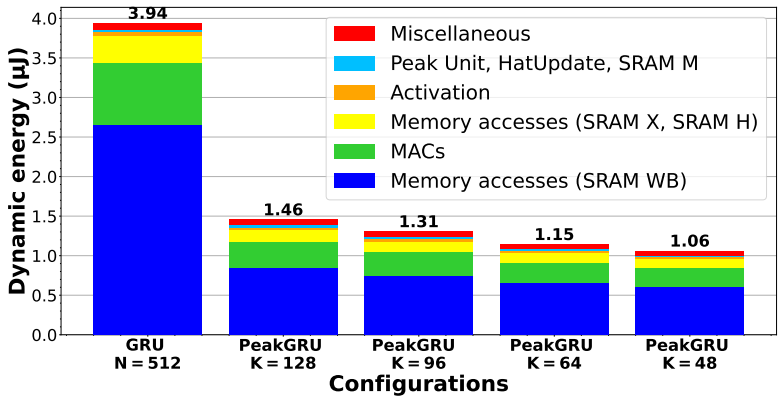


Figure 8.11: Comparison of dynamic energy for GRU and four PeakGRU configurations per inference.

configurations (66.7%, 71.1%, 72.1%, 73.5%, and 74.4%, left to right). Almost all the static energy comes from *SRAM WB*. Although the PeakGRU algorithm targets dynamic pruning, it also assists in decreasing the total leakage by 54.6-61%, i.e., from 7.89 μJ (GRU) to only 3.58-3.08 μJ per inference. These savings are a result of completing the inference faster and consequently putting *SRAM WB* into retention along with clock-gating the *PeakEngine* modules. Without the leakage of *SRAM WB*, *PeakEngine* in total dissipates 4.08 μJ for GRU and only 1.56-1.16 μJ for the PeakGRU configurations per inference.

Figure 8.11 shows the total dissipated dynamic energy per inference, where our objective was minimizing energy spent on weight memory accesses and MACs that dominate the computations. GRU dissipates 3.94 μJ per inference. MACs account for 0.78 μJ and weight memory accesses for 2.66 μJ, which corresponds to 19.8% and 67.5% of the total dynamic energy. All the PeakGRU-related modules are clock-gated in the GRU setup, namely the *Peak Unit* (*Delta*, *SCMs*, *MinHeap*) and *HatUpdate*, while the unused *SRAM M* is powered down. A noticeable portion of dynamic energy (0.34 μJ, 8.5%) is spent on *SRAM X* and *SRAM H* memory accesses. The *Activation* and the rest of the system (*Miscellaneous*) constitute only 1.5% (0.057 μJ) and 2.2% (0.09 μJ) of the total, respectively.

PeakGRU DNNs dissipate only 1.46-1.06 μJ per inference. Although the PeakGRU-related modules (light blue bar) are clock-gated in the GRU DNN, at the same time, they dissipate less than 0.028 μJ per timestep for each of the four K configurations, while saving 2.26-2.6 μJ of the original dynamic energy spent on MACs and memory fetches. Moreover, as it can be noticed in Figure 8.11, the PeakGRU configurations also decrease $\sim$55.2-67.1% of dynamic energy spent on

SRAM X and *SRAM H* (yellow bar). This reduction corresponds to fewer $x(t)$ and $h(t-1)$ reads since their delta versions for the MAC operations are fetched from smaller and cheaper SCMs instead. *Activation* and *Miscellaneous* dissipate only $\sim 0.03\,\mu$J and $\sim 0.07\,\mu$J, respectively.

Latency

The baseline GRU model performs 2,097,152 MACs and 175,104 96-bit vector memory fetches of weights to compute 512 gain values every timestep. Approximately 75% of both MACs (1,572,864) and vector memory fetches (131,072) are needed for the GRU layer itself. Executing this amount of computations and memory fetches is not only energy-intensive, but it also exceeds the real-time budget of 30 ms when running at a low clock frequency necessary for HIs. The baseline GRU DNN executes inference in 44.04 ms, where the GRU layer needs ~ 33 ms and each FC layer ~ 5.52 ms. Pruning considerably reduces the total inference latency down to $\sim$ 20-14.83 ms (2.2-2.97×), i.e., below both the limit and the 25 ms input data rate defined in Section 8.4.1. PeakGRU layers require only $\sim$ 8.8-3.7 ms, which is comparable with or even below the latency of a FC layer. The latency of the first FC layer in the PeakGRU DNN slightly increases to ~ 5.68 ms. This is a result of configuring the inputs and outputs of the first FC layer to be stored in the same memory (*SRAM X*). Hence the *Mac* unit is stalled for few cycles every time *Activation* writes the final outputs. This setup enables a fast selection of the top K_h elements since the other memory (*SRAM H*) is only accessed by the *Peak Unit*.

Overall, the achieved reductions enable DNN inference to be completed within the real-time and energy budget of a HI.

8.9.3 PeakEngine vs State-of-the-Art

Table 8.3 shows a comparison of *PeakEngine* to the state-of-the-art accelerators. These span a wide variety of pruning (model compression, skipping updates/-computations) and non-pruning techniques, target different platforms (FPGAs, microcontrollers, ASICs), network architectures, use cases, and requirements in terms of latency, power consumption, and model size. All the stated ASIC accelerators report *synthesis* results.

The *EdgeDRNN* [215] accelerator is most similar to our work in terms of pruning approach, network type, and the number of MAC operations. It utilizes threshold-based *Delta Networks* [138], a GRU layer, and performs eight multiplications per cycle. The weights are stored in an off-chip memory. However, the accelerator is developed for FPGA platforms, consumes 2.3 W, and targets extremely low

Table 8.3: A comparison of *PeakEngine* with prior state-of-the-art works. All the stated ASIC accelerators present *synthesis* results.

	EdgeDRNN [215]	**SpAtten [248]**	**TinyLSTM [214]**	**E-PUR [172]**	**SHARP [258]**	**Skip [182]**	**This Work (Peak128)**
Platform	FPGA XC7Z007S	ASIC 40 nm	STM32F746VE Arm Cortex-M7	ASIC 28 nm	ASIC 32 nm	ASIC 65 nm	ASIC 22 nm
NN Architecture	2×DeltaGRU (768)	GPT-2 Medium	2×LSTM(256) 2×FC(128)	17× LSTM(1,024)	17× LSTM(1,024)	Embedding(300) -LSTM(300)	FC(512)-PeakGRU (128/512)-FC(512)
Sparsity type	Temporal	Temporal	Weight/Temporal	None	None	Temporal	Temporal
Pruning approach	Threshold	Top K (quick-select)	Threshold	None	None	Threshold	Top K (min-heap)
Voltage (V)	N/A	N/A	N/A	0.78	0.85	N/A	0.6 (logic), 0.8 (bit cells)
Freq (MHz)	125	1,000-2,000	216	500	500	200	4
#Params (millions)	5.4	345	0.46	N/A	N/A	N/A	2
Activation/weight WL	FX16/FX8	FX12/FX8-FX12	FX8/FX8	FP16-FP32	FP16	FX8/FX8	FX16/FX8
Weight MEM (MB)	N/A (off-chip)	N/A (off-chip)	0.43 (flash memory)	272 (off-chip)	272 (off-chip)	N/A	2
Accelerator MEM (MB)	0.25	0.383	0.313	14	28.3	N/A	0.013
Area (mm^2) (synthesis)	-	18.71 (w/o param storage)	-	64.6 (storage for 1 layer)	101.1 (storage for 1 layer)	1.1 (w/o param storage)	2.95
Latency (ms)	0.536	27.88	2.39	N/A	N/A	N/A	25
Accelerator power (mW)	66	1,360	-	975 (averaged)	N/A	N/A	0.029
Total power (mW)	2,290	8,300	540	N/A	8,110 (averaged)	N/A	0.202
Application	Speech recognition	Text Generation	Speech enhancement	Neural Machine Translation	Neural Machine Translation	Language Modeling	Speech enhancement
Dataset	TIDIGIT	Wikitext, Penn Tree Bank, One-Billion Word	CHiME2	WMT	WMT	Penn Treebank	VCTK [204], ADD [48], ES [128] Demant [260]
#MACs	8	1,024	N/A	16	1,000	192	12

latency in the range of μs. Additionally, the computation time is *unbounded* due to using a threshold-based pruning.

SpAtten is [248] an ASIC accelerator that focuses on dynamic pruning in huge transformer neural networks [146] for natural language processing tasks. It also applies the top K selection, however by using a quick-select algorithm instead. Nonetheless, the average and worst-case time complexity of quick-select is $O(N)$ and $O(N^2)$, compared to the min-heap's $O(1)$ and $O(N \log K)$, respectively. Moreover, quick-select requires significant memory space $O(N)$, while the min-heap only $O(K)$. Furthermore, SpAtten runs at 1-2 GHz and comprises of two parallel top-K engines that together perform 1,024 MACs. This considerably higher parallelism results in area of 18.71 mm^2, excluding huge memories for storing 345 million weights, and it has a total power consumption of 8.3 W.

TinyLSTM [214] also focuses on supporting RNNs for SE in HIs. The authors reduce computations and memory accesses via static pruning and additionally introduce a scheme for skipping LSTM state updates, which can be seen as a form of dynamic temporal pruning. While the application and use case match perfectly with ours, this work does not propose an actual hardware accelerator. Instead, the STM32 microcontroller with ARM Cortex-M7 is used to run the inference, consuming 0.54 W.

