UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Reducing memory requirements for Neural Networks trained for Direction of Arrival Estimation

Making Neural Networks more practical
for use on a Low-Cost Real-Time Embedded System

Bram H.T. Wesselink

B.Sc. Thesis
June 2021

Supervisors:
prof. dr. ir. M.J.G. Bekooij
M.L. Lima de Oliveira MSc

Computer Architecture for
Embedded Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Preface

This thesis has been written as part of my bachelor's programme Electrical Engineering at the University of Twente. The research that we present in this document was conducted in the Computer Architecture for Embedded Systems (CAES) Group and is part of an ongoing and, at the time of writing this, unpublished PhD research by Marcio Lima de Oliveira.

I would like to thank my main supervisor Marco for providing me the main topic of my research, Marco has always provided very informative comments on my research and pointed out which parts of it are interesting to explore. Furthermore, I would like to thank Marcio for providing me a good introduction in Neural Networks and how they can be used for DoA estimation. Both Neural Networks and DoA estimation were completely new to me before starting this research, so this introduction was very valuable to me. Marcio has provided me python scripts that I have used as a basis for the c++ code I created for this research. Marcio also provided me the parameters of the neural networks he trained such that I could evaluate them for this research.

This research was fully conducted remotely and all communication took place digitally due to the COVID-19 pandemic. This posed little difficulty mainly due to everyone always attempting to communicate and respond in a timely fashion whenever necessary. I have had multiple great online meetings with Marco and Marcio in which I have been able to present my progress and in which they have always given me very useful constructive feedback and insights.

Finally I would like to thank the final member of my thesis committee Andrés Alayon Glasunov for his time and willingness to assess my thesis together with Marco and Marcio.

Abstract

The Direction of Arrival estimation problem is a common problem in sensor array signal processing. Many estimation algorithms exist, one such algorithm is the maximum likelihood estimator. MLE is a widely used Direction of Arrival estimator and is suitable for use with non-uniform radar antenna arrays. One of the main disadvantages of the MLE algorithm is the fact that its computational requirements scale exponentially with the number of targets to be estimated. In situations where more than two targets have to be estimated, MLE can be very expensive and not practical for use on a low-cost system. More recent research and experiments have shown that artificial neural networks can be used for the Direction of Arrival estimation.

The computational requirements of neural networks do not depend on the number of targets to estimate. This causes the networks to have lower required computational power than MLE when a large number of targets have to be estimated. One disadvantage of using neural networks is the fact that their often large topology requires a large number of parameters to be stored. These large memory requirements make them less practically applicable on a low-cost embedded radar system. We are unaware of any published research that mentions and addresses this disadvantage.

In this report, we have investigated the memory and computational requirements of different types of neural networks. We also have described techniques to reduce the large memory requirements of these neural networks. Using simulations, we show that the required memory of these neural networks can be reduced up to a factor of 6.8, without losing a significant amount of estimation performance.

These results were obtained using two types of neural networks, specifically a fully connected neural network (FCNN) and a residual neural network (ResNet). The reduction in memory usage was obtained by using smaller number formats than the standard IEEE-754 32-bit float. This change was done on a per-layer basis on an already trained network. After changing the number format, the network was evaluated without retraining to ensure that it had no significant impact on estimation performance.

Contents

List of acronyms

DoA	Direction of Arrival
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
FCNN	Fully Connected Neural Network
ResNet	Residual Neural Network
MLE	Maximum Likelihood Estimator

Introduction

1.1 Direction of Arrival estimation

In sensor array signal processing one common problem of interest is finding the Direction of Arrival (DoA) of a given signal using a radio antenna array [1]. Algorithms that attempt to find these angles are called DoA estimators. DoA estimation has recently become an active problem of interest due to the interest in mapping the 3-dimensional surroundings of an autonomous robot or vehicle [2]. DoA estimators can be used to both map the azimuth and elevation of obstacles with respect to the position of the radar [3].

1.2 Classical estimators

There is a lot of research on DoA estimators and there exist a lot of well performing estimation algorithms. In [1], an overview is provided of the most common classical estimators. Most of these algorithms can be divided into two categories, namely spectral based estimators and parametric based estimators. The former category takes the received power from the radio antennas and uses some mathematical function to generate a spectrum with peaks at the angles of interest. Parametric based estimators start with a model of the incoming signal using the angles of interest as unknown parameters in this model. They then try to find values for these parameters such that the difference between the observed input and the predicted input using the model is minimal [1].

1.2.1 Common estimators

One of the simplest spectral based estimators is the conventional beamformer. The idea behind this algorithm is to mathematically ”steer” the antenna array in different directions and measuring the power of the input signal in each direction. Then a spectrum is generated and the algorithm finds peaks located at the angles of interest [1].

Another common spectral based estimator is the Multiple SIgnal Classification (MUSIC) algorithm. This estimator uses properties from the eigen-structure of the covariance matrix to reduce the impacts of noise. The algorithm provides a good balance between computational complexity and estimation accuracy. Arbitrary accuracy can be obtained if data is collected for a long enough period [1].

