# Razvoj učinkovitog modela dubokog učenja za raspoznavanje govora

**Božić, Matej**

SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

DIPLOMSKI RAD br. 469

# RAZVOJ UČINKOVITOG MODELA DUBOKOG UČENJA ZA RASPOZNAVANJE GOVORA

Matej Božić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

DIPLOMSKI RAD br. 469

# RAZVOJ UČINKOVITOG MODELA DUBOKOG UČENJA ZA RASPOZNAVANJE GOVORA

Matej Božić

Zagreb, lipanj 2024.

**SVEUČILIŠTE U ZAGREBU**
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

Zagreb, 4. ožujka 2024.

# DIPLOMSKI ZADATAK br. 469

| | |
|---|---|
| Pristupnik: | **Matej Božić (0036525981)** |
| Studij: | Računarstvo |
| Profil: | Računarska znanost |
| Mentor: | doc. dr. sc. Marko Horvat |

Zadatak: **Razvoj učinkovitog modela dubokog učenja za raspoznavanje govora**

Opis zadatka:

Modeli dubokog učenja postali su temelj u razvoju naprednih algoritama za obradu govora, postižući izuzetne rezultate u točnosti i izražajnosti. Međutim, njihova složenost često ograničava primjenu u stvarnom vremenu gdje su brzina i učinkovitost ključni. Ovaj rad je usmjeren na istraživanje i razvoj novog modela dubokog učenja koji je optimiran za primjenu u raspoznavanju govora iz zvučnog signala s naglaskom na smanjenje složenosti modela. Cilj je razviti model koji nudi optimalan kompromis između brzine rada, broja parametara za učenje i performansi, uspoređujući s trenutno zastupljenim rješenjima u ovom području primjene. Posebna pažnja bit će posvećena kreiranju modela s manjim brojem učenih parametara koji zadržavaju visoku učinkovitost uz poboljšanu brzinu rada. Eksperimentalno usporediti performanse izrađenog modela i odabranih postojećih modela nad reprezentativnim skupom podataka. Vizualizirati rezultate usporedbe modela. Prikazati arhitekturu izrađenog modela i bitne isječke izvornog programskog koda uz potrebna dodatna objašnjenja i dokumentaciju. U diplomskom radu prikazati komponente izrađene programske podrške te definirati sva programska sredstva i potrebne postupke. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 28. lipnja 2024.

# Contents

# 1 Introduction

A study has shown that weights within a layer can be accurately predicted from only a small fraction of them, which indicates that deep learning models are highly over-parametrized [1]. The over-parametrization of deep learning models shows that there is a significant redundancy in those models [2]. Because of this, high accuracy and high resource consumption became the defining characteristics of deep learning [3]. Already in 2019, various deep learning models could be found in nearly any mobile device [4]. The main issues in deploying Deep Neural Networks (DNNs) on resource-limited platforms are high computational complexity and huge model storage requirements [5]. As an example, in the object classification task on the ImageNet dataset, the top-5 classification accuracy increased from 71% in 2012 to 97% in 2016, while the models became 20x computationally more expensive [6]. Even older image classification models, such as the 8-layer AlexNet, have over 60M parameters and require more than 729M FLOPs to classify a single image [7]. As the deep learning model's accuracy keeps rising, the amount of computation and storage requirements may become unbearable for mobile and embedded devices even during inference [8]. This will make usages, such as autonomous vehicles or video surveillance, impossible, as the model will not be able to perform the real-time decision-making required for such tasks. Especially because these tasks have to be done on-device due to privacy and reliability concerns [9].

Traditional deep learning workloads have widely used cloud-based solutions to handle mobile applications with constrained hardware resources [10]. DNN inference generally has high computational cost, and its execution on resource-constrained devices can result in prohibitively large processing delays. For us to be able to meet the computational requirements posed by deep learning models, a common approach is to utilize cloud computing [3]. In this configuration, the data is moved from the embedded device

to the cloud, which does the DNN inference and then sends the results back to the device. While cloud-based approaches were initially successful, many newer applications have privacy concerns because sensitive data is uploaded to publicly available servers. Additionally, cloud-based solutions are subject to communication overhead that depends on network conditions between the device and the used cloud server, resulting in high latency. Because of the large upkeep cost, cloud computing services are primarily provided by a limited number of large companies, as they are the only ones with resources to support such large systems. The edge computing paradigm has emerged as a solution to address the problems of cloud-based inference. It aims to bring computation close to the data source to reduce latency, bandwidth use, and power consumption. This approach uses edge devices, which have low power-consumption and high efficiency, which makes them promising for accelerated deep learning algorithms [11]. Edge devices could be installed in locations such as cellular base stations or Wi-Fi access points, devices that are close to the end-user. They provide computational capacity significantly lower than the cloud but higher than mobile devices. Therefore, there is a trade-off between communication and processing delay. To fulfill these computational demands, vendors have designed and launched low-power hardware accelerators for machine learning [11]. There are many benefits to performing computing locally on edge devices instead of relying on cloud computing solutions in terms of privacy, apprehension, and restricted connectivity [12]. Companies involved in information technology are strongly interested in running deep learning models at the edge to improve security and privacy while also improving delays experienced by end users [13]. There have been many proposed solutions to enable inference on edge, including model redesign, network pruning, parameter quantization, hardware acceleration based on parallel computing, and software acceleration focused on optimizing resource management and pipeline design [14]. In addition to performing computation on the edge, thanks to the advent of dedicated hardware accelerators, multi-core processors, and gigabytes of RAM, there is an emerging trend to perform inference locally on the mobile device [8]. Similarly to edge computing, it better protects user privacy, greatly reduces response time, communication costs, and improves scalability. Due to the benefits of on-device computation, it is the preferred mode for various applications, especially in real-time applications [8].

To benefit from edge computing, we need to reduce the costs of performing inference

on deep learning models. We will split the optimization approaches into two categories, depending on if the method changes the model output or not. The first category encompasses methods that have no cost in model accuracy. The advantage of these methods is that the user does not have to know the details of their implementation, as they can only change the speed of model inference but never their outputs. An example of such a method is developing and using application-dependent hardware or implementing optimized GPU kernels for faster algorithm execution. In contrast, the second category, called accuracy-altering, contains methods that have a possibility of changing model accuracy. These methods promise greater gains in model performance at the cost of possibly lowering model accuracy. Knowledge distillation is an example method; it involves creating a smaller DNN (student) that imitates the behaviour of a larger, more capable model (teacher) [3]. This is done by training the smaller model using the output predictions from the larger DNN.

In this work we will be focusing on only accuracy-altering methods that can be applied on a typical computer with an x86 CPU and an Nvidia GPU. We want our results to be easily reproducible, and therefore we will not use any special hardware such as FPGAs or ASICs. Because edge computing solutions usually only perform model inference and do the training in the cloud, our focus will be on optimizing inference time model accuracy and speed. Best-performing model architectures constantly change. For example, between 2010 and 2023, there have been more than 5 significant architecture changes in the audio generation domain [15]. Therefore, instead of doing architecture-specific optimizations, we will be focusing on generally applicable or layer-specific optimizations. The goal of this work isn't to design a new deep learning model optimized for speed, but instead, to apply methods that speed up an already existing model. By restricting the scope of utilized methods, we hope our research will be more broadly applicable.

In this work we will explore model compression techniques, which are general techniques that look at optimizing model architecture typically by compressing its layers [16]. We will explore three different model compression methods: pruning, quantization, and low-rank factorization. Chapter 2 will describe some of the DNN optimization methods that are not explored in this work; Chapter 3 will explain speech separation and why creating optimized deep learning models is important for the future; Chapter 4 will

introduce models used to test optimization methods and their respective architectures; Chapter 5 will explain the specifics of every optimization method and some of the research surrounding them; Finally, chapter 6 will describe the experimental setup and discuss results.

# 2 Related work

In [17], the execution of the deep learning model was partitioned between the edge and the cloud, where the first DNN layers were executed on the edge and the remaining were sent to the cloud. Therefore, the problem they faced was to determine how much computation to offload to the cloud. Because some deep learning models use non-invertible functions, it could possibly be used as a privacy-safe cloud-based method. But, in that case, the initial layer would have to be done on the edge, where more layers computed locally provide higher privacy. [18] is a similar work to ours that utilizes many software optimization techniques. While useful, it doesn't contain recent model compression techniques. Also in [19] a method called ITLUMM was used, where they replaced matrix multiplication with lookups. The sizes of the lookups determined the trade-off between accuracy and performance. Another approach attempts to run efficient DNN inference without affecting accuracy by determining at runtime which of the available models is the best based on input and evaluation criteria [20]. They construct a machine learning predictor that can dynamically select the optimal model for inference. While this approach will not make large models applicable to edge devices, it will reduce the latency of simpler tasks.

Another approach is the optimization of model architectures. In this category there are models such as ShuffleNet [21], YOLO [22], MobileNet [23], and SqueezeNet [24]. For example, YOLO object detection takes over two hundred milliseconds on most major mobile processors [25]. MobileNet is nearly as accurate as VGG16 but 32 times smaller and 27 times less computationally intensive, while SqueezeNet achieves a model size of less than 0.5 MB with 50 times fewer parameters than AlexNet but with the same accuracy [26].

Another approach is using inference compilers such as TensorRT, TensorFlow Lite,

Relay, and TVM, which optimize DL inference models for use in edge devices [14]. These compilers work with standard models defined in popular DL frameworks to accelerate model execution. TensorFlow Lite and TensorRT encompass most of the compiler optimization techniques that have been proposed for edge computing [14].

Another approach is developing specialized SoC chips for deep learning inference, such as Google's TPU or Intel's VPU, which can then be used with deep learning compilers [14]. For CPUs, at the instruction level, Intel has added AVX-512 Vector Neural Network Instruction (AVX-512 VNNI) to the AVX-512 instruction set to accelerate CNNs, along with support for Brain Floating Point (bfloat16) operations [14]. Nvidia combined Tensor cores with traditional CUDA cores in some platforms, where Tensor cores accelerate large matrix operations. They also added support for a new numerical format called Tensor Format (TF32) that provides 10x performance in A100 GPU architecture when compared to FP32 in V100 GPU architecture [27].

# 3 Problem

In this work, we focus on a specific task in deep learning: speech separation. In the broad sense, speech separation is focused on separating speech from the background interference. Remarkably, humans have the ability to extract one audio source from a mixture of sources. For example, when listening to a song, a trained ear can separate different instruments. Or, in an environment like a cocktail party, humans have the ability to extract one speaker from a group of people without much effort. Speech separation is commonly called the "cocktail party problem"[28]. It is a special case of sound source separation. Speech is the main method humans use to communicate; therefore, methods such as speech separation and speech enhancement become paramount for increasing the quality of human communication. We will focus on a special task within speech separation we will call speaker separation. The goal of this task is to take an audio sequence containing a mixture of people speaking at once and transform it into distinct sequences containing only one speaker where each separated audio will contain only one speaker identity throughout the sequence. In figure 3.1 we show a simple example of speaker separation task with only two speakers and no added noise.