The *E-PUR* ASIC accelerator [172] targets the execution of large LSTM networks (1-272 MB) in low-power mobile devices for various use cases such as video classification, speech recognition, and neural machine translation. Table 8.3 states the biggest supported model (272 MB) that is used for machine translation. The smallest model demonstrated for speech recognition (LibriSpeech dataset) uses 4× Bi-directional LSTMs and requires 42 MB of weight memory. The accelerator does not apply pruning. Instead, it exploits a novel technique, called Maximizing Weight Locality (MWL), that improves the temporal locality of the synaptic weights. *E-PUR* provides 14 MB of on-chip memory - 8 MB for weights to store one layer at a time (MWL applied) and 6 MB as intermediate storage. The area and average power consumption of the accelerator itself, excluding the large off-chip memory, are 64.6 mm^2 and ∼1 W (averaged across applications), which is amenable for mobile devices, however, not for HIs.

Another ASIC accelerator called *SHARP* is presented in [258]. Like *E-PUR* [172], it is demonstrated on the same models (smallest 40 MB) and use cases, and does not apply any pruning method. The accelerator is benchmarked for different LSTM dimensions as well. *SHARP* also emphasizes the importance of adaptive computations, however, via a tiled-based dispatching mechanism to handle the data dependencies, and not via dynamic pruning. Table 8.3 again shows the biggest supported network. *SHARP* provides 26 MB of on-chip weight memory to store one LSTM layer at a time and 2.3 MB of buffers. The smallest configuration

has 1,000 MACs, an area of 101.1 mm^2, and consumes 8.1 W (averaged across applications) - all infeasible for HIs.

Last but not least, the *Skip* [182] ASIC accelerator for edge devices performs dynamic pruning of hidden state vectors in an LSTM unit. The proposed method, however, requires training and cannot be applied directly during inference like PeakGRU. Additionally, it focuses on language modeling task, i.e., predicting a next word in the sequence. The authors do not report total power or energy, only the area of 1.1 mm^2, energy efficiency and peak performance compared to previous works.

Our proposed *PeakEngine* serves as a first ASIC accelerator for dynamic and deterministic pruning of RNNs that targets HI applications and the SE use case. The example configuration with K=128 (*Peak128*, Table 8.3) consumes 29 μW without the big weight memory and 202 μW in total. *PeakEngine* has a small total area of 2.95 mm^2 while supporting big DNNs within the given time and energy constraints. The weight memory occupies 2.9 mm^2 and the accelerator itself only 0.053 mm^2. *PeakEngine* is configurable and easily portable, and it can be used as a co-processor to a typical digital signal processor in HIs to take off the neural processing workload from the system. Overall, *PeakEngine* is suitable for low-power and resource-constrained embedded devices such as HIs.

8.10 Conclusion

This paper presented *PeakEngine*, a configurable ASIC accelerator for low-power edge devices, such as HIs, that supports dynamic and deterministic pruning of input and hidden state sequences in a GRU layer. This is accomplished via the top K element selection by utilizing the small and efficient *Min-heap engine*. The algorithm-hardware co-design of the PeakGRU algorithm and *PeakEngine* significantly reduces total energy and latency up to 2.86 $\times$ and 2.97 $\times$, respectively, making the energy-efficient and real-time execution of even bigger RNNs viable within the constraints imposed by HIs. The accelerator was demonstrated in the SE task, and it could be used as a co-processor to a typical digital processor found in HIs. Additionally, we developed a framework for parameter space exploration to identify the most suitable data wordlength, Q formats, and K values for DNNs executed by *PeakEngine*. The accelerator is synthesized in a 22 nm CMOS technology and evaluated at a 4 MHz clock frequency while operating at two supply voltages: 0.6 V (logic) and 0.8 V (bit cells). It occupies 0.053 mm^2 without the weight memory and 2.95 mm^2 in total. To the best of our knowledge, *PeakEngine* is the first ASIC accelerator for

low-power dynamic and deterministic pruning of RNNs that targets support of HI-relevant use cases such as SE.

Chapter 9

Conclusion

This thesis was written as a collection of five papers presented in Chapters 4-8. The following sections summarize the contributions of the papers (Section 9.1), outline several ideas for future research (Section 9.2), and state a couple of final remarks on the topic of the explored algorithms as well as neural networks for hearing instruments in general (Section 9.3).

9.1 Summary

The presented papers can be thematically divided into two main areas:

1. *Algorithms* - Development of computationally-efficient *deep learning algorithms*
2. *Hardware* - Design and implementation of *custom hardware accelerators* for deep learning algorithms

Therefore, together, the papers form an *algorithm-hardware co-optimization* approach for embedded deep learning, where the proposed algorithms significantly reduce the amount of computations and the custom hardware accelerators efficiently support the optimized algorithms.

9.1.1 Deep Learning Algorithms

Chapters 5 and 6 proposed deep learning algorithms for pruning that reduced the amount of computations dynamically during inference in recurrent and attention layers, respectively. While the *PeakRNN* and *StatsRNN* algorithms in Chapter 5 were demonstrated on the SE task using GRU-based DNNs, the *Delta* algorithm in Chapter 6 pruned FC-based MHSA layers in a TNN trained for KWS. Both works showed that using the proposed pruning techniques resulted in significant savings of up to 70% for SE and 80% for KWS in terms of MACs and memory accesses without degrading the original evaluation metrics. Such computational savings make execution of DNNs viable in low-energy and battery-powered devices such as hearing instruments. Moreover, *PeakRNN* offers determinism, worst-case execution guarantees, and robustness to the variations in input data since it selects a constant number of top elements every timestep. Last but not least, the *patent application* for *PeakRNN* and *StatsRNN* algorithms was successfully granted [O2].

9.1.2 Custom Hardware Accelerators

The hardware implementation of deep learning algorithms was explored in Chapters 4, 7, and 8. A dedicated and configurable neural network accelerator for FCNNs was introduced in Chapter 4. It implemented a novel two-step scaling technique for efficient quantization of output activations on-the-fly. The method ensures deterministic execution for any arbitrary FCNN without impacting latency. The accelerator was demonstrated on the KWS task, and it outperformed a typical digital signal processor in all aspect, including power consumption, memory accesses, and area (5×, 5.5×, and 3.7× less). Chapters 7 and 8 focused on the *PeakRNN* algorithm. While Chapter 7 presented a *Min-heap-based engine* for the efficient selection of the top K elements, Chapter 8 introduced *PeakEngine*, a configurable accelerator that encompasses the *Min-heap engine* and supports inference of both dense and pruned DNNs. The experiments on the SE task showed that the area and energy overhead of the *Min-heap engine* and other *PeakRNN*-related modules together is insignificant compared to the achieved energy savings of 57.4-65% when running the pruned DNNs. Additionally, the original latency was decreased 2.2-2.97×, making a real-time execution of big DNNs feasible within the time constraints imposed by a hearing instrument. The most area- and energy-dominating factor was a big SRAM that stored the weights for a DNN.

9.2 Future Work

This section discusses possible extensions and improvements of the current work. A few preliminary studies for some of the suggestions have been conducted, as described in Subsection 9.2.4 and 9.2.5.

9.2.1 Memory Leakage

As shown in Chapter 8, dynamic energy can be significantly reduced with our algorithm-hardware co-optimization. However, the (weight) memory leakage is a dominating factor of the total energy. Putting unused memories into a retention mode saved a substantial amount of energy. The *PeakRNN* (also referred to as *PeakGRU*) algorithm considerably contributes to these savings since as soon as the inference is done, the system idles, including the weight memories.

To alleviate this issue further, the weights could be reorganized and the efficient *heap sort* algorithm could be applied to sort the top K elements based on indices, allowing even more memory banks to be put into retention - and for a longer period of time. Another solution that also reduces area is to combine dynamic pruning with weight compression, while still preserving large enough representation power of a DNN. The compression could be achieved via several well-established techniques such as network architecture search, knowledge distillation, or the previously mentioned static pruning.

9.2.2 Pruning of FC layers

The main goal of our study was to prune a computationally intensive GRU layer that has a quadratically growing complexity with respect to the hidden state $h(t)$. In the case of our tested networks (Chapters 5, 7, and 8), 75% of the total number of MAC operations and vector memory fetches are needed by a GRU, while the remaining 25% by the two FC layers together. We reduced the number of processed values (top K) by 4-10.7×, which resulted in comparable or even lower latency ($\sim$ 8.8-3.7 ms) than for a FC layer ($\sim$ 5.52 ms), thus making the FC layers a bottleneck in the system.

Therefore, we could adapt the *PeakGRU* algorithm to the needs of an FC layer, which would actually correspond to simplifying the equations (5.1)-(5.8) to computing only $\hat{x}(t)$ and Δx since FC layers do not have a feedback connection, and hence do not need any delta memory M. The only overhead would therefore

be $\hat{x}(t)$. We already demonstrated the power of the *Delta* algorithm (Chapter 6) on the FC-based attention layers in a TNN. Furthermore, *PeakGRU* could be easily adjusted to LSTMs as they are very similar to GRUs.

9.2.3 Smaller DNNs

The results in Chapter 8 (and Chapter 7) proved that the area and energy overhead of the *PeakGRU*-related modules and memories is negligible. Therefore, the method and the *PeakEngine* will be also tested on smaller networks to study the trade-off between the reduced number of computations (and latency) and the algorithm overhead. This study will also help us to observe the performance knee point for different DNNs and use cases.

9.2.4 Additional DNNs, Datasets, and Use Cases

The *PeakGRU* algorithm and the *PeakEngine* were demonstrated on GRU-based DNNs and the SE task. They can, however, be both applied to other use cases as well as network architectures.

