Tampere University

Tapio Honka

# AUTOMATIC MIXED PRECISION QUANTIZATION OF NEURAL NETWORKS USING ITERATIVE CORRELATION COEFFICIENT ADAPTATION

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Prof. Pekka Jääskeläinen
Jouni Siirtola
November 2021

# ABSTRACT

Tapio Honka: Automatic Mixed Precision Quantization of Neural Networks using Iterative Correlation Coefficient Adaptation
Master of Science Thesis
Tampere University
Master's Degree Programme in Computing and Electrical Engineering
November 2021

Recent research of deep learning approaches has resulted in many novel and high-performing models being developed. Simultaneously, the interest in hardware acceleration of neural networks has been constantly growing. This has led to research being targeted at transforming deep learning models to a hardware-implementable form by using techniques such as quantization, quantization-aware training and pruning. These techniques aim to optimize a neural network for hardware resource efficiency with minimal loss in the quality of network output.

Quantization is a commonly used technique, as it maps high precision floating-point models to integer-only models. In the simple case, a certain precision is used uniformly throughout the whole model. Mixed precision quantization extends this by allowing a mix of different integer precision to be used for different parts of the network. The problem which arises in mixed precision quantization is finding a suitable configuration of different precisions, which in practice usually means a trade-off between network accuracy and resource consumption. As models get larger, this problem becomes difficult to solve manually, thus requiring intelligent automatic solutions.

In this thesis, a novel lightweight approach for automatic mixed precision quantization is proposed. Compared to many already existing methods, the proposed iterative correlation coefficient adaptation method is lightweight and easy to implement, as it does not use any form of gradient-based optimization or complex algorithms. The proposed method is evaluated on a CNN-based radio receiver DeepRx using the Open Neural Network Exchange (ONNX) format. Furthermore, as the ONNX format does not currently support mixed precision quantization fully, an explicit list of changes and additions needed to enable this is proposed.

The experiments done in this thesis first explore the hyperparameters for the method, and then perform automatic mixed precision quantization for both memory consumption and compute latency optimization. The resulting mixed precision quantization configurations are compared against uniformly quantized baselines and manually chosen mixed precision configurations by assessing the radio receiver performance, memory usage, and compute latency of each model. The results show that the method is able to find feasible mixed precision models within the chosen resource limitations, and can be thus utilized for finding more complicated mixed precision configurations outside the common manually chosen configurations.

Keywords: neural networks, deep learning, quantization, mixed precision, Open Neural Network Exchange, hardware accelerator, radio receiver

# TIIVISTELMÄ

Tapio Honka: Automaattinen monitarkkuuskvantisointi neuroverkoille iteratiivista korrelaatiokerroinadaptaatiota käyttäen
Diplomityö
Tampereen yliopisto
Marraskuu 2021

Neuroverkkopohjaisten lähestymistapojen laaja tutkimustyö on johtanut monien uusien ja tarkkojen neuroverkkomallien kehittämiseen. Samanaikaisesti myös kiinnostus neuroverkkomallien laitteistokiihdytykselle on ollut jatkuvassa nousussa. Tämä on luonut nostetta neuroverkkojen laitteistototeusten tutkimukselle, jossa malli pyritään muuntamaan laitteistolle sopivammaksi käyttäen tekniikoita kuten kvantisointi, kvantisointitietoinen koulutus, sekä verkon rakenteen karsiminen.

Kvantisointi on yleisesti käytetty tekniikka, sillä sen avulla liukulukuja käyttävät neuroverkot pystytään muuntamaan monelle laitteistolle sopivimmiksi kokonaisluvuiksi. Tavallisessa tapauksessa kvantisointi tapahtuu yhdenmukaisesti samaa tarkkuutta käyttäen koko mallille. Monitarkkuuskvantisointi vie tämän pidemmälle käyttämällä eri tarkkuuksia neuroverkkomallin eri osille. Tämän tekniikan suurin ongelma on sopivan eri tarkkuuksien yhdistelmän löytäminen, joka tavallisesti on valinta neuroverkon tarkkuuden ja laitteistoresurssien välillä. Suurempien mallien tapauksessa tämä ongelma vaikeutuu entisestään ja yhdistelmien manuaalinen valitseminen muuttuu epäkäytännölliseksi, jolloin eri tarkkuuksien yhdistelmien löytämiseen tarvitaan automaattisia ratkaisuja.

Tässä diplomityössä esitetään uusi kevyt menetelmä automaattiselle sekoitettujen tarkkuuksien kvantisoinnille. Muihin saatavilla oleviin menetelmiin verrattuna tämän työn menetelmä on kevyt ja helposti toteutettavissa, sillä se ei käytä gradientteihin perustuvaa optimointia tai muita monimutkaisia algoritmeja. Tämän työn menetelmä evaluoidaan neuroverkkopohjaisella radiovastaanottimella nimeltä DeepRx, käyttäen Open Neural Network Exchange (ONNX) esitysmuotoa. Koska ONNX ei nykyisessä muodossaan tue täysin sekatarkkuuksia, osana tätä työtä esitetään myös selkeä lista muutosehdotuksista nykyiseen ONNX määritelmään, jotka mahdollistavat monitarkkuuskvantisoinnin.

Tämän työn kokeellinen osuus tutkii aluksi menetelmän hyperparametreja, jonka jälkeen menetelmällä suoritetaan automaattinen sekoitettujen tarkkuuksien kvantisointi sekä muistinkäytölle että laskentalatenssille optimoiden. Optimoinnin tuloksia vertaillaan yhdenmukaisesti kvantisoitujen ja manuaalisesti valittujen sekoitetuilla tarkkuuksilla kvantisoitujen mallien kanssa, tarkastellen mallien radiovastaanoton suorituskykyä, muistinkäyttöä ja laskentalatenssia. Tulosten perusteella voidaan todeta, että työssä esitetty menetelmä kykenee löytämään sopivia sekoitettujen tarkkuuksien yhdistelmiä optimointiin valitun resurssin rajoissa. Näin ollen menetelmää voidaan käyttää manuaalisesti valittuja yhdistelmiä monimutkaisempien sekoitettujen tarkkuuksien yhdistelmien löytämiseen.

Avainsanat: neuroverkot, kvantisointi, sekatarkkuus, Open Neural Network Exchange, laitteistokiihdytin, radiovastaanotin

# PREFACE

This thesis was done while working on a machine learning related project at Nokia. I'd first like to acknowledge and thank both Jouni Siirtola and Andrew Baldwin from Nokia who were the key people when coming up and working with the topic of this thesis. The topics related to this thesis have resulted in many great discussions and development ideas within the project. Pekka Jääskeläinen was the key person from Tampere University and I'd like to also thank him for giving clear and constructive feedback during the writing process. Moreover, I'd like to thank my parents and my girlfriend who have supported me throughout this journey ultimately culminating to a finished master's thesis (and hopefully a degree as well), and I'd also like to encourage my sister who will be experiencing the same journey. One journey completed, though certainly not the last; let us see what the future holds.

In Tampere, 29th November 2021

Tapio Honka

# CONTENTS

# LIST OF SYMBOLS AND ABBREVIATIONS

| | |
|---|---|
| AI | Artificial Intelligence |
| ANN | Artificial Neural Network |
| CNN | Convolutional Neural Network |
| CP | Cyclic Prefix |
| DMRS | Demodulation Reference Signal |
| DNN | Deep Neural Network |
| FFT | Fast Fourier Transform |
| FNN | Feed-forward Neural Network |
| GPU | Graphics Processing Unit |
| IFFT | Inverse Fast Fourier Transform |
| INT16 | 16-bit signed integer |
| INT4 | 4-bit signed integer |
| INT8 | 8-bit signed integer |
| LDPC | Low-Density Parity Check Code |
| LLR | Log-Likelihood Ratio |
| LMMSE | Linear Minimum Mean Square Error |
| MAPE | Mean Average Percentage Error |
| OFDM | Orthogonal Frequency-Division Multiplexing |
| ONNX | Open Neural Network Exchange |
| PCC | Pearson Correlation Coefficient |
| QAM | Quadrature Amplitude Modulation |
| QAT | Quantization-aware Training |
| ReLU | Rectified Linear Unit |
| RL | Reinforcement Learning |
| SRAM | Static Random Access Memory |

# 1. INTRODUCTION

During the recent years deep learning has become a widely researched topic, extending to many different applications such as image processing, speech recognition, and even radio receivers. This active research has resulted in many novel and high-performing neural networks being developed, and new state-of-the-art approaches for different applications are discovered every year. Although different novel approaches, architectures, and training procedures are constantly being researched and optimized, many of these state-of-the-art networks are not suitable for devices with distinct hardware limitations as is. Generally, hardware implementation of a deep learning model requires transforming the highly optimized network to a hardware-compatible form, usually sacrificing some of the model accuracy in the process. Consequently, this has led to research being targeted at making networks more hardware implementable, ultimately assisting in the model deployment process for mobile devices and custom hardware chips. There exists many different techniques for transforming neural networks to a more hardware suitable form, such as quantization, quantization-aware training and pruning. Quantization is a frequently used approach as it transforms floating-point operations to fixed integer arithmetic operations commonly required by the target hardware. As a result of quantization, some information is inherently lost and thus in most cases a trade-off between accuracy and model output quality is required. In the simple cases, this is done by manually deciding how much precision is required for the quantized integer values.

Many of the currently available neural network libraries already offer quantization and representation of quantized models, e.g. TensorFlow Lite [1] and PyTorch's built-in quantization [2]. Open Neural Network Exchange (ONNX) [3] is a quite recent format for representing neural networks, and has since become a common way to bridge the gap between the development framework and the hardware deployment environment. The quantization support currently found in ONNX is still somewhat limited and only aimed towards 8-bit quantization. Despite its current limitations, ONNX is still a popular approach for representing quantized models in a common format, and furthermore it is been constantly developed by its open community.

To further optimize the trade-off between accuracy and model resources, a mix of different integer precisions may be used. This mixed precision quantization can be done in a very straightforward way, e.g. by using certain higher precision on all intermediate quantizations and keeping weights in a lower precision, or by intelligently choosing a mix of different precision for all different parts of the network. The latter is essentially a type of neural architecture search problem, where the search consists of finding a suitable mixed precision quantization configuration according to some predefined trade-off between model accuracy and resource cost. Although this problem is highly researched with multiple different methods proposed [4, 5, 6, 7, 8, 9, 10], many of these utilize gradient-based optimization or other computation heavy approach.

In this thesis, a novel gradient-free optimization algorithm for automatic mixed precision quantization is proposed. The proposed algorithm is first described in detail, and the hyperparameters for the method are explored and suitable values for these suggested. The method is then evaluated using a CNN-based radio receiver neural network, and the baseline, manual mixed precision and automatic mixed precision results are thoroughly reported and discussed. Furthermore, as the environment used for the experiments utilizes the ONNX format which does not currently support mixed precision quantization, the changes required for the current ONNX specification for it to fully enable mixed precision quantization are proposed. The main contributions of this thesis therefore consists of:

- A novel gradient-free method for finding mixed precision quantization configurations according to a trade-off between accuracy drop and a chosen target resource.
- Suggested hyperparameters for the proposed method based on thorough experimental results.
- Well documented and analyzed experiments, where the method is utilized for DeepRx, a CNN-based radio receiver.
- Conclusive list of changes required to the current ONNX specification to fully enable mixed precision quantization

The rest of this thesis is arranged as follows. Background regarding neural networks, optimization of these for resource efficiency, and the ONNX format are discussed in Chapter 2. A novel lightweight method for finding suitable mixed precision quantization configurations is proposed in Chapter 3, followed by a review of other research work related to automatic mixed precision quantization in Chapter 4. The target environment for the experiments is discussed in Chapter 5, including extended quantization using ONNX, target hardware, and an example neural network of interest, DeepRx. Experimental results, including suggested hyperparameters for the method and utilization of the method for DeepRx is presented in Chapter 6, followed by a conclusion to the thesis in Chapter 7. Moreover, some suggestions for future research and evaluation of the proposed method, and how it could be developed further to improve the experimental findings are discussed at the end of Chapter 7.

# 2. NEURAL NETWORKS

**Artificial neural networks** (ANN) refer to computing systems defined as networks of interconnected computing nodes. These networks were originally inspired by the biological neurons found in the brain which form a biological neural network, which consequently is one of the reasons why neural networks are commonly associated with the concept of artificial intelligence (AI). The reason why ANNs are particularly interesting is their capability to model very complex behaviour by learning from data. By significantly expanding these networks and optimizing them using very large and diverse data sets, they're able to tackle complex problems such as image classification and speech recognition. Activity regarding ANNs has seen many busy and quiet periods, the latter being sometimes referred to as *AI winters*. Currently neural networks is a widely researched and invested topic, most notably due to inference and training acceleration enabled by general-purpose computing on Graphics Processing Units (GPU).

## 2.1 Overview

**Feed-forward neural network** (FNN) is the very first and simplest type of a neural network consisting of computation nodes fully connected in a feed-forward only manner [11]. A single computation node processes an input vector $\boldsymbol{x}$ by first computing an affine transformation by multiplying by a weight matrix $\boldsymbol{W}^T$ followed by addition of a bias vector $\boldsymbol{b}$, and then applying a nonlinear *activation function* $g(\boldsymbol{z})$ to produce output vector $\boldsymbol{y}$ [12]:

$$\boldsymbol{y} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}) \tag{2.1}$$

A commonly used activation function is the *Rectified Linear Unit* (ReLU), which replaces all negative values with zeros:

$$g(z) = max(0, z) \tag{2.2}$$

ReLU is commonly used due to its simple and efficient implementation, and its robustness to vanishing gradients when compared to sigmoid activation function [13]. The basic concept of a FNN can be seen in Figure 2.1, where input, hidden and output layers are

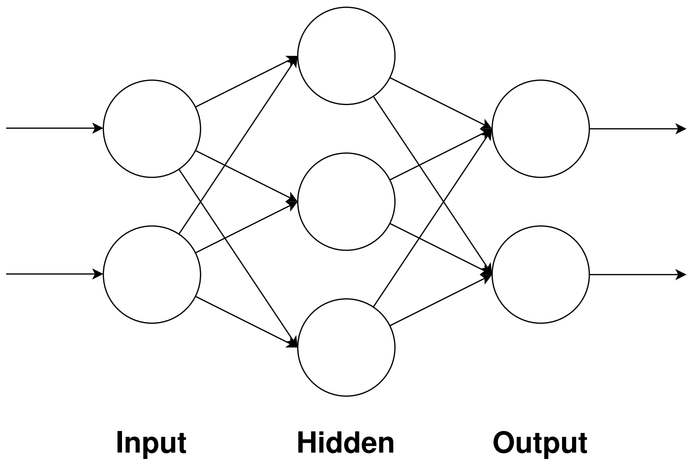


***Figure 2.1.*** *A simple FNN with one hidden layer between the input and output layers*

illustrated. Each unit in a given layer is fully connected to all of the units in the next layer. This is the reason why feed-forward layers are also commonly referred to as fully connected layers.

**Convolutional neural network** (CNN) is a specific type of neural network which consists of convolutional layers and optional fully connected layers. The concept of a CNN was introduced in 1980 by Fukushima [14], where layers utilizing convolution operations and layers applying downsampling were first proposed. The convolution operation is commonly denoted using an asterisk, and in a two-dimensional case can be written as

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n), \tag{2.3}$$

where $K$ is a two-dimensional kernel matrix and $I$ is a two-dimensional input matrix [12]. In the case of neural networks, a single convolution result is commonly referred to as a *feature map*. Moreover, an additional dimension is usually used for different channels. Depending on the use case, the edges of $I$ may need to be handled using padding, i.e. increase the size of $I$ by adding values, e.g. zeros, outside of its edges. This is done to ensure the convolution result shape matches the shape of its input. In Figure 2.2 the convolution operation with zero padding for ensuring the input and output shapes match is visualized.

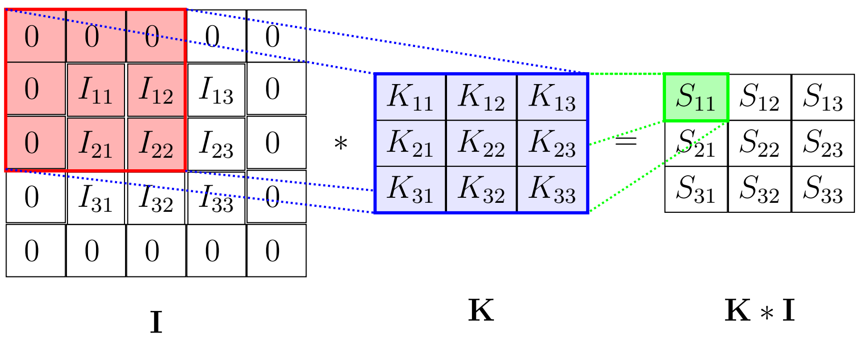

0 0 0 0 0
0 $I_{11}$ $I_{12}$ $I_{13}$ 0
0 $I_{21}$ $I_{22}$ $I_{23}$ 0
0 $I_{31}$ $I_{32}$ $I_{33}$ 0
0 0 0 0 0
$*$
$K_{11}$ $K_{12}$ $K_{13}$
$K_{21}$ $K_{22}$ $K_{23}$
$K_{31}$ $K_{32}$ $K_{33}$
$=$
$S_{11}$ $S_{12}$ $S_{13}$
$S_{21}$ $S_{22}$ $S_{23}$
$S_{31}$ $S_{32}$ $S_{33}$
$\mathbf{I}$
$\mathbf{K}$
$\mathbf{K} * \mathbf{I}$

***Figure 2.2.*** *Convolution operation with zero padding visualized*

Depthwise separable convolution extends the standard convolution by reducing the computing complexity, thus making the convolutional neural network more efficient. This concept was first introduced in [15], and has since been adopted in many architectures [16, 17, 18]. Instead of performing kernel-wise operations at once for all of the channels of an input array local neighborhood, each channel is first considered separately using depthwhise convolution and the resulting intermediate result is interpreted using pointwise convolution. In depthwise convolution, a separate one-channel convolution kernel is used for each channel, resulting in an array with the same size as in normal convolution. The pointwise convolution following this performs convolution with a kernel matching the channel count but the neighborhood reduced to one, i.e. a single point, in all other dimensions. This procedure is able compute a result matching to standard convolution but with significantly less total multiplications.