A commonly used parametric based approach is the deterministic Maximum Likelihood Estimator (MLE). This algorithm assumes a deterministic model of the input data determined by the angles of N targets. To find the angles of interest, the model performs an N-dimensional search for all possible angles, within some range and using some resolution. The set of N angles that result in the closest match with the observed input are the most likely output of the estimator [1].

1.2.2 Shortcomings

One of the major shortcomings of a lot of classical estimators is the fact that they only have a good estimation performance when multiple snapshots of data are available. Not all applications (e.g. real-time radar systems) have the luxury of being able to take multiple snapshots before requiring an output of the estimator. It has been shown that the conventional beamformer and the MUSIC algorithm both require multiple snapshots to estimate the directions of arrival with acceptable accuracy [4]. MLE does not have this shortcoming [4], but it does have another significant disadvantage. The multi-dimensional search that MLE performs gets very expensive for an increased number of targets [1].

1.3 Artificial Neural Networks for DoA estimation

Recent experiments have shown that artificial neural networks can be used for DoA estimation even when only a single snapshot of data is available [5]. Like the classical estimators, neural networks can be implemented in two ways for the DoA estimation. The two types mainly differ in the way they output the angles. Firstly neural networks can directly output the angles as values in the last layer of the network [5]. Alternatively, the networks can generate some form of spectrum with peaks at the angles of interest, similar to spectral based estimators [6]. The major advantage of this spectrum-like output is that it does not directly impose a limit on the number of targets that can be estimated by the network. The maximum number of targets that can be estimated using this type of output is limited by the resolution of the generated spectrum. As shown in [5], the neural network is competitive to the ML estimator in terms of computational complexity.

1.3.1 Shortcomings

One of the major shortcomings of neural networks is their often large topologies. Plenty of research shows that large deep neural networks have better performance than smaller networks [7], [8]. A popular convolutional neural network (CNN) for image recognition, AlexNet, for example, has more than 60 million parameters [9]. When standard 32-bit numbers are used to store these parameters this network requires around 230 MiB of storage. The large memory requirements of good performing neural networks make them difficult to implement on a low-cost system.

As described in Section 1.5, these large memory requirements have been addressed for image recognition problems, however, we were unable to find any published research on these memory requirements for neural networks trained for DoA estimation.

1.4 Research question

In this report, we aim to address the large memory requirements of neural networks for DoA estimation. The aim of this research is to reduce the large memory requirements, without significantly affecting the estimation performance, of a neural network trained for DoA estimation, without retraining the network.

1.5 Related Work

Reducing the implementation cost of a neural network has been studied extensively. One technique is removing weights and/or neurons to reduce the number of parameters and operations required to evaluate the network. This technique is called pruning [10]. In [10], a compression rate of 4.1$\times$ for AlexNet is shown, and a compression of 13.6$\times$ for a network called VGGNet. Likewise, [11] were able to compress AlexNet with a factor of 9$\times$ and showed similar pruning compression statistics for other networks used in computer vision.

In [11], multiple compression techniques were combined to find reductions up to 49$\times$ in memory usage for neural networks used for image recognition. They found that the biggest savings came from the quantization of the parameters. Quantization is also the technique that is explored in this report for networks performing DoA estimation. In contrast to [11], we did not explore retraining the network after quantizing the parameters.

1.6 Structure of this report

In Chapter 2, some insights are given in the costs of a neural network with the operations performed by a neural network described in Section 2.1 and the storage requirements for the parameters described in Section 2.2. In Chapter 3, we describe the neural networks that were examined and we explain how we quantized the floating point numbers. In Section 3.3, the assumptions made during the research are described. In Chapter 4, we show the results of the research, and in Chapter 5, we point to some directions that should be explored in future research on this topic. Chapter 6 contains the conclusion of this report.

Theory

In this chapter, we provide an overview of the implementation costs of a neural network. In this chapter, both the operations and the storage requirements are described for both a 1-dimensional convolutional layer and a fully connected layer.

2.1 Operations in a neural network

2.1.1 convolutional layer

A convolutional layer is a layer in a neural network that uses convolution to produce an output. In the networks described in this report, only one-dimensional convolution is used.

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] \tag{2.1}$$

The 1-dimensional discrete convolution operation on input f using kernel g is shown in Equation 2.1. Due to the finite nature of the input of a layer in a neural network, the input is often padded in such a way that the output has the same size as the input, assuming a stride of 1. Assuming this form of padding and a stride of 1 the number of Multiply-ACcumulate (MAC) operations required for a single convolution is shown in the following equation.

$$\text{MAC}_{convolution} = NK \tag{2.2}$$

Where N is the size of input f and K is the size of kernel g.

The number of MAC operations in a convolutional layer can then be described by the following equation.

$$\text{MAC} = C_{in}\frac{N}{S}KC_{out} \tag{2.3}$$

Where C_{in} is the number of channels in the input, N is the size of the input, S is the stride of the convolution, K is the size of the filter used and C_{out} is the number of filters used.