The future work therefore involves experimenting with *PeakGRU* on different DNN architectures, use cases, and datasets. Some of our preliminary studies on, e.g., KWS using GRU-based DNN topologies from [147] and GSCD [178] indicate that 50% (K_h=77) of computations and memory accesses from even the smallest setup with 154 hidden states can be skipped without any accuracy degradation, and 74% (K_h=40) while decreasing the accuracy by only ∼1.74% (the starting accuracy is ∼91.1%) when reusing weights from the baseline GRU. Similarly for a bigger network with 400 hidden states [147], only 75% of computations (K_h=100) are necessary to achieve 90.4% accuracy. When the networks are trained for a specific number of K elements, the reductions can be pushed even further. For instance, reducing 91% of operations (K_h=14) in the smallest network still produces high 88% accuracy.

9.2.5 Adaptive Thresholding

Chapter 5 explored a threshold-based pruning technique, *StatsRNN*, that tries to mimic the behavior of *PeakRNN*, i.e., selecting the top K elements every timesteps. The thresholds for input and hidden state sequences in *StatsRNN* are derived based on a dataset analysis that is performed prior to inference.

Ideally, the thresholds should not be static as the environment around us changes constantly, which impacts the fluctuations between the desired and the actual selected number of K elements every timestep. Hence, the thresholds should adapt to these changes as well.

We have studied this behavior and performed preliminary experiments with *adaptive thresholding*. The baseline idea of *adaptive thresholding* is simple. If the number of processed elements deviates from the desired K elements within tolerable boundaries, the threshold is preserved. However, if the number of processed elements is too high or too low, the thresholds are increased and decreased for the next timestep, respectively. Additionally, in our experiments, the amount of threshold tweaking was proportional to the deviation from the ideal K number of elements - the bigger the deviation, the higher/lower the threshold. The experiments showed promising results, especially for a low number of K values (e.g., $\sim$5% and 10% of the total operations), where the performance was on par with *PeakRNN*. However, the fluctuations from one timestep to another were still significant, which became prominent in regions with a higher K value. Therefore, further analysis and simulations are required to find the right feedback-loop control mechanism and amount of tweaking to achieve a relatively stable number of processed elements every timestep.

9.3 Final Remarks

Although the *PeakGRU* algorithm was developed to reduce the amount of computations in DNNs and hence save energy, it could find its application in other scenarios and areas.

For instance, a hearing instrument runs on a miniature battery that needs to last for almost an entire day. If the battery is low, the number of the processed top K values could adapt accordingly so that less energy is spent on running a DNN - while accepting a certain degradation of audio quality/overall performance. Therefore, the top K selection would be guided by the needs of the battery instead of the most optimal number of K elements for different environments.

Another example is *signal processing* and a use case of identifying the most likely direction of the sound, as shown in Figure 9.1). All the arrows in the figure mark possible sound targets, while the red arrows indicate the actual sound direction. In order to find the most likely target, we need to calculate a likelihood function for all the directions, i.e., the elements of a dictionary. However, the number of elements in a dictionary might sometimes exceed the number of likelihood functions that can be computed. Therefore, it could be beneficial to apply a

scheme such as *PeakGRU* to select and only evaluate K likelihood functions out of N dictionary elements.

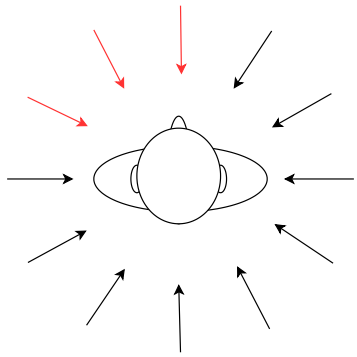


Figure 9.1: An illustration of the possible target sound directions, with the red arrows denoting the actual sound target.

In general, the potential of DNNs is vast. If we look into the future, we could envision a hearing instrument trained to specific voices, which would improve a speech-in-noise listening experience even further. Moreover, other sensors such as *Electrooculography (EOG)* and *Electroencephalography (EEG)* could be embedded into a hearing instrument to steer the directionality of the device in situations with multiple sound sources. For instance, in a face-to-face communication, human speech perception depends on both auditory and visual information [255]. With EOG, the eye movement of the hearing instrument user could be used to amplify the voice of the person that the user is looking at [156]. Lip-reading information [245, 224] can be also retrieved via a video/images. Since visual information is not affected by acoustic distortions, both approaches would be particularly beneficial in difficult listening situations. Furthermore, recent studies have shown that brain signals might also be extracted via EEG electrodes placed in the ear canal [251], referred to as Ear-EEG.

Yet, in order to exploit the power of DNNs directly in devices such as hearing instruments, it is of prime importance to co-design efficient hardware architectures and deep learning algorithms. This is the challenge we have tried to tackle in our research. We believe that with this project, we have made a step closer towards the future with intelligent, DNN-based, and energy-efficient hearing instruments capable of providing seamless adaptation in daily life and, as a result, a highly personalized and user-friendly experience for the hearing instrument users.

Bibliography

[1] Stanley Smith Stevens, John Volkmann, and Edwin Broomell Newman. "A Scale for the Measurement of the Psychological Magnitude Pitch". In: *The Journal of the Acoustical Society of America* 8.3 (1937), pp. 185–190.

[2] Warren S. McCulloch and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *The Bulletin of Mathematical Biophysics* 5 (1943), pp. 115–133.

[3] E. C. Cherry. "Some Experiments on the Recognition of Speech, with One and with Two Ears". In: *The Journal of the Acoustical Society of America* 25.5 (1953), pp. 975–979.

[4] David H Hubel and Torsten N Wiesel. "Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex". In: *The Journal of Physiology* 160.1 (1962), pp. 106–154.

[5] J. W. J. Williams. "Algorithm 232: Heapsort". In: *Communications of the ACM* 7.6 (1964), pp. 347–348.

[6] L. Chua. "Memristor-The Missing Circuit Element". In: *IEEE Transactions on Circuit Theory* 18.5 (1971), pp. 507–519.

[7] C. J. van Rijsbergen. "Information Retrieval". In: *Library Quarterly* 47.2 (1977), pp. 198–199.

[8] Steven Davis and Paul Mermelstein. "Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 28.4 (1980), pp. 357–366.

[9] Y. Ephraim and D. Malah. “Speech Enhancement Using a Minimum-Mean Square Error Short-Time Spectral Amplitude Estimator”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 32.6 (1984), pp. 1109–1121.

[10] Y. Ephraim and D. Malah. “Speech Enhancement Using a Minimum Mean-Square Error Log-Spectral Amplitude Estimator”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 33.2 (1985), pp. 443–445.

[11] L. Rabiner and B. Juang. “An Introduction to Hidden Markov Models”. In: *IEEE Acoustics, Speech and Signal Processing Magazine* 3.1 (1986), pp. 4–16.

[12] Stephen Hanson and Lorien Pratt. “Comparing Biases for Minimal Network Construction with Back-propagation”. In: *Advances in Neural Information Processing Systems* 1 (1988).

[13] Geoffrey J McLachlan and Kaye E Basford. *Mixture Models: Inference and Applications to Clustering*. Vol. 38. M. Dekker New York, 1988.

[14] Yann LeCun, John S. Denker, and Sara A. Solla. “Optimal Brain Damage”. In: *Advances in Neural Information Processing Systems (NIPS)*. Vol. 2. 1989, pp. 598–605.

[15] Albert S Bregman. *Auditory Scene Analysis: The Perceptual Organization of Sound*. MIT press, 1990.

[16] Babak Hassibi and David Stork. “Second Order Derivatives for Network Pruning: Optimal Brain Surgeon”. In: vol. 5. 1992, pp. 164–171.

[17] Michael Weintraub. “Keyword-spotting Using SRI’s DECIPHER Large-Vocabulary Speech-Recognition System”. In: *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. Vol. 2. 1993, pp. 463–466.

[18] Corinna Cortes and Vladimir Vapnik. “Support-Vector Networks”. In: *Machine Learning* 20.3 (1995), pp. 273–297.

[19] Zoubin Ghahramani and Michael I. Jordan. “Factorial Hidden Markov Models”. In: *Machine Learning* 29.2–3 (1997), pp. 245–273.

[20] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[21] Wolfgang Maass. “Networks of Spiking Neurons: The Third Generation of Neural Network Models”. In: *Neural Networks* 10.9 (1997), pp. 1659–1671.

[22] Alvin F. Martin, George R. Doddington, Terri Kamm, Mark Ordowski, and Mark A. Przybocki. “The DET Curve in Assessment of Detection Task Performance”. In: *European Conference on Speech Communication and Technology, (EUROSPEECH)*. 1997.

[23] Sergei Kochkin. "MarkeTrak V: "Why My Hearing Aids are in the Drawer" The Consumers' Perspective". In: *The Hearing Journal* 53.2 (2000), pp. 34–36.

[24] Akira Murata, Vittorio Gallese, Giuseppe Luppino, Masakazu Kaseda, and Hideo Sakata. "Selectivity for the Shape, Size, and Orientation of Objects for Grasping in Neurons of Monkey Parietal Area AIP". In: *Journal of Neurophysiology* 83.5 (2000), pp. 2580–2601.

[25] International Telecommunication Union (ITU-T). *Recommendation P.862: Perceptual Evaluation of Speech Quality (PESQ): An Objective Method for End-to-end Speech Quality Assessment of Narrow-band Telephone Networks and Speech Codecs*. 2001.

[26] Antony W Rix, John G Beerends, Michael P Hollier, and Andries P Hekstra. "Perceptual Evaluation of Speech Quality (PESQ) - A New Method for Speech Quality Assessment of Telephone Networks and Codecs". In: *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. Vol. 2. 2001, pp. 749–752.