A typical convolution layer pipeline applies an activation function after convolution, such as the aforementioned ReLU, followed by a *pooling function*. The basic idea of pooling is to take local neighbourhoods of a tensor and summarize each of them with a statistic, such as maximum or average value. This is very similar to the shifting kernel in convolution, where the stride between each shift and the size of the local neighbourhood determine the output size of the pooling. Pooling is commonly used to reduce the dimensions of a tensor and additionally help the representation become more invariant to small input translations. [12]

A key part of a neural network is the optimization of its parameters according to the use case, commonly referred to as learning. Automated gradient-based back-propagation optimization of neural networks was first introduced by LeCun et. al in 1998 [19], and has since been expanded and implemented in the majority of deep learning frameworks. Moreover, the first GPU supported CNN implementation was introduced in [20] where GPU provided 4 times faster inference compared to CPU implementation, and the first GPU accelerated supervised learning using back-propagation was introduced in [21] and [22] for FNNs and CNNs respectively. GPU accelerated learning and inference has since become the standard used in the majority of deep learning libraries [2, 23, 24].

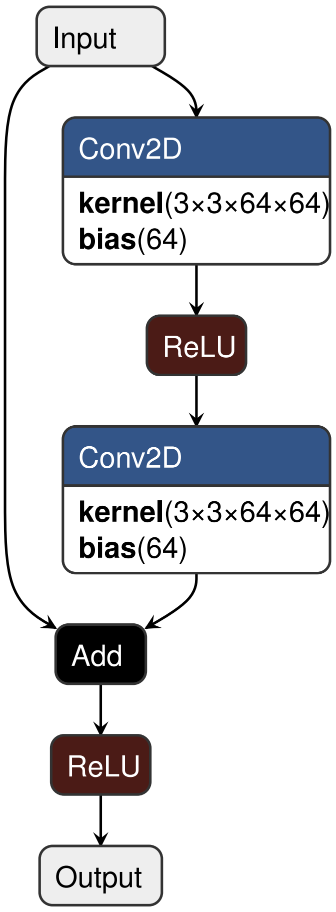


***Figure 2.3.*** *A simple residual neural network with a single residual block*

**Residual neural networks** utilize shortcut connections (also known as skip or residual connections) to jump over one or more layers. The concept of a residual building block was first introduced in [25], where they were used to address the degradation problem associated with deep neural networks. As networks get deeper, i.e. the number of layers is increased significantly, the accuracy saturates and then starts to rapidly degrade. The authors of [25] showed that residual networks are easier to optimize and are able to gain accuracy boosts from increased network depth. The original residual block consists of two convolution layers with a ReLU activation in the middle followed by the residual connection and a final ReLU activation. Residual blocks have since been adopted in many deep neural network architectures [25, 26, 27]. A simple residual neural network consisting of a single residual block can be seen in Figure 2.3.

Although neural networks are currently popular and highly researched, they still are not always suitable for every problem and come with certain drawbacks. As networks get deeper and more complex, the amount of diverse training and evaluation data required becomes very high. This may cause problems with applications where a very complex neural network is required but relevant data is hard to gather. Moreover, if data is not diverse enough or the neural network model is too complex for the application, the model might *overfit* to the training data. This means that the network becomes too optimized for the training data and is not able to perform as well on data outside the training data set. Another drawback is the network architecture which might be arbitrarily chosen by the network designers to maximize the model accuracy without considering model execution time or possible hardware implementations.

## 2.2 Quantization of Neural Networks

There exists many techniques for transforming a neural network to a more hardware-implementable form. Representing neural network operations using limited precision integer values is known as *quantization*, more specifically *post-training quantization* if the training procedure is not affected. This method is commonly used to map a neural network to integer hardware, and also to reduce the size and complexity of the network. A quantization scheme defines how a full-precision floating-point neural network is converted to fixed-precision integer form. Although many quantization approaches have been proposed, not all of them aim to deliver improvements on hardware implementation, but rather to reduce the storage size of the model [28, 29, 30].

One of the biggest motivations behind quantization is the hardware platforms developed for real-time hardware-accelerated inference of neural networks. As an example, NVIDIA is looking more into using 4-bit integers in model inference, which is supported by their Turing architecture [31] where INT4 precision mode was introduced as an option to provide speed-up to calculations. They showed that an INT4 implementation of a residual neural network model achieved a 59% throughput increase while only losing less than 1% of accuracy when compared to INT8 implementation [32]. Moreover, in their 2020 white paper [33], Xilinx describe how their devices perform with 4-bit integer optimized convolutional neural networks. They concluded that a network with 4-bit integer weights and activations delivered 77% performance boost compared to 8-bit integer quantization, and that the accuracy drop from 8-bit to 4-bit precision in a variety of computer vision tasks was not too drastic, although still present in all of the tasks. These examples indicate that lower precision modes are starting to appear more frequently on machine learning hardware applications, and could provide significant boosts in terms of resource utilization and inference time when exploited appropriately.

### 2.2.1 Quantization Scheme

Let $r \in \mathbb{R}$ be a full-precision real number, and $q \in \mathbb{Z}$ be a quantized integer number. The mapping between these two can be written as

$$r = S(q - Z) \tag{2.4}$$

$$q = r/S + Z \tag{2.5}$$

where $S \in \mathbb{R}_+$ is a scaling factor and $Z \in \mathbb{Z}$ is zero-point offset. $S$ and $Z$ are the main quantization parameters which are configured for each weight array, bias vector and pre-activation layer. Additionally, bias vectors are handled in a special manner specific to the target hardware where the number of bits used for bias vector values can be decreased while preserving the scale, discussed more in detail in Section 5.2.

The scaling factor $S$ aims to scale $r$ to the target integer type range, and is typically represented as a floating-point value in software. Section 2.2.2 describes more in detail how scaling can be done in a hardware implementable way. The scale value needs to be determined based on the target integer type and the range of the associated real values. For quantization of weight arrays, the minimum and maximum values for each weight array are gathered, while for pre-activation layers, the minimum and maximum values are observed by inferring a representative dataset through the network. The representative dataset should be as close to the final use case as possible to ensure the observed quantization ranges correspond to those present during deployment. The scaling factor $S$ can be then calculated as:

$$S = \frac{2^{b-1}}{max(|x_{max}|, |x_{min}|)} \tag{2.6}$$

where $x_{max}$ and $x_{min}$ are the minimum and maximum values of a weight array or pre-activation layer output, and $b$ is the number of bits available for a signed integer, e.g. 8 for INT8.

Zero-point offset $Z$ is the quantized value corresponding to the value $r = 0$, and is represented as the same type as the target quantized value $q$. $Z$ translates the scaled values so that the real value $r = 0$ can be precisely represented by the quantized value $q$. As mentioned in [34], one of the motivations for this is the zero-padding of arrays required by many neural network operators.

An essential part of converting real numbers to integers is rounding. There are multiple different ways of performing rounding, such as the commonly used *rounding down* and *rounding up* methods, and the less conventional methods *rounding half towards zero* and *rounding half away from zero*. The latter two consider positive and negative values symmetrically, and are thus free from any bias related to the sign of the values.

To store the intermediate result from a partial convolution or matrix multiplication product, there needs to be enough bits available to not overflow the intermediate result. On the hardware, this effectively means using an accumulator register with a high enough bit-width. For accumulation of multiplications with 8-bit integer operands, a 32-bit accumulator is sufficient as discussed in [34]. With both operands being 4-bits, a 16-bit accumulator suffices, whereas with 16-bit operands the accumulator is expanded to 48 bits.

### 2.2.2 Hardware Implementable Scaling

One way to implement data scaling on hardware is to first apply an integer pre-multiplier followed by bit-shifting, also referred to as *dyadic scaling* in [10]. Let $S_{opt} \in \mathbb{R}_+$ be the optimal quantization scaling factor based on the ranges observed from a representative data set or a weight array. Moreover, let's denote the pre-multiplier as $M$ and the amount of right shifting applied as $n$, where $\{M, n\} \in \mathbb{N}$. In the hardware case we want to find a hardware implementable scale $S_{hw}$ best corresponding to $S_{opt}$ within the hardware limitations. The hardware implementable scale can be written as

$$S_{hw} = \frac{2^n}{M}, \tag{2.7}$$

and we want to find $M$ and $n$ which minimize the difference between the scaling factors $S_{opt}$ and $S_{hw}$ while also making sure that $S_{opt} \leq S_{hw}$ to prevent scaling values over the target ranges thus saturating or overflowing them. First, we calculate the optimal shift $n_{opt} \in \mathbb{R}$ and pre-multiplier values $n_{opt}$ as

$$n_{opt} = \log_2(S_{opt}) \tag{2.8}$$

$$M_{opt} = 2^{\lceil n_{opt} \rceil - n_{opt}} \tag{2.9}$$

At this point the optimal scale can be written as

$$S_{opt} = \frac{2^{\lceil n_{opt} \rceil}}{M_{opt}}, \tag{2.10}$$

Next, the optimal pre-multiplier is scaled according to the magnitude bits $b \in \mathbb{N}$, $b \geq 1$, available in the hardware pre-multiplier. Now $2^0 \leq M_{opt} < 2^1$, and by scaling $2^b$ we get $2^b \leq M_{opt}2^b < 2^{b+1}$, where the most significant bit is always 1. To preserve equality, this scaling is also done to the numerator in the right side of Equation 2.10, thus resulting in

$$S_{opt} = \frac{2^{\lceil n_{opt} \rceil}2^b}{M_{opt}2^b}, \tag{2.11}$$

which is then finally converted to hardware implementable integer values

$$S_{hw} = \frac{2^{\lceil n_{opt} \rceil + b}}{\lfloor 2^{\lceil n_{opt} \rceil - n_{opt} + b} \rfloor}, \tag{2.12}$$

where $\lfloor . \rfloor$ denotes the floor function which effectively prevents the denominator from going over $2^{b-1}$, while also ensuring that $S_{opt} \leq S_{hw}$ to prevent overflowing the scaled values. We can now write explicitly:

$$n = \lceil n_{opt} \rceil + b \tag{2.13}$$

$$M = \lfloor 2^{\lceil n_{opt} \rceil - n_{opt} + b} \rfloor. \tag{2.14}$$

In practice, these values are calculated beforehand and then moved to the hardware during deployment. Furthermore, as $2^b \leq M < 2^{b+1}$, the most significant magnitude bit of $M$ is always 1 and thus still only $b$ bits need to be stored. In the special case where pre-multiplier is not available, i.e. $b = 0$, and only right-shifting is available, then $M = 1$ while $n = \lceil n_{opt} \rceil$.

### 2.2.3 Mixed Precision Quantization

In mixed precision quantization (also known as heterogeneous quantization), the bit-widths are not constant for the whole model, but rather a mixed configuration of different bit-widths are used across the model. The bit-widths can be set separately for each weight tensor and activation output [4, 5], or additionally for each convolution channel separately [6, 7]. The latter is often referred to as *channel-wise quantization* of weights and it significantly extends the possible mixed precision quantization configurations, but could also provide additional resource consumption optimization. The more common way of quantizing whole kernels separately is commonly referred to as *kernel-wise quantization*. Both channel-wise and kernel-wise quantization may result in a combination of different multiplication operand bitwidths. Regardless, the scaling factor in Equation 2.6 is still calculated similarly using the pre-activation layer output value ranges for the multiplication result.

Generally, the goal of mixed precision quantization is to assign more bits to the parts of the network which are more significant in terms of keeping the quantization error minimal, and use less bits for the less significant parts, ultimately keeping the network output as close to the high-precision representation as possible while making sure too many bits are not used in parts where it's unnecessary. It should be noted that the benefits of assigning less bits are very hardware dependent, and might not always provide direct gains in terms of latency or even memory consumption. Therefore generalizing a search method for all possible hardware types is very complicated, and some hardware-specific definitions are usually needed to find an optimal or close to optimal mixed precision configuration for the target hardware. As the network architecture gets larger in terms of layer count, the amount of mixed precision quantization configurations gets exponentially larger, thus drastically expanding the search space. This means that sophisticated search methods are required to find the optimal or close to the optimal solution in reasonable search time.

## 2.3 Quantization-aware Training

The post-training quantization discussed up until now does not affect the model training procedure which is targeting the full-precision floating-point environment. Quantization-aware training (QAT) aims to train the model directly to the quantized deployment environment by emulating the behaviour of quantized inference during forward-propagation, while keeping the back-propagation normal. In [34], quantized inference was emulated by inserting fake quantization operations in parts of the floating-point graph where the intermediate tensors would be converted to lower bit-width representation after actual quantization. Specifically, these fake quantization operations divide floating-point values into evenly spaced discrete bins, emulating the behaviour of integer values. Furthermore, Zhou et. al. [35] used low bit-width quantization of parameter gradients during back-propagation to train convolutional neural networks. QAT has since become a widely used method when targeting neural networks to applications requiring quantization [34, 36, 37], as it prevents the common failures, such as large differences in ranges of weights and outlier weight values [34], associated with simple post-training quantization.

## 2.4 Pruning

Another completely different approach for reducing the size and complexity of neural networks is called *pruning*. In pruning, individual parts of a model are analyzed and removed if declared non-essential, e.g. if there are no significant accuracy drop observed from the absence of the specific part. Pruning can be done in either an unstructured or structured manner. In unstructured pruning, the connections to certain individual parameters are removed by setting the parameter to zero or introducing a zero during the multiplication, resulting in a sparse network which usually requires a special implementation to be fully exploited. In structured pruning, grouped structures, such as neurons, filters, or channels, are completely removed from the architecture and can be thus fully exploited using standard neural network computation hardware or software. [38]

The method proposed in [39] combines pruning and QAT in a so-called Quantization-aware multi-stage pruning algorithm. The main idea of this method is to perform pruning on a quantization-aware trained model to ensure that the pruned model is already aimed to the target quantized environment. Additionally, the algorithm combines unstructured and structured pruning by first disabling connections in an unstructured manner for faster execution time during optimization, and removing connections in a structured manner only when the pruned model has been ensured to work appropriately.

## 2.5 Open Neural Network Exchange

Open Neural Network Exchange (ONNX) [3] is an open format for representing neural network models using a common set of well-defined operators, standard data types, and a common file format. ONNX is an open community project with many large companies as partners, and encourages its users to contribute their ideas and code implementations. It was originally developed by the PyTorch team under the name Toffee, and was later renamed to ONNX when announced by Facebook and Microsoft. Due to its common format, ONNX is often used as a bridge between the training framework and the deployment environment as models produced by most common frameworks, e.g. Tensorflow [23] and PyTorch [2], can be converted into the ONNX format. This effectively makes the hardware deployment process independent of the framework used during research, thus streamlining the deployment from research to the hardware environment.

ONNX format also offers a representation for quantized neural networks. This was first introduced in version 1.5.0, which introduced the operator set version 10 including quantization specific operators. This included operators such as integer convolution and matrix multiplication, and a specific operator for scaling and converting data from an integer type to another. Additionally, ONNX runtime [40], a cross-platform open source project for inference and training of ONNX models, offers 8-bit quantization of ONNX models. The current ONNX specification (version 1.9.0) is mainly targeted for 8-bit quantization, and does not fully support e.g. 16-bit or mixed precision quantization. This is due to quantization being a relatively new feature within ONNX, the aforementioned operator set version 10 being released in 2019. However, as ONNX is a constantly evolving open community project, quantization support is expected to be extended in the coming years.

# 3. ITERATIVE CORRELATION COEFFICIENT ADAPTATION FOR MIXED PRECISION OPTIMIZATION

As discussed previously, mixed precision quantization results in a large amount of possible quantization configurations with larger networks. Therefore, manually finding an optimized configuration is usually non-practical and an automatic algorithmic approach is needed. The automatic mixed precision optimization method proposed in this thesis consists of combining normal distributed sampling and Pearson correlation coefficient adaptation with dynamic weighted sampling center updating. The benefit of this method is that it does not require any gradient information, and is therefore relatively lightweight and applicable to non-differentiable cost functions. Furthermore, the implementation of this method is not too complex and also well-documented here, and can be thus easily implemented for different target environments.

The proposed optimization algorithm starts as a pure random search where quantization configurations are sampled from a normal distribution around the current sampling center. For each configuration, the corresponding cost function value is calculated to evaluate its performance. As the algorithm advances, the Pearson correlation coefficients between each configuration state and cost value are updated incrementally, and used for calculating a heuristic candidate solution. The sampling center is updated based on the best configurations found from the latest iteration and the heuristic candidate. Moreover, the proportion in which these are used when updating the center is dynamically changed, first by using only the best sampled configurations and gradually using more of the heuristic candid solution. This whole process is described in more detail in the following sections.