2.1.2 fully connected layer

A fully connected layer is a layer in a neural network that simply connects each input of the layer to each output of the layer using a scalar weighting factor.

If the input has size N and the output has size M a total of

$$\text{MAC} = NM \tag{2.4}$$

MAC operations are required.

2.2 Storage requirements of a neural network

2.2.1 convolutional layer

In the convolutional layer, there are two types of parameters required, the kernels used in the convolution and a set of additive bias values. In a single convolutional layer with C_{in} input channels and C_{out} output channels, there are $C_{in} \times C_{out}$ convolutional kernels. Each kernel of size K contains K parameters. For each output channel, there is also a single bias value that is added to each value in that channel. This means that, on top of the storage requirements for the kernels, there are also C_{out} bias values.

So the total amount of parameters to be stored for each convolutional layer is given by Equation 2.5.

$$Parameters = C_{in}C_{out}K + C_{out} \tag{2.5}$$

2.2.2 fully connected layer

Similar to the convolutional layer a fully connected layer, also has two types of parameters, the weights and a set of bias values. In a layer with N input neurons and M output neurons, there are $N \times M$ weights and M biases.

The total number of parameters to be stored for a fully connected layer is given by Equation 2.6.

$$Parameters = NM + M \tag{2.6}$$

Experimental Setup

3.1 Examined neural networks

The neural networks examined in this research are similar to spectral based classical estimators (see Section 1.2). The networks take the input data from the antenna arrays and convert it into a spectrum-like output, where the peaks of angles of interest can be located. In this research, a model of 8 antennas is used, the locations of the antennas are shown in Figure 3.1. The complex input values for the network are split into their real and imaginary parts, the first 8 neurons in the input layer are the real values of the power and the last 8 neurons are the imaginary values. This gives a total input layer of 16 neurons.

All neural networks used in this research were created and trained in python using the Tensorflow library as part of an ongoing unpublished research. After the networks were trained and it was confirmed that they had satisfactory performance the parameters were exported to an HDF5 file. This file was then read using c++ and the parameters were imported into a network of similar structure as the one created in python.

3.1.1 Fully Connected Neural Network

The fully connected neural network we researched has 9 fully connected layers and 3 batch normalization steps. The biggest layer in the network is the 7th fully connected layer, as it has 1024 inputs and 2048 outputs. The full structure of the network is shown in Figure 3.2. Using Equation 2.6, we can see that the total number

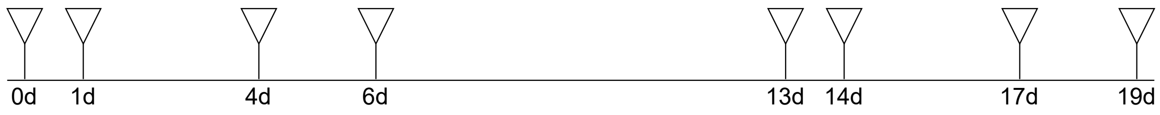

Figure 3.1: Locations of the antenna array

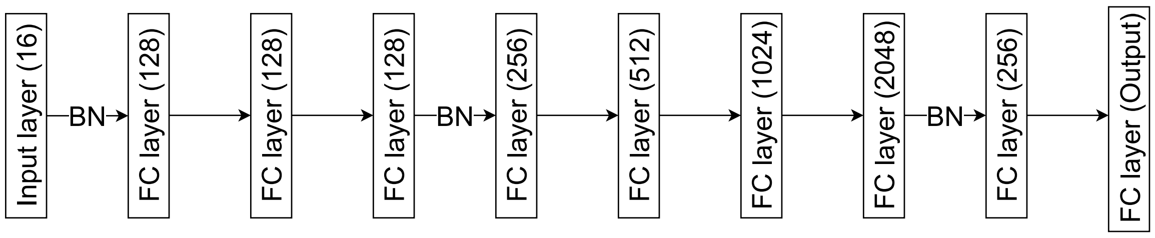

Figure 3.2: Fully connected neural network structure

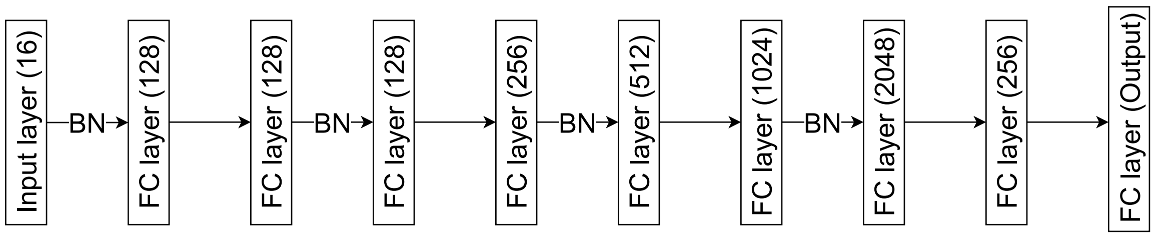

Figure 3.3: Fully connected neural network with an increased number of normalization layers

of parameters required for the fully connected layers in the network is 3,395,381 [1]. If 32 bit floating point numbers are used for these parameters $\frac{3{,}395{,}381 \times 4}{1024 \times 1024} \approx 12.95\,\text{MiB}$ of storage is required.