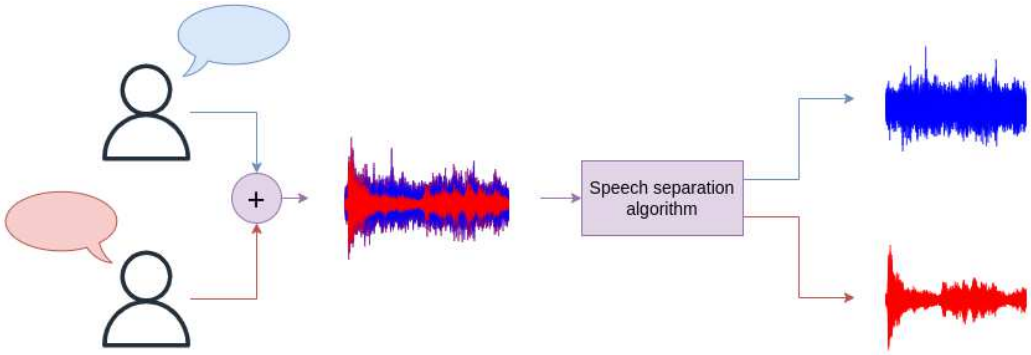


**Figure 3.1:** Speech separation problem with two speakers

While the trend in deep learning is to rely on over-parametrization [29] we want to

create portable models which can be used on low energy devices like hearing aids. It was shown that model over-parametrization aids in gradient descent convergence [29]. After model training, the over-parametrization is kept going into model inference, and depending on the percentage of redundancy, we can have models performing slowly and taking up a lot of memory for no increase in model accuracy. Therefore, instead of developing new low-power models, we will focus on transforming already successful speaker separation models by employing optimization methods. The methods we will introduce will reduce the number of model parameters, and because we cannot perfectly choose only the unimportant parameters, they will impact model performance. The goal is to find or transform a subset of parameters while keeping accuracy as close to the original as possible. For optimization methods, we wanted methods that were architecture-agnostic, meaning they could be used in most models without having to alter the model architecture. While some of the methods used change the layer architecture, from the outside perspective, viewing each layer as a black box, nothing changes. In other words, optimization methods do not change the required input or the resulting output data requirements of the optimized layer. These requirements were important to ensure the methods can be translated to other fields of deep learning and to allow people without the knowledge of the inner workings of these optimization methods to still be able to apply them to their models.

The next section will describe the architectures of the models used; after that, we will explore three different optimization methods, namely quantization, pruning, and low-rank factorization. Finally, we will conduct many experiments with different optimization hyperparameter configurations and compare them to the original models in terms of speed, memory, and quality.

# 4 Model architectures

When selecting model architectures, we used two criteria. The first criterion we used for choosing models is the openness of implementation. We want our results to be reproducible and our model architecture code to be open. For the second criterion we have unique architectures. Because we use architecture-agnostic optimization methods, we want to choose models with very different architectures so we can gauge how well the optimization methods will perform on a random model. Even though our methods are architecture-agnostic, they are expected to perform differently for different architectures, as methods, such as low-rank factorization, work differently based on the layer that the method is applied on. Based on these criteria, we have selected three unique speaker separation models. All of the selected models operate on the audio sequence in the time domain. The models are: LSTM-TasNet [30], Conv-TasNet [31], and DPT-Net [32].

The first model is LSTM-TasNet (LSTM Time-domain Audio Separation Network). Figure 4.1 shows a high-level view of the model. The model consists of three main parts: encoding, separation, and decoding. The encoding transforms the audio into an intermediate representation used by the separation module. The decoder is the reverse process returning the sequence back to the human intelligible audio. The main module, separation, contains two paths. The first path takes the intermediate representation and uses the recurrent and linear layers to estimate a mask. The second path takes the source audio in intermediate representation and multiplies it with the mask, thereby separating speech. The main hyperparameters of the model include the number of LSTM layers and the number of hidden units in the LSTM and linear layer. The main standout of this model is its use of the LSTM layer.

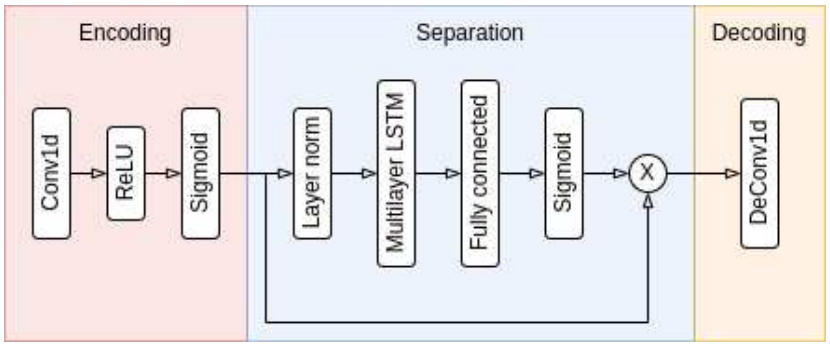


**Figure 4.1:** LSTM-TasNet model architecture

The second model is Conv-TasNet (Convolution Time-domain Audio Separation Network). Similar to the first model, it consists of three main modules: encoding, separation, and decoding. Figure 4.2 shows the different components of the model's architecture. The encoder and decoder have the job of converting to and from the intermediate representation of the model. In the separation module, the encoded input is multiplied by the estimated mask in order to get separated speech. The mask is estimated using a submodule that uses the encoded audio as input. The mask estimation submodule consists of stacked one-dimensional dilated convolutional blocks, which allows the network to model long-term dependencies of speech. The dilation factors of the stacked convolutional blocks increase exponentially with depth to achieve a large context window. Each block in the stack consists of one input and two outputs, where one of the outputs from the last block is ignored. They contain a sequence of convolution, PReLU, and normalization layers. Compared to the first model, here we use stacks of dilated convolutional blocks in order to replace LSTM layers.

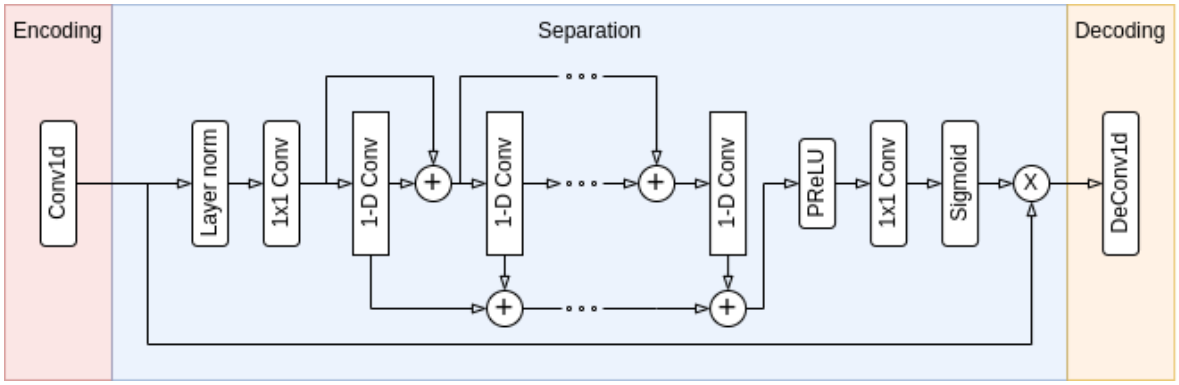


**Figure 4.2:** Conv-TasNet model architecture

DPT-Net (Dual-Path Transformer Network) is the final model we will be exploring

in this work. Like the previous two models, it consists of three main modules: encoder, separator, and decoder. Figure 4.3 shows the general architecture of the model. The role of the encoder and decoder is the same as with the previous two models. The outside path of the separation module is also the same; in other words, the outside path carries the encoded input to be multiplied by the estimated mask. The separation module consists of three stages: segmentation, dual-path transformer, and overlap-add [32]. The first stage splits the encoded input into overlapped chunks, which are then concatenated. The second stage consists of intra- and inter-transformers. The intra processing block models the local chunk independently, while the inter block is used to summarize the information from all chunks to learn global dependency [32]. In the final stage, the output of the last inter-transformer is transformed back into sequences by folding. After the masks have been estimated, they are additionally fed through a small gated network, which can only remove or silence values. Looking at 4.3, the first stage is the norm and unfold layers, the second is the transformer feedback/loop, and the final is up to and including the fold layer. The "transformers" used in this model are only the encoder part of the transformer, which is comprised of scaled dot-product attention, multi-head attention, and position-wise feed-forward network. Unlike in traditional transformers, this model does not use positional encodings and instead replaces the first fully connected layer with a recurrent neural network, as the positional encodings usually lead to model divergence [32]. While usually transformers deal with sequences with lengths of hundreds, in speech separation we have to model extremely long sequences, which is why the transformer utilizes the dual-path network as a solution. The author of the model mentions that RNN- and CNN-based-models cannot model long sequences as well as transformer-based models, as RNNs suffer from many intermediate states while CNNs suffer from limited receptive fields.

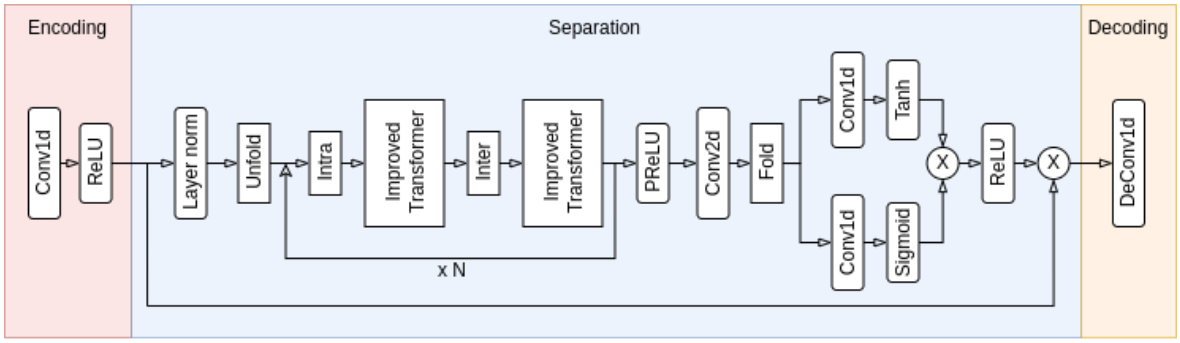


**Figure 4.3:** DPT-Net model architecture

## 4.1 Layers

We have described the high-level view of model architectures we use. As we deal with optimizing such architecture, we want to get more specific without delving into all the different hyperparameters of the model. The middle ground we chose is to display the number and type of layer we have in the models we use. In addition, we will show the amount of parameters, as depending on the depth of the model, the number of parameters in the same layer can change drastically. Table 4.1 shows the amount of layers in each architecture. From the table, we can see the numbers differ from the architecture images shown. This is because with images we wanted to display the barebones model, while here we are using an improved version with added layers that do not change the high-level architecture. Using a barebones model was possible, but we wanted to achieve better accuracy. With low-accuracy models, there is a lot of room for improvement, which could result in optimization methods improving accuracy when in reality they don't.