## 3.1 Algorithm

Let $\mathbf{S} = [s_0, s_1, \ldots, s_{n-1}]$, be a mixed precision quantization configuration state-vector with $n$ state-variables, each being $s_i \in [0, 1]$ for all $i$ from $0$ to $n-1$. In this case, each state-variable can be interpreted as amount of precision given to a certain part of the neural network, i.e. layer weights, bias, and activation output quantization. As the state-variables are now represented as floating point values, they need to be mapped to actual usable precision values. For layer weights and activations, values in range $[0, 0.333...)$ are considered to be INT4, values in $[0.333..., 0.666...)$ INT8, and values in $[0.666..., 1]$

INT16, however, this could be extended to even wider variety of integer precisions. For all bias vectors, a single common precision is used to reflect the target hardware considered in this thesis as described in Section 5.2. The floating point value for the common bias vector precision is mapped to the range $[8, 32]$ and then rounded to the nearest integer to acquire the bias bits used. These mappings are done when evaluating each state and calculating the corresponding cost value.

The proposed search algorithm starts by initializing the current sampling center $\mathbf{m} = [m_0, m_1, \dots, m_{n-1}]$ where $m_i \in [0, 1]$ for all $i$ from $0$ to $n-1$, to $\mathbf{m} = [1.0, 1.0, \dots, 1.0]$. For learning rates, the sample learning rate $\alpha_s \in [0, 1]$ is set to $\alpha_s = 1$, and the correlation coefficient candidate learning rate $\alpha_c \in [0, 1]$ to $\alpha_c = 0$. Furthermore, the states for static INT4, INT8 and INT16 configurations are evaluated and the state with the lowest cost is used as the initial best state $\mathbf{S_{best}}$ and best cost $c_{best}$.

For each start of an iteration (also called epoch), a set $\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{\lambda-1}]$ of $\lambda$ samples are sampled from a normal distribution with current center $\mathbf{m}$ as mean and variance $\sigma$ for each state variable, which are then evaluated based on a cost function to acquire the corresponding cost values $\mathbf{c} = [c_0, c_1, \dots, c_{\lambda-1}]$. The sample set is then sorted according to ascending cost values denoted as $\mathbf{X}_{sorted}$, and $\mathbf{S_{best}}$, $c_{best}$ are updated if a new lowest cost is found. Next, the first $\mu \in (0, 1, \dots, \lambda]$ samples from the sorted sample set are used to calculate the sample based center $\mathbf{m_s}$ using weights $\mathbf{w} = [w_0, w_1, \dots, w_{\mu-1}]$ as

$$\mathbf{m_s} = \sum_{i=0}^{\mu-1} x_{sorted,i} w_i, \tag{3.1}$$

where

$$\sum_{i=0}^{\mu-1} w_i = 1, w_i \in \mathbf{R}_+, \tag{3.2}$$

and

$$w_0 \geq w_1 \geq \cdots \geq w_{\mu-1}. \tag{3.3}$$

These weights implement a weighted average of the top samples from the current sample set, generally favoring significantly more samples with a lower cost.

In some cases the top performing samples from a sample set might be drastically worse than the best overall sample so far, and therefore $\mathbf{m_s}$ may start to drift to an inferior direction. To mitigate this, if the best sample set cost is higher than $c_{best}$, the best solution so far, $\mathbf{S_{best}}$, and previously calculated $\mathbf{m_s}$ are averaged as

$$w_{best} = min((\frac{min(\mathbf{c})}{c_{best}})^2 - 1, 1) \tag{3.4}$$

$$\mathbf{m_s} = (1 - w_{best})\mathbf{m_s} + w_{best}\mathbf{x_{best}} \tag{3.5}$$

Doing purely random search with dynamically changing sampling center works well in cases where the target model is not too deep. However, when the model gets deeper, this approach does not always converge and may fail completely. The approach proposed here is to introduce a heuristic candidate solution, which is not guaranteed to be the optimal solution, but rather a sophisticated approximate solution based on the sample Pearson correlation coefficients (PCC) $\mathbf{r} = [r_0, r_1, \ldots, r_{n-1}]$ between each state-variable and cost value. For all $i$ from $0$ to $n - 1$, the correlation coefficient based candidate solution $\mathbf{m_c}$ is calculated by linearly mapping the coefficient as:

$$\mathbf{m_{c,i}} = 0.5(1 - r_i) \tag{3.6}$$

A positive correlation coefficient indicates that higher variable values, i.e higher precision, increase the cost and therefore lower precision is preferred, and a negative coefficient indicates that higher values decrease the cost and therefore higher precision is preferred. Equation 3.6 implements this behaviour by setting state-variables to range $[0.5, 1]$, i.e. INT8 and INT16 range, for negative coefficients, and to range $[0, 0.5)$, i.e. INT4 and INT8 range, for positive coefficients. For coefficients with low magnitude, i.e. changing the precision does not drastically affect the cost, the INT8 range is used. This same logic applies to the state variable used for bias bits, where 16 is considered as the conservative middle area.

After $\mathbf{m_s}$ and $\mathbf{m_c}$ are calculated, they are weighted using $\alpha_c$, the sample learning rate, and $\alpha_s$, the correlation coefficient candidate learning rate, and summed to acquire the updated sampling center $\mathbf{m}$:

$$\mathbf{m} = \alpha_s\mathbf{m_s} + \alpha_c\mathbf{m_c} \tag{3.7}$$

Recall that $\alpha_s$ is initialized to 1, therefore at the start of the algorithm only $\mathbf{m_s}$ is considered when updating $\mathbf{m}$. As the algorithm advances, the learning rates are updated at each iteration using the following formulas respectively:

$$\alpha_c = \alpha_c + \alpha_s(1 - \gamma) \tag{3.8}$$

$$\alpha_s = \gamma\alpha_s \tag{3.9}$$

where $\gamma$ is the discount factor effectively diminishing $\alpha_s$ and augmenting $\alpha_c$ at each iteration so that $\alpha_s + \alpha_c = 1$. The value of $\gamma$ should be $0 \ll \gamma \leq 1$, typically somewhere in range $[0.9, 1]$ as the correlation coefficient candidate requires a broad selection of explored states before being a meaningful candidate solution.

To conclude, the hyperparameters for the algorithm consist of sample set size $\lambda$, number of top samples $\mu$, weights $\mathbf{w}$, sampling variance $\sigma$, and learning rate discount factor $\gamma$. The values proposed for these are explored and discussed more thoroughly in Section 6.1. The full algorithm with all of the discussed steps is presented in 1 as pseudocode.

**Algorithm 1** CorrelationCoefficientAdaptation

```
1:  Set $\lambda, \sigma, \gamma$
2:  Initialize $\mathbf{m}, \alpha_s, \alpha_c, c_{best}, x_{best}$
3:  for number of epochs do
4:      for $i = 0, 1, \ldots \lambda - 1$ do
5:          $x_i \leftarrow sample\_normal(\mathbf{m}, \sigma)$
6:          $c_i \leftarrow cost(x_i)$
7:          Update $\mathbf{r}$ incrementally
8:      end for
9:      $\mathbf{x_{sorted}} \leftarrow sort\_ascending\_cost(\mathbf{x}, \mathbf{c})$
10:     $\mathbf{m_s} \leftarrow \sum_{i=0}^{\mu} x_{sorted,i} w_i$
11:     if $min(c) > c_{best}$ then
12:         $w_{best} \leftarrow (min(\mathbf{c})/c_{best})^2 - 1$
13:         if $w_{best} > 1$ then
14:             $w_{best} \leftarrow 1$
15:         end if
16:         $\mathbf{m_s} \leftarrow (1 - w_{best}) m_s + w_{best} x_{best}$
17:     else
18:         $\mathbf{S_{best}} = x_{sorted,0}$
19:     end if
20:     for $i = 0, 1, \ldots variables - 1$ do
21:         $\mathbf{m_{c,i}} \leftarrow 0.5(1 - r_i)$
22:     end for
23:     $\mathbf{m} \leftarrow \alpha_s \mathbf{m_s} + \alpha_c \mathbf{m_c}$
24:     $\alpha_c \leftarrow \alpha_c + \alpha_s (1 - d)$
25:     $\alpha_s \leftarrow \alpha_s d$
26: end for
```

## 3.2 Incremental Pearson Correlation Coefficients

Generally for a pair of random variables $X$ and $Y$, the population Pearson correlation coefficient is defined as

$$\rho_{xy} = \frac{cov(X, Y)}{\sigma_X \sigma_Y}, \tag{3.10}$$

where $cov$ is the covariance between two random variables, and $\sigma_X, \sigma_Y$ are the standard deviations for $X$ and $Y$ respectively [41]. The Pearson correlation coefficient can be thus considered as a normalised measurement of covariance. The sample Pearson correlation coefficient $r_{xy}$ for a pair of sample sets $\mathbf{x}_N = [x_0, x_1, \ldots, x_{N-1}]$ and $\mathbf{y}_N = [y_0, y_1, \ldots, y_{N-1}]$ with $N$ samples each is then defined as

$$r_{xy} = \frac{\sum_{i=0}^{N-1}(x_i - \hat{x})(y_i - \hat{y})}{\sqrt{\sum_{i=0}^{N-1}(x_i - \hat{x})^2}\sqrt{\sum_{i=0}^{N-1}(y_i - \hat{y})^2}}, \tag{3.11}$$

where $\hat{x}, \hat{y}$ are the mean values over each sample set [41]. Furthermore, calculating $r_{xy}$ can be implemented in an incremental way, where each update remains the same in terms of complexity. The mean $\hat{x}$ can be written as:

$$\hat{x} = \frac{1}{N}\sum_{i=0}^{N-1} x_i \tag{3.12}$$

Which can be implemented incrementally with a running sum of the data and sample number $N$. To estimate variance, Bessel's Correction [42] is applied to estimate unbiased sample variance $\sigma^2$ using biased sample variance $s_N^2$:

$$\begin{aligned} s_x^2 &= \frac{1}{N}\sum_{i=0}^{N-1}(x_i - \hat{x})^2 \\ &= \frac{1}{N}(\mathbf{x}_N - \hat{x}\mathbf{1}_N)^T(\mathbf{x}_N - \hat{x}\mathbf{1}_N) \end{aligned} \tag{3.13}$$

$$\sigma^2 = \frac{N}{N-1}s_x^2 \tag{3.14}$$

where $\mathbf{1}_N$ is a vector with all $N$ elements set to $1$. By expanding Equation 3.13, we get

$$\begin{aligned} s_x^2 &= \frac{1}{N}(\mathbf{x}_N - \hat{x}\mathbf{1}_N)^T(\mathbf{x}_n - \hat{x}\mathbf{1}_N) \\ &= \frac{1}{N}(\mathbf{x}_n^T\mathbf{x}_N - N\hat{x}^2) \\ &= \frac{1}{N}\mathbf{x}_n^T\mathbf{x}_N - \hat{x}^2 \end{aligned} \tag{3.15}$$

where $\mathbf{x}_N^T\mathbf{x}_N$ can be implemented as a running sum of squares. Therefore, $\sigma^2$ can be calculated incrementally by keeping track of the sample number $N$, and keeping a running sum and sum of squares of the incoming data. For covariance, the same principle is applied:

$$
\begin{aligned}
s_{xy}^2 &= \frac{1}{N}\sum_{i=0}^{N-1}(x_i - \hat{x})(y_i - \hat{y}) \\
&= \frac{1}{N}\mathbf{x}_N^T\mathbf{y}_N - \hat{x}\hat{y}
\end{aligned}
\tag{3.16}
$$

$$
cov(\mathbf{x}, \mathbf{y}) = \frac{N}{N-1}s_{xy}^2 \tag{3.17}
$$

which needs to keep a running product of the incoming data pairs instead of sum of squares. In the case of the search method discussed here, the variances for state variables and cost as well as the covariances between each state variable and cost is calculated to acquire the PCCs for all state variables. A pseudocode implementation of incremental PCC is described in Algorithm 2.

**Algorithm 2** IncrementalPCC

```
1:  $n \leftarrow 0$
2:  $sums\_state \leftarrow 0$
3:  $sums\_cost \leftarrow 0$
4:  $products \leftarrow 0$
5:  $squares\_state \leftarrow 0$
6:  $squares\_cost \leftarrow 0$
7:  $PCC \leftarrow 0$
8:  for every new $state$ and $cost$ do
9:      $n \leftarrow n + 1$
10:     $sums\_state \leftarrow sums\_state + state$
11:     $sums\_cost \leftarrow sums\_cost + cost$
12:     $products \leftarrow products + state * cost$
13:     $squares\_state \leftarrow squares\_state + state^2$
14:     $squares\_cost \leftarrow squares\_cost + cost^2$
15:     if correlation coefficients needed then
16:         $mean\_state = sums\_state/n$
17:         $mean\_cost = sums\_cost/n$
18:         $s_2\_cov = products/n - mean\_state * mean\_cost$
19:         $s_2\_var\_state = squares\_state/n - mean\_state^2$
20:         $s_2\_var\_cost = squares\_cost/n - mean\_cost^2$
21:         $cov = n * s_2\_cov/(n-1)$
22:         $var\_state = n * s_2\_var\_state/(n-1)$
23:         $var\_cost = n * s_2\_var\_cost/(n-1)$
24:         $PCC \leftarrow cov/\sqrt{var\_state * var\_cost}$
25:     end if
26: end for
```

## 3.3 Estimating Memory Usage and Compute Latency

For estimating the memory required to store the weight parameters of a neural network, the shape and bit-width used for each parameter array, i.e. weight arrays and bias vectors, is observed. The total memory resource consumption for a model with $l$ convolutional or matrix multiplication layers is therefore estimated as:

$$C_{memory} = \sum_{i=0}^{l-1} b_{w,i} n_{w,i} + \sum_{i=0}^{l-1} b_b n_{b,i}, \tag{3.18}$$

where $n_{w,i}$ is the number of elements in $i$th weight array using bit-width $b_{w,i}$, and similarly for number of bias elements $n_{b,i}$ with the constant bias most significant bits $b_b$ available. The bit-widths for weight values considered here are $4$, $8$ and $16$, and the number of bias magnitude bits limited to range $[8, 9, \ldots, 32]$.

For estimating the multiplication latency of a single the network inference, the estimated computed cycles for multiplication between each data type pair is used. The latency due to memory reading is omitted here, as usually memory utilization is done partially or fully in parallel while computing parts of the network, and is therefore complicated to estimate. The latency estimation for a model with $l$ convolutional or matrix multiplication layers is defined here as:

$$C_{latency} = \sum_{i=0}^{l-1} m_i o_i, \tag{3.19}$$

where $m_i$ is the number of individual multiplications in $i$th weight array and $o_i$ is the corresponding latency in cycles for the two operand data types. For full element-wise matrix multiplications the number of multiplications is simply the number of elements in the weight array, as the batch size is omitted and assumed to be 1. For convolutions, the weight kernel is reused and shifted and thus the number of multiplications is not as straightforward to calculate. The number of convolution multiplications $o_c$ is calculated as

$$o_c = w_k \cdot h_k \cdot f \cdot c \cdot w_o \cdot h_o, \tag{3.20}$$

where $w_k$ and $h_k$ are the kernel width and height, $f$ is the number of feature maps, $c$ is the channel count, and $w_o$ and $h_o$ are the output tensor width and height. This way of estimating matrix multiplication and convolution latency completely ignores parallelism. However, in the common case the amount of parallel operations stays constant across different data types, thus the relative difference between latencies with different data types is still very relevant despite omitting parallelism. The amount of cycles for multiplications between each available data type considered here are shown in Table 3.1. These values are not accurate for any specific hardware, but rather give a general direction of how each

***Table 3.1.*** *Latency values used for compute latency estimation*

| Operand 1 data type | Operand 2 data type | Multiplication latency in cycles |
|---|---|---|
| INT4 | INT4 | 1 |
| INT4 | INT8 | 2 |
| INT4 | INT16 | 4 |
| INT8 | INT8 | 4 |
| INT8 | INT16 | 8 |
| INT16 | INT16 | 16 |

data type would behave relative to each other. These values are only used as part of the proposed optimization method to give an estimated change in latency between different data types, but could be easily changed to be more hardware specific.