Normalization layers

In Figure 3.2 it can be seen that 3 batch normalization layers were used for the fully connected neural network. We wanted to explore if a network with more normalization layers reduced the dynamic range of the parameters of the network. To test this we also explored the network as shown in Figure 3.3.

3.1.2 Residual Neural Network

The Residual Neural Network (ResNet) we explored contains 12 1-dimensional convolutional layers and 3 fully connected layers. The main benefit of ResNets is that the forward skips of layers ease and improve trainability of deep neural networks [8]. The structure of the ResNet we examined is shown in Figure 3.4. Using Equation 2.5 it can be shown that the total number of parameters required for the convolutional layers in the network is 437,760. Using Equation 2.6 we can find that the fully con-

[1]There are more parameters in the network such as the ones used for batch normalization, these parameters are however not explored as discussed in Section 3.3

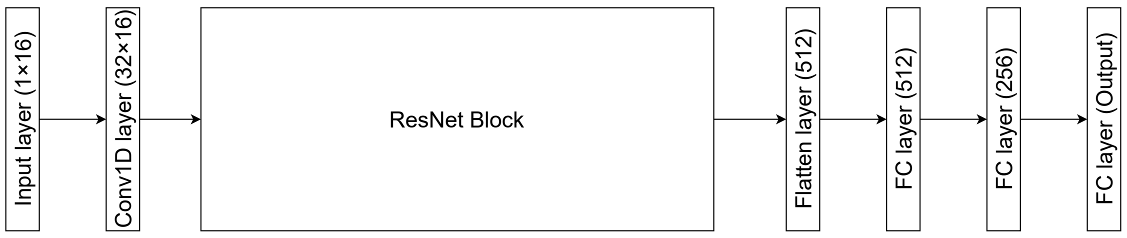

Figure 3.4: Residual neural network structure[2]

nected layers in this network require 440,501 parameters. Hence the ResNet has a total of 878,261 parameters in the convolutional and fully connected layers. Using 32 bit floating point numbers this requires $\frac{878{,}261\times4}{1024\times1024} \approx 3.35$ MiB of storage.

3.2 Arbitrary floating point number types

To explore the effects of different floating point number formats in C++ the FloatX library was used. This header only library adds a new templated floating point number type where the number of bits for the mantissa and exponent can be specified as template arguments. To easily perform linear algebra operations using these FloatX floating point numbers the Eigen library was used.

The FloatX library fully conforms to the IEEE-754 standard, hence all numbers used have special numbers like infinity, -infinity, NaN, and -NaN. The FloatX numbers also have denormalized numbers [12].

3.2.1 Infinity in neural networks

When reducing the number of bits in the exponent of the floating point numbers, overflow can occur. In the IEEE-754 standard, infinity and -infinity are used to indicate that a value is outside the range of the used number format. These infinities cause problems in a neural network. Whenever an infinity occurs, for example in a fully connected layer, all values in the next layer will either be infinity or -infinity. This happens because a scalar multiplied with infinity or -infinity will become either infinity or -infinity. Once all values in a layer are infinity or -infinity a situation like infinity + (-infinity) is likely to occur. The result of that expression is NaN and at that point, all information in the values of the network is lost.

[2]Because this neural network is part of unpublished research, the full architecture is not shown in this figure.

3.3 Assumptions

This section attempts to provide a complete list of assumptions made during the research. The first assumption that was made is that the signals from the sources have equal power. This is, of course, not very realistic but we suspect that it provides an acceptable model for testing the influences of compression on a neural network for DoA estimation. The second assumption is that the storage size for the biases in the convolutional and fully connected layers, is insignificant compared to the kernels and weights. Because of this, we have chosen not to compress these parameters. Similar to the biases, the parameters for the batch normalization layers have also not been studied and compressed in this research. It is also assumed that the Root Mean Square Error (RMSE) for 1000 samples and using 50 Monte-Carlo trials is a sufficient metric for determining if the network has any performance losses. See Appendix A for more details on the performance metrics used in this report.

3.4 Performed experiments

In this report, two techniques for reducing the memory requirements are explored. The first technique focuses on reducing the floating point numbers used for all calculations in the network. This is done by reducing the mantissa size for all calculations and evaluating the performance of the network.

The second technique focuses on compressing purely the weights and convolution kernels to reduce storage requirements.

3.4.1 Estimation performance using reduced mantissa size

To test how sensitive the neural networks are to reduced precision numbers the networks are evaluated with different sizes for the mantissa. Using the FloatX library the mantissas are swept from 1 bit to 23 bit in size. For every mantissa size, the RMSE is calculated using a set of 1000 randomly generated samples, and this RMSE is averaged over 50 Monte-Carlo trials (see Appendix A). This sweep is done for 15 dB SNR conditions and 30 dB SNR conditions. Both the 2 target and 3 target cases are evaluated. This RMSE is then plotted against the mantissa size using MATLAB to find a suitable lower limit for the mantissa size.