| Layer name | LSTM-TasNet | Conv-TasNet | DPTNet |
|---|---|---|---|
| Conv2d | 0 | 0 | 1 |
| Conv1d | 3 | 100 | 4 |
| LSTM | 4 | 0 | 4 |
| Multihead-attention | 0 | 0 | 4 |
| Linear | 1 | 0 | 4 |
| Layer norm | 1 | 49 | 9 |
| PReLU | 0 | 49 | 1 |

Table 4.1: Number of layers in architectures

While table 4.1 gives us a good overview of the models, when performing optimizations we will attempt to reduce the number of parameters while keeping accuracy. With this in mind, table 4.2 shows the number of parameters for the baseline models. We use the term baseline for models without any optimization methods applied to them. Looking at the table, we can see where the focus of each model is: LSTM-TasNet focuses the parameters on LSTM layers, Conv-TasNet focuses on Conv layers, while DPTNet spreads out to many different layers. It should be noted that one should not compare the number of parameters between the models, as the selected models do not perform the same. We chose the models not for the accuracy similarity but for the difference in architectures.

| Layer name | LSTM-TasNet | Conv-TasNet | DPTNet |
|---|---|---|---|
| Conv2d | 0 | 0 | 33,280 |
| Conv1d | 62,465 | 5,388,417 | 12,481 |
| LSTM | 22,080,000 | 0 | 2,637,824 |
| Multihead-attention | 0 | 0 | 66,560 |
| Linear | 4,100,096 | 0 | 131,328 |
| Layer norm | 1,024 | 50,176 | 1,152 |
| PReLU | 0 | 49 | 1 |
| Total | 26,243,585 | 5,438,642 | 2,882,626 |

Table 4.2: Number of parameters in architectures

# 5 Optimization methods

## 5.1 Quantization

In the general sense, quantization is the process of mapping a continuous or a larger set to a discrete smaller set by applying a function. This definition is not very useful when facing neural networks, as they contain many large sets and therefore have a lot of possibilities to choose from. Quantization in neural networks reduces the size of the model by decreasing the precision of values. While at first it doesn't seem much, the simple act of reducing model size gives rise to many benefits. Firstly, it reduces the amount of storage required from the hardware, not just static memory but also runtime memory like GPUs and CPUs. The reduction of required storage space in turn reduces the memory bandwidth, and by reducing the memory bandwidth we get faster loading and faster execution. One important concept of quantization is the quantization error. This error signifies how well the quantized values approximate the full-precision value. The main goal for quantization is to keep this error as close to zero while reducing the precision as much as possible. A simple way to calculate this error is to take the absolute difference of the approximated and real value. One study has shown that with quantization we can get networks that are 2 to 8 times more efficient than their full-precision counterparts [33].

In neural networks, there are three types of values we can quantize: the weights, the activations, and the gradients [34]. Quantizing weights, for most people, is what comes to mind when they hear DNN quantization, as they were initially the focus of research [35]. Quantizing weights has two immediate advantages. First, the model takes less working memory, thereby allowing GPU training or higher batch processing. Second, by changing from floating-point numbers to low-bit width numbers, we avoid costly floating-point multiplications, which can have a significant effect on performance [3].

Additionally, in case we quantize the weights only after the model training, we get additional benefits. As the computation is done only once, we can apply more computationally demanding quantization schemes, possibly giving better model accuracy. Also, in inference, we will not change the weights, meaning we can additionally fine-tune the quantized weights, providing even more accuracy. While the weights can be quantized after the training phase, the activations require quantization at each execution, thus creating computational overhead. In this case, while activation quantization may reduce memory requirements, depending on the quantization scheme, it may decrease performance. These kinds of quantizations are useful in low memory tasks, as activations can take several magnitudes more storage than weights for certain applications like speech separation. The added complexity of handling activation quantization makes it a secondary choice when considering model quantization. We will not consider quantizing the gradients because we are focused on inference while keeping most of the accuracy. Gradients are only ever seen in the training phase, and because they directly alter the accuracy of the model, if we reduce their precision, we can get worse accuracy, which will transfer to inference and impact our end results.

### 5.1.1 Number formats and quantization types

After selecting which layers require quantization, we face the problem of selecting the quantization number format. Floating-point has the advantage of keeping model parameters the same, thus ensuring the same accuracy; no special conversion or re-training is necessary. Using full-precision number format is only viable in simpler tasks, the disadvantages come when more advanced methods are employed. Many previously state-of-the-art deep learning models that work with high resolution images require more than 8 gigabytes of memory in addition to many layers of calculation [4]. These kinds of capabilities are unseen in the general public hardware, especially in mobile devices.

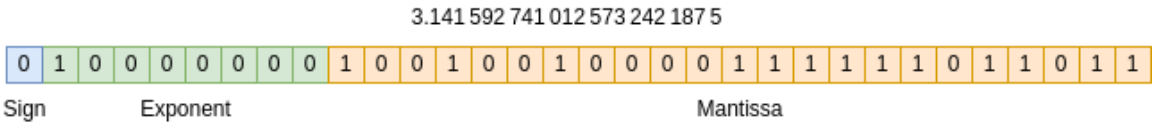


**Figure 5.1:** IEEE 754 floating point representation of $\pi$

The three types of low-precision formats are floating-point, fixed-point, and integer.

One type of the floating-point format is the IEEE 754 standard, shown in figure 5.1. In a 32-bit version (fp32), one bit will express the sign of the number, the next 8 bits represent the exponent, and the remaining 23 bits represent the mantissa. It can cover the range of $10^{-38}$ to $10^{38}$ [35]. The exponent can be considered as the range of the value, while the mantissa will determine how precise the value is. There are other types of floating-point numbers; brain floating point (bfloat16) takes the IEEE 754 standard but reduces the mantissa to 8 bits, while TensorFloat-32 (tf32) reduces the mantissa to 10 bits. The last type we will mention are posit numbers; they require fewer bits than fp32 for representing certain ranges, thus reducing storage requirements and increasing memory bandwidth [36]. In contrast to floating-point format, fixed-point and integer formats are extremely similar. The fixed-point format consists of a whole number multiplied by a scalar whose value is between zero and one. This way, the integer can be thought of as a special case of fixed-point format where the scaling factor is equal to one. The most popular representation of the fixed-point number is the two's complement. One advantage of the two's complement is that it doesn't contain two zeros but instead uses the second zero to represent an additional number, meaning it has $2^b$ unique values it can represent, where $b$ is the number of bits. While there are some differences, the fixed-point number can be thought of as a floating-point number with the exponent being constant and implicitly defined in the type. Exponent being constant means we can add its bits to the mantissa, making the number more precise. For example, if we wanted to handle currency, we could define a fixed-point type for money that had an implicit scaling factor of $10^{-2}$.

Low-precision formats offer several benefits. First, many hardware platforms support higher throughput math on low-bit formats, decreasing latency. Second, lower precision means lower memory requirements, allowing us to store larger models in memory-constrained devices while also speeding up execution by having more model parameters stored in cache. Third, the same input sizes in lower precision reduce memory bandwidth requirements, improving performance in cases of bandwidth-limited computation. For example, conversion of 16-bit floating point to 8-bit integer format reduces the size and memory consumption by a factor of 4 and potentially speeds up its execution by 2-3 times [4]. Since integer computation consumes less energy on many platforms, quantization also makes inference more power efficient, which is critical for battery-powered

devices like smartphones or IOT devices [4]. For example, MAC (Multiply-ACcumulate) operations performed on an 8-bit fixed point number lead to a power consumption reduction of 20x when compared to their floating-point counterparts [34].

Quantization has an inherent trade-off between the two data types. Floating-point models will always show better accuracy, while integer models enable faster inference [4]. Concrete differences between data types depend on the used hardware. AI accelerators for floating-point models are becoming faster and are reducing the differences between the speed of int-8 and fp16 inference. Thus the choice will depend on the particular task. Floating-point models excel when quality is necessary, while integer models are more beneficial in low-cost or low-power devices.

The act of reducing the bitwidth of the network weights almost always leads to accuracy loss; while in some cases the loss is minimal, in others it can be detrimental [4]. Quantization schemes are methods used to define how we convert between the full precision parameters and the quantized values. One possible way to split quantization schemes is into linear and non-linear. The key idea of linear quantization is to reduce the number of bits that represent each activation or weight, while the key idea of non-linear quantization is to divide weights into a few groups, and each group shares a single weight [8].

Linear quantization is characterized by evenly-spaced quantization intervals. An example of linear quantization is fixed-point coding. It has been widely studied and applied to neural networks because its hardware implementation is well known [34]. For example, the Nvidia Tesla GPU and Google Tensor Processing Units (TPUs) support 8-bit fixed-point operations. It has been demonstrated that both the weights and activations can be quantized to 8-bit dynamic fixed-point values without significantly affecting the accuracy [34]. Additionally, it was shown that dynamic fixed point was superior to floating point and fixed point formats in DNN training [8].

An example of linear quantization is the affine quantizer. The quantizer requires three values: minimum($x_{min}$), maximum($x_{max}$), and precision($n$). Equation 5.1 defines linear quantization. The round and clamp functions are the only lossy parts of the transform. It can be brought down to only one round if the minimum and maximum values

are well chosen. In the equation, $s$ represents the quantization step while $z$ represents the zero-point. The quantization step is based on the range of values and the set precision and can be calculated as $s = \frac{x_{max} - x_{min}}{n-1}$. The zero point is much more important as it represents the value of zero in the quantized space. Unlike other numbers, zero is used as an empty value, for example in padding, and can therefore introduce many more quantization errors. For this reason we need to make sure that the zero value is represented exactly the same when dequantizing it back: dequantize (quantize (0)) = 0. We can calculate the zero-point by taking the minimal quantized value and subtracting the scaled minimum: $z = \text{round}\left(x_{q_{min}} - \frac{x_{min}}{s}\right)$. When we have defined quantization, the reverse process is done by just reversing the order of operations while ignoring the lossy functions: $x_{dq} = s \cdot (x_q - z)$. One work recommends using scale quantization for quantizing weight and affine quantization for activations without having any performance penalty [37]. Another minor tweak can be performed on affine quantization, for certain number formats, to enable a substantial optimization opportunity by keeping the minimum and maximum values symmetric [38].

$$x_q = \text{round}\left(\frac{\text{clamp}\,(x, x_{min}, x_{max})}{s}\right) + z \tag{5.1}$$

On the other hand, non-linear quantization is recognized by having unevenly distributed quantization intervals. These types of quantization schemes can benefit NN as weights and activations usually have non-uniform distributions [34]. Having more values close to zero would be better since tensor distributions tend to be bell-shaped [39]. Unlike linear quantization, non-linear have less hardware support since they have to be carefully designed to be hardware-friendly [39]. One type of non-linear quantization is logarithmic quantization, which, when compared to linear quantization, incurs lower accuracy loss with the same bitwidth [34]. This type of quantization gives finer granularity for smaller magnitude values. By using base-2 logarithms we can get very efficient hardware implementations allowing us to replace multiplications with bit-wise shifts [39]. An example would be the VGG16 network. With linear quantization, the accuracy loss is 6.2%, while it's only 0.6% with logarithmic [34]. Another type of non-linear quantization is vector quantization. It consists of applying clustering algorithms to the weights of the NN, where the centroids of the clusters are used as the quantized

values. Weight sharing can be thought of as another form of non-linear quantization, as it forces multiple weights to contain a single value, thereby removing some of the precision. Unlike other forms of quantization, weight sharing does not reduce the precision of calculation, only the storage requirements. Non-linear quantization tends to be more computationally intensive, meaning it is usually applied in one-off quantizations like inference quantization.