[27] Barbara Hammer and Kai Gersmann. "A Note on the Universal Approximation Capability of Support Vector Machines". In: *Neural Processing Letters* 17.1 (2003), pp. 43–53.

[28] International Telecommunication Union (ITU-T). *Recommendation P.862.1: Mapping Function for Transforming P.862 Raw Result Scores to MOS-LQO*. 2003.

[29] Eugene M Izhikevich. "Simple Model of Spiking Neurons". In: *IEEE Transactions on Neural Networks* 14.6 (2003), pp. 1569–1572.

[30] Michael A Stone and Brian CJ Moore. "Tolerable Hearing Aid Delays. III. Effects on Speech Production and Perception of Across-Frequency Variation in Delay". In: *Ear and Hearing* 24.2 (2003), pp. 175–183.

[31] J. Tchorz and B. Kollmeier. "SNR Estimation Based on Amplitude Modulation Analysis with Applications to Noise Suppression". In: *IEEE Transactions on Speech and Audio Processing* 11.3 (2003), pp. 184–192.

[32] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. "Model Compression". In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2006, pp. 535–541.

[33] Jesse Davis and Mark Goadrich. "The Relationship Between Precision-Recall and ROC Curves". In: 2006, pp. 233–240.

[34] Tom Fawcett. "An Introduction to ROC Analysis". In: *Pattern Recognition Letters* 27.8 (2006), pp. 861–874.

[35] Aggelos Ioannou and Manolis Katevenis. "Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks". In: *IEEE/ACM Transactions on Networking* 15.2 (2007), pp. 450–461.

[36] David RH Miller, Michael Kleber, Chia-Lin Kao, Owen Kimball, Thomas Colthurst, Stephen A Lowe, Richard M Schwartz, and Herbert Gish. "Rapid and Accurate Spoken Term Detection". In: *Annual Conference of the International Speech Communication Association (INTERSPEECH)*. 2007, pp. 314–317.

[37] Suzanne Rigler, William Bishop, and Andrew Kennings. "FPGA-based Lossless Data Compression Using Huffman and LZ77 Algorithms". In: *Canadian Conference on Electrical and Computer Engineering*. 2007, pp. 1235–1238.

[38] M Angoletta. "Digital Signal Processor Fundamentals and System Design". In: *CERN Accelerator School: Digital Signal Processing* (Jan. 2008).

[39] Che-Wei Lin and Jeen-Shing Wang. "A Digital Circuit Design of Hyperbolic Tangent Sigmoid Function for Neural Networks". eng. In: 2008, pp. 856–859.

[40] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2009.

[41] J. Hovorka. "Methods for Evaluation of Speech Enhancement Algorithms". In: *Advances in Military Technology* 4.2 (2009), pp. 27–35.

[42] Gibak Kim, Yang Lu, Y. Hu, and Philipos C. Loizou. "An Algorithm that Improves Speech Intelligibility in Noise for Normal-Hearing Listeners". In: *The Journal of the Acoustical Society of America* 126.3 (2009), pp. 1486–1494.

[43] Yipeng Li and DeLiang Wang. "On the Optimality of Ideal Binary Time–Frequency Masks". In: *Speech Communication* 51.3 (2009), pp. 230–239.

[44] Toby Litovitz, Nicole Whitaker, and Lynn Clark. "Preventing Battery Ingestions: An Analysis of 8648 Cases". In: *Pediatrics* 125.6 (2010), pp. 1178–1183.

[45] Pramod Kumar Meher. "An Optimized Lookup-Table for the Evaluation of Sigmoid Function for Artificial Neural Networks". In: *IEEE/IFIP International Conference on VLSI and System-on-Chip*. 2010, pp. 91–95.

[46] P. Meinerzhagen, C. Roth, and A. Burg. "Towards Generic Low-Power Area-Efficient Standard Cell Based Memory Architectures". In: *Proceedings of the IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2010, pp. 129–132.

[47] Cees H. Taal, Richard C. Hendriks, Richard Heusdens, and Jesper Jensen. "A Short-Time Objective Intelligibility Measure for Time-Frequency Weighted Noisy Speech". In: *IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings (ICASSP)* (2010), pp. 4214–4217.

[48] Gisle Andersen. *Akustiske Database for Dansk*. 2011. URL: https://www.nb.no/sbfil/dok/nst_taledat_dk.pdf.

[49] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. "Natural Language Processing (almost) from Sscratch". In: *Journal of Machine Learning Research* 12 (2011), pp. 2493–2537.

[50] Philipos C. Loizou. "Speech Quality Assessment". In: *Multimedia Analysis, Processing and Communications*. Vol. 346. 2011, pp. 623–654.

[51] Cees H. Taal, Richard C. Hendriks, Richard Heusdens, and Jesper Jensen. "An algorithm for Intelligibility Prediction of Time-Frequency Weighted Noisy Speech". In: *IEEE/ACM Transactions on Audio, Speech and Language Processing* 19.7 (2011), pp. 2125–2136.

[52] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. "Improving the Speed of Neural Networks on CPUs". In: *Deep Learning and Unsupervised Feature Learning NIPS Workshop*. 2011.

[53] Zhiyao Duan, Gautham J. Mysore, and Paris Smaragdis. "Speech Enhancement by Online Non-Negative Spectrogram Decomposition in Non-stationary Noise Environments". In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2012, pp. 594–597.

[54] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. "Learning Hierarchical Features for Scene Labeling". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8 (2012), pp. 1915–1929.

[55] Kun Han and Deliang Wang. "A Classification Based Approach to Speech Segregation". In: *The Journal of the Acoustical Society of America* 132.5 (2012), pp. 3475–3483.

[56] D. Harris and S. Harris. *Digital Design and Computer Architecture, Second Edition*. 2nd. Morgan Kaufmann Publishers Inc., 2012.

[57] Ji Liu, Przemyslaw Musialski, Peter Wonka, and Jieping Ye. "Tensor Completion for Estimating Missing Values in Visual Data". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.1 (2012), pp. 208–220.

[58] Pascal Andreas Meinerzhagen, Oskar Andersson, Babak Mohammadi, S. M. Yasser Sherazi, Andreas Peter Burg, and Joachim Neves Rodrigues. "A 500 fW/Bit 14 fJ/Bit-Access 4kb Standard-cell Based Sub-VT Memory in 65nm CMOS". In: *Proceedings of the European Solid-State Circuit Conference (ESSCIRC)*. 2012, pp. 321–324.

[59] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation". In: *arXiv preprint arXiv:1308.3432* (2013).

[60] L. Deng et al. "Recent Advances in Deep Learning for Speech Research at Microsoft". In: *IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013, pp. 8604–8608.

[61] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. “Speech Recognition with Deep Recurrent Neural Networks”. In: *IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013, pp. 6645–6649.

[62] Eric W. Healy, Sarah E. Yoho, Yuxuan Wang, and Deliang Wang. “An Algorithm to Improve Speech Recognition in Noise for Hearing-Impaired Listeners”. In: *The Journal of the Acoustical Society of America* 134.4 (2013), pp. 3029–3038.

[63] Richard C. Hendriks, Timo Gerkmann, and Jesper Jensen. “DFT-Domain based Single-Microphone Noise Reduction for Speech Enhancement: A Survey of the State of the Art”. In: *Synthesis Lectures on Speech and Audio Processing* 9.1 (2013), pp. 1–80.

[64] Philipos C. Loizou. *Speech Enhancement: Theory and Practice*. 2nd. CRC Press, 2013.

[65] A. McCormack and H. Fortnum. “Why Do People Fitted with Hearing Aids not Wear Them?” In: *International Journal of Audiology*. 2013, pp. 360–368.

[66] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. “Playing Atari with Deep Reinforcement Learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[67] A. Narayanan and D. Wang. “Ideal Ratio Mask Estimation Using Deep Neural Networks for Robust Speech Recognition”. In: *IEEE International Conference on Acoustics, Speech and Signal Processing* (2013), pp. 7092–7096.

[68] NIST. “OpenKWS 13 Keyword Search Evaluation Plan”. In: *National Institute of Standards and Technology, Evaluation plan*. 2013.

[69] Yuxuan Wang and DeLiang Wang. “Towards Scaling Up Classification-Based Speech Separation”. In: *IEEE Transactions on Audio, Speech, and Language Processing* 21.7 (2013), pp. 1381–1390.

[70] Guoguo Chen, Carolina Parada, and Georg Heigold. “Small-Footprint Keyword Spotting Using Deep Neural Networks”. In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2014, pp. 4087–4091.

[71] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. “Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation”. In: *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1724–1734.

[72] Andrew Davis and Itamar Arel. “Low-rank Approximations for Conditional Feedforward Computation in Deep Neural Networks”. In: *International Conference on Learning Representations (ICLR)*. 2014.

[73] M. Horowitz. “1.1 Computing’s Energy Problem (and what we can do about it)”. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. Vol. 57. 2014, pp. 10–14.

[74] Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. “On Using Very Large Target Vocabulary for Neural Machine Translation”. In: *Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (ACL)*. 2014, pp. 1–10.

[75] Jesper Jensen and Cees H. Taal. “Speech Intelligibility Prediction Based on Mutual Information”. In: *IEEE/ACM Transactions on Audio, Speech and Language Processing* 22.2 (2014), pp. 430–440.

[76] N. G. Chetan Kumar, Sudhanshu Vyas, Ron K. Cytron, Christopher D. Gill, Joseph Zambreno, and Phillip H. Jones. “Hardware-Software Architecture for Priority Queue Management in Real-time and Embedded Systems”. In: *International Journal of Embedded Systems (IJES)* 6.4 (2014), pp. 319–334.