## 3.4 Cost Function

The cost function proposed for the method used here consists of a weighted sum of normalized mean absolute percentage error (MAPE) between floating-point and quantized model outputs, and normalized resource cost measuring either memory usage or model compute latency. The mean absolute percentage error, denoted as $\epsilon$ here, between floating point model outputs $\mathbf{y}_f = [y_{f,0}, y_{f,1}, \dots, y_{f,n-1}]$ and quantized integer model outputs $\mathbf{y}_q = [y_{q,0}, y_{q,1}, \dots, y_{q,n-1}]$ is calculated as

$$\epsilon = \frac{1}{n}\sum_{i=0}^{n-1} \frac{|y_{f,i} - y_{q,i}|}{s_i}, \tag{3.21}$$

where scales $s_i$ for all $i$ from 0 to $n-1$ are

$$s_i = \begin{cases} |y_{f,i}|, & y_{f,i} \neq 0 \\ 1, & y_{f,i} = 0 \end{cases} \tag{3.22}$$

The observed MAPE $\epsilon$ is then normalized to $[0, 1]$ using the maximum and minimum MAPEs, i.e. static INT4 and INT16 configurations $\epsilon_{i4}$ and $\epsilon_{i16}$ respectively, and mapped to a logistic sigmoid function:

$$C_{err} = f_\sigma\left(\frac{\epsilon - \epsilon_{i16}}{\epsilon_{i4} - \epsilon_{i16}}\right), \tag{3.23}$$

where the logistic sigmoid function $f_\sigma$ is defined as

$$f_\sigma(x) = \frac{1}{exp(-k(x - x_0))}. \tag{3.24}$$

Here $x_0$ is the middle-point of the curve and is set to the desired MAPE threshold normalized to $[0, 1]$ using the maximum and minimum MAPEs similarly as above. The steepness $k$ of the curve determines how fast the curve goes from $0$ to $1$, and is set now to an arbitrary high value of $200$ to keep the accuracy cost values very close to $0$ with MAPEs below the threshold, and saturate quickly to $1$ with MAPEs above the threshold. It should be noted that as the logistic function implementing the threshold is still a continuous curve, there is a slight slope in the threshold area. Combined with the fact that $f_\sigma(x_0) = 0.5$, this means that the $C_{err}$ starts to already increase significantly with values slightly below the threshold. This effectively means that the search method aims to find models with MAPEs slightly below the threshold, instead of trying to directly match the threshold.

Memory usage and compute latency are calculated as discussed in Section 3.3. For both memory usage and compute cycles, the values are again normalized to $[0, 1]$ using the worst- and best-case resource costs, i.e. INT16 and INT4 configurations respectively:

$$C_{memory,norm} = \frac{C_{memory} - C_{memory,i4}}{C_{memory,i16} - C_{memory,i4}} \tag{3.25}$$

$$C_{latency,norm} = \frac{C_{latency} - C_{latency,i4}}{C_{latency,i16} - C_{latency,i4}} \tag{3.26}$$

The target resource cost $C_{resources}$ used during optimizing is chosen between $C_{memory,norm}$ and $C_{latency,norm}$. Finally, the cost function $C$ can be written as

$$C = 0.51C_{err} + 0.49C_{resources} \tag{3.27}$$

The reasoning behind this weighted sum is to always prefer just slightly more models with lower output errors, in case multiple models all having different values for $C_{err}$ and $C_{resources}$ but with their sum being equal in all cases is met. Generally these cases mainly happen when initializing $\mathbf{S_{best}}$ based on the static baseline configurations. Finally, as the sampling center $\mathbf{m}$ is initialized to the corresponding static INT16 configuration, the search always starts below the MAPE threshold and starts to optimize the resource until the threshold is reached. This prevents the search from getting stuck to the search space area where the MAPE is always above the given threshold.

When the resource used for optimizing is memory consumption, INT4 option for model input, intermediate quantizations, and output is left completely out of the search. This is to reduce the search space and ease the overall search process, as INT4 representation of dynamic data does not affect the memory consumption and is generally a bad choice in terms of model accuracy. It is also common to leave activations completely out of the search and set these according to the target hardware. Similarly with latency as resource cost, global bias vector significant bits is left out of the search as it only affects the memory consumption of the model.

# 4. RELATED WORK

Integer representation of neural networks is a very popular reasearch topic due to e.g. general hardware accelerator chips and model-specific FPGA implementations, which has consequently resulted in many solutions and frameworks being developed. Jacob et. al [34] explored the quantization and training of neural networks using integer-only arithmetics for inference. The authors proposed a quantization scheme and inference framework which together enable more efficient inference on integer-only hardware as opposed to floating point inference. Later this became the implementation used in TensorFlow Lite [1]. Their proposed quantization scheme consisted of representing both weights and activations as 8-bit integers and bias vectors as 32-bit integers. This effectively means that operations, such as convolution or matrix multiplication, involves 8-bit input operands and a 32-bit accumulator for the multiplication result and bias addition. The resulting 32-bit integer is then quantized to an 8-bit integer value and fed to the activation involving only 8-bit arithmetic. Although this quantization scheme is effective, the authors did not discuss INT16 or INT4 implementation, or heterogeneous configuration of weight, input, bias and activation precisions.

In the extreme cases a neural network can be quantized using only binary or ternary representation of weights and two- or three-valued discrete activations. In [43], the authors investigated how decreasing a network's numerical precision to binary or ternary affects the model's performance and resource consumption. They considered two multi-class classification tasks where the models were trained with their floating point representation and compared the quantized models to the floating point models. They showed how binary and ternary networks are able to preserve a competitive accuracy regardless of their low precision representation. Moreover, they concluded that ternary models reach superior accuracy values when compared to binary models, and that generally these extremely quantized models use significantly less resources with low inference latency.

Park et. al [44] proposed a deep neural network with two different precision modes in a single model. This dual-precision neural network enables easy switching between the two different precision modes. The purpose of this is to support scalable dynamic tradeoff between model accuracy and complexity for applications in dynamic environments. They achieved this by sharing the common bits between the weights from both precision modes, and by proposing a two-part training process consisting of shared-bit training and full high-precision training.

Training a model simultaneously under different bit-widths was investigated by Jin et. al. [45]. Their approach was to use shared weights for different bit-widths during training and adopt the switchable batch normalization introduced in [46] to moderate the different variances of weights and activations caused by different quantization bit-widths. Moreover, they proposed using independent clipping levels [47] for different bit-widths during training to avoid quantization error due to too large or small clipping levels for different bit-widths.

Adaptive layer-wise mixed precision quantization of deep neural networks was studied in a 2018 publication by Zhu et. al [4], where they proposed a method for using different bit-widths for different layers. Their method aims to quantize most of the layers using low bit representations and assign more bits only to the most important layers. They used entropy as an indicator of importance for each layer by observing the distribution of weight and activation values. These distributions were approximated by uniformly dividing the values into discrete bins and then calculating the entropy for this discrete distribution. To mitigate the large values produced by unconstrained activations, an additional penalty term for preventing too large activation values was introduced.

In [48], the authors explored using periodic functions as reqularizers for training mixed-precision neural network models, while simultaneously learning the bit-widths of different layers by distinguishing the respective importance concerning model accuracy for each layer. The periodic functions, such as continuous sine and cosine or non-continuous hat function, were applied as a regularization term to the training loss instead of using a quantization function. Their proposed regularization term pushes the weight and activation values to a set of discrete points during training, effectively acting as a quantization function. The frequencies of the periodic regularizer functions are directly related to the number of bits assigned, thus each selected frequency is used for determining the model bit-widths.

Mixed precision quantization for convolutional networks was studied by Wu et. al in their paper [5], where they approached the problem of finding different quantization precisions for different layers as a model architecture search problem. They proposed a differentiable neural architecture search framework which uses gradient-based optimization to explore the neural network architecture search space. This framework represents the architecture search space as a stochastic super net, where intermediate data tensors, e.g. feature maps, are represented as nodes, and operators, e.g. convolution layers, are represented as edges. Any architecture can be then seen as a sub-graph of this super net. To solve the mixed precision quantization problem, a super net with the same number of layers as the target network and with each layer containing several parallel edges representing different quantization precisions is constructed. The optimal architecture is then solved by using stochastic gradient descent with respect to quantized weights and parameterized network architecture. They showed that their framework for layer-wise precision assignments surpassed state-of-the-art compression on ResNet models. Moreover, they concluded that their pipeline is faster when compared to previous network architecture search algorithms where search time is typically several days using hundreds of GPUs as opposed to several hours with only a few GPUs using their method.

Coelho Jr. et. al [6] explored automatic mixed precision quantization of neural networks and provided software libraries *QKeras* and *AutoQKeras* for using their methods on Keras [49] models. They introduced a method for heterogeneously quantizing deep neural networks using an automatic procedure which aims to minimize area and power consumption while maximizing the accuracy. This method enables users to trade off model area or energy consumption for accuracy according to the application. This is achieved by defining a forgiving factor which indicates how big of an accuracy drop is tolerated for a given energy consumption or area reduction, while also using an estimation of model energy consumption to allow the algorithm to simultaneously tune the model quantization and architecture. This means that the model architecture is changed according to the level of quantization done to each part of the model, e.g. fewer convolutional filters might be needed for a certain layer when lower bit-width quantization is used. The actual tuning of quantization and architecture is treated as a hyper parameter search by using either random search methods, hyperband [50] or Gaussian processes.

Yang et. al. [7] further investigated mixed precision quantization by utilizing differentiable fractional bit-widths, where the transition between neighbouring quantized bits is smoothed. The pipeline for their method consisted of first searching with fractional bit-width using gradient-based optimization and then fine-tuning with mixed bit-width quantization. During the optimization, the model converges to optimal bit-widths for both weights and activations of each unit by using a resource constraint as a penalty loss for the optimizer. Fine-tuning consists discretizing the fractional bit-widths using a threshold value which results in a model resource cost within 1% deviation from the target constraint. They also noted how this method also supports kernel-wise quantization where each convolution kernel producing a single-channel feature map is quantized separately.

A method for automated kernel-wise quantization was proposed in a paper by Lou et. al. [8], where they used a hierarchical deep reinforcement learning -based schemes to search distinct quantization bit-widths for each convolution weights in kernel-wise manner and activation in layer-wise manner. The reinforcement learning agent considers the inference latency and energy as well as FPGA area overhead in its extrinsic reward function. The method consists of a high-level controller agent which performs layer-wise quantization, and a low-level controller agent which finds the kernel-wise bit-widths.

Reinforcement learning leveraged automated quantization with mixed precision was also studied in a paper by Wang et. al. [9]. They introduced a hardware-aware automated quantization method, which includes additional feedback from the target hardware in the loop by utilizing a hardware simulator. The reinforcement learning agent processes the target network layer by layer, taking an action for both the quantization of weights and activations at each layer. The action space they used is continuous to preserve relative order between different bit-widths during search, and an action is rounded into a practical discrete bit-width value before quantization. The direct latency and energy feedback from the hardware accelerator is used as resource constraints to determine the bit-widths for each layer. If a policy exceeds the resource limits, the bit-widths of each layer is sequentially decreased until the policy is withing limits.

In HAWQv3, Yao et. al. [10] introduced a hardware-aware mixed precision quantization framework that proposed a fast method for finding the optimal bit precision for a given application-specific constraint, while only relying on integer multiplication, addition and bit shifting. Their method formalized the problem as an Integer Linear Programming problem, which tries to minimize a sensitivity metric for each layer to find the optimal bit-widths. They also addressed the precision used for residual connections and ended up using INT32 for the skip connection. Moreover, the authors also discussed the common question about hardware efficiency of mixed-precision quantization by deploying models on NVIDIA T4 GPUs, ultimately showing that up to 50% speed up is achievable with INT4/INT8 mixed-precision quantization compared to 8-bit integer quantization.

Bruin et. al [51] explored layer-wise quantization of deep neural networks for processors with limited accumulation registers. Their heuristic technique aims to perform layer-wise optimization of quantization to maximize the classification accuracy with a given fixed accumulator size and maximum data width. In their method, they define several accumulator constraints to first reduce the search space, and use a straightforward iterative approach to find the best input data and accumulator bit width configuration for each layer. Layers are processed in a sequential manner starting from the first eligible layer. For each layer, the solutions proposed by the chosen constraint are tested and the best solution for data bus width and accumulator bit width is set to the layer. The accumulator constraints they proposed were divided into pessimistic, conservative and optimistic, which all provide an upper bound for the accumulator width. Moreover, they also provided a novel procedure for fine-tuning the solution from the heuristic optimization by combining reduced-precision forward computation pass with high-precision backward pass. During fine-tuning, the loss value for the network is calculated using a simulated fixed-point network, which is then used to update the full-precision weights.

In [52], the authors proposed a method for adapting neural networks to use 8-bit additions in the accumulators instead of the commonly used 32-bits, and naming the architecture utilizing this method WrapNet. To enable the use of low precision accumulation, WrapNet includes a cyclic activation function, an overflow penalty, and a special strategy for quantization activation scale selection. The cyclic activation function is inserted after every convolutional and linear layer to emulate the wrap-around behaviour associated with overflows while also ensuring the activation is continuous thus enabling gradient-based training. The proposed overflow penalty acts as a regularizer during training by penalizing outputs exceeding the accumulator bit-width. Finally, the quantization activation scale for each layer is adjusted according to the overflow rate to balance accumulation and quantization errors.

As seen from the overview above, mixed precision quantization and automatic solutions to it are widely researched topics. Although many novel and intriguing approaches for mixed precision quantization optimization have been proposed, many of these aim to highly optimize the quantization configuration using very compute costly methods. The methods discussed in [5] and [9] both utilize reinforcement learning, which is a tedious resource heavy process comparable to the actual network training process. Similarly in [7], gradient-based optimization was used which isn't a lightweight solution. The approach in [48] embeds the quantization optimization to the training process as training is always needed for neural networks, however, this approach always requires costly gradient-based training to tune the quantization configuration. As discussed in Chapter 3, the proposed iterative correlation coefficient adaptation does not require any gradient information during optimization, and is thus lightweight and relatively simple to implement. The drawback of this is that the proposed method might not be able to optimize the architecture as extremely as gradient-based methods, as it is a random search method at its core with a heuristic technique embedded in it.

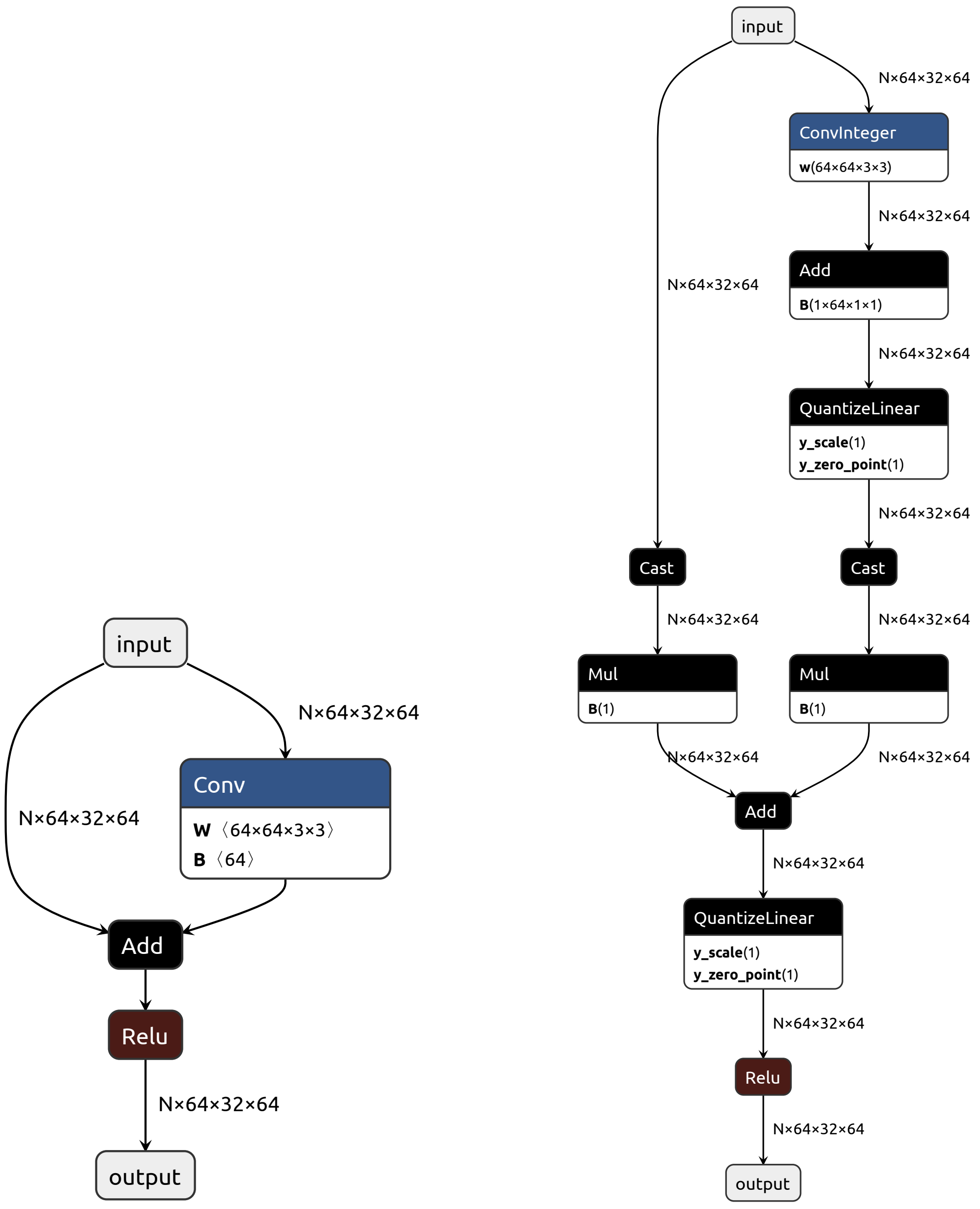
# 5. EVALUATION ENVIRONMENT

In this chapter, the target environment used for the experiments is discussed in more detail. The pre-hardware representation used here is the Open Neural Network Exchange (ONNX) format [3], as it is independent of the training framework. The ONNX quantization method discussed in Section 5.1 is not the same as the one implemented in ONNX runtime [40], but rather a method which ensures that the quantized ONNX model is implementable on the target hardware discussed in Section 5.2. Furthermore, in Section 5.3 the static quantization configurations used as baselines during the experiments are defined explicitly. Finally, the actual model being quantized and optimized, a CNN-based radio receiver named DeepRx, is overviewed in Section 5.4.