We can apply equal quantization throughout the entire network (fixed) or choose to use different precisions for different layers (variable). The second method is also called mixed precision quantization (MPQ). Here the most important layers would be assigned higher precision while the less important would be in lower precision [40]. It was shown that bit-width used for weights can decrease approaching the last layers of the NN, while the bit-width of the activations remains approximately constant [34].

### 5.1.2 Quantization techniques

Quantization can be applied both at training and at inference time. At training time, applying quantization reduces the training time while possibly slowing down convergence or accuracy. Quantization at inference allows us to load large neural networks on memory-constrained devices while reducing the latency. Additionally, the networks can be fine-tuned after inference quantization to bring back more accuracy. For example, math-intensive tensor operations executed on 8-bit integer types can gain up to a 16x speed increase when compared to fp32, while in memory-limited operations the speed up can be up to 4x [37].

Post-training quantization (PQT) is where the weights and activations are quantized after the full-precision model training [41]. Before PQT, we need to determine the range for the to-be-quantized values. For weights, this is simple, as we have already trained the model and can learn these values by reading the weights. On the other hand, the range for activations can be determined from the training, or if using a pretrained model, we can run a few forward passes on the dataset. Even if we run the whole training dataset through, it still doesn't guarantee we found the real minimum and maximum, as it is impossible to get the inference-time dataset. It was shown that applying post-training 8-bit integer quantization usually leads to minor or no loss of accuracy [41]. Although

in more complex tasks it may cause noticeable accuracy degradation [39, 16]. These could be because the outliers stretch the range, making many possible values unused, or because, in layers like the convolutional, not all filters have the same distribution while utilizing the same quantization values, which may be more pronounced in low bitwidths due to the loss of precision [16]. These problems can be alleviated with the next method.

Quantization-aware training (QAT) emulates inference-time quantization during training. In this setting, the training happens in floating-point, but the forward pass simulates the quantization behavior during inference. Both weights and activations are passed through a function that simulates this quantization behavior [16], sometimes also referred to as fake-quantized. The intuition behind QAT is that while the forward pass will accumulate quantization errors, the optimization methods applied on the forward pass will attempt to reduce it with gradients that won't be quantized. This allows the network to adapt to tolerate the noise introduced by the clamping and rounding behavior during inference [16], leading to less performance degradation due to quantization. While the forward pass is well-defined, the backward pass can be done in many different ways since the quantization function is non-differentiable [8]. One common way is to estimate the gradient by pretending the quantization function is an identity function. At inference time, fake quantization operations are removed, and the network uses actual quantized weights and activations.

Unlike the previous two methods, another type of quantization called data-free quantization aims to reduce the bit-precision without access to training data. This type of quantization is used in privacy-sensitive applications like learning in the cloud. If the previously mentioned methods still cause too large of an accuracy drop, another option is partial quantization. This method only quantizes a few layers, which are mostly unaffected by quantization. Finding out which layers respond better to quantization can be done using sensitivity analysis.

Some examples of really low-precision models are ternary and binary weight networks, which use only three and two values respectively. In the extremes, we can utilize 1-bit weights and activations, which can achieve 32x compression of model size and replace expensive mathematical operations with fast XNOR operations [36]. Multipliers are one of the most power-consuming elements in DL processing. By removing them,

we also achieve energy consumption reduction, useful for battery-powered devices [36]. Binarization of both weights and activations yields extreme complexity reductions for DNN inference since MAC operations can be completely eliminated and replaced by binary operations [39]. Unlike other forms of quantization, binary networks do not only reduce the precision of operation but they radically change it. This means that the benefits derived from the usage of such networks are even higher than the intuitive 32x reduction in model size and memory bandwidth when compared to the floating-point alternative [39]. While there is a trend toward less and less precision, there is a floor. As precision gets lower than 1 byte or 8 bits, most CPUs will not be able to fetch these memory locations quickly as they will have to unpack the values. When we reach the stage of unpacking values, the only solution is to use specialized hardware to reap the benefits of low-precision models. Several accelerators were proposed to support the usage of binary NNs [34]. While binary weights can take on values of $-1$ and 1, it may prove beneficial to introduce a third value that allows for weights to be zero. Although this requires additional storage, the sparsity of weights can be exploited to reduce both storage and performance costs, thereby canceling out the cost of an additional bit [35].

## 5.2 Pruning

DNNs are usually over-parameterized to make training easier [35]. In extremes, DNNs can memorize the entire training dataset along with random patterns or get such parameters where 95% of the parameters can be predicted from the remaining 5% [29]. For example, the Inception-V3 network, a highly accurate object recognition model, requires 5.7 billion arithmetic operations and 27 million parameters to be evaluated, while GPT-3, a large language model, requires 175 billion parameters to be evaluated [29]. Because of their over-parametrization, they can tolerate sparsification [36, 41, 39, 42]. Studies have shown that most DNNs have redundancy in their weights, and therefore it is possible to sparsify them without affecting the accuracy [34]. Sparsification (or pruning), in simple terms, is the process of removing the least important parameters (by setting them to zero) while keeping most of the models accuracy. Typically, the weight parameters can be reduced by a factor of 10 using sparsification [43]. A study showed that the DNN model after sparsification could achieve up to a 25.6 times reduction in transmission workload, 6 times acceleration in total computation, and a 4.81 times reduction in latency when

compared to the original DNN model [26]. Additionally, studies have shown that DNNs can retain their performance after sparsification even if plenty of weights are removed [40]. We can reduce the sparsity of the model by the usage of sparsification algorithms. Sparsification algorithms rank parameters based on their contributions [44]. The goal of these algorithms is to find the set of least important weights in the model and to set them to zero. We can gauge the performance of these algorithms by inspecting the accuracy and the compression ratio achieved. One simple way of sparsifying the network is to choose weights that fall below a certain threshold. The intuition is that weights of small magnitudes have a negligible effect on results, and therefore their removal will not have a high accuracy penalty. The majority of sparsification algorithms operate after the initial training and then employ iterative sparsification while fine-tuning the model to recover the drop in accuracy [39]. This means most sparsification algorithms can be employed on pretrained models. Later we will see that sparsification can be applied for each weight or for a group of weights like the filters in the convolutional layer or the heads in multi-head attention. Figure 5.2 shows the ranking of model weights before sparsification. The weights in red are determined to have the lowest contribution to the model and are therefore removed (set to zero). Empirical evidence suggests that different layers should be treated differently and even the same layer types should be sparsified differently depending on their position in the network [29].

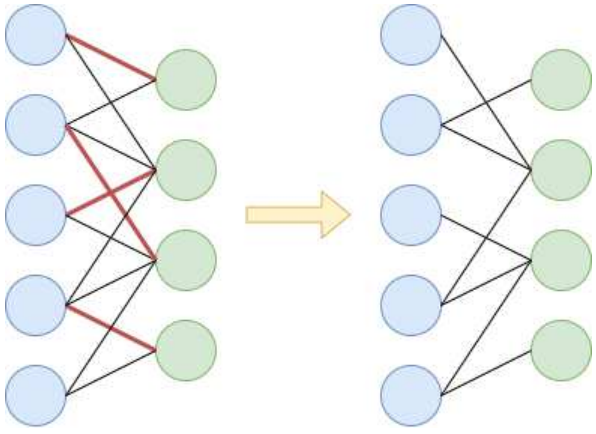

**Figure 5.2:** Sparsification of a sparse linear layer

Biological brains, especially in humans, are hierarchical, sparse, and recurrent structures. Sparsity plays a large role in brains, as the more neurons the brain has, the sparser it gets [29]. Over-parameterized models tend to overfit to the data and degrade general-

ization [29]. In this context, sparsification can be seen as a form of regularization, which can improve the model quality by reducing the noise in the model. In addition, with today's trend of bigger and bigger models, sparsification gives us a way to improve the explainability and interpretability of the model for sensitive use cases, such as in medicine. Similarily to quantization, the benefit of sparsification is the reduction of the amount of weights in the model, thereby reducing storage and computational requirements. But in addition to the model reduction, sparsification also improves the generalization and the robustness of the model [29]. When compared to quantization, the possibly achieved compression ratio is usually much higher in methods that allow for the alteration of the model architecture, such as sparsification [26]. From the hardware perspective, sparsification can help speed up basic operations. For instance, multiplying matrices will be much faster if they are sparse, as the multiplication with zero returns zero and therefore can be skipped. Additionally, because sparse structures by definition contain large portions of the same value, that value being zero, we can utilize compression methods, which would decrease memory storage requirements and increase memory bandwidth. An example of a simple compression we can utilize is where instead of directly storing values, we store the values and counts. The efficiency of such a method would grow with the level of data sparsity. More specifically, we can describe sparsification by defining a model as a function of input and weights $f(X, W)$ where $X$ is the input and $W$ are the weights. In this context, sparsification is a technique for estimating a minimal subset of weights $W' \in W$ where the missing weights are set to zero while also ensuring the accuracy of the model is minimally reduced [16]. Keeping up with the notation, we can define the compression ratio of the technique as the ratio between the original and pruned weights: $1 - \frac{|W'|}{|W|}$. Another way we can define sparsification is with a multiplication of weight with a mask. The mask is a tensor consisting of ones and zeros, and the sparsified weight is defined as the multiplication of the original weights $W$ and the mask $M$: $W' = W * M$. Additionally, in this configuration we can define the mask of multiple sparsification techniques as a chain of multiplications: $M = M_1 * M_2 * \cdots * M_N$. Using masks instead of just setting weight to zero allows us to rollback certain sparsification attempts if we find it degrades the accuracy too much. As we start to sparsify the network, the accuracy initially increases due to the reduction of learned noise. This phenomenon has been observed both in natural language processing and computer vision [29]. Intu-

itively, we can see that the smaller models form a kind of regularizer, forcing the learning algorithm to focus on a more general feature of the data. After the initial increase, the performance will remain stable or perhaps slightly decrease. At the end, we will see a large drop in accuracy as the sparsity is too high for the model to learn anything. In general, the performance increases with higher sparsities, where only for extreme sparsities will reach an area of diminishing returns, an area deep learning models have yet to reach [29].