[77] Michael KK Leung, Hui Yuan Xiong, Leo J Lee, and Brendan J Frey. “Deep Learning of the Tissue-Regulated Splicing Code”. In: *Bioinformatics* 30.12 (2014), pp. 121–129.

[78] Ding Liu, P. Smaragdis, and Minje Kim. “Experiments on Deep Learning for Speech Denoising”. In: *International Speech Communication Association (INTERSPEECH)*. 2014, pp. 2685–2689.

[79] Paul Merolla et al. “A Million Spiking-Neuron Integrated Circuit with a Scalable Communication Network and Interface”. In: *Science* 345.6197 (2014), pp. 668–673.

[80] Karen Simonyan and Andrew Zisserman. “Two-Stream Convolutional Networks for Action Recognition in Videos”. In: *Advances in Neural Information Processing Systems (NIPS)*. Vol. 27. 2014, pp. 568–576.

[81] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to Sequence Learning with Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 27. 2014, pp. 3104–3112.

[82] Yuxuan Wang, Arun Narayanan, and DeLiang Wang. “On Training Targets for Supervised Speech Separation”. In: *IEEE/ACM Transactions on Audio, Speech and Language Processing* 22.12 (2014), pp. 1849–1858.

[83] Harvey B Abrams and Jan Kihm. “An Introduction to MarkeTrak IX: A New Baseline for the Hearing Aid Market”. In: *Hearing Review* 22.6 (2015), p. 16.

[84] S. Anwar, K. Hwang, and W. Sung. “Fixed Point Optimization of Deep Convolutional Neural Networks for Object Recognition”. In: *Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2015, pp. 1131–1135.

[85] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. “Compressing Neural Networks with the Hashing Trick”. In: *International Conference on Machine Learning (ICML)*. 2015, pp. 2285–2294.

[86] Matthieu Courbariaux, Y. Bengio, and Jean-Pierre David. “Low Precision Arithmetic for Deep Learning”. In: *International Conference on Learning Representations (ICLR)* (2015).

[87] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations”. In: *Advances in Neural Information Processing Systems*. Vol. 28. 2015, pp. 3123–3131.

[88] Hakan Erdogan, John R. Hershey, Shinji Watanabe, and Jonathan Le Roux. “Phase-Sensitive and Recognition-Boosted Speech Separation Using Deep Recurrent Neural Networks”. In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2015, pp. 708–712.

[89] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. “Deep Learning with Limited Numerical Precision”. In: *Proceedings of the International Conference on Machine Learning*. 2015, pp. 1737–1746.

[90] Song Han, Jeff Pool, John Tran, and William Dally. “Learning both Weights and Connections for Efficient Neural Network”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 28. 2015, pp. 1135–1143.

[91] Eric W. Healy, Sarah E. Yoho, Jitong Chen, Yuxuan Wang, and Deliang Wang. “An Algorithm to Increase Speech Intelligibility for Hearing-Impaired Listeners in Novel Segments of the Same Noise Type”. In: *The Journal of the Acoustical Society of America* 138.3 (2015), pp. 1660–1669.

[92] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, et al. “Distilling the Knowledge in a Neural Network”. In: *CoRR* abs/1503.02531 (2015). URL: http://arxiv.org/abs/1503.02531.

[93] Søren Jørgensen, Jens Cubick, and Torsten Dau. “Speech Intelligibility Evaluation for Mobile Phones”. In: *Acustica United With Acta Acustica* 101.5 (2015), pp. 915–919.

[94] Junshui Ma, Robert P Sheridan, Andy Liaw, George E Dahl, and Vladimir Svetnik. “Deep Neural Nets as a Method for Quantitative Structure–Activity Relationships”. In: *Journal of Chemical Information and Modeling* 55.2 (2015), pp. 263–274.

[95] Rohit Prabhavalkar, Raziel Alvarez, Carolina Parada, Preetum Nakkiran, and Tara N Sainath. "Automatic Gain Control and Multi-Style Training for Robust Small-Footprint Keyword Spotting with Deep Neural Networks". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2015, pp. 4704–4708.

[96] O. Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252.

[97] Tara N. Sainath and Carolina Parada. "Convolutional Neural Networks for Small-Footprint Keyword Spotting". In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2015.

[98] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going Deeper with Convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1–9.

[99] Y. Xu, J. Du, L. Dai, and C. Lee. "A Regression Approach to Speech Enhancement Based on Deep Neural Networks". In: *IEEE/ACM Transactions on Audio, Speech and Language Processing* 23.1 (2015), pp. 7–19.

[100] Tianhao Zhang, Gregory Kahn, Sergey Levine, and Pieter Abbeel. "Learning Deep Control Policies for Autonomous Aerial Vehicles with MPC-Guided Policy Search". In: *IEEE International Conference on Robotics and Automation (ICRA)* (2015), pp. 528–535.

[101] Oskar Andersson, Babak Mohammadi, Pascal Meinerzhagen, Andreas Burg, and Joachim Neves Rodrigues. "Ultra Low Voltage Synthesizable Memories: A Trade-Off Discussion in 65 nm CMOS". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 63-I.6 (2016), pp. 806–817.

[102] James Atwood and Don Towsley. "Diffusion-Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems* 29 (2016).

[103] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks". In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 367–379.

[104] Jitong Chen, Yuxuan Wang, Sarah E. Yoho, Deliang Wang, and Eric W. Healy. "Large-Scale Training to Increase Speech Intelligibility for Hearing-Impaired Listeners in Novel Noises." In: *The Journal of the Acoustical Society of America* 139.5 (2016), pp. 2604–2612.

[105] Matthieu Courbariaux and Yoshua Bengio. "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". In: *ArXiv* abs/1602.02830 (2016).

[106] Yiming Cui, Shijin Wang, and Jianfeng Li. "LSTM Neural Reordering Feature for Statistical Machine Translation". In: *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL:HLT)* (2016), pp. 977–982.

[107] Sukru Burc Eryilmaz, Siddharth Joshi, Emre Neftci, Weier Wan, Gert Cauwenberghs, and H-S Philip Wong. "Neuromorphic Architectures with Electronic Synapses". In: *International Symposium on Quality Electronic Design (ISQED)*. 2016, pp. 118–123.

[108] Yin Fan, Xiangju Lu, Dian Li, and Yuanliu Liu. "Video-based Emotion Recognition Using CNN-RNN and C3D Hybrid Networks". In: *ACM International Conference on Multimodal Interaction*. 2016, pp. 445–450.

[109] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.

[110] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: 2016.

[111] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 29. 2016, pp. 4107–4115.

[112] Morten Kolbaek, Zheng-Hua Tan, and Jesper Jensen. "Speech Enhancement Using Long Short-Term Memory Based Recurrent Neural Networks for Noise Robust Speaker Verification". In: *IEEE spoken language technology workshop (SLT)*. 2016, pp. 305–311.

[113] Vadim Lebedev and Victor Lempitsky. "Fast ConvNets Using Group-Wise Brain Damage". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2554–2564.

[114] Fengfu Li and Bin Liu Liu. "Ternary Weight Networks". In: *NIPS Workshop on Efficient Methods for Deep Neural Networks*. 2016.

[115] Daisuke Miyashita, Edward H Lee, and Boris Murmann. "Convolutional Neural Networks using Logarithmic Data Representation". In: *CoRR* abs/1603.01025 (2016). URL: http://arxiv.org/abs/1603.01025.

[116] Bert Moons, Bert De Brabandere, Luc Van Gool, and Marian Verhelst. "Energy-Efficient ConvNets through Approximate Computing". In: *IEEE Winter Conference on Applications of Computer Vision (WACV)*. 2016, pp. 1–8.

[117] Bert Moons and Marian Verhelst. "A 0.3–2.6 TOPS/W Precision-Scalable Processor for Real-Time Large-Scale ConvNets". In: *IEEE Symposium on VLSI Circuits*. 2016, pp. 1–2.

[118] J. Qiu et al. "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network". In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2016, pp. 26–35.

[119] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks". In: *European Conference on Computer Vision*. 2016, pp. 525–542.

[120] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars". In: *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2016, pp. 14–26.

[121] D. Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529 (2016), pp. 484–489.

[122] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. "Learning Structured Sparsity in Deep Neural Networks". In: vol. 29. 2016, pp. 2074–2082.

[123] Hao Zhou, Jose M Alvarez, and Fatih Porikli. "Less is More: Towards Compact CNNs". In: *European Conference on Computer Vision*. 2016, pp. 662–677.

[124] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. "Structured Pruning of Deep Convolutional Neural Networks". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13.3 (2017), pp. 1–18.

[125] S. Bang et al. "14.7 A 288μW Programmable Deep-Learning Processor with 270KB On-Chip Weight Storage Using Non-Uniform Memory Hierarchy for Mobile Intelligence". In: *IEEE International Solid-State Circuits Conference (ISSCC)*. 2017, pp. 250–251.

[126] A. Esteva, B. Kuprel, R. Novoa, J. Ko, S. Swetter, H. Blau, and S. Thrun. "Dermatologist-Level Classification of Skin Cancer with Deep Neural Networks". In: *Nature* 542 (2017), pp. 115–118.

[127] Xin Fan, Jan Stuijt, Rui Wang, Bo Liu, and Tobias Gemmeke. "Re-addressing SRAM Design and Measurement for Sub-Threshold Operation in View of Classic 6T vs. Standard Cell Based Implementations". In: *Proceedings of the International Symposium on Quality Electronic Design (ISQED)*. 2017, pp. 65–70.