3.4.2 Compression of network parameters

To reduce the storage requirements of the neural networks, the kernels of the convolutional layers and the weights of the fully connected layers are quantized to smaller floating point number formats. For each layer in the network, the parameters are quantized in two sweeps, first by sweeping the exponent from 8 down to 1 bit, followed by a sweep of the mantissa from 23 down to 1 bit. For each parameter and quantization, the error introduced by quantizing is calculated using Equation 3.7.

$$E_{quantization,i} = \bar{x}_i - x_i \tag{3.7}$$

Where $E_{quantization,i}$ is the quantization error for the i^{th} parameter in that layer, $\bar{x}_i$ is the i^{th} quantized parameter and x_i is the i^{th} original, 32 bit parameter. After determining the quantization errors for each parameter and a given floating point format, a root mean square average for each layer is found using Equation 3.8.

$$\text{RMSE}_{quantization} = \sqrt{\frac{1}{N}\sum_{i=0}^{N} E^2_{quantization,i}} \tag{3.8}$$

Where N is the number of parameters in that layer.

After finding the $RMSE_{quantization}$ for each layer and quantization level, a threshold, for both the mantissa RMSE and the exponent RMSE, is used to find a suitable quantization level. The quantized parameters are then used in an evaluation of the network again using 1000 samples and 50 Monte-Carlo trials. If using the quantized parameters no significant loss in estimation performance is observed, then the threshold is increased to increase the compression levels. This process is repeated until a significant loss in estimation performance occurs.

3.4.3 Overflow and saturation

As discussed in Subsection 3.2.1, infinity, in a neural network, should be avoided. To achieve this, whenever it was found that a value would overflow to infinity, this value would be saturated. This saturation was done by finding the maximum finite possible value for the floating point number type using Equation 3.9.

$$\text{max} = 2^{2^{S_E-1}-1} \times \left(2 - 2^{-S_M}\right) \tag{3.9}$$

Where S_E is the number of bits in the exponent and S_M is the number of bits in the mantissa. The saturated value was multiplied with the sign of the original value and then used instead of infinity or -infinity. In the special case, where there is only one exponent bit, there are only denormalized numbers. In this case, Equation 3.10 was used.

$$\text{max}_{denormalized} = 2 \times (2 - 2^{-S_M} - 1) \tag{3.10}$$

Results

4.1 Estimation performance using reduced mantissa size

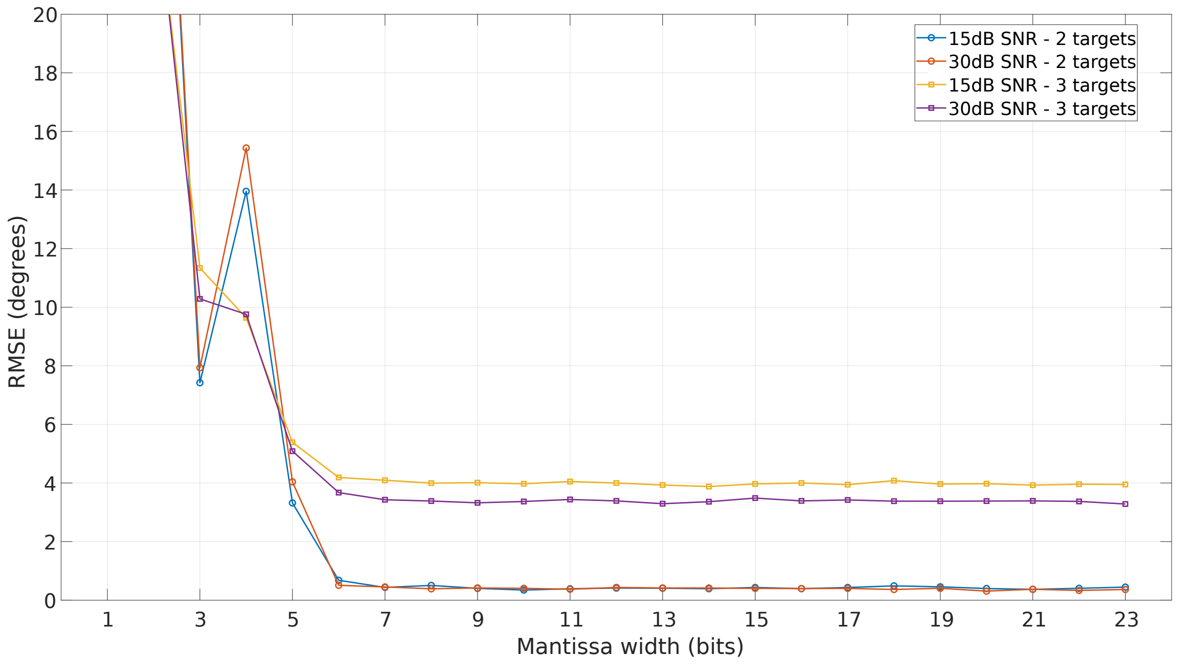

Figure 4.5: FCNN performance with reduced mantissa size

In Figure 4.5, the performance of the FCNN (shown in Figure 3.2) can be seen, in which reduced precision numbers are used. In this figure, it can be seen that when mantissas of 3 or fewer bits are used, the estimation performance is reduced significantly. In the region from 7 up to 23 bit, the performance is similar. For a 7 bit mantissa, on average, a performance loss of 6.15% is observed. Whereas for an 8 bit mantissa, a performance loss of 5.40% is observed and for a 9 bit mantissa 1.15% loss.