In addition to weights, we can also sparsify model activations. In certain tasks, activations tend to consume more memory than model parameters, which makes them a high-priority target for sparsification. In addition to the memory savings, the sparsity of activations allows us to enable the hardware to allocate computation selectively [40]. This means hardware may skip certain arithmetic operations, or in extreme cases, skip network paths, leading to major improvements in model latency.

We can describe the process of model sparsification as a series of steps [29]. Depending on circumstances, some steps may be skipped while others will be iterated multiple times. The first step involves initializing network structure. We can define our own architecture or use a popular implementation. After creating the model architecture, we need to initialize model weights. This is usually done randomly according to a distribution that is appropriate for the certain layer or network. In case we wish to skip the process of training, we can use a pretrained model weights. This is preferable as we save a lot of time and energy if we can skip the training step. Note that this can only be done for post-training sparsification methods. If we don't use a pretrained model, we have to run training until the model converges. At this point we have an over-parametrized model, which we can start to sparsify. We run the model through the sparsification algorithm, and in case of deterioration, we run fine-tuning to "regrow" the weights back. Many works have shown that retraining immediately following each pruning step and also fine-tuning after the last pruning step are both crucial for well-performing sparsification schedules [29]. After we are satisfied with the level of pruning, we can do more fine-tuning to recover the most accuracy possible. This step is often skipped. Finally, we have two options: use the pruned module in the current state or go back to step two, reinitialize the weights, and start over.

During training, we can differentiate between static and dynamic sparsity. Dynamic sparsity uses pruning with regrowth of the parameter during the training process, while static sparsity prunes only once before the training starts and then does not update the model structure until the training is over [29]. Additionally, model spasity is often trained with a pruning scheduler. We can differentiate between three different classes of training schedules [29]. First schedule class is called train-then-sparsify. It is the most common schedule type. As the name suggests, we use a standard training procedure until model convergence. Following the training, we apply the sparsification algorithm to the fully-trained model. It is common for the model to be retrained after the sparsification, as there is no guarantee the optimal weights after sparsification are the same as the ones before. Compared to other schedules, it provides the best baseline performance for model quality as it allows for simple comparison between the original and the sparsified model. Another advantage of this schedule is that we can re-use existing model hyperparameters and learning methods as training is not modified. The second schedule class is sparsify-during-training. In contrast to the first method, this schedule is involved in the training phase of the model. It starts sparsification before the model reaches convergence and is therefore usually cheaper than a train-then-sparsify schedule. These schedules also include methods to correct for approximation errors from the training sparsification. At the start, they often train the model a few iterations before sparsification. Unlike the previous schedule class, at the end of training we end up with a sparsified model, meaning we don't have to waste additional iterations on fine-tuning. Also, because we are gradually sparsifying during training, the performance of the model increases, making the overall training time lower. Gradually sparsifying the model during training "reduces" the amount of parameters we have available to train, which could lead to less efficient convergence, and as the model changes during training, we cannot re-use hyperparameters. Final schedule class is called fully-sparse-training. Similarily to the previous class, it too removes parameters during training, but in addition, it can re-add other parameters. Many fully-sparse-training schedules add parameters while pruning to ensure the model stays approximately the same size. The process of finding parameters to add is similar to architecture search as it searches for all possible architectures. When we have to remove parameters, we can at least take into account the value of weights. The same isn't true for adding, as all their values are zero. This makes the task of adding weights

even more difficult than removing them.

We can split the pruning techniques into data-free and data-driven methods. Data-free pruning techniques do not consider the training data and can be useful in tasks that require privacy and security. They often require expensive retraining to recover the lost accuracy [29]. Data-driven methods consider the statistical sensitivity of the output with respect to the training data. By inspecting which weights have an approximately constant variation, meaning they don't change much for different training data inputs, we can determine which elements don't contribute much to the end output. The intuition is that if elements with radically different inputs show very little change in values computed, then they have no role in the network and can therefore be pruned. Because data-driven methods only consider the input-output behaviour of the network, it becomes useless in low-accuracy models as they will give wrong information on the importance of weights.

### 5.2.1 Pruning granularities

Another consideration when selecting parameters to remove is what granularity to choose. We will describe three types of granularities: fine-grained, pattern-based, and coarse-grained. Fine-grained pruning removes individual parameters, meaning it does not care about the structure of the layer to perform sparsification. This allows us to choose the parameter to remove arbitrarily. For example, it reduces the number of parameters of AlexNet by a factor of 9, while for VGG-19 by a factor of 13 with no loss of accuracy [41]. Pattern-based can be viewed a as special kind of fine-grained pruning that has the added benefit of better hardware acceleration with compiler optimizations. It works by assigning a set of fixed masks to each 3x3 kernel. The number of masks is usually limited to ensure hardware efficiency. Finally, coarse-grained pruning removes an entire tensor block for better hardware efficiency. Depending on the block size, we can remove entire vectors, kernels, or channels. Coarse-grained pruning can provide better hardware acceleration on popular GPU deep learning libraries. But this comes at a cost, as it usually achieves less accuracy than fine-grained pruning [41].

A more popular separation of pruning granularities is into structured and unstructured. Unstructured pruning removes the weights but preserves the neuron if at least on connection to that neuron exists [36]. Because unstructured pruning poses no restric-

tions on the structure, it contains irregular memory access that limits efficient hardware implementations and parallelism. It may lead to irregular structure, which cannot be accelerated directly, while also possibly giving rise to memory and cache access issue due to the non-structured connectivity [8]. To alleviate these issue structured pruning is introduced to obtain regular network connections. Structured pruning removes weights in a group-wise manner [40]. It works by constraining portions of the mask matrix to contain exactly the same number of non-zero weights. For example, in CNNs, we can perform structured pruning by removing entire convolutional filters. While CNN has an intuitive direction for structural pruning, other types of layers, for example recurrent, aren't as trivial and thus requires more elaborate approaches. It has many benefits, as the weights are removed structurally it aligns better with data-parallel architectures which results in more efficient processing. Additionally, it amortizes the overhead cost required to gather non-zero weights resulting in improved compression and reduced storage costs.

In terms of flexibility, unstructured pruning is unmatched as it poses no restriction on how the pruning should be done. We can view structured pruning as a subset of unstructured pruning meaning there may be cases where structured pruning is unable to match the accuracy of the unstructured because the optimal pruning method breaks restrictions. While we may get higher accuracy with unstructured methods, we will lose on inference-time performance. Because structured pruning removes "blocks" of weights it allows hardware to skip computation on large portions of the model decreasing latency and reducing the amount of stored activations. Note that unstructured pruning can be viewed as structured pruning with a block size of one [16]. Structured pruning is better for computationally restricted devices such as mobile and embedded devices. According to granularity we can decompose structured pruning into two groups: vector-/kernel-level and channel-/filter-level pruning [8]. Vector-/kernel-level pruning techniques focus on the vectors in the convolutional kernels or complete convolution kernels in a structured way. Channel-/filter-level pruning technique identifies the importance of channels and filters based on the assumption that the amount of information contributed by each channel can be evaluated by the channel's activation output variance. Unstructured sparsity requires storing the offsets of non-zero elements and handling the structure explicitly during processing. This amounts to a significant cost in processing and storing such networks. Structured sparsity solves this by constraining the sparsity

patterns in weights in such a way that they can be described with low-overhead representations such as strides or blocks [29]. This reduces the index storage overhead and simplifies processing. While structural pruning promises higher performance and lower storage overhead, it does so at the cost of limiting methods available, in other words by posing restrictions in order to increase performance it has reduced the possibility of finding an optimal method. One example of structured sparsity is the removal of whole neurons in fully-connected layers. One type of structured pruning called strided sparsity considers sparsification at the granularity of channels, kernels, or a strided kernel structure [29]. For example, we can define a stride-2 weight vector as [0.2, 1.9, 0, 1.3, 0, 0.3, 0, 1.2, 0, 0.4] where after the initial offset the rest of the elements are alternating between zero and non-zero. This means we would only have to remember the offset, stride and the non-zero elements [0.2, 1.9, 1.3, 0.3, 1.2, 0.4].

### 5.2.2 Pruning techniques

The selection of parameters to prune can be a difficult task. The most precise data-driven way to select parameters for removal is to evaluate all the possible subsets of model parameters. This means we would have to evaluate the performance of approximately $\binom{n}{k}$ networks, where $n$ is the number of network parameters and $k$ is the number of weights to be removed. This is unrealistic, especially for larger models, as we have seen the amount of parameters in those networks is in the orders of millions or even billions. Another method we can use is to select elements at random, which can be effective in certain settings [29].

Probably the simplest approach we can consider is to track the weight change during training. The intuition being that weights that changed the least during training are less important. The problem with this method is that it adds additional memory cost during training as it requires a separate storage of weight change for every weight in the network we wish to prune. While this may not be a problem for every network, for large models that have weights in the order of billions, it can make model training substantially slower.

One of the first published selection techniques was proposed in 1989 and was called optimal brain damage [35]. It was based on removing weights with the smallest saliency, where the weight saliency quantifies the weight's impact on the training loss [39]. The

weights with the smallest saliency, that is, the smallest impact on training loss, were removed, and the remaining model was fine-tuned. As models became larger, calculating weight saliency became more expensive. Today, in the era of deep learning, weight saliency has become too expensive and was replaced by magnitude-based pruning approaches [39]. In this approach, we assume the weights with the lowest magnitudes will have the least amount of impact on model accuracy and can therefore be pruned. After we remove the weight with the lowest magnitude, we also need to fine-tune the model to restore accuracy. It was shown that without fine-tuning, 50% of the weight could be pruned, while that number increases to 80% when we fine-tune [35]. It is often applied during training to maintain an approximately constant connection density during training [29]. It can also be extended to structured pruning, like kernel pruning, where the norm of the tensor is utilized as the criterion. Other criteria include second-order derivatives, loss-approximating Taylor expansion, and output sensitivity [41]. It was shown that magnitude-based sparsity can increase the speed of convergence [36]. To get effective magnitude-based pruning, we have to select the threshold carefully. Additionally, it may not be optimal to choose the same threshold for all layers, as their means and deviations can differ. It was found that weights in fully connected layers can be safely pruned with magnitude-based pruning having negligible impact on accuracy [39]. While magnitude pruning can be effective, several works have shown that more precise methods can achieve significantly better results, especially for high sparsity [29].