## 5.1 Quantization with Open Neural Network Exchange

As discussed in Section 2.5, operators enabling quantization were first introduced in operator set version 10 as part of ONNX version 1.5.0. All operators and their behaviour discussed here refer to the current ONNX specification version 1.9.0, which has functional but somewhat limited quantization support. The main operations which will be discussed are convolution, matrix multiplication, element-wise addition, and ReLU activation function, as they form the core operators utilized by the target hardware.

First, the basic ONNX quantization procedure used in the target environment is defined. For convolution layers with bias vector addition and ReLU activations, the floating-point ONNX operators *Conv* and *ReLU* are converted into *ConvInteger*, *Add*, *ReLU* and *QuantizeLinear* integer ONNX operators respectively. Similarly for matrix multiplications, the floating-point operators *MatMul* and *ReLU* are converted into *MatMulInteger*, *Add*, *ReLU* and *QuantizeLinear* integer operators respectively. The *Add* operator introduced for convolutions and matrix multiplications performs bias addition, which is already integrated into the floating point convolution *Conv*, but not in *MatMul*. The *QuantizeLinear* operator performs data scaling and quantization by using a pre-calculated scaling value to scale its high precision input data tensor, followed by rounding and conversion of the data to a lower precision data type. In case addition of two non-constant inputs are needed, e.g. in residual blocks, *Cast* and *Mul* operators are introduced to cast and scale both inputs of the original *Add* operator into a common format, followed by a *QuantizeLinear* operator.



***(a)*** *Original model* ***(b)*** *Quantized model*

***Figure 5.1.*** *Example of ONNX quantization procedure in the target environment*

A comparison between an example ONNX model and its corresponding quantized model is illustrated in Figure 5.1.

Operators *ConvInteger* and *MatmulInteger* function similarly as their floating point counterparts, but instead using integer values. The weight arrays for these are converted to integer values as discussed in 2.2.1 and stored as constant initializers. The same is repeated for *Add* operators performing bias addition by quantizing each bias vector and storing it as a constant initializer. With non-constant *Add* operators, both inputs are first cast to a common data type, then scaled by multiplication to a common scale based on the data types and magnitudes of both inputs, and finally the addition can be calculated.

The scaling discussed in Sections 2.2.1 and 2.2.2 is performed during each *QuantizeLinear* operator. Each scale is calculated during model quantization and stored as a constant initializer. For *ReLU* operators, the only change is the input and output data types which are changed from floating point to integer values.

Mixed precision quantization is something that isn't thoroughly supported yet, as quantization is a relatively new feature within the ONNX specification. Next, the changes and additions needed to fully enable mixed precision quantization on the current ONNX specification (version 1.9.0) will be discussed. The aim is to support a mix of INT4, INT8 and INT16 quantizations. The changes proposed here enable mixed precision quantization within the in-house framework used for manipulating ONNX models in this thesis, but can be additionally used as a reference when extending other frameworks or the ONNX specification for mixed precision quantization.

The first obvious change needed in the context of this thesis, is the addition of a 4-bit integer data type, which in practice means simply extending the existing data type enumeration. The behaviour of 4-bit integers may need to be emulated by software, as the byte size of x86, ARM, and similar architectures is commonly 8 bits. The input data type constraints for operators *ConvInteger*, *MatMulInteger* need to be extended to also include 16-bit and 4-bit integers. As accumulated 16-bit multiplications may need more than 32 bits, the output type constraints of these operators need to additionally include 64-bit integers. Moreover, the output type constraints may also include 16-bit integers for 4-bit multiplications. For *Add* operators, all types are already supported in the current specification. Inputs with different types for *Add* are not supported, however, this is not required as constant bias values are bit-shifted to the same scale as will be discussed in Section 5.2, and non-constant inputs are cast and scaled using *Cast* and *Mul* operators as discussed above. *QuantizeLinear* operators are only able to quantize 32-bit integer values to 8-bit integers in the current specification. Thus, the input type constraint need to be extended to include 64-bit integers (and 16-bit integers if included in *ConvInteger* or *MatmulInteger* output constraints), and the quantized output constraints to also include 4-bit and 16-bit integers.

## 5.2 Target Hardware

In this section, the target hardware and parts of its implementation in a high level of abstraction will be discussed. Here target hardware refers to an example hardware platform the quantization scheme from Section 2.2.1 and ONNX representation from Section 5.1 aim for, however the proposed optimization method is not bound to this platform. The goal of this hardware is to efficiently run models quantized using the techniques discussed previously. For the target hardware used in this thesis, the integer-arithmetic matrix multiplication, bias-addition, activation function and quantization are fused into a single pipeline.

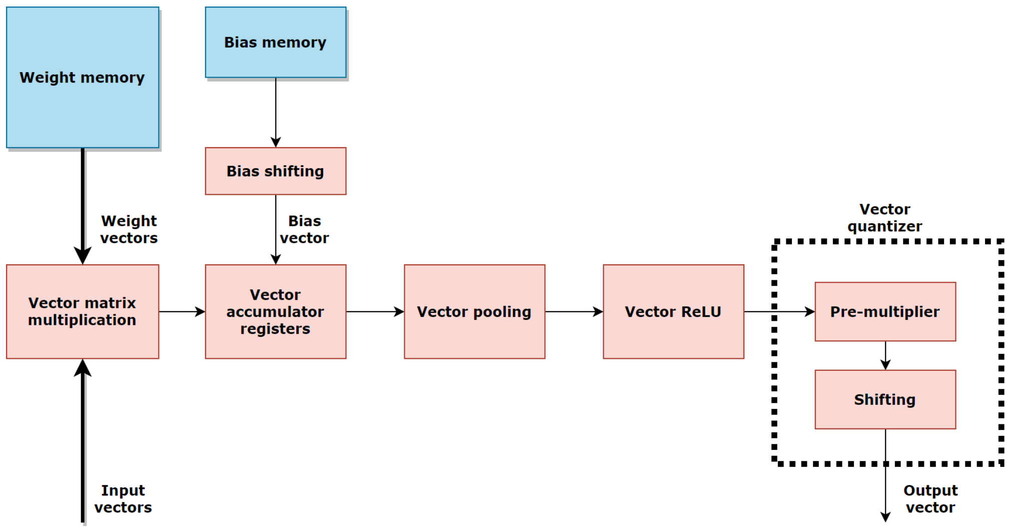


***Figure 5.2.*** *High level implementation of a fused layer on hardware*

The exact compute capabilities and specific bit-widths will not be disclosed, but rather a general example of the functionality is discussed. The general pipeline of a fused layer is illustrated in Figure 5.2

The fused layer pipeline starts by first loading matrix multiplication weights from a fast local Static Random Access Memory (SRAM) block, and setting these to the vector matrix multiplication unit. The vector matrix multiplication unit considers matrices as sets of vectors and is responsible for calculating kernel-wise operations and full element-wise matrix multiplications. Additionally, the accumulator registers for a single vector is initialized to the bias vector loaded from another SRAM block for bias values. Bias loading includes a special bias shifting stage, which can be used to dynamically scale bias values larger according to the target value range, e.g. from 16 bits to 32 bits with the least significant bits being zeros. The vector multiplication results from the vector matrix multiplication unit are accumulated to the initialized accumulators, and then passed to the vector pooling and vector ReLU activation units respectively. The final stage is the quantizer, which quantizes the higher precision activation output to a lower precision, e.g. from 32 bits to 8 bits, and consists of a pre-multiplier stage followed by a power-of-2 downscaler using bit-shifting as formulated in Section 2.2.2. For the hardware considered here, *half away from zero* rounding is used, which can be formulated as

$$round(x) = -sgn(x)\lceil -|x| - 0.5 \rceil \tag{5.1}$$

where $sgn$ is the sign function which returns the sign of its input, and $\lceil . \rceil$ is the ceiling function. The benefit of this rounding mode is that its behaviour is efficiently implementable on hardware.

The kernel and matrix multiplication weights are stored as is in the weight memory block, whereas bias memory includes only the available most significant bits for each value. Therefore, the data type (INT4, INT8 or INT16) of each weight array determines how much weight memory they consume, and the amount of bits used for bias values determines how much bias values consume bias memory. The amount of most significant bias bits is kept constant for all bias vectors, as assigning different amount of bits for each bias vector wouldn't provide significant memory optimization compared to weight memory, and would also expand the quantization configuration search space drastically. To implement this, full precision bias vectors are processed during quantization to ensure they fit to the available bits. For each bias vector separately, the maximum magnitude value is observed, and if this exceeds the maximum value representable using the available bias bits the values are right shifted (sacrificing the least significant bits) until the remaining bits are representable. The amount of shifting done is stored and then later applied as left shift in the bias shifting stage.

In practice, multiple parallel fused matrix multiplication pipelines can be used to leverage parallelism for faster network graph computation. For some element-wise vector operations which cannot be efficiently executed by the discussed pipeline, such as residual connections, additional special hardware may be required. The details of this hardware are not disclosed here, but are still mentioned to highlight the limitations of the discussed fused matrix multiplication pipeline. The target hardware aims for 100% utilization rate, meaning that the computational resources are fully utilized at all times during the computation of a model graph fulfilling the hardware requirements, e.g. sizes of inputs and weight arrays.

## 5.3 Static Quantization Configurations

Static quantization configurations in the context of this thesis will now be explicitly defined, as they are a key part of the experiments performed in Chapter 6. Static INT8 quantization is a commonly used configuration, as it's usually a good balance between model output error and resource consumption. In this configuration, each input, weight array and quantization output uses 8-bit integer precision, and the number of bits used for bias vector values is set to 16. For static INT4 configuration, inputs, weight arrays, and outputs use 4-bit integers, and the number of bias vector bits used is set to 8. For static INT16 configuration, each input, weight array and quantization outputs uses 16-bit integer precision, number of bias vector bits used set to 32. For all of the configurations, the pre-multiplier bits, i.e. $b$ in Equations 2.13 and 2.14, is set to 3.

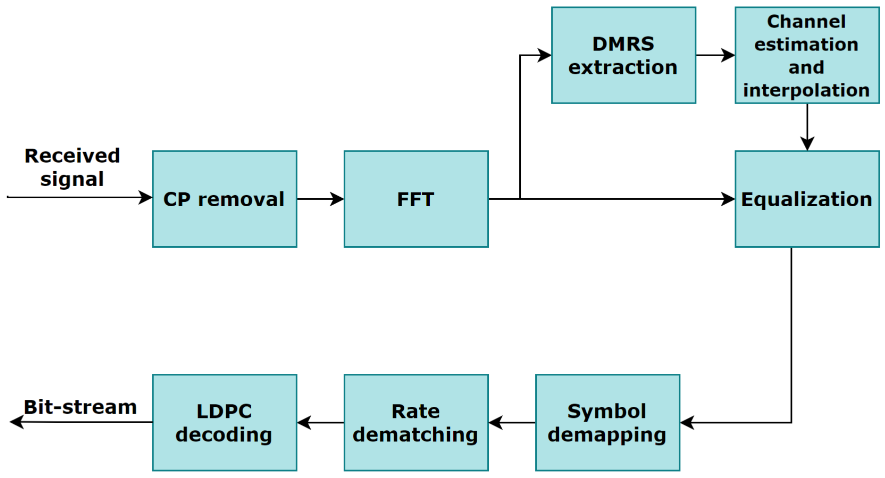


***Figure 5.3.*** *Traditional OFDM receiver pipeline stages illustrated*

## 5.4 DeepRx

Replacing a specific part of a traditional radio receiver pipeline with a neural network has been proposed by multiple researchers [53, 54, 55]. In DeepRx [26], almost the whole receiver pipeline is executed in a 5G-compliant fashion using a CNN developed in Nokia Bell Labs. As shown in the original paper, DeepRx performs significantly better than a traditional OFDM receiver pipeline. Therefore this novel approach could be beneficial if made implementable on the radio receiver hardware with feasible inference latency and power consumption.

A traditional digital wideband radio receiver commonly uses Orthogonal Frequency-Division Multiplexing (OFDM) [56] as its digital data transmission method, as it is capable of encoding bit-streams into multiple subcarrier frequencies for carrying data in parallel. In OFDM transmission, a bit-stream is first encoded using an error correction code, such as Low-Density Parity Check code (LDPC), followed by rate matching and OFDM symbol mapping. Next, Demodulation Reference Signals (DMRS) (also known as pilot signals) are inserted to the symbols for later signal demodulation and transmit channel estimation. Finally, the Inverse Fast Fourier Transform (IFFT) is calculated and a Cyclic Prefix (CP) is added as a guard interval to prevent interference between different transmissions. This is the final transmitted signal. The receiver needs to reverse all of the transmitter steps and also assess and mitigate any distortion caused by the transmission channel. Consequently, the receiver pipeline includes an additional channel estimation and interpolation stage utilizing the pilot reference symbols, followed by equalization where the effects of the transmission channel are mitigated.

From the traditional ODFM receiver, illustrated in Figure 5.3, DeepRx replaces the DMRS extraction, channel estimation and interpolation, equalization, and symbol demapping tasks with a single CNN. The network is a fully convolutional deep neural network utilizing pre-activation blocks similarly as in [25], and also depthwise separable convolutions. The original paper focused mainly on Quadrature Amplitude Modulation (QAM) with 16 values, however the model is designed to also handle other QAM schemes. The modulation scheme considered in this thesis is QAM with 256 values, i.e. 8 bits per symbol.

The input for DeepRx consists of frequency-domain representation of the received antenna data with CP removed, reference pilot symbols positioned to correspond to the pilot positions in antenna data with zero-padding in non-pilot positions, and a pre-computed raw channel estimate. All of these are stacked to separate channels to form the full network input. The output of the network represents the raw bit log-likelihood ratios (LLR) similarly as in the output of symbol demapper in the OFDM receiver pipeline. Hence the output of DeepRx requires additional post-processing to acquire the final bit stream.

Although the proposed model shows potentially very significant receiver accuracy boosts, the hardware implementation of even the smaller single receiver antenna model limits the practicality of DeepRx drastically. Very large neural network computation graphs implemented on hardware require more power and silicon area which are crucial factors especially when designing large scale telecommunication products. The work done in [39] aims to prune DeepRx to a much smaller size using a so-called quantization-aware multi-stage pruning algorithm. The proposed algorithm is a crucial part of the model deployment pipeline which the method proposed in Chapter 3 also belongs to. Therefore, DeepRx and more specifically a pruned version of it is the key model used for evaluation in Chapter 6, as it’s very relevant to the work done in this thesis at Nokia.

# 6. EXPERIMENTS

The iterative correlation coefficient adaptation method proposed in Chapter 3 will be evaluated on DeepRx in this chapter. The experiments are arranged as follows. First, the hyperparameters for the proposed method are explored and discussed using a small test network, and appropriate values for each are suggested. Next, the DeepRx neural network is quantized using the static quantization configurations defined in Section 5.3 and used as a baseline for the coming mixed precision quantization results. This is followed by common manual mixed precision configurations and the proposed automatic method, which are both evaluated in terms of DeepRx performance, memory usage, and compute cycle latency. The automatic method is used for both memory and latency optimization, and all of the automatically optimized models are additionally compared with the manual mixed precision configurations to conclude the functionality of the method.

## 6.1 Hyperparameters for the Proposed Method

In this section, different hyperparameter values for the method proposed in Section 3 are explored, effects of each parameter discussed, and best choices concluded. The model used for examining the method behaviour consists of an input convolutional layer with three residual blocks and a fully-connected layer all utilizing a ReLU activation. All searches performed started with the same sampling center and random seed, and the target cost function maximum MAPE set to $\epsilon_{i8}$ with latency being the resource target.