In Figure 4.6, the performance of the ResNet (shown in Figure 3.4) is shown. Compared to the FCNN, it can be seen that the ResNet is slightly more sensitive to quantization losses. For the ResNet, when using 7 bit for the mantissa in 30 dB

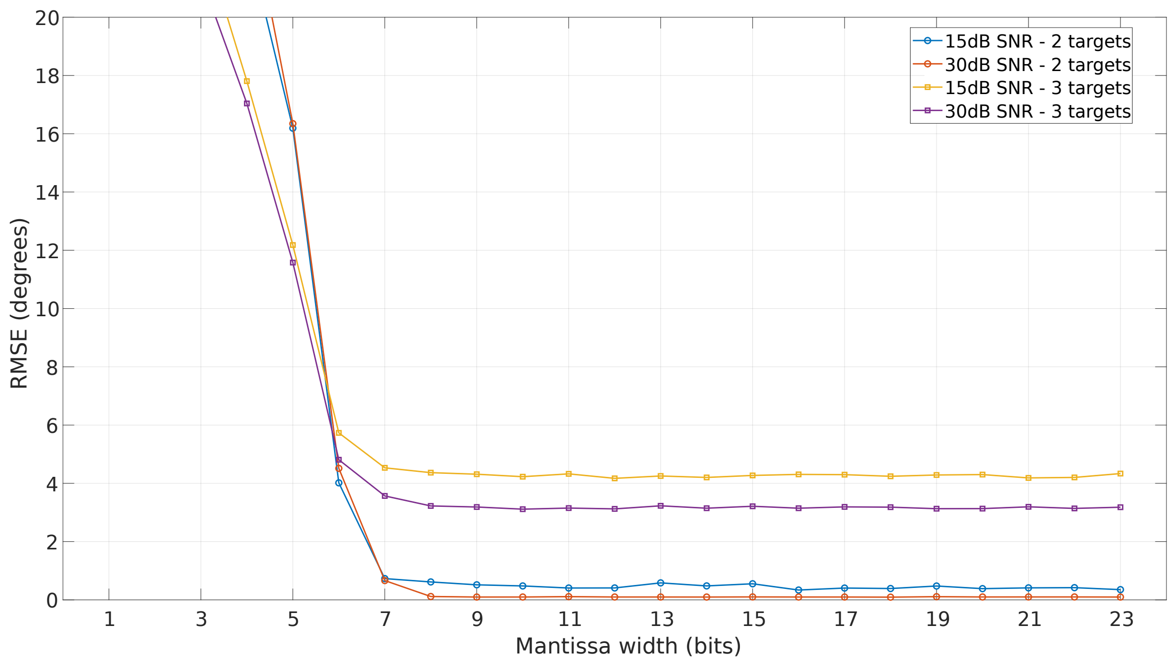

Figure 4.6: ResNet performance with reduced mantissa size

SNR conditions with two targets, the network performs 85.36% worse than when a standard 23 bit mantissa is used. On average, for 7 bit, the network performs 38.11% worse. For 8 bit this is 15.23% and for 9 bit it performs 7.98% worse.

4.2 Compression of network parameters

4.2.1 Compression of the FCNN

To compress the fully connected layers in the FCNN, the parameters were quantized, and for each layer, the Root Mean Square quantization error was found, as described in section 3.4.2. For the FCNN (shown in Figure 3.2) a threshold of 0.0001 was found for the quantization of the exponents and a threshold of 0.25 was found for the quantization of the mantissas. This resulted in a performance loss of 8.50%. Using these thresholds, the floating point formats (as shown in Table 4.1) are obtained. This resulted in a total compression factor of 5.47. Reducing the total memory required from 12.95 MiB to 2.37 MiB.

4.2.2 Compression of the FCNN with more normalization layers

In Table 4.2, the results for the FCNN (shown in Figure 3.3) can be seen. In this case, thresholds of 0.005 and 0.05 were found. This resulted in a performance loss of 6.63%. It can be seen that the normalization layers allowed the exponent

Table 4.1: FCNN compression details per layer

Layer	Format	Bits per parameter	Bits	Compression
FC_0	1-3-1	5	10,240	6.40
FC_1	1-4-2	7	114,688	4.57
FC_2	1-4-2	7	114,688	4.57
FC_3	1-4-1	6	196,608	5.33
FC_4	1-4-1	6	786,432	5.33
FC_5	1-4-1	6	3,145,728	5.33
FC_6	1-4-1	6	12,582,912	5.33
FC_7	1-3-1	5	2,621,440	6.40
FC_8	1-3-2	6	278,016	5.33
Total			**19,850,752**	**5.47**