When our focus is on energy minimization, some works have shown that utilizing saliency- or magnitude-based pruning approaches does not always correspond to an optimal solution [39]. In such situations, there is a third kind of selection technique called energy-driven pruning or energy-aware pruning. With this technique, the impact of the weight is estimated from energy consumption. We can estimate the energy utilized by considering the number of MACs, data sparsity, and movement in the memory hierarchy [36]. For AlexNet, a popular computer vision model, it has demonstrated an increase of 1.74 in energy efficiency when compared to magnitude-based pruning [36].

While it may not be immediately obvious, regularization methods, such as L1-norm, are a form of a pruning algorithm as they drive the weights towards zero. We call such pruning methods regularization-based. Unlike previous methods, here we are not di-

rectly modifying the weights. Instead, we modify the loss function, which will in turn make the model sparse through training. Its biggest advantage is that it doesn't impact the weights directly and therefore provides better accuracy, but has the disadvantage of requiring a lot of iterations to allow for the model to converge [36]. This enables us to proactively find the optimal sparse pattern by forcing the unimportant weights towards zero [40]. With these methods, we are able to choose which weights to prune by changing which group of weights the regularization methods affect.

Most training algorithms for DNNs utilize gradient-based methods. This means throughout training we have the weight gradient information available without having to do additional computation. Therefore, we can utilize the available information in order to make our model sparse. We will call this method gradient-based pruning. It works by assigning importance to weights based on the value of their gradients. Additionally, we can combine this method with gating elements to select arbitrary elements for removal [29]. One approach we can take is similar to magnitude-base pruning, but instead of focusing on the weight's magnitude, we will focus on the weight gradient's magnitude.

We can also form pruning as an optimization problem where the goal is to minimize the amount of weight while keeping accuracy high. This allows us to use a generic optimization algorithms like the genetic algorithm. For the population, we can choose a random subset of model weights, in other words, a pruned version of the network where each is trained separately. Every network is rewarded based on the number of parameters and accuracy. Then we can make use of operators to create new networks and repeat the cycle until we are satisfied. An example of operators could be pruning a random weight for mutation, intersection for crossover, and tournament for selection. The problem we face with this algorithm is we will have a lot of wasted computation. Every time we remove a chromosome from the population, we throw away a fully trained model. When we also add multiple iterations, it becomes impossible for large DNNs to utilize this method.

### 5.2.3 Sparse storage formats

While pruning zeroes out weights, it does not remove them. This means that the pruned model will have the same amount of parameters and therefore the same memory usage.

While our models still have the same number of parameters, a large amount of those parameters are set to zero, which means we should be able to reduce the storage cost by changing the way weights are stored. However, not all compression formats are suitable for neural networks. For example, using complex compression algorithms will hurt our inference performance while also increasing energy consumption. This is due to the fact that for every forward pass, the weights have to be decompressed in order to use them. We may choose to decompress the weight only once before inference, but this defeats the purpose of optimized storage in the first place as we could just use the initial format. Therefore, compression algorithms for DNN inference have been proposed. They attempt to simultaneously provide a low-cost decompression algorithm while also providing a way to exploit sparsity for skipping computational paths [39]. With compression algorithms on sparse parameters, we can reduce memory access bandwidth by 20%–30% [35].

One simple format is the compressed sparse row (CSR). CSR uses three vectors, one to store the non-zero values of the network's weight matrix and two to recover their original location. The first indexing vector, called the row vector, describes which rows contain non-zero values. The second indexing vector, called the column vector, describes which column indexes in the current row contain non-zero values. The size of the row vector is fixed for the same matrix size, while the column vector and non-zero vector depend on the number of non-zero elements in the matrix. For instance, let's use a 4x4 matrix described in equation 5.2. The row vector is [0, 2, 2, 2, 4], the column vector is [0, 2, 1, 3], and the non-zero vector is [0.1, 0.3, 0.6, 0.9]. We interpret the row vector from left to right, where every two values describe the start and end index of the column/non-zero vector for each row. In addition, the end index of one row is the start index of the next row. For the current example, this means the elements of the first row start from index 0 up to index 2 of the column/non-zero vector. The second and third rows start and end in index 2, meaning they have no elements, and finally, fourth row starts at index 2 and ends at index 4 of the column/non-zero vector.

The compression may not seem as high for the current example matrix, but keep in mind that in practice the matrices are much larger and the number of elements in a matrix grows polynomially with the matrix size. The size of the row vector is always

$R + 1$ where $R$ is the number of rows, but for column and non-zero vectors the size is the number of non-zero elements in the matrix. We can easily conclude that for this compression format to be better than just storing the elements normally, our matrix has to have the sparsity of $S > 1 + \frac{R+1-RC}{2RC}$ where $R$ and $C$ denote the number of rows and columns respectively. For very large matrices, the $R + 1$ part will become negligible, and the sparsity required will approach 50%. CSR decoding is efficient for matrices in row-major order, meaning the consecutive elements in the row are next to each other. Every row can be accessed in constant time, and reconstructing the row is linear in the number of non-zero elements. The problem of this format occurs when trying to skip computations related to zero-weights, in particular when accessing the activation's tensor with which the compressed matrix is multiplied [39]. Since the non-zero elements in the CSR matrix are stored in rows, the activations corresponding to the element have to be accessed multiple times. Alternatively, we can store the decompressed vector in memory, but this might not be an option in memory-constrained devices.

CSR focused on the rows in the matrix. Similarly, we have the compressed sparse column format (CSC) which uses the opposite approach, swapping the roles of the row and column vector. This change allows the matrix to be read by columns while performing multiplications, which removes the problem of multiple accesses to input activations [39]. We can see this is because in matrix-vector product, each matrix column is multiplied with the same input element, which guarantees that the matrix will be read once and in order. While this fixes the problem of multiplication with the input, it creates the same problem for the output activations, which again have to be accessed multiple times or stored in memory during computation. This might not be as big of a problem as the output size is usually smaller than the input size in deep learning models [39].

CSR and CSC belong to the class of compressed stripe storage [34]. Both CSR and CSC are flexible compression formats, meaning they can be used for unstructured pruning algorithms where the position of non-zero elements is arbitrary. Without restrictions on the structure, it prevents hardware for optimizing computation as there is no possibility of exploiting parallelism. Because every multiplication with a row or column requires a different number of operations based on the number of non-zero elements, it prevents hardware from parallelly executing such operations due to the ever-changing position of

non-zero elements. The hardware can still skip some of the computation once it detects the whole rows or columns are filled with zeros, but this does not fully exploit parameter sparsity as we cannot predict how many rows or columns will be sparse. In the worst-case scenario, we can have a highly sparse matrix, such as a diagonal matrix, where none of the rows or columns are fully non-zero, preventing us from skipping computation in a parallel manner.

One example of structured pruning compressed formats is the compressed sparse banks (CSB) format. Unlike CSR/CSC, this format uses only two vectors. The first one stores the non-zero values, and the second stores their column indexes. This means we have effectively removed the row vectors. Now the final size after compression only depends on the number of non-zero elements. While this is a big improvement, this format requires a specific configuration of the matrix. This means we cannot use just any pruning algorithm with this format. The pruning algorithm used with this storage format is called bank-balanced pruning. In it, we remove the same number of elements in each group called a "bank". Going back to our example matrix defined in equation 5.2, it has been pruned in a compatible format with banks of $2x4$, meaning each group of 2 rows and 4 columns has the same number of elements. Because we are using banks, the row vector can be inferred, and now our matrix can be stored with [0.1, 0.3, 0.6, 0.9] as the non-zero vector and [0, 2, 1, 3] as the column vector. When we decode this format, we are going row by row, starting from the top, and because each bank is guaranteed to have the same number of elements, we always know which column index corresponds to the row element. Under the assumption that the entire activation vector can be split into the same banks, it will allow us to perform interbank parallelization where weights from different banks are simultaneously multiplied [39].

The simplest storage scheme is called the bitmap (BM) scheme, also called compressed image size (CIS). It is so called because it stores a map of N bits, where $N$ is the number of elements, including zeros. Every bit signifies whether the element is zero or non-zero. It is efficient for denser formats as the storage cost is proportional to the total number of elements. Another simple storage format is the coordinate offset (COO). It stores each non-zero element along with its absolute offset, which depends on how the tensor is stored. Unlike the previous format, this format is more efficient for very

sparse structures as its storage cost is proportional to the number of non-zero elements. An extension of this format is called run length encoding (RLE), where only the difference between two element offsets are stored. Using offset differences may be preferable for larger tensors, as we can use integer formats with fewer bits. Additionally, if the differences between offsets are too large, we can insert zero elements in the non-zero vector between the largest offset differences in order to reduce them. Note that in some works, RLE can mean a different algorithm. We will call this algorithm run length coding (RLC), and while it can be used to store sparse tensors, it does so by storing zeroes too. It works by storing value and its repetition count. This means it favors tensors with the least amount of unique elements, especially if they are consecutive. Following our example matrix in equation 5.2 all the schemes have the same vector of non-zero elements [0.1, 0.3, 0.6, 0.9], and assuming row-major order, BM scheme would store [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.6, 0, 0.9], COO would store [0, 2, 13, 15] and RLE would store [0, 2, 11, 2]. If we added a zero in the RLE, the "non-zero" vector would change into [0.1, 0.3, 0, 0.6, 0.9] and the offset differences vector would become [0, 2, 6, 5, 2]. This way we can represent the elements of the vector with only 3 bits, saving us 1 bit on each element. With RLC, we do not have a vector of non-zero values but instead get one vector of unique consecutive values [0.1, 0, 0.3, 0, 0.6, 0, 0.9] and another of their repetition [1, 1, 1, 10, 1, 1, 1].

$$\begin{bmatrix} 0.1 & 0 & 0.3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0.6 & 0 & 0.9 \end{bmatrix} \tag{5.2}$$

An example of a sparse matrix

While we have introduced many sparse-specific storage formats, there is a question of why we aren't using any of the already established compression methods. Huffman coding is the most efficient method to encode scattered data due to its optimal compression rate [34]. It can be used to gain even higher compression ratios than the previous method, but it does so by using computation-heavy encoding and decoding schemes.

Therefore, these kinds of methods are only ever useful for long-term storage, as they would introduce a large overhead in each operation where a sparse tensor is involved.

## 5.3 Low-rank factorization

Low-rank factorization, or specifically, tensor decomposition technique, attempts to find an approximate low-rank tensor that is close to the original tensor. Low-rank factorization methods have long been used to accelerate and compress neural networks. The two most popular tensor decomposition models are CANDECOMP/PARAFAC (CP) and the Tucker model [45]. The technique is often applied to reduce the complexity of fully-connected or convolutional layers in DNNs. It has been long investigated in the field of signal processing in order to accelerate convolution [41]. When we use decomposition methods on a network's layer, we are breaking apart that layer into many layers, which will attempt to approximate the output of the original layer while also having a reduced number of parameters. Some works have shown that factorization can reduce the model size up to 75% without a decrease in accuracy [41].