[128] Marc Ciufo Green and Damian Murphy. *EigenScape*. Zenodo, 2017. DOI: 10.5281/zenodo.1012809. URL: https://doi.org/10.5281/zenodo.1012809.

[129] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA". In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 75–84.

[130] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens Van Der Maaten, and Kilian Q Weinberger. "Multi-Scale Dense Convolutional Networks for Efficient Prediction". In: *CoRR* abs/1703.09844 (2017). URL: http://arxiv.org/abs/1703.09844.

[131] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6869–6898.

[132] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[133] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. "Pruning Filters for Efficient ConvNets". In: *International Conference on Learning Representations (ICLR)*. 2017.

[134] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. "Runtime Neural Pruning". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 30. 2017, pp. 2181–2191.

[135] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. "Learning Efficient Convolutional Networks through Network Slimming". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 2736–2744.

[136] Bert Moons, Koen Goetschalckx, Nick Van Berckelaer, and Marian Verhelst. "Minimum Energy Quantized Neural Networks". In: *Asilomar Conference on Signals, Systems, and Computers*. 2017, pp. 1921–1925.

[137] Bert Moons, Roel Uytterhoeven, Wim Dehaene, and Marian Verhelst. "14.5 Envision: A 0.26-to-10TOPS/W Subword-Parallel Dynamic-Voltage-Accuracy-Frequency-Scalable Convolutional Neural Network Processor in 28nm FDSOI". In: *IEEE International Solid-State Circuits Conference (ISSCC)*. 2017, pp. 246–247.

[138] Daniel Neil, Junhaeng Lee, Tobi Delbrück, and Shih-Chii Liu. "Delta Networks for Optimized Recurrent Network Computation". In: *International Conference on Machine Learning (ICML)*. Vol. 70. 2017, pp. 2584–2593.

[139] Santiago Pascual, Antonio Bonafonte, and Joan Serrà. "SEGAN: Speech Enhancement Generative Adversarial Network". In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2017, pp. 3642–3646.

[140] M. Pfeiffer, M. Schaeuble, J. Nieto, R. Siegwart, and C. Cadena. "From Perception to Decision: A Data-Driven Approach to End-to-End Motion Planning for Autonomous Ground Robots". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 1527–1533.

[141] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer". In: *International Conference on Learning Representations (ICLR)*. 2017.

[142] Dongjoo Shin, Jinmook Lee, Jinsu Lee, and Hoi-Jun Yoo. "14.2 DNPU: An 8.1 TOPS/W Reconfigurable CNN-RNN Processor for General-Purpose Deep Neural Networks". In: *International Solid-State Circuits Conference (ISSCC)*. 2017, pp. 240–241.

[143] V. Sze, Y. Chen, T. Yang, and J. S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[144] Cesar Torres-Huitzil and Bernard Girau. "Fault and Error Tolerance in Neural Networks: A Review". In: *IEEE Access* 5 (2017), pp. 17322–17341.

[145] Karen Ullrich, Edward Meeds, and Max Welling. "Soft Weight-Sharing for Neural Network Compression". In: *International Conference on Learning Representations (ICLR)*. 2017.

[146] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017, pp. 5998–6008.

[147] Y. Zhang, N. Suda, L. Lai, and V. Chandra. "Hello Edge: Keyword Spotting on Microcontrollers". In: *CoRR* abs/1711.07128 (2017). URL: http://arxiv.org/abs/1711.07128.

[148] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. "Trained Ternary Quantization". In: *International Conference on Learning Representations, (ICLR)*. 2017.

[149] Byungmin Ahn and Taewhan Kim. "Memory Access Driven Memory Layout and Block Replacement Techniques for Compressed Deep Neural Networks". In: *Proceedings of the IEEE International System-on-Chip Conference (SOCC)*. 2018, pp. 221–226.

[150] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B Milde, Federico Corradi, Alejandro Linares-Barranco, Shih-Chii Liu, et al. "NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps". In: *IEEE Transactions on Neural Networks and Learning Systems* 30.3 (2018), pp. 644–656.

[151] R. Banner, I. Hubara, E. Hoffer, and D. Soudry. "Scalable Methods for 8-Bit Training of Neural Networks". In: *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*. 2018, pp. 5151–5159.

[152] DL Beck et al. “Audiologic Considerations for People with Normal Hearing Sensitivity yet Hearing Difficulty and/or Speech-in-Noise Problems”. In: *The Hearing Review* 25.10 (2018), pp. 28–38.

[153] Thomas Bentsen, Tobias May, Abigail A. Kressner, and Torsten Dau. “The Benefit of Combining a Deep Neural Network Architecture with Ideal Ratio Mask Estimation in Computational Speech Segregation to Improve Speech Intelligibility”. In: *PLOS ONE* 13.5 (2018), pp. 1–13.

[154] D. Das et al. “Mixed Precision Training of Convolutional Neural Networks using Integer Operations”. In: *International Conference on Learning Representations (ICLR)*. 2018.

[155] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. “Universal Transformers”. In: *CoRR* abs/1807.03819 (2018). URL: http://arxiv.org/abs/1807.03819.

[156] Antoine Favre-Felix, Carina Graversen, Renskje K. Hietkamp, Torsten Dau, and Thomas Lunner. “Improving Speech Intelligibility by Hearing Aid Eye-Gaze Steering: Conditions with Head Fixated in a Multitalker Environment”. In: *Trends in Hearing* 22 (2018), pp. 1–13.

[157] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. “DeltaRNN: A Power-Efficient Recurrent Neural Network Accelerator”. In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2018, pp. 21–30.

[158] Juan Sebastian P Giraldo and Marian Verhelst. “Laika: A 5uW Programmable LSTM Accelerator for Always-On Keyword Spotting in 65nm CMOS”. In: *IEEE European Solid State Circuits Conference (ESSCIRC)*. 2018, pp. 166–169.

[159] Matthew B Hoy. “Alexa, Siri, Cortana, and More: An Introduction to Voice Assistants”. In: *Medical Reference Services Quarterly* 37.1 (2018), pp. 81–88.

[160] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. “Multi-Scale Dense Networks for Resource Efficient Image Classification”. In: *International Conference on Learning Representations (ICLR)*. 2018.

[161] Gao Huang, Shichen Liu, Laurens Van der Maaten, and Kilian Q Weinberger. “CondenseNet: An Efficient DenseNet Using Learned Group Convolutions”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2752–2761.

[162] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2704–2713.

[163] Morten Kolbæk. "Single-Microphone Speech Enhancement and Separation Using Deep Learning". In: *arXiv preprint arXiv:1808.10620* (2018).

[164] Aditya Kusupati, M. Singh, Kush Bhatia, A. Kumar, P. Jain, and M. Varma. "FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2018, pp. 9031–9042.

[165] Craig Macartney and Tillman Weyde. "Improved Speech Enhancement with the Wave-U-Net". In: *ArXiv* abs/1811.11307 (2018).

[166] Adrien Meynard and Bruno Torresani. "Spectral Analysis for Nonstationary Audio". In: *IEEE/ACM Transactions on Audio Speech and Language Processing* 26.12 (2018), pp. 2371–2380.

[167] Samuel Myer and Vikrant Singh Tomar. "Efficient Keyword Spotting Using Time Delay Neural Networks". In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2018, pp. 1264–1268.

[168] Junki Park, Jaeha Kung, Wooseok Yi, and Jae-Joon Kim. "Maximizing System Performance by Balancing Computation Loads in LSTM Accelerators". In: *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2018, pp. 7–12.

[169] Bruno U Pedroni, Sadique Sheik, Hesham Mostafa, Somnath Paul, Charles Augustine, and Gert Cauwenberghs. "Small-Footprint Spiking Neural Networks for Power-Efficient Keyword Spotting". In: *IEEE Biomedical Circuits and Systems Conference (BioCAS)*. 2018, pp. 1–4.

[170] Changhao Shan, Junbo Zhang, Yujun Wang, and Lei Xie. "Attention-Based End-to-End Models for Small-Footprint Keyword Spotting". In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2018, pp. 2037–2041.

[171] Anil Shanbhag, Holger Pirk, and Samuel Madden. "Efficient Top-K Query Processing on Massively Parallel Hardware". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2018, pp. 1557–1570.

[172] Franyell Silfa, Gem Dot, Jose-Maria Arnau, and Antonio Gonzalez. "E-PUR: An energy-efficient processing unit for recurrent neural networks". In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 2018, pp. 1–12.

[173] Ke Tan and DeLiang Wang. "A Convolutional Recurrent Neural Network for Real-Time Speech Enhancement". In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2018, pp. 3229–3233.

[174] Raphael Tang and Jimmy Lin. "Deep Residual Learning for Small-Footprint Keyword Spotting". In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2018, pp. 5484–5488.

[175] Jean-Marc Valin. "A Hybrid DSP/Deep Learning Approach to Real-Time Full-Band Speech Enhancement". In: *International Workshop on Multimedia Signal Processing (MMSP)*. 2018, pp. 1–5.

[176] Yiyan Wang and Yanhua Long. "Keyword Spotting Based on CTC and RNN for Mandarin Chinese Speech". In: *International Symposium on Chinese Spoken Language Processing (ISCSLP)*. 2018, pp. 374–378.

[177] Zhisheng Wang, Jun Lin, and Zhongfeng Wang. "Hardware-Oriented Compression of Long Short-Term Memory for Efficient Inference". In: *IEEE Signal Processing Letters* 25.7 (2018), pp. 984–988.

[178] Pete Warden. "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition". In: *CoRR* abs/1804.03209 (2018). URL: http://arxiv.org/abs/1804.03209.