Table 4.2: FCNN with more normalization compression details per layer

Layer	Format	Bits per parameter	Bits	Compression
FC_0	1-3-3	7	14,336	4.57
FC_1	1-4-4	9	147,456	3.55
FC_2	1-4-3	8	131,072	4.00
FC_3	1-4-3	8	262,144	4.00
FC_4	1-3-3	7	917,504	4.57
FC_5	1-3-2	6	3,145,728	5.33
FC_6	1-2-1	4	8,388,608	8.00
FC_7	1-3-1	5	2,621,440	6.40
FC_8	1-3-3	7	324,352	4.57
Total			**15,952,640**	**6.80**

to be lower in the large layers, resulting in an increase in compression, with a final compression factor of 6.8. The total memory requirement after compression for this network is 1.90 MiB.

4.2.3 Compression of the ResNet

Similar to the FCNN, the ResNet was compressed layer by layer. For the ResNet, both the parameters in the convolutional layers and the fully connected layers were quantized. For the ResNet (shown in Figure 3.4) a threshold of 0.2 was found for the exponent and a threshold of 0.05 was found for the mantissa. Notably, these thresholds are lower than the ones used for the fully connected neural networks. This means that the ResNet could be quantized less before showing performance loss. The final floating point formats are shown in Table 4.3. The ResNet reached

Table 4.3: ResNet compression details per layer

Layer	Format	Bits per parameter	Bits	Compression
Conv1D_0	1-3-5	9	864	3.56
Conv1D_1	1-4-4	9	27,648	3.56
Conv1D_2	1-3-3	7	21,504	4.57
Conv1D_3	1-3-4	8	49,152	4.00
Conv1D_4	1-3-4	8	98,304	4.00
Conv1D_5	1-4-4	9	18,432	3.56
Conv1D_6	1-3-4	8	196,608	4.00
Conv1D_7	1-3-4	8	393,216	4.00
Conv1D_8	1-3-4	8	65,536	4.00
Conv1D_9	1-3-4	8	786,432	4.00
Conv1D_10	1-3-4	8	1,572,864	4.00
Conv1D_11	1-3-4	8	262,144	4.00
FC_0	1-1-2	4	1,048,576	8.00
FC_1	1-3-2	6	786,432	5.33
FC_2	1-3-5	9	417,024	3.56
Total			**5,744,736**	**4.88**

a compression of 4.88 times resulting in a final memory requirement of 0.68 MiB compared to 3.35 MiB when using 32 bit floating point numbers. This compression resulted in a 5.45% performance loss.

Directions for future work

5.1 Retraining

All experiments that we have performed were done without retraining the network. To ensure no significant losses in estimation performance, we had to apply all techniques conservatively. Whenever we encountered a level of compression that significantly reduced the performance of the network we stopped compressing further. As explained in Section 1.5, other researchers have shown that this might not be necessary. We suspect that higher compression rates can be achieved, with a similarly small loss of estimation performance, if the network is fine-tuned after quantizing the parameters.

5.2 Pruning

To achieve higher levels of compression, it should be explored if every weight or neuron in the network is truly contributing to the final output. By removing weights close to 0 and storing these weights in efficient sparse data structures, we suspect that even more compression can be achieved [10], [11]. If this technique is applied before the quantization of the weights, higher levels of quantization can likely be achieved. This happens due to the fact that numbers close to zero have a large negative exponent, which means that, using the described compression technique, these numbers might have limited the levels of compression. Another major advantage of pruning is that it does not require any additional instructions, and if neurons in a network could be pruned, it would additionally reduce the number of instructions to be performed.

5.3 Weight sharing

Another technique that should be explored in the future is the idea of sharing one weight value for multiple connections in the network [11]. It is unclear, if the weights, in the neural networks for DoA estimation, are similar enough to enable this technique. However we suspect that, this technique combined with retraining the network after compression can give a significant increase in compression. One disadvantage of weight sharing is the fact that it requires a lookup process during the evaluation of the network. This will require additional instructions and thus slow down the estimator.

5.4 Huffman coding

As a last layer of compression, Huffman coding can be used. Since this is a lossless technique no loss in estimation performance should occur. After all the previous layers of compression, we do not suspect that Huffman coding will greatly increase the total compression level. However similar to the networks explored in [11], it might reduce the memory costs a little bit. One disadvantage of compression using Huffman coding is the fact that it requires a Huffman decoder.

5.5 Fixed point

To further reduce the memory and computational requirements of the neural networks, it might be possible to implement the networks using fixed point numbers. If the dynamic range of the parameters in the network is sufficiently small, storing them using fixed point number formats would be beneficial.

Conclusion

The main goal of this work was to explore different techniques to reduce the memory costs of a neural network trained for the Direction of Arrival (DoA) estimation problem. The fully connected neural network, that we examined in this report, required 3.4 million parameters. When using a 32 bit number format this would require around 13 MiB of storage. The residual neural network had 880,000 parameters, and when 32 bit numbers are used, this would require 3.4 MiB of storage. On a low-cost embedded system, this amount of memory might not be available, making these networks not feasible to be implemented without a significant reduction in memory requirements.