The most widely used matrix decomposition is called Truncated Singular Value Decomposition (TSVD). It is effective for speeding up the execution of fully connected layers [41]. While SVD decomposes the matrix in an exact way, TSVD is only an approximation. From equation 5.3 we can see the decomposition of the matrix by the application of the SVD algorithm. As previously mentioned, the equation is only true for SVD because TSVD is only an approximation, and thus equality is not satisfied. The decomposed matrix is defined as $\mathbf{X} \in \mathbb{R}^{m \times n}$, while the decomposed factors depend on which method we use. For SVD they are $\mathbf{U} \in \mathbb{R}^{m \times m}$, $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$, and $\mathbf{V} \in \mathbb{R}^{n \times n}$, while for TSVD they are $\mathbf{U} \in \mathbb{R}^{m \times t}$, $\mathbf{\Sigma} \in \mathbb{R}^{t \times t}$, and $\mathbf{V} \in \mathbb{R}^{t \times n}$ where $t \ll r$ and $r$ is the number of non-zero singular values. By utilizing the TSVD, we can reduce the number of elements from $m \cdot n$ to $t \cdot (m + n + t)$ which will be less if $t < 0.5(\sqrt{m^2 + 6mn + n^2} - m - n)$. Because of its structure, TSVD can be easily applied to fully connected layer, breaking it apart into two different layers where the bias is appended to the last. It requires no retraining and thus can be applied for run-time layer compression [46]. Even though the decomposition gives us three factors, we only use two matrices by merging the $\mathbf{\Sigma}$ into one of the two other matrices, thereby reducing the amount of parameters by $t^2$ while keeping the end

result the same.

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{5.3}$$

Singular value decomposition

While matrix decomposition methods such as SVD, LU, QR, and Cholesky decomposition are widely supported, tensor decomposition methods have scarce support. For example, while popular Python deep learning libraries PyTorch and TensorFlow support all the mentioned matrix decomposition methods, they do not support neither Tucker nor CP decomposition. Additionally, in deep learning we are much more likely to come across higher-order tensors than matrices, making tensor decomposition methods much more useful. There are many uses for tensor decompositions in the area of machine learning, such as relation inference and latent variable modeling [47].

The Tucker decomposition, shown in equation 5.4, is a form of higher-order PCA [48]. In Tucker decomposition, we decompose the tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ into a core tensor $\mathbf{G} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times R_N}$ and $N$ matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R_n}$. The parameter $R_i$, which we will call rank, is freely chosen based on the compression required. If we choose ranks for which $R_i < I_i$ is true, then the core tensor can be thought of as a compressed version of the original tensor $\mathbf{X}$. There are $N$ ranks in total, and in case we choose ranks for which $R_n = rank_n(\mathbf{X})$ our decomposition becomes exact, where $rank_n(\mathbf{A})$ denotes the column rank of unfolded tensor $\mathbf{A}$ on dimension $n$. Note that Tucker decompositions are generally not unique, as we can modify the core tensor without affecting the result as long as we apply the inverse to the factor matrices [48]. This allows us to change the structure of the factors so most of the elements are zero We can view this as a form of pruning, allowing us to utilize the introduced sparse tensor techniques. In addition, we will introduce partial Tucker decomposition. Equation 5.5 shows the structure of the decomposition. As we will see later in the chapter, the partial Tucker decomposition has a desirable structure. The only difference between the full and the partial decomposition is in the core tensor, where now only two dimensions correspond to the ranks while the others correspond to the initial decomposed tensor's dimensions.

$$x_{i_1,i_2,\ldots,i_N} = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \cdots \sum_{r_N=1}^{R_N} g_{r_1,r_2,\ldots,r_N} a^{(1)}_{i_1,r_1} a^{(2)}_{i_2,r_2} \cdots a^{(N)}_{i_N,r_N} \tag{5.4}$$

Tucker decomposition

$$x_{i_1,i_2,\ldots,i_N} = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} g_{r_1,r_2,i_3,i_4,\ldots,i_N} a^{(1)}_{i_1,r_1} a^{(2)}_{i_2,r_2} \tag{5.5}$$

Partial Tucker decomposition with two ranks

CANDECOMP (canonical decomposition)/PARAFAC (parallel factors), or CP for short, is an extension of SVD [8]. While in Tucker we had a core tensor, CP decomposes the tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ into $N$ matrices defined as $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$. Each matrix has the second dimension size set to the same value $R$ we call rank. CP can be viewed as a special case of Tucker where the core tensor is diagonal and $R_1 = R_2 = \cdots = R_n$ [48]. It can often be useful to normalize the matrices to length one, which will add factor $\lambda \in \mathbb{R}^R$ to the decomposition. If we choose rank for which $R = rank(\mathbf{X})$ is true, then our CP decomposition will be exact. While there are upper bounds for rank on tensors of specific number of dimensions, calculating the rank of an arbitrary tensor is an NP-hard problem [48], and the best low-rank approximation of a higher rank tensor may not even exist [49]. An exact CP decomposition is called the rank decomposition [48]. Unlike matrix decomposition, rank decomposition of higher-order tensors is often unique [48]. As finding tensor rank is a difficult problem, most algorithms try to fit multiple CP decompositions and take the best approximation [47].

$$x_{i_1,i_2,\ldots,i_N} = \sum_{r=1}^{R} a^{(1)}_{i_1,r} a^{(2)}_{i_2,r} \cdots a^{(N)}_{i_N,r} \tag{5.6}$$

CP decomposition

### 5.3.1 Convolutional layer decomposition

While the decomposition methods can be utilized on many different layers, we chose to focus on convolutional layers as they exist in all the models chosen. Decomposition methods are applied to the kernels of convolutional layers. Depending on the type of

convolutional layer, kernel sizes go from three dimensions to more. We will be focusing on one (1-D) and two (2-D) dimensional convolutions, meaning our kernel sizes will have three and four dimensions respectively. Equations 5.7 and 5.8 describe 1-D and 2-D convolution operations respectively. We do not include additional terms like dilution factors or padding, as they would only add unnecessary notation. For 1-D convolution, our input tensor is $\mathbf{X} \in \mathbb{R}^{B \times L_x \times I}$, output is $\mathbf{Y} \in \mathbb{R}^{B \times L_y \times O}$, and kernel is $\mathbf{K} \in \mathbb{R}^{L_k \times I \times O}$. While for 2-D convolution our input tensor is $\mathbf{X} \in \mathbb{R}^{B \times H_x \times W_x \times I}$, output is $\mathbf{Y} \in \mathbb{R}^{B \times H_y \times W_y \times O}$, and kernel is $\mathbf{K} \in \mathbb{R}^{H_k \times W_k \times I \times O}$. Each convolution consists of batch $B$, input $I$ and output $O$ channels, and size. For 1-D convolution we can interpret the input as an audio recording of length $L$ with $I$ channels, while for 2-D the input can be an image with $H$ height, $W$ width, and $I$ color channels. Additionally, we used batch notation, as it's common in deep learning libraries to perform convolution on multiple data points at once. It should be noted that kernel sizes, designated with a $k$ subscript, are usually much smaller than input ($x$ subscript) and output ($y$ subscript) sizes.

$$y_{b,l,o} = \sum_{dl=1}^{dL} \sum_{i=1}^{I} x_{b,s_l l+dl,i} k_{dl,i,o} \tag{5.7}$$

1-D convolution

$$y_{b,h,w,o} = \sum_{dh=1}^{dH} \sum_{dw=1}^{dW} \sum_{i=1}^{I} x_{b,s_h h+dh,s_w w+dw,i} k_{dh,dw,i,o} \tag{5.8}$$

2-D convolution

We apply the decomposition methods to the kernel $k$, decomposing it into many factors. Because we now have several kernels, our single convolution will turn into a series of convolutions. If we replace the kernel with the regular Tucker decomposition equation, we will notice an issue. The core tensor in the Tucker decomposition consists of all user-specified ranks. This prevents us from trivially using the core tensor as a part of the convolution equation. Therefore, instead of applying the normal Tucker, we will use the partial Tucker decomposition, where only two dimensions of the core tensor will be ranks while the rest will be from the kernel. For 1-D convolution our core tensor will become $\mathbf{G} \in \mathbb{R}^{L_k \times R_1 \times R_2}$, while for 2-D it will become $\mathbf{G} \in \mathbb{R}^{H_k \times W_k \times R_1 \times R_2}$. In equation 5.9

we show how to split up a single 2-D convolution into three where each factor of the partial Tucker decomposition is a kernel in their convolution operation. Keep in mind that to implement this operation additional steps must be taken to ensure the tensors are of the correct size. In equation 5.9 this would mean expanding the first dimension of factor $a^{(1)}$, and the first dimension of factor $a^{(2)}$, this will allow us to index them on three dimensions. There are additional parameters such as padding, dilation, and bias, which have been excluded for readability as they are trivial to implement.

$$y_{b,h,w,o} = \sum_{r_2=1}^{R_2} \left( \sum_{dh=1}^{dH} \sum_{dw=1}^{dW} \sum_{r_1=1}^{R_1} \left( \sum_{i=1}^{I} x_{b,s_h h+dh,s_w w+dw,i} a^{(1)}_{i,r_1} \right) g_{dh,dw,r_1,r_2} \right) (a^{(2)}_{o,r_2})^T \tag{5.9}$$

2-D convolution with partial Tucker applied

Similarly to Tucker decomposition, we apply the CP decomposition to the convolutional kernel $k$. In equation 5.10 we show the decomposition of 2-D convolution. Unlike Tucker decomposition, here we have four different convolutions where every kernel is a matrix. Just as in Tucker decomposition, we have omitted optional parameters of the convolution operation to help readability. While implementing this operation, great care must be taken to correctly expand the matrices, as 2-D kernels are four dimensional tensors.

$$y_{b,h,w,o} = \sum_{r=1}^{R} \left( \sum_{dh=1}^{dH} \left( \sum_{dw=1}^{dW} \left( \sum_{i=1}^{I} x_{b,s_h h+dh,s_w w+dw,i} a^{(3)}_{i,r} \right) a^{(2)}_{dw,r} \right) a^{(1)}_{dh,r} \right) (a^{(4)}_{o,r})^T \tag{5.10}$$

2-D convolution with CP applied

We have shown how to decompose 2-D convolution composition for both partial Tucker and CP decompositions. We do not show the decomposition for 1-D convolution as it contains fewer dimensions and therefore should be simpler to derive. The final issue we have to face is how to decide the ranks for the decomposition methods. As we do not want to have an exact decomposition, we will not be calculating the tensor $rank_n$ or $rank$. While there are different methods of finding ranks for decompositions, such as Variational Bayesian Matrix Factorization (VBMF) [50], we wanted a simple method that only needed a single parameter that signifies the level of compression. We can equate the

compression level parameter to sparsity, but instead of zeroing weights, we are removing them. The main condition we have for determining ranks is that the number of parameters of the decomposed convolution divided by the number of parameters of the original convolution is equal to the compression level, or mathematically $S = \frac{\#estimated}{\#original}$. For CP decomposition we can just calculate the rank giving us $R = S\frac{\prod_i I_i}{\sum_i I_i}$ where $I_i$ is the size of the $i$th dimension. On the other hand, for partial Tucker, we cannot calculate the ranks directly as there are many solutions for two ranks with only one equation. Therefore, we will include another restriction, which says that the ratio of the input rank and input channel has to be equal to the ratio of the output rank and output channel. Mathematically, we can write $\frac{R_1}{I_3} = \frac{R_2}{I_4}$. More broadly, we can say the ratio is between the rank and the dimension the rank is applied to. Solving the equation will give us $R_1 = \frac{-A+\sqrt{A^2+4BC^2S}}{2BC}$ where $A = \frac{I_3^2+I_4^2}{I_3}$, $B = I_3^{-2}$, $C = \prod_i I_i$, and the $R_2$ can be derived from the ratio.