[179] Kevin W. Wilson, Michael Chinen, Jeremy Thorpe, Brian Patton, John R. Hershey, Rif A. Saurous, Jan Skoglund, and Richard F. Lyon. "Exploring Tradeoffs in Models for Low-Latency Speech Enhancement". In: *International Workshop on Acoustic Signal Enhancement (IWAENC)* (2018), pp. 366–370.

[180] S. Yin, P. Ouyang, S. Zheng, D. Song, X. Li, L. Liu, and S. Wei. "A 141 UW, 2.46 pJ/Neuron Binarized Convolutional Neural Network Based Self-Learning Speech Recognition Processor in 28nm CMOS". In: *IEEE Symposium on VLSI Circuits*. 2018, pp. 139–140.

[181] Raziel Alvarez and Hyun-Jin Park. "End-to-End Streaming Keyword Spotting". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 6336–6340.

[182] Arash Ardakani, Zhengyun Ji, and Warren J Gross. "Learning to Skip Ineffectual Recurrent Computations in LSTMs". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019, pp. 1427–1432.

[183] Ye Bai, Jiangyan Yi, Jianhua Tao, Zhengqi Wen, Zhengkun Tian, Chenghao Zhao, and Cunhang Fan. "A Time Delay Neural Network with Shared Weight Self-Attention for Small-Footprint Keyword Spotting". In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2019, pp. 2190–2194.

[184] Shijie Cao, Chen Zhang, Zhuliang Yao, Wencong Xiao, Lanshun Nie, Dechen Zhan, Yunxin Liu, Ming Wu, and Lintao Zhang. "Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity". In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2019, pp. 63–72.

[185] Xi Chen, Shouyi Yin, Dandan Song, Peng Ouyang, Leibo Liu, and Shaojun Wei. "Small-Footprint Keyword Spotting with Graph Convolutional Network". In: *IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. 2019, pp. 539–546.

[186] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. 2019, pp. 4171–4186.

[187] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural Architecture Search: A Survey". In: *The Journal of Machine Learning Research* 20.1 (2019), pp. 1997–2017.

[188] Jonathan Frankle and Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks". In: *International Conference on Learning Representations, (ICLR)*. 2019. URL: https://openreview.net/forum?id=rJl-b3RcF7.

[189] Xitong Gao, Yiren Zhao, Łukasz Dudziak, Robert Mullins, and Cheng-zhong Xu. "Dynamic Channel Pruning: Feature Boosting and Suppression". In: *International Conference on Learning Representations (ICLR)*. 2019.

[190] Anil Kag, Ziming Zhang, and Venkatesh Saligrama. "RNNs Evolving in Equilibrium: A Solution to the Vanishing and Exploding Gradients". In: *ArXiv* abs/1908.08574 (2019).

[191] Jaeha Kung, Junki Park, Sehun Park, and Jae-Joon Kim. "Peregrine: A Flexible Hardware Accelerator for LSTM with Limited Synaptic Connection Patterns". In: *Proceedings of the Annual Design Automation Conference (DAC)*. 2019, pp. 1–6.

[192] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. "Improving Multi-Task Deep Neural Networks via Knowledge Distillation for Natural Language Understanding". In: *CoRR* abs/1904.09482 (2019). URL: http://arxiv.org/abs/1904.09482.

[193] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *CoRR* abs/1907.11692 (2019). URL: http://arxiv.org/abs/1907.11692.

[194] Yun Liu, Hui Zhang, Xueliang Zhang, and Linju Yang. "Supervised Speech Enhancement with Real Spectrum Approximation". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 5746–5750.

[195] J. S. McCarley. "Pruning a BERT-based Question Answering Model". In: *CoRR* abs/1910.06360 (2019). URL: http://arxiv.org/abs/1910.06360.

[196] Paul Michel, Omer Levy, and Graham Neubig. "Are Sixteen Heads Really Better than One?" In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 32. 2019, pp. 14014–14024.

[197] B. Moons, D. Bankman, and M. Verhelst. *Embedded Deep Learning: Algorithms, Architectures and Circuits for Always-on Neural Network Processing*. Springer, 2019.

[198] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. "Language Models are Unsupervised Multitask Learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[199] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. "DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter". In: *CoRR* abs/1910.01108 (2019). URL: http://arxiv.org/abs/1910.01108.

[200] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. "Patient Knowledge Distillation for BERT Model Compression". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2019, pp. 4322–4331.

[201] Raphael Tang, Yao Lu, Linqing Liu, Lili Mou, Olga Vechtomova, and Jimmy Lin. "Distilling Task-Specific Knowledge from BERT into Simple Neural Networks". In: *CoRR* abs/1903.12136 (2019). URL: http://arxiv.org/abs/1903.12136.

[202] Jin Tao, Urmish Thakker, Ganesh Dasika, and Jesse Beu. "Skipping RNN State Updates Without Retraining the Original Model". In: *Proceedings of the Workshop on Machine Learning on Edge in Sensor Systems*. 2019, pp. 31–36.

[203] *TMS320C6652 and TMS320C6654 Fixed and Floating-Point Digital Signal Processor*. TMS320C6652 TMS320C6654. Texas Instruments. 2019.

[204] C. Veaux, J. Yamagishi, and Kirsten Macdonald. "CSTR VCTK Corpus: English Multi-Speaker Corpus for CSTR Voice Cloning Toolkit". In: 2019. URL: https://datashare.ed.ac.uk/handle/10283/3443.

[205] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. "Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned". In: *Proceedings of the Conference of the Association for Computational Linguistics (ACL)*. 2019, pp. 5797–5808.

[206] Pete Warden and Daniel Situnayake. *Tinyml: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2019.

[207] Jie-Fang Zhang, Ching-En Lee, Chester Liu, Yakun Sophia Shao, Stephen W. Keckler, and Zhengya Zhang. "SNAP: A 1.67 - 21.55TOPS/W Sparse Neural Acceleration Processor for Unstructured Sparse Deep Neural Network Inference in 16nm CMOS". In: *Symposium on VLSI Circuits*. 2019, pp. 306–307.

[208] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. "Sparse Tensor Core: Algorithm and Hardware Co-design for Vector-Wise Sparse Neural Networks on Modern GPUs". In: *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2019, pp. 359–371.

[209] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. "What is the State of Neural Network Pruning?" In: 2 (2020), pp. 129–146.

[210] Tom Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems, (NeurIPS)*. Vol. 33. 2020, pp. 1877–1901.

[211] Qinyu Chen, Yan Huang, Rui Sun, Wenqing Song, Zhonghai Lu, Yuxiang Fu, and Li Li. "An Efficient Accelerator for Multiple Convolutions from the Sparsity Perspective". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.6 (2020), pp. 1540–1544.

[212] Alexandre Défossez, Gabriel Synnaeve, and Yossi Adi. "Real Time Speech Enhancement in the Waveform Domain". In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2020, pp. 3291–3295.

[213] Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. "Depth-Adaptive Transformer". In: *International Conference on Learning Representations (ICLR)*. 2020. URL: https://openreview.net/forum?id=SJg7KhVKPH.

[214] Igor Fedorov, Marko Stamenovic, Carl Jensen, Li-Chia Yang, Ari Mandell, Yiming Gan, Matthew Mattina, and Paul N. Whatmough. "TinyLSTMs: Efficient Neural Speech Enhancement for Hearing Aids". In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2020, pp. 4054–4058.

[215] Chang Gao, Antonio Rios-Navarro, Xi Chen, Shih-Chii Liu, and Tobi Delbruck. "EdgeDRNN: Recurrent Neural Network Accelerator for Edge Inference". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 10.4 (2020), pp. 419–432.

[216] J. S. P. Giraldo, S. Lauwereins, K. Badami, and M. Verhelst. "Vocell: A 65-nm Speech-Triggered Wake-Up SoC for 10-μW Keyword Spotting and Speaker Verification". In: *IEEE Journal of Solid-State Circuits* 55.4 (2020), pp. 868–878.

[217] Mitchell Gordon, Kevin Duh, and Nicholas Andrews. “Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning”. In: *Proceedings of the Workshop on Representation Learning for NLP*. 2020, pp. 143–155.

[218] Saurabh Goyal, Anamitra Roy Choudhury, Saurabh Raje, Venkatesan T. Chakaravarthy, Yogish Sabharwal, and Ashish Verma. “PoWER-BERT: Accelerating BERT Inference via Progressive Word-Vector Elimination”. In: *International Conference on Machine Learning (ICML)*. Vol. 119. 2020, pp. 3690–3699.

[219] Anmol Gulati et al. “Conformer: Convolution-Augmented Transformer for Speech Recognition”. In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2020, pp. 5036–5040.

[220] Siddhant M. Jayakumar, Razvan Pascanu, Jack W. Rae, Simon Osindero, and Erich Elsen. “Top-KAST: Top-K Always Sparse Training”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.

[221] Deepak Kadetotad, Shihui Yin, Visar Berisha, Chaitali Chakrabarti, and Jae-Sun Seo. “An 8.93 TOPS/W LSTM Recurrent Neural Network Accelerator Featuring Hierarchical Coarse-Grain Sparsity for On-Device Speech Recognition”. In: *IEEE Journal of Solid-State Circuits* 55.7 (2020), pp. 1877–1887.

[222] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. “ALBERT: A Lite BERT for Self-supervised Learning of Language Representations”. In: *International Conference on Learning Representations, (ICLR)*. 2020.

[223] Ximin Li, Xiaodong Wei, and Xiaowei Qin. “Small-Footprint Keyword Spotting With Multi-Scale Temporal Convolution”. In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2020, pp. 1987–1991.