In this report, we have described techniques to reduce this large storage requirement. The coefficients of each layer have been quantized to smaller floating point types. This was done by reducing the size of the mantissa and exponent fields. By doing so, an overall compression factor of 6.8 was achieved for the FCNN with 4 normalization layers, while a compression factor of 5 was achieved for a FCNN with 3 normalization layers. The ResNet was also compressed by a factor of 5. From this, it can be concluded that extra normalization layers can increase the compression ratio using our method. All three networks showed no major loss in performance after this compression.

During this research, we have found that certain layers were more sensitive to quantization noise. We observed that the first couple of layers and the last layer of the network required more precision than the layers in the middle of the network. The large fully connected layers, near the end of both the FCNN and the ResNet, were quite robust to quantization noise, therefore, it was possible to reduce the number format for the parameters in these layers to less than 6 bit.

The final memory requirements, using the smaller floating point number formats, are 1.75 MiB for the FCNN and 0.68 MiB for the ResNet. These levels of compression, combined with the reduction in computation time required from using smaller floating point numbers, make these networks more practically feasible to be implemented on a low-cost embedded system.

We suspect, however, that these optimizations are just a small part of what might be possible. As discussed in section 1.5, researchers from other fields have shown much more impressive numbers regarding the compression of the parameters of a convolutional neural network. Besides using reduced size floating point number formats, they have also applied techniques like weight sharing, Huffman coding, and pruning.

For future research, we propose that these techniques are implemented on neural networks for the application of DoA estimation. Another potential limitation of this research is that our techniques were applied quite carefully and conservatively. We suspect that slightly retraining the network, after applying these techniques, could compensate for losses in the performance of the network. By doing so, our techniques might be applied more aggressively, thus further reducing the memory and computational requirements.

Bibliography

[1] H. Krim and M. Viberg, “Two decades of array signal processing research: the parametric approach,” *IEEE Signal Processing Magazine*, vol. 13, no. 4, pp. 67–94, 1996.

[2] S. Xu, J. Wang, and A. Yarovoy, “Super resolution doa for fmcw automotive radar imaging,” in *2018 IEEE Conference on Antenna Measurements Applications (CAMA)*, 2018, pp. 1–4.

[3] A. van der Veen, P. Ober, and E. Deprettere, “Azimuth and elevation computation in high resolution doa estimation,” *IEEE Transactions on Signal Processing*, vol. 40, no. 7, pp. 1828–1832, 1992.

[4] P. Häcker and B. Yang, “Single snapshot DOA estimation,” *Advances in Radio Science*, vol. 8, pp. 251–256, 10 2010.

[5] O. Bialer, N. Garnett, and T. Tirer, “Performance advantages of deep neural networks for Angle of Arrival estimation,” in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019, pp. 3907–3911.

[6] A. M. Elbir, “Deepmusic: Multiple signal classification via deep learning,” *IEEE Sensors Letters*, vol. 4, no. 4, pp. 1–4, 2020.

[7] G. Huang, Z. Liu, and K. Q. Weinberger, “Densely connected convolutional networks,” *CoRR*, vol. abs/1608.06993, 2016. [Online]. Available: http://arxiv.org/abs/1608.06993

[8] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS’12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[10] C. Jiang, G. Li, C. Qian, and K. Tang, “Efficient dnn neuron pruning by minimizing layer-wise nonlinear reconstruction error,” in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI’18. AAAI Press, 2018, p. 2298–2304.

[11] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding,” in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1510.00149

[12] G. Flegar, F. Scheidegger, V. Novaković, G. Mariani, A. E. Tom´s, A. C. I. Malossi, and E. S. Quintana-Ortí, “Floatx: A c++ library for customized floating-point arithmetic,” *ACM Trans. Math. Softw.*, vol. 45, no. 4, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3368086

Appendix A

Performance metric

To evaluate the performance of the neural networks used in this report a Root Mean Square Error (RMSE) of a 1000 input samples is evaluated, this is done 50 times and using those 50 RMSE values an average RMSE value is obtained.

The 1000 input samples are generated from a random set of 2 or 3 target integer angles and white Gaussian noise is added with a given Signal-Noise Ratio (SNR) condition. For each input sample the network is then evaluated and a peak detector function is applied on the output spectrum to find the estimated angles. These estimated angles are compared to the known input angles to find the error for the prediction of the network. Using these errors Equation A.1 is used to find the RMSE.

$$RMSE = \sqrt{\frac{1}{N}\sum_{n=0}^{N-1} E_n^2} \tag{A.1}$$

Where N is the number of angles evaluated and E_n is the error for the nth angle. As shown in Equation A.2 this process is repeated 50 times to find the final RMSE used for comparing the networks performance.

$$RMSE_{final} = \frac{1}{50}\sum_{n=0}^{49} RMSE_n \tag{A.2}$$