**Sample set size** $\lambda$ defines how frequently the sampling center is updated, consequently also affecting the time complexity of each epoch. Together with $\mu$ and $\mathbf{w}$, they define how many samples and in what proportions are used for calculating the sample based sampling center update. For all hyperparameter tests performed, $\mu$ was set to $\lfloor \frac{\lambda}{3} \rfloor$ and $\mathbf{w}$ were calculated as

$$\mathbf{w} = \left[ \frac{\mu^5}{\sum_{i=1}^{\mu} i^5}, \frac{(\mu-1)^5}{\sum_{i=1}^{\mu} i^5}, \ldots, \frac{1^5}{\sum_{i=1,}^{\mu} i^5} \right], \tag{6.1}$$

which satisfies both Equations 3.2 and 3.3. The sample set sizes included in the tests here were $\{5, 10, 15, 20\}$, and $\sigma = 0.10, \gamma = 0.95$ was used for all tests. The search results are illustrated in Figure 6.1. As the sample set size increases, the mean cost of

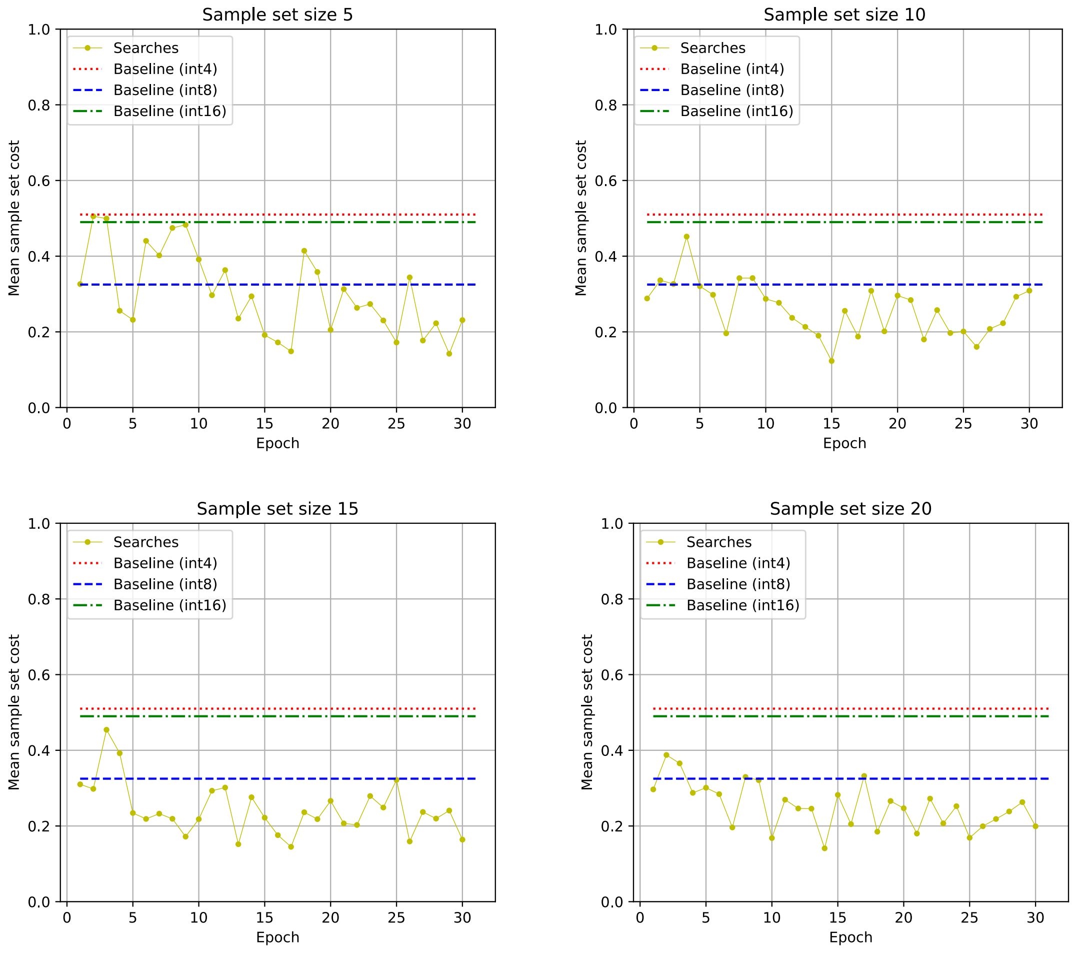


***Figure 6.1.*** *Search results with different sample set sizes*

each set becomes less stochastic, while simultaneously the time it takes to complete a single epoch increases. Based on more extensive tests, $\lambda = 5$ is a too small value and the search does not converge easily, whereas with $\lambda = 20$, the search time increases significantly. Hence, a sample set size of 10 or 15 or similar is a good choice.

**Sampling variance** $\sigma$ affects how widely the current search center is sampled. Too small variance makes it hard for the new sample set to contain new unexplored states, whereas too big variance is not able to fully exploit the learned sampling center. Figure 6.2 shows results for pure random ($\gamma = 1$) searches with different sampling variances. The sample set size used here is $\lambda = 10$. When $\sigma^2 = 0.05$, the search finds and stays around a good solution but does not explore many new states. As the variance is increased to $\sigma^2 = 0.10$, the search starts to explore more states where the cost function occasionally saturates to larger values due to too high accuracy drops, but also states below the threshold. With $\sigma^2 = 0.15$ and larger the sampling starts to become too random, i.e. updating the sampling center does not have a big effect to the sampling process, and a good solution might be missed completely. Hence, the preferred variance is approximately $\sigma^2 = 0.10$.

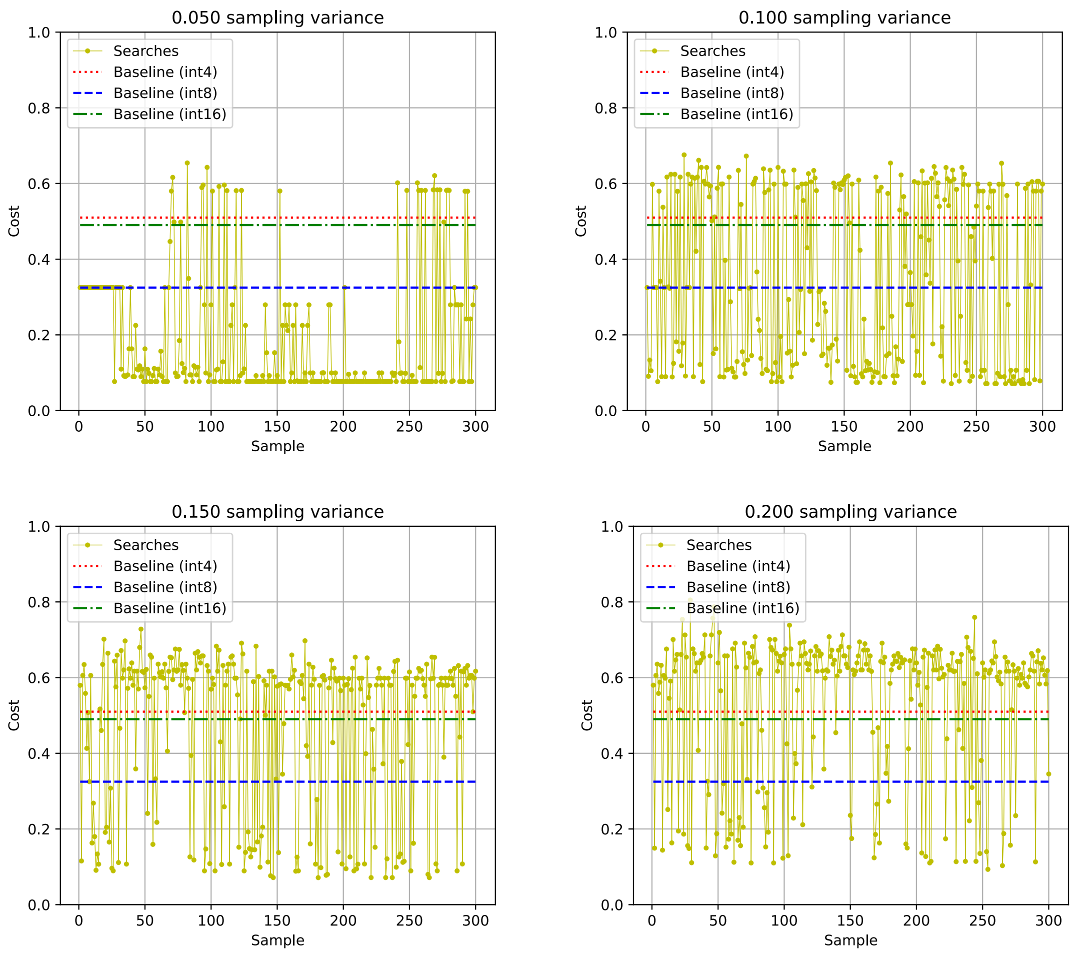


***Figure 6.2.*** *Pure random search results with different sampling variances*

**Learning rate discount factor**, i.e. $\gamma$ in Equations 3.9 and 3.8, defines how fast the correlation coefficient candidate solution is used. As the candidate solution requires a diverse set of explored states, $\gamma$ should not be set too low as then random sampling is not executed to the required extend and the candidate solution is used prematurely. Furthermore, a too large value makes it hard for the algorithm to exploit the candidate solution. A total of 10 searches with $\gamma \in [0.90, 0.91, \ldots, 0.99]$ were performed, all having $\lambda = 10$ and $\sigma = 0.10$. Figure 6.3 illustrates some of the search results with different values of $\gamma$. With lower values of $\gamma$, e.g. $\gamma = 0.91$, the search converges fast and keeps sampling around a suitable solution. However, especially when the search space gets significantly larger, the candidate solution may be significantly worse than the global optimum, or a sampled new best solution might be missed as the center converges too quickly. With higher values, e.g. $\gamma = 0.99$, the search takes more epochs to converge and the candidate solution is not exploited to its full extent. Based on more extensive tests, $\gamma = 0.95$ is a good choice for many model architectures, however, smaller values such as $\gamma = 0.92$ can be used to speed-up the search. Lastly, it should be mentioned that very small models with small search spaces should prefer larger values of $\gamma$, and possibly larger values of $\sigma$, as the time cost of performing more stochastic search is noticeably smaller and a more optimal solution may be thus discovered.

## 6.2 Baselines

For evaluating DeepRx, the uncoded Bit Error Rates (BER) and signal-to-noise ratio (SNR) are utilized. Here uncoded BER refers to the log-likelihood ratios before LDPC decoding, and coded refers to the LDPC decoded bits. Signal-to-noise ratio indicates how much background noise there is by comparing the noise-free signal to the background noise. A lower SNR value indicates a more noisy signal. The BER against SNR curves are plotted for each DeepRx model with 1 and 2 pilots, and are compared against a traditional OFDM receiver with Linear Minimum Mean Square Error (LMMSE) used for channel estimation with 1 and 2 pilots, as well as an optimal (non-practical) receiver with perfect channel knowledge. By representing SNR as a function of BER, the robustness of each method based on the error rates for different amounts of background noise can be observed. The evaluation data set used consists of 256 QAM modulation scheme data, i.e. each OFDM symbol is represented using 8 bits, and was provided by the original authors of [26].

As the main goal of DeepRx is to outperform traditional methods, quantization should not introduce too drastic errors. As a baseline, all static quantization configurations from Section 5.3 are evaluated. The DeepRx model here consists of an input convolutional layer and 11 residual blocks utilizing depthwise separable convolutions, and is quantization-aware trained and pruned using the method proposed in [39]. The original floating point model and all quantized models are evaluated and illustrated in Figure 6.4. Static INT16 quantization does not introduce notable errors and is very close to the original model, however the resource consumption is the worst possible. Static INT8 quantization introduces clear errors and does not perform as desired, but is still somewhat usable and has a good overall resource consumption. Errors introduced by static INT4 quantization are far too big and make the model unusable, even though the minimal resources would be beneficial. Therefore, a suitable MAPE threshold for the search method is the static INT8 configuration MAPE.

## 6.3 Manual Mixed Precision

Next, some of the common manual mixed precision configurations are evaluated. The configurations considered here are INT8 activations with INT4 weights, INT8 activations with INT16 weights, and INT16 activations with INT8 weights. These are referred to as 8x4, 8x16, and 16x8, respectively. For all of these configurations, bias bits is set to 16. The BER against SNR curves for each configuration are illustrated in Figure 6.5, and the

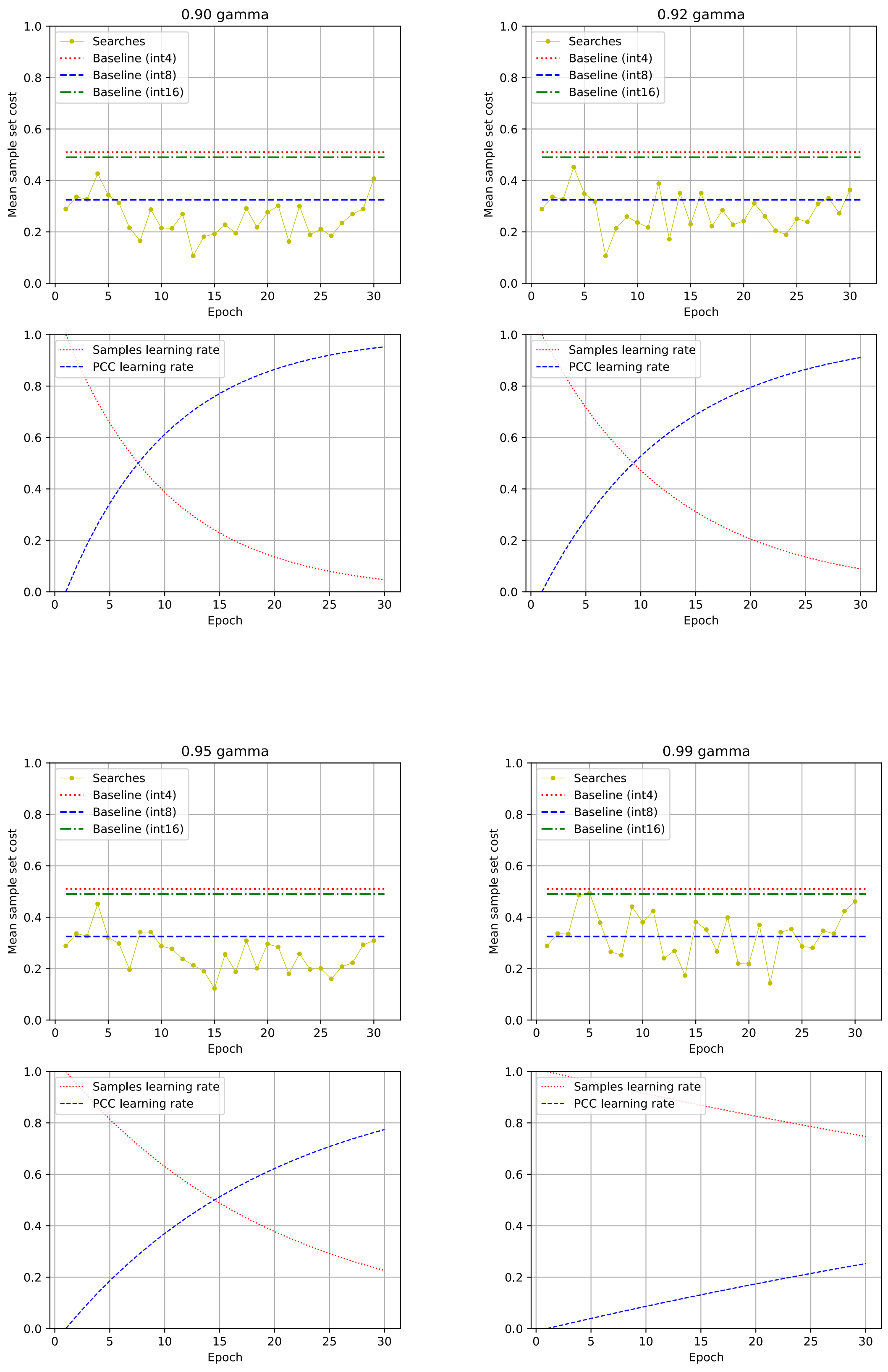
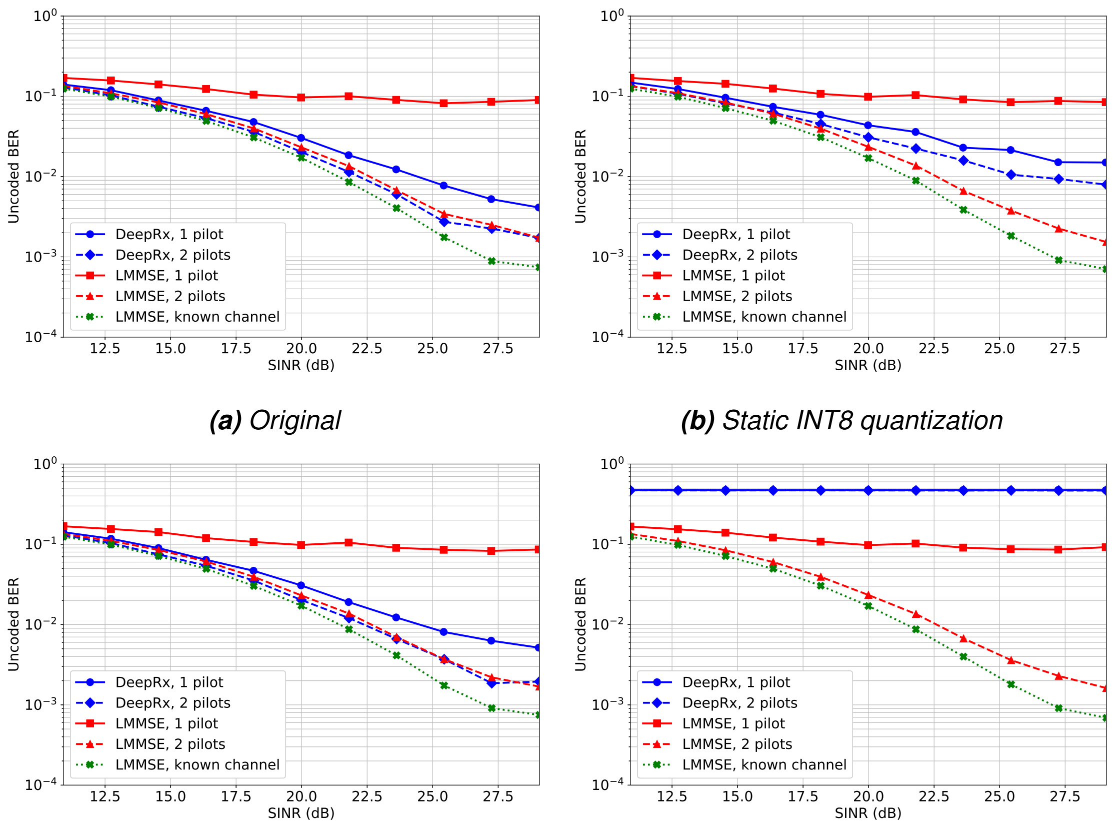


***Figure 6.3.*** *Mean sample set cost for each epoch with different values of* $\gamma$



***(a)** Original*

***(b)** Static INT8 quantization*

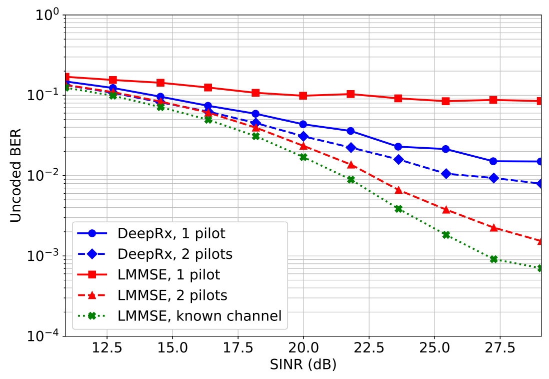
***(c)** Static INT16 quantization*

***(d)** Static INT4 quantization*

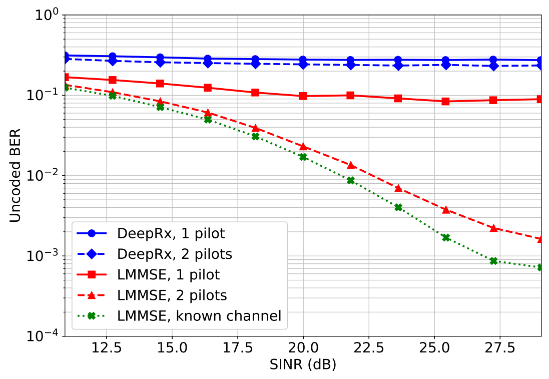
***Figure 6.4.** Original and quantized DeepRx models with pruning and QAT*

corresponding DeepRx models are summarized in Table 6.1. The weight values in Table 6.1 refer to the percentage of all convolution kernel weight parameters using a specific data type, and the activations refer to the number of intermediate *QuantizeLinear* operator outputs with a certain data type as discussed in Section 5.1.