[224] Liliane Momeni, Triantafyllos Afouras, Themos Stafylakis, Samuel Albanie, and Andrew Zisserman. “Seeing Wake Words: Audio-Visual Keyword Spotting”. In: *British Machine Vision Conference 2020 (BMVC)*. 2020.

[225] Oticon. *OticonMore*. 2020. URL: https://wdh01.azureedge.net/-/media/oticon/main/pdf/master/more/pbr/223211uk_pbr_concept_brochure_oticon_more.pdf?rev=918D&la=tr-TR.

[226] W. Shan, M. Yang, J. Xu, Y. Lu, S. Zhang, T. Wang, J. Yang, L. Shi, and M. Seok. “14.1 A 510nW 0.41V Low-Memory Low-Computation Keyword-Spotting Chip Using Serial FFT-Based MFCC and Binarized Depthwise Separable Convolutional Neural Network in 28nm CMOS”. In: *IEEE International Solid-State Circuits Conference (ISSCC)*. 2020, pp. 230–232.

[227] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. "MobileBERT: A Compact Task-Agnostic BERT for Resource-Limited Devices". In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. 2020, pp. 2158–2170.

[228] Daiki Takeuchi, Kohei Yatabe, Yuma Koizumi, Yasuhiro Oikawa, and Noboru Harada. "Real-Time Speech Enhancement Using Equilibriated RNN". In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2020, pp. 851–855.

[229] Yulin Wang, Kangchen Lv, Rui Huang, Shiji Song, Le Yang, and Gao Huang. "Glance and Focus: A Dynamic Approach to Reducing Spatial Redundancy in Image Classification". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 2432–2444.

[230] Menglong Xu and Xiao-Lei Zhang. "Depthwise Separable Convolutional ResNet with Squeeze-and-Excitation Blocks for Small-Footprint Keyword Spotting". In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2020, pp. 2547–2551.

[231] Le Yang, Yizeng Han, Xi Chen, Shiji Song, Jifeng Dai, and Gao Huang. "Resolution Adaptive Networks for Efficient Inference". In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 2369–2378.

[232] World Health Organization (WHO). *Deafness and Hearing Loss*. 2021. URL: https://www.who.int/news-room/fact-sheets/detail/deafness-and-hearing-loss (visited on 09/12/2022).

[233] Axel Berg, Mark O'Connor, and Miguel Tairum Cruz. "Keyword Transformer: A Self-Attention Model for Keyword Spotting". In: *Conference of the International Speech Communication Association (INTERSPEECH)*. 2021, pp. 4249–4253.

[234] Feng-Ju Chang, Martin Radfar, Athanasios Mouchtaris, Brian King, and Siegfried Kunzmann. "End-to-End Multi-Channel Transformer for Speech Recognition". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2021, pp. 5884–5888.

[235] Xie Chen, Yu Wu, Zhenghao Wang, Shujie Liu, and Jinyu Li. "Developing Real-Time Streaming Transformer Transducer for Speech Recognition on Large-Scale Dataset". In: *IEEE International Conference on Acoustics, Speech and Signal Processing, (ICASSP)*. 2021, pp. 5904–5908.

[236] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *International Conference on Learning Representations (ICLR)*. 2021.

[237] Amin Edraki, Wai Yip Chan, Jesper Jensen, and Daniel Fogerty. "Speech Intelligibility Prediction using Spectro-Temporal Modulation Analysis". In: *IEEE/ACM Transactions on Audio Speech and Language Processing* 29 (2021), pp. 210–225.

[238] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. "A Survey of Quantization Methods for Efficient Neural Network Inference". In: *CoRR* abs/2103.13630 (2021). URL: https://arxiv.org/abs/2103.13630.

[239] Amirhossein Habibian, Davide Abati, Taco S. Cohen, and Babak Ehteshami Bejnordi. "Skip-Convolutions for Efficient Video Processing". In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021, pp. 2695–2704.

[240] Christian Janiesch, Patrick Zschech, and Kai Heinrich. "Machine Learning and Deep Learning". In: *Electronic Markets* 31.3 (2021), pp. 685–695.

[241] Gyuwan Kim and Kyunghyun Cho. "Length-Adaptive Transformer: Train Once with Length Drop, Use Anytime with Search". In: *Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing (ACL/IJCNLP)*. 2021, pp. 6501–6511.

[242] Jiaoda Li, Ryan Cotterell, and Mrinmaya Sachan. "Differentiable Subset Pruning of Transformer Heads". In: *Transactions of the Association for Computational Linguistics* 9 (2021), pp. 1442–1459.

[243] Tailin Liang, C. John Glossner, Lei Wang, and Shaobo Shi. "Pruning and Quantization for Deep Neural Network Acceleration: A Survey". In: *Neurocomputing* 461 (2021), pp. 370–403.

[244] Andy T. Liu, Shang-Wen Li, and Hung-yi Lee. "TERA: Self-Supervised Learning of Transformer Encoder Representation for Speech". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 29 (2021), pp. 2351–2366.

[245] Daniel Michelsanti, Zheng-Hua Tan, Shi-Xiong Zhang, Yong Xu, Meng Yu, Dong Yu, and Jesper Jensen. "An Overview of Deep-Learning-Based Audio-Visual Speech Enhancement and Separation". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 29 (2021), pp. 1368–1396.

[246] Daniel Neimark, Omri Bar, Maya Zohar, and Dotan Asselmann. "Video Transformer Network". In: *IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*. 2021, pp. 3156–3165.

[247] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. "Training Data-Efficient Image Transformers & Distillation through Attention". In: *International Conference on Machine Learning (ICML)*. Vol. 139. 2021, pp. 10347–10357.

[248] Hanrui Wang, Zhekai Zhang, and Song Han. "SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning". In: *IEEE International Symposium on High-Performance Computer Architecture, (HPCA)*. 2021, pp. 97–110.

[249] Kai Wang, Bengbeng He, and Wei-Ping Zhu. "TSTNN: Two-Stage Transformer Based Neural Network for Speech Enhancement in the Time Domain". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2021, pp. 7098–7102.

[250] Li Yuan, Yunpeng Chen, Tao Wang, Weihao Yu, Yujun Shi, Zi-Hang Jiang, Francis E.H. Tay, Jiashi Feng, and Shuicheng Yan. "Tokens-to-Token ViT: Training Vision Transformers From Scratch on ImageNet". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021, pp. 558–567.

[251] Tirdad Seifi Ala, Emina Alickovic, Alvaro Fuentes Cabrera, William M Whitmer, Lauren V Hadley, Mike L Rank, Thomas Lunner, and Carina Graversen. "Alpha Oscillations During Effortful Continuous Speech: From Scalp EEG to Ear-EEG". In: *IEEE Transactions on Biomedical Engineering* (2022).

[252] Chang Gao, Tobi Delbruck, and Shih-Chii Liu. "Spartus: A 9.4 TOp/s FPGA-based LSTM Accelerator Exploiting Spatio-Temporal Sparsity". In: *IEEE Transactions on Neural Networks and Learning Systems* (2022).

[253] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. "Dynamic Neural Networks: A Survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.11 (2022), pp. 7436–7456.

[254] Sehoon Kim, Sheng Shen, David Thorsley, Amir Gholami, Woosuk Kwon, Joseph Hassoun, and Kurt Keutzer. "Learned Token Pruning for Transformers". In: *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2022, pp. 784–794.

[255] Iván López-Espejo, Zheng-Hua Tan, John Hansen, and Jesper Jensen. "Deep Spoken Keyword Spotting: An overview". In: *IEEE Access* 10 (2022), pp. 4169–4199.

[256] Xiaolong Ma, Sheng Lin, Shaokai Ye, Zhezhi He, Linfeng Zhang, Geng Yuan, Sia Huat Tan, Zhengang Li, Deliang Fan, Xuehai Qian, et al. "Non-Structured DNN Weight Pruning–Is It Beneficial in Any Platform?" In: *IEEE Transactions on Neural Networks and Learning Systems* 33.9 (2022), pp. 4930–4944.

[257] Dominika Przewlocka-Rus, Syed Shakib Sarwar, H Ekin Sumbul, Yuecheng Li, and Barbara De Salvo. "Power-of-Two Quantization for Low Bitwidth and Hardware Compliant Neural Networks". In: *arXiv preprint arXiv: 2203.05025* (2022).

[258] Reza Yazdani, Olatunji Ruwase, Minjia Zhang, Yuxiong He, Jose-Maria Arnau, and Antonio González. "SHARP: An Adaptable, Energy-Efficient Accelerator for Recurrent Neural Networks". In: *ACM Transactions on Embedded Computing Systems (TECS)* (2022).

[259] *Acceptable Processing Delay in Digital Hearing Aids*. https://hearingreview.com/practice-building/practice-management/acceptable-processing-delay-in-digital-hearing-aids.

[260] *Demant A/S*. https://www.demant.com/.

[261] IBM Cloud Education. *Machine Learning*. https://www.ibm.com/cloud/learn/machine-learning. Accessed: 2022-10-06.

[262] *Livio AI*. https://www.starkey.com/hearing-aids/livio-artificial-intelligence-hearing-aids. Accessed: 2022-10-03.

[263] *Simple Python Fixed-Point Module (SPFPM)*. https://github.com/rwpenney/spfpm.

[264] *SoundSense Learn*. https://www.widex.com/en/blog/global/soundsense-learn/. Accessed: 2022-10-03.

[265] *Tiny machine learning*. https://www.tinyml.org/. Accessed: 2022-10-03.