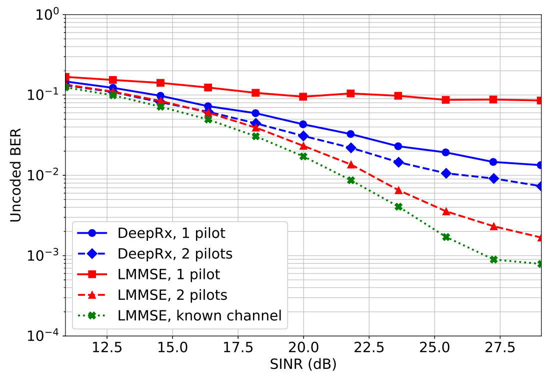
As seen in Figure 6.5, 16x8 configuration performs significantly better than others and is very close to the static INT16 configuration, 8x16 performs similarly as static INT8 quantization, and 8x4 configuration is completely unusable. Moreover, as 8x16 configuration does not provide any significant accuracy gains and has worse latency when compared to INT8, it is not considered as a feasible mixed precision configuration, and therefore 16x8 is the only usable configuration here. Table 6.1 shows that 16x8 has the same memory consumption as static INT8 configuration, and 50% of static INT16 latency while keeping the model bit error rates very close to static INT16 configuration. Thus, it can be concluded that for DeepRx 16x8 manual mixed precision configuration is very suitable and a good compromise between static INT8 and INT16 configurations. This configuration has also been implemented in Tensorflow Lite [1], where it is mentioned that the configuration can significantly improve the accuracy of quantized models.



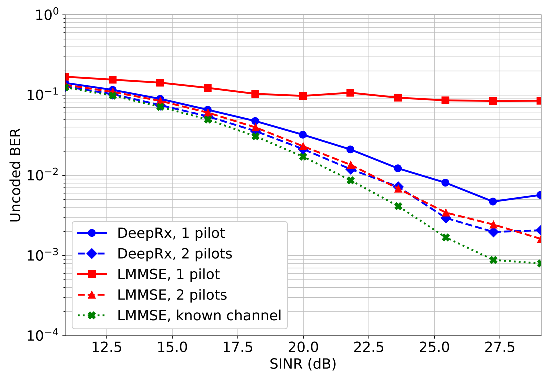
***(a)*** *Static INT8 quantization*



***(b)*** *INT8 activations with INT4 weights*



***(c)*** *INT8 activations with INT16 weights*



***(d)*** *INT16 activations with INT8 weights*

***Figure 6.5.*** *Manual mixed precision quantized DeepRx models*

***Table 6.1.*** *Manual mixed precision quantization models*

| | **8x4** | **8x16** | **16x8** | **Static INT4** | **Static INT8** | **Static INT16** |
|---|---|---|---|---|---|---|
| *MAPE* | 2.172 | 0.589 | 0.187 | 2.894 | 0.677 | 0.003 |
| *% of worst memory case* | 25.5% | 99.1% | 50.0% | 25.0% | 50.0% | 100.00% |
| *INT4 weight values* | 100.0% | 0.0% | 0.0% | 100.0% | 0.0% | 0.0% |
| *INT8 weight values* | 0.0% | 0.0% | 100.0% | 0.0% | 100.0% | 0.0% |
| *INT16 weight values* | 0.0% | 100.0% | 0.0% | 0.0% | 0.0% | 100.0% |
| *Bias bits* | 16 | 16 | 16 | 8 | 16 | 32 |
| *% of worst latency case* | 12.5% | 50.0% | 50.0% | 6.25% | 25.0% | 100.0% |
| *INT4 activations* | 0 | 0 | 0 | 61 | 0 | 0 |
| *INT8 activations* | 61 | 61 | 0 | 0 | 61 | 0 |
| *INT16 activations* | 0 | 0 | 61 | 0 | 0 | 61 |
| *Input type* | INT8 | INT8 | INT16 | INT4 | INT8 | INT16 |
| *Output type* | INT8 | INT8 | INT16 | INT4 | INT8 | INT16 |

## 6.4 Proposed Method

The proposed method is next applied separately for both optimizing the memory consumption and compute latency of DeepRx. The top 3 models discovered by the search algorithm are saved, and then evaluated similarly as the baselines to ultimately choose the most suitable model for the use case. Following the propositions in Section 6.1, the hyperparameters were set to $\lambda = 10$, $\sigma = 0.10$, $\gamma = 0.95$, and $\mathbf{w}$ was set according to Equation 6.1 for both searches. Finally, the cost function MAPE threshold was set to $\epsilon_{i8}$ and a small subset of the whole data set was used to calculate the MAPE values for each configuration in both search cases.

First, the search was performed for optimizing memory consumption, where the input and activations were completely left out of the search and manually set to INT16. A total of 30 iterations were performed, which are illustrated in Figure 6.6. The total elapsed search time was approximately one hour without any GPU accelerated inference. The search smoothly converges to a good solution in approximately 10 epochs, and keeps sampling around the good solutions to find the first, second and third best cost values at epochs 13, 21, and 25 respectively. Additionally, the explored models shown in Figure 6.6 shows that as a result of the $\epsilon_{i8}$ threshold, most of the searches were done around the INT8 baseline configuration. The top 3 models from the search are summarized in Table 6.2, and evaluated in Figure 6.7. All of the models have lower memory consumption than static INT8 and 16x8 configurations, have lower latency than 16x8, and perform better than static INT8 but slightly worse than 16x8. An interesting observation is the amount of INT4 weight values used for all of the top 3 models which is close to 30 % for the best model and around 19% for second and third best models. Based on manual 8x4 mixed precision configuration, INT4 weights don't seem to be feasible at all, however, the memory optimized search results show that parts of the DeepRx weight arrays can indeed be quantized using INT4 precision.

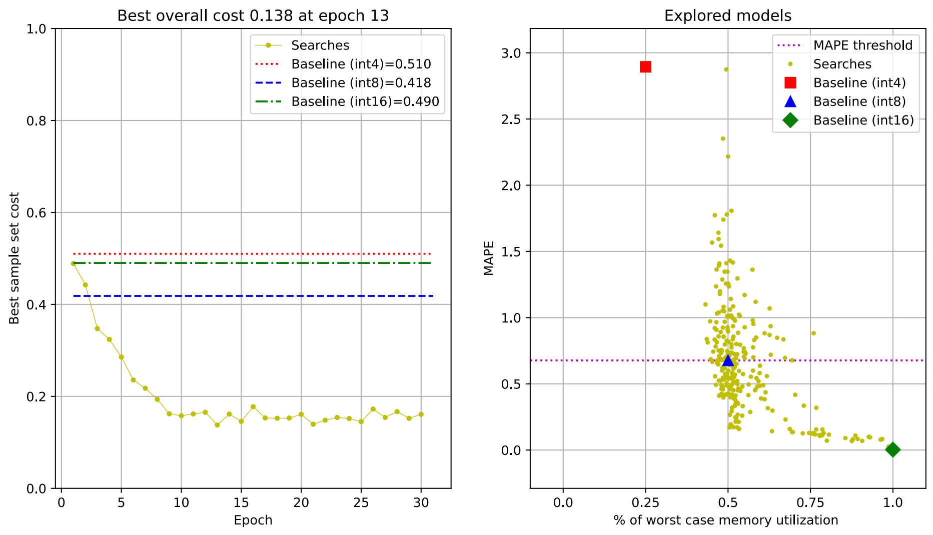
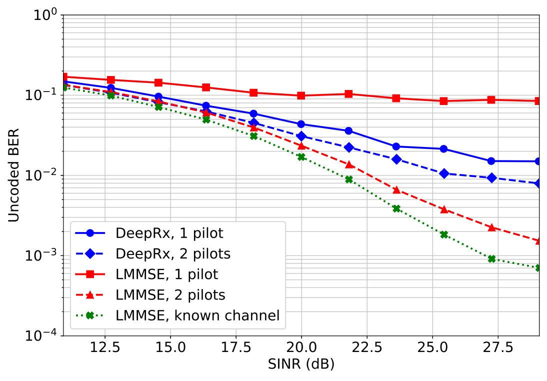
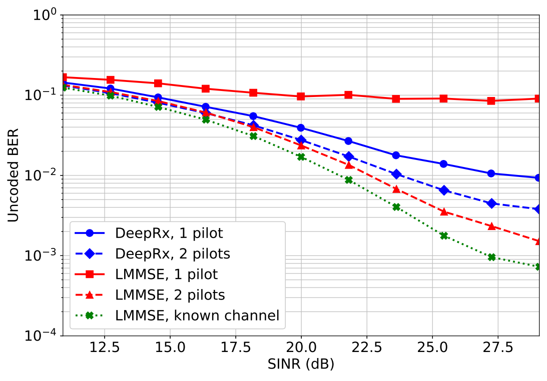


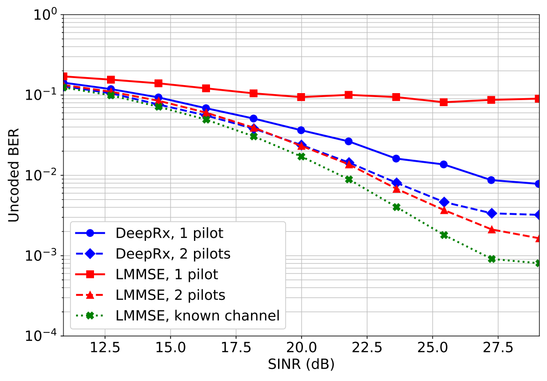
***Figure 6.6.*** *Memory optimized search*



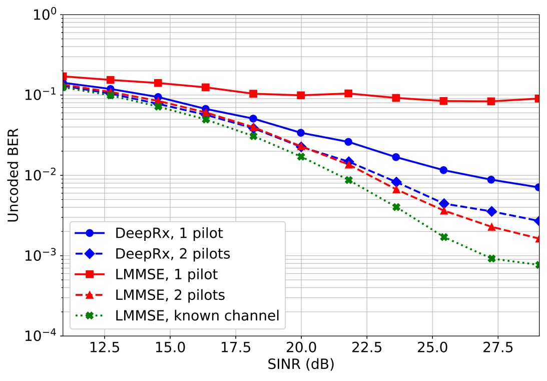
***(a)*** *Static INT8 quantization*



***(b)*** *Best search result*



***(c)*** *Second best search result*



***(d)*** *Third best search result*

***Figure 6.7.*** *Original and top 3 best models from memory optimized search*

***Table 6.2.*** *Memory optimized top 3 models and baselines*

| | **1st** | **2nd** | **3rd** | **Static INT4** | **Static INT8** | **Static INT16** |
|---|---|---|---|---|---|---|
| *MAPE* | 0.557 | 0.492 | 0.525 | 2.894 | 0.677 | 0.003 |
| *% of worst memory case* | 46.1% | 46.3% | 47.2% | 25.0% | 50.0% | 100.00% |
| *INT4 weight values* | 27.7% | 19.4% | 18.9% | 100.0% | 0.0% | 0.0% |
| *INT8 weight values* | 66.4% | 78.7% | 77.5% | 0.0% | 100.0% | 0.0% |
| *INT16 weight values* | 5.9% | 1.9% | 3.6% | 0.0% | 0.0% | 100.0% |
| *Bias bits* | 16 | 19 | 18 | 8 | 16 | 32 |
| *% of worst latency case* | 46.0% | 46.1% | 47.0% | 6.25% | 25.0% | 100.00% |
| *INT4 activations* | 0 | 0 | 0 | 61 | 0 | 0 |
| *INT8 activations* | 0 | 0 | 0 | 0 | 61 | 0 |
| *INT16 activations* | 61 | 61 | 61 | 0 | 0 | 61 |
| *Input type* | INT16 | INT16 | INT16 | INT4 | INT8 | INT16 |
| *Output type* | INT16 | INT16 | INT16 | INT4 | INT8 | INT16 |

Next, the search was performed for latency optimization, where the bias bits were left out of the search and set to a constant value of 32. Additionally, the input data type was excluded from the search and manually set to INT16 to ensure the precision is not already lost at start of the model. A total of 30 iterations were again performed, illustrated in Figure 6.8. Similarly as in the memory optimization search, the elapsed search time was approximately one hour without any GPU accelerated inference. In this case, the search does not converge as smoothly and exceeds the threshold limit especially in the first epochs, but is still able to discover good solutions. The first, second and third best cost values were found at epochs 29, 19 and 22 respectively. The visualization of explored models show that there can be very drastic changes in MAPE with models having similar latency, consequently making the latency optimization search much harder than memory optimization. The resulting top 3 latency optimized models are summarized in Table 6.3, and evaluated in Figure 6.9. The latency of all of the top 3 models is close but slightly higher than static INT8, however the MAPEs are superior in all of the top models. After evaluation, it can be seen that all of the top 3 models perform better than static INT8 configuration but still worse than 16x8, however, as mentioned the latency of the top models is very close to static INT8 and thus the models can be considered as good compromises between static INT8 and 16x8 configurations.

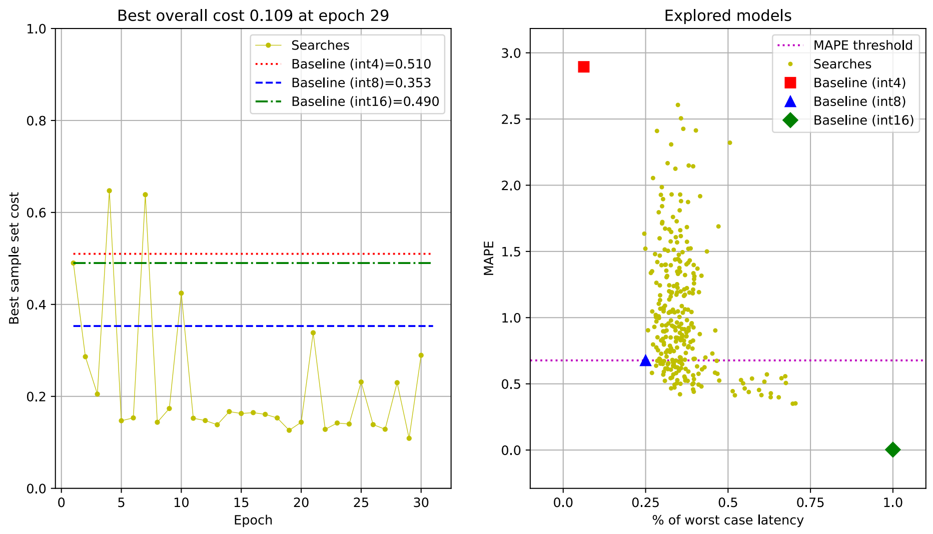
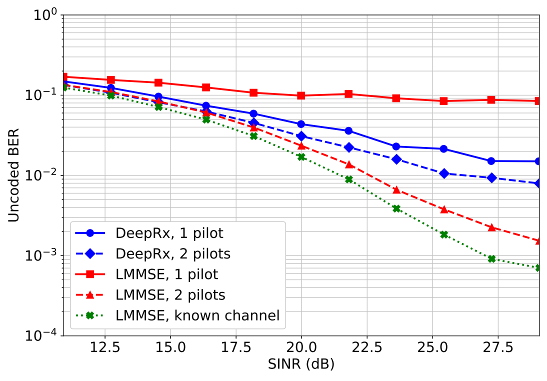
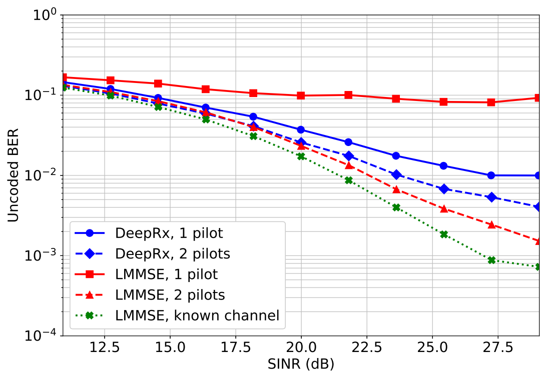


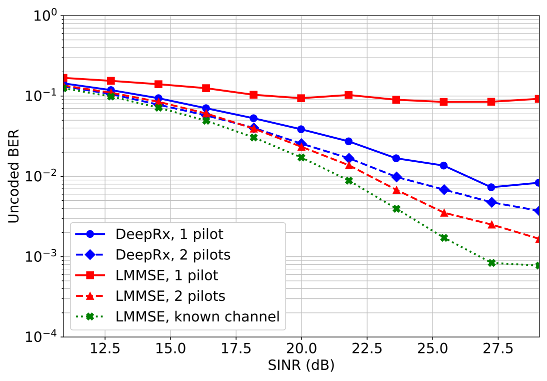
***Figure 6.8.*** *Latency optimized search*



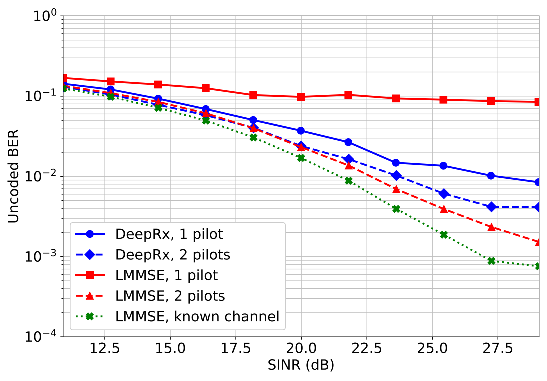
***(a)*** *Static INT8 quantization*



***(b)*** *Best search result*



***(c)*** *Second best search result*



***(d)*** *Third best search result*

***Figure 6.9.*** *Static INT8 and top 3 best models from latency optimized search*

***Table 6.3.*** *Latency optimized top 3 models and baselines*

| | **1st** | **2nd** | **3rd** | **Static INT4** | **Static INT8** | **Static INT16** |
|---|---|---|---|---|---|---|
| *MAPE* | 0.583 | 0.570 | 0.592 | 2.894 | 0.677 | 0.003 |
| *% of worst memory case* | 52.1% | 56.7% | 53.3% | 25.0% | 50.0% | 100.00% |
| *INT4 weight values* | 0.0% | 0.6% | 0.0% | 100.0% | 0.0% | 0.0% |
| *INT8 weight values* | 97.7% | 87.2% | 95.2% | 0.0% | 100.0% | 0.0% |
| *INT16 weight values* | 2.3% | 12.2% | 4.8% | 0.0% | 0.0% | 100.0% |
| *Bias bits* | 32 | 32 | 32 | 8 | 16 | 32 |
| *% of worst latency case* | 26.9% | 30.4% | 30.5% | 6.25% | 25.0% | 100.00% |
| *INT4 activations* | 0 | 0 | 0 | 61 | 0 | 0 |
| *INT8 activations* | 58 | 51 | 48 | 0 | 61 | 0 |
| *INT16 activations* | 3 | 10 | 13 | 0 | 0 | 61 |
| *Input type* | INT16 | INT16 | INT16 | INT4 | INT8 | INT16 |
| *Output type* | INT8 | INT16 | INT16 | INT4 | INT8 | INT16 |

# 7. CONCLUSION

In this thesis, the goal was to study mixed precision quantization of neural networks and the ONNX specification, and propose and evaluate a novel mixed precision quantization search method. After studying the current ONNX specification, an explicit list of changes required to the current version for it to fully support the mixed precision quantization scheme considered in this thesis was proposed in Section 5.1. These change proposals are yet to be officially forwarded to the ONNX community, but the plan of doing this is under preparation. The automatic mixed precision quantization search method proposed in Chapter 3 is a lightweight, gradient-free approach with an easily tunable cost function. The method is able to find multiple slightly different configurations optimized for the chosen target resource while keeping the error introduced due to quantization below the specified threshold. The proposed target resources, though not being perfectly accurate to the target hardware, function as intended and the resulting quantization configurations favor either smaller bit-width parameters or a combination of smaller parameter and intermediate quantization bit-widths. Furthermore, the hyperparameters for the search method were explored and suitable values for each proposed in Section 6.1.

The experimental results done in Sections 6.4 and 6.3 on a CNN-based radio receiver, DeepRx, show that significant optimizations are achievable by utilizing mixed precision quantization. From the manual mixed precision configurations, 16-bit activations and 8-bit weights is clearly the most suitable configuration, effectively being a compromise between static INT8 and INT16 configurations. The proposed search method was able to find 3 feasible mixed precision quantization configurations for both memory and latency optimization, and can be thus utilized to find quantization configurations in a more specific area of model output quantization error. Although DeepRx is a large network, both memory and latency optimization searches took approximately one hour without any GPU inference acceleration. This is significantly faster than e.g. the gradient-based architecture search in [5] which took several hours using multiple GPUs. The MAPE metric used for estimating the model accuracy drop due to quantization is generally usable in many models, however, in more complex models such as DeepRx where additional post-processing is needed, it does not directly capture the performance degradation of the model.

Although the experiments done in this demonstrate the functionality of the proposed search method, future work should evaluate the method more thoroughly on a wider set of different models. As the framework developed here is very specific to the quantization scheme, proposed modified ONNX specification, and the target hardware, other frameworks were not used to compare the search results. Therefore a rough comparison between different frameworks could be performed to estimate how well the proposed method performs in terms of resource optimization and search time when compared to other existing methods. This comparison should also include GPU inference acceleration for the proposed method which is currently absent. Moreover, as shown during the experiments, MAPE does not fully capture the accuracy degradation of complex models, and therefore a more model specific accuracy drop function could be used instead of the general MAPE metric when optimizing very use case specific neural networks. Future work could also expand the proposed method, e.g. by utilizing multivariate normal distribution sampling by utilizing a full covariance matrix, investigating dynamic sampling variance where variance is adjusted according to the current sampling center, or by exploring different mappings of the correlation efficients in the heuristic candidate as opposed to the current simple linear mapping. Finally, the whole mixed precision quantization scheme could be extended from kernel-wise to channel-wise quantization to allow even more freedom in the mixed precision configurations, and possibly optimize models even further.

# REFERENCES

[1] *TensorFlow Lite*. https://www.tensorflow.org/lite/. 2017.

[2] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[3] *ONNX: Open Neural Network Exchange*. https://onnx.ai/. 2019.

[4] Zhu, X., Zhou, W. and Li, H. Adaptive Layerwise Quantization for Deep Neural Network Compression. eng. *2018 IEEE International Conference on Multimedia and Expo (ICME)*. Vol. 2018-. IEEE, 2018, pp. 1–6. ISBN: 9781538617373.

[5] Wu, B., Wang, Y., Zhang, P., Tian, Y., Vajda, P. and Keutzer, K. Mixed Precision Quantization of ConvNets via Differentiable Neural Architecture Search. *CoRR* abs/1812.00090 (2018). arXiv: 1812.00090. URL: http://arxiv.org/abs/1812.00090.

[6] Coelho, C. N., Kuusela, A., Li, S., Zhuang, H., Ngadiuba, J., Aarrestad, T. K., Loncar, V., Pierini, M., Pol, A. A. and Summers, S. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. eng. *Nature machine intelligence* 3.8 (2021), pp. 675–686. ISSN: 2522-5839.

[7] Yang, L. and Jin, Q. FracBits: Mixed Precision Quantization via Fractional Bit-Widths. *CoRR* abs/2007.02017 (2020). arXiv: 2007.02017. URL: https://arxiv.org/abs/2007.02017.

[8] Lou, Q., Liu, L., Kim, M. and Jiang, L. AutoQB: AutoML for Network Quantization and Binarization on Mobile Devices. *CoRR* abs/1902.05690 (2019). arXiv: 1902.05690. URL: http://arxiv.org/abs/1902.05690.

[9] Wang, K., Liu, Z., Lin, Y., Lin, J. and Han, S. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.

[10] Yao, Z., Dong, Z., Zheng, Z., Gholami, A., Yu, J., Tan, E., Wang, L., Huang, Q., Wang, Y., Mahoney, M. and Keutzer, K. HAWQ-V3: Dyadic Neural Network Quantization. *Proceedings of the 38th International Conference on Machine Learning*. Ed. by M. Meila and T. Zhang. Vol. 139. Proceedings of Machine Learning Research.

PMLR, 18–24 Jul 2021, pp. 11875–11886. URL: https://proceedings.mlr.press/v139/yao21a.html.

[11] Schmidhuber, J. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), pp. 85–117. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0893608014002135.

[12] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[13] Xu, L., Choy, C.-S. and Li, Y.-W. Deep sparse rectifier neural networks for speech denoising. *2016 IEEE International Workshop on Acoustic Signal Enhancement (IWAENC)*. 2016, pp. 1–5. DOI: 10.1109/IWAENC.2016.7602891.

[14] Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics* 36 (Feb. 1980), pp. 193–202. DOI: 10.1007/BF00344251.

[15] Sifre, L. Rigid-Motion Scattering For Image Classification. PhD thesis. Ecole Polytechnique, CMAP, Oct. 2014.

[16] Chollet, F. Xception: Deep learning with depthwise separable convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1251–1258.

[17] Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International conference on machine learning*. PMLR. 2015, pp. 448–456.

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

[19] Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[20] Chellapilla, K., Puri, S. and Simard, P. High Performance Convolutional Neural Networks for Document Processing. *Tenth International Workshop on Frontiers in Handwriting Recognition*. Ed. by G. Lorette. http://www.suvisoft.com. Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006. URL: https://hal.inria.fr/inria-00112631.

[21] Ciresan, D. C., Meier, U., Gambardella, L. M. and Schmidhuber, J. Deep, Big, Simple Neural Nets for Handwritten Digit Recognition. eng. 22.12 (2010), pp. 3207–3220.

[22] Cireşan, D. C., Meier, U., Masci, J., Gambardella, L. M. and Schmidhuber, J. Flexible, High Performance Convolutional Neural Networks for Image Classification. *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelli-*

*gence - Volume Volume Two*. IJCAI'11. Barcelona, Catalonia, Spain: AAAI Press, 2011, pp. 1237–1242. ISBN: 9781577355144.

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

[24] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T. Caffe: Convolutional Architecture for Fast Feature Embedding. eng. *Proceedings of the 22nd ACM international conference on multimedia*. MM '14. ACM, 2014, pp. 675–678. ISBN: 1450330630.

[25] He, K., Zhang, X., Ren, S. and Sun, J. Deep Residual Learning for Image Recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[26] Honkala, M., Korpi, D. and Huttunen, J. M. J. DeepRx: Fully Convolutional Deep Learning Receiver. *IEEE Transactions on Wireless Communications* 20.6 (2021), pp. 3925–3940. DOI: `10.1109/TWC.2021.3054520`.

[27] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L.-C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[28] Gong, Y., Liu, L., Yang, M. and Bourdev, L. D. Compressing Deep Convolutional Networks using Vector Quantization. *CoRR* abs/1412.6115 (2014). arXiv: `1412.6115`. URL: `http://arxiv.org/abs/1412.6115`.

[29] Han, S., Mao, H. and Dally, W. J. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: `1510.00149 [cs.CV]`.

[30] Zhou, A., Yao, A., Guo, Y., Xu, L. and Chen, Y. *Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights*. 2017. arXiv: `1702.03044 [cs.CV]`.

[31] Kilgariff, E., Moreton, H., Stam, N. and Bell, B. Sept. 2018. URL: `https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/`.

[32] Salvator, D., Wu, H., Kulkarni, M. and Emmart, N. Oct. 2019. URL: `https://developer.nvidia.com/blog/int4-for-ai-inference/`.

[33] Han, T., Zhang, T., Li, D., Liu, G., Tian, L., Xie, D. and Shan, Y. *Convolutional Neural Network with INT4 Optimization on Xilinx Device*. Tech. rep. Xilinx, June 2010.

[34] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H. and Kalenichenko, D. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2704–2713. DOI: `10.1109/CVPR.2018.00286`.

[35] Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H. and Zou, Y. *DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients*. 2018. arXiv: `1606.06160 [cs.NE]`.

[36] Fan, A., Stock, P., Graham, B., Grave, E., Gribonval, R., Jégou, H. and Joulin, A. Training with Quantization Noise for Extreme Model Compression. *CoRR* abs/2004.07320 (2020). arXiv: `2004.07320`. URL: `https://arxiv.org/abs/2004.07320`.

[37] *TensorFlow: Quantization aware training*. `https://www.tensorflow.org/model_optimization/guide/quantization/training`.

[38] Blalock, D. W., Ortiz, J. J. G., Frankle, J. and Guttag, J. V. What is the State of Neural Network Pruning?: *CoRR* abs/2003.03033 (2020). arXiv: `2003.03033`. URL: `https://arxiv.org/abs/2003.03033`.

[39] Henri, K. *Quantization-aware pruning for a CNN-based radio receiver model*. eng. Informaatioteknologian ja viestinnän tiedekunta - Faculty of Information Technology and Communication Sciences, 2021.

[40] *ONNX Runtime*. `https://onnxruntime.ai/`. 2021.

[41] Donnelly, R. A. *Statistics*. eng. Indianapolis, Indiana, 2016.

[42] Weisstein, E. W. *Bessel's Correction.* From MathWorld–A Wolfram Web Resource. `https://mathworld.wolfram.com/BesselsCorrection.html`. 2017.

[43] Guglielmo, G. D., Duarte, J. M., Harris, P. C., Hoang, D., Jindariani, S., Kreinar, E., Liu, M., Loncar, V., Ngadiuba, J., Pedro, K., Pierini, M., Rankin, D., Sagear, S., Summers, S., Tran, N. and Wu, Z. Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml. eng. *Machine Learning: Science and Technology* 2.1 (2020). ISSN: 2632-2153.

[44] Park, J. H., Choi, J. S. and Ko, J. H. Dual-Precision Deep Neural Network. *Proceedings of the 2020 3rd International Conference on Artificial Intelligence and Pattern Recognition*. AIPR 2020. Xiamen, China: Association for Computing Machinery, 2020, pp. 30–34. ISBN: 9781450375511. DOI: `10.1145/3430199.3430228`. URL: `https://doi.org/10.1145/3430199.3430228`.

[45] Jin, Q., Yang, L. and Liao, Z. AdaBits: Neural Network Quantization With Adaptive Bit-Widths. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020.

[46] Yu, J., Yang, L., Xu, N., Yang, J. and Huang, T. S. Slimmable Neural Networks. *CoRR* abs/1812.08928 (2018). arXiv: `1812.08928`. URL: `http://arxiv.org/abs/1812.08928`.

[47] Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I.-J., Srinivasan, V. and Gopalakrishnan, K. PACT: Parameterized Clipping Activation for Quantized Neural Networks.

*CoRR* abs/1805.06085 (2018). arXiv: 1805.06085. URL: http://arxiv.org/abs/1805.06085.

[48] Naumov, M., Diril, U., Park, J., Ray, B., Jablonski, J. and Tulloch, A. *On Periodic Functions as Regularizers for Quantization of Neural Networks*. 2018. arXiv: 1811.09862 [cs.LG].

[49] Chollet, F. et al. *Keras*. https://keras.io. 2015.

[50] Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A. and Talwalkar, A. Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits. *CoRR* abs/1603.06560 (2016). arXiv: 1603.06560. URL: http://arxiv.org/abs/1603.06560.

[51] Bruin, B. d., Zivkovic, Z. and Corporaal, H. Quantization of deep neural networks for accumulator-constrained processors. eng. *Microprocessors and microsystems* 72 (2020), pp. 102872–. ISSN: 0141-9331.

[52] Ni, R., Chu, H.-M., Castañeda, O., Chiang, P.-Y., Studer, C. and Goldstein, T. WrapNet: Neural Net Inference with Ultra-Low-Resolution Arithmetic. *CoRR* abs/2007.13242 (2020). arXiv: 2007.13242. URL: https://arxiv.org/abs/2007.13242.

[53] Neumann, D., Wiese, T. and Utschick, W. Learning the MMSE Channel Estimator. *IEEE Transactions on Signal Processing* 66.11 (2018), pp. 2905–2917. DOI: 10.1109/TSP.2018.2799164.

[54] Chang, Z., Wang, Y., Li, H. and Wang, Z. Complex CNN-Based Equalization for Communication Signal. *2019 IEEE 4th International Conference on Signal and Image Processing (ICSIP)*. 2019, pp. 513–517. DOI: 10.1109/SIPROCESS.2019.8868708.

[55] He, H., Wen, C.-K., Jin, S. and Li, G. Y. Deep Learning-Based Channel Estimation for Beamspace mmWave Massive MIMO Systems. *IEEE Wireless Communications Letters* 7.5 (2018), pp. 852–855. DOI: 10.1109/LWC.2018.2832128.

[56] Chang, R. W. Synthesis of band-limited orthogonal signals for multichannel data transmission. eng. *Bell System Technical Journal* 45.10 (1966), pp. 1775–1796. ISSN: 0005-8580.