# 6 Experiments

## 6.1 Datasets

To test our optimization methods, we will be using three different datasets: Aishell1Mix, LibriMix [51], and LibriCSS [52]. Aishell1Mix is an open-source Mandarin version of the speech separation dataset. It mixes two or three speaker sources from the open-source Aishell1 [53] and can contain noise from the dataset WHAM! [54]. The base Aishell1 contains around 165 h of speech from 400 speakers, while WHAM! noise contains 80 h of audio from 44 different locations. The second dataset, called LibriMix, is an open-source dataset for generalizable noisy speech separation. It contains a two- or three-speaker mixture where each mixture can be set to contain noise. The dataset is created from speech utterances taken from LibriSpeech [55] and noise samples taken from WHAM! [54]. In total, the dataset contains around 470 h of speech from 1252 speakers with a 60k vocabulary. In our experiments, we will be using only the two-speaker mixture with noise included, which is around 292 h of speech. Finally, LibriCSS is an open source dataset derived from LibriSpeech [55] by concatenating utterances to simulate conversation. It consists of 10 hours of audio recordings, where each session is an hour long and each session is split into 10 minute segments with different overlap ratios. The dataset was recorded by playing each LibriSpeech utterance from a different loudspeaker in a meeting room and capturing the acoustics with a microphone array.

Aishell1Mix and LibriMix are quite similar because they both use WHAM! as their noise source. The only difference is that Aishell1Mix uses Aishell1 as the source dataset, while LibriMix uses LibriSpeech. While the datasets share a lot of similarities, each of the chosen datasets serves a purpose in our testing. Aishell1Mix is used to test how the models respond to a different language; LibriMix is the generic noisy dataset; and LibriCSS is used to test how models handle many speakers with different overlap ratios.

## 6.2 Evaluation methods

For evaluation, we are using objective speech separation performance metrics: Scale-Invariant Signal-to-Distortion ratio (SI-SDR) and Signal-to-Noise ratio (SNR). In addition to the standard metrics, we include evaluation methods specific to optimization method evaluation. We will include the evaluation of model duration, memory consumption, latency, throughput, and number of parameters. In equations 6.1 and 6.3 we describe the calculation of SNR and SI-SDR respectively, where $T$ is the target separation and $P$ is the predicted separation. For both the target and prediction tensors, the first dimension is the audio channel, or the separated speaker, and the second dimension is the length of the audio. We can write this mathematically as $P, T \in \mathbb{R}^{C \times L}$.

$$SNR = 10 \log_{10} \left( \frac{||T||^2}{||P - T||^2} \right) \tag{6.1}$$

$$T_s = \frac{\sum(T \cdot P)}{||T||^2} \cdot T \tag{6.2}$$

$$SI\text{-}SDR = 10 \log_{10} \left( \frac{||T_s||^2}{||P - T_s||^2} \right) \tag{6.3}$$

When we feed the input audio to the chosen model, we get $N$ audio channels, which correspond to the set number of speakers. Because each channel can be any speaker, we don't know how to match it with the output of the dataset. While we can require the model to match the output channels from the dataset exactly, we aren't using speaker identities, and thus our output channel orderings do not have any meaning. This problem is termed the label ambiguity or permutation problem [56]. Therefore, instead of matching the channel ordering exactly, we will attempt to match every possible output channel from the model with every possible output channel from the dataset. With this, the model is free to choose which channels to pick for each speaker without a loss penalty. This method is called Permutation Invariant Training (PIT), and it works by calculating a cost matrix where each row corresponds to the model output channel and each column to the dataset output channel. To construct the matrix PIT requires an additional

algorithm that can take two tensors of the same size and output a scalar, for example, an algorithm such as SI-SDR. Then PIT will select a unique column for each row such that the sum of all the selected elements is minimal (or maximal). In equation 6.4, we show an example cost matrix with the solution highlighted. The model channel to dataset channel matching is then: $1 \rightarrow 3, 2 \rightarrow 1, 3 \rightarrow 2, 4 \rightarrow 4$.

$$\begin{bmatrix} 12 & 11 & \boxed{8} & 10 \\ \boxed{23} & 25 & 21 & 24 \\ 17 & \boxed{15} & 30 & 12 \\ 15 & 20 & 17 & \boxed{10} \end{bmatrix} \tag{6.4}$$

An example of a cost matrix with the solution highlighted

On the side of performance measurements there are also problems that need to be addressed. Due to the intricacies of today's processors and the prevalence of non-uniform memory access, we cannot guarantee the latency is measured reliably. We have taken steps in order to ensure smaller measurement variation, such as repeating measurements, setting CUDA clocks to stable, and synchronizing CUDA devices after each measurement. In order to determine the variations in measurements caused by the hardware, we will measure the expected performance metric deviation by calculating the standard deviation of the repeated performance measurements.

## 6.3 Testing setup

Before using any optimizations, we will first create a baseline to compare. Each model will be trained on each dataset, and results will be stored. We will use the baseline to compare how the optimization methods have impacted the accuracy and performance of the model. All the models use the same optimizer, scheduler, and loss function. For the loss function, we use SI-SDR. We use SI-SDR instead of SDR as it takes into consideration the scale of given estimates and doesn't allow the model to artificially boost the value by rescaling the estimate. Additionally, the computation of SI-SDR is much faster than that of SDR [57]. Along with SI-SDR, we use PIT, where we select the maximum loss out of the

cost matrix. For the optimizer, we use Adam with a learning rate of 0.001 and no weight decay. Our scheduler reduces the learning rate by a factor of 0.5 once the validation error stops improving for 5 epochs. Finally, we added an early stop mechanism that watches the validation error, and if it hasn't improved in 30 epochs, the training is concluded. All the model training and optimizations are written in Python programming language using the PyTorch library. To prevent code duplication, we only added functionalities that PyTorch didn't already support. Our library, which was used to perform all testing, is publicly available and open source [1].

For our pruning setup, we are using 7 different pruning strategies: random structured, L2-norm structured, L1-norm structured, gradient change structured, random unstructured, L1-norm unstructured, and gradient change unstructured. Every one of the strategies has an additional parameter that sets the percentage of weights to prune; we are using 10%, 20%, and 30%. In total, this makes 21 different pruning configurations for each model and dataset. We implemented pruning only for a select number of layers, which were most abundant in our models. These layers are: linear, conv1d, conv2d, and multihead-attention. Every pruning strategy is done iteratively with a maximum of 5 iterations of pruning/fine-tuning, and in case the validation error decreases by more than 5% we terminate the pruning and keep the previous result. While with pruning there is an additional multiplication of weight with the mask, at the end of fine-tuning we are merging the masks with the weights to remove this tiny overhead. Because pruning only sets weights to zero, we will not get a reduction in memory or computation. Therefore, we implemented many sparse storage formats, but because PyTorch tensors do not support many operations with this format, we could not use them for our models.

For quantization, we are using PTQ with an affine quantizer. The implementation quantizes weights ahead-of-time, but activations are dynamically quantized. This can add overhead to the model and slow down the performance during inference. Other forms of quantization, such as PTQ with ahead-of-time activation quantization and QAT were not tested as there is not support for them in PyTorch using CUDA GPUs. In addition, we tested two quantization number formats: 8-bit integer and 16-bit floating-point IEEE 754 with 1 sign bit, 5 exponent bits, and 11 mantissa bits. Because of the limited

[1] `https://github.com/mb52598/SSepOptim`

support for quantization, we are only using it on convolutional layers, as they are the most used.

To the best of our knowledge, PyTorch does not support tensor factorization. Therefore, we had to implement it in its entirety. We implemented CP and Tucker decomposition for both one- and two-dimensional convolutional layers. Both decomposition methods will do 5 iterations of the decomposition algorithm. As we are not doing a lot of iterations, our approximation error will be high, which is why we additionally perform fine-tuning. Same as for pruning, we will be testing the decomposition methods for three different sparsities: 10%, 20%, and 30%. But unlike pruning, tensor factorization does reduce the amount of parameters, which makes us expect a reduction in model latency and throughput. We focus on convolutional layers as they are substantially more expensive than linear layers [58].

This research was performed using the Advanced computing service provided by University of Zagreb University Computing Centre - SRCE [2]. This is a publicly available supercomputer for High Performance Computing (HPC). The supercomputer, called Supek after a famous Croatian academic, allows users to execute applications with high computational or memory demands on CPUs or GPUs. It is realized with HPE Craya technology, an operating system based on standard SUSE Enterprise Linux which is designed to run complex applications at scale. The supercomputer has in total 8384 processing cores, 81 graphical processors, 32 TB of work memory, which amounts to 1.25 PFLOPS. For the CPUs, the supercomputer uses an AMD Epyc 7763 2,45 GHz CPU, while it uses an NVIDIA A100 40 GB for the GPUs. Because the supercomputer is publicly available and the resources are given on first come, first served principle, the amount of available computational resources at any time fluctuated rapidly, and therefore, some of the testing was done in slightly different hardware configurations. We attempted to compensate by changing the node configuration as little as possible, but this still makes the results less reliable.

[2] `https://www.srce.unizg.hr/napredno-racunanje`

## 6.4 Results and discussion

We will start with baseline results, as they provide us with a starting point. Tables 6.1, 6.2, and 6.3 show the baseline measurements for LSTM-TasNet, Conv-TasNet, and DPTNet respectively. We can see that Conv-TasNet performs best in terms of SI-SDR, while DPTNet is better with SNR. Also, we can notice the amount of parameters is the same for Aishell1Mix and LibriMix while it changes for LibriCSS. This is because model size depends on the number of speakers. Aishell1Mix and LibriMix have only two speakers, while LibriCSS has eight. Latency and throughput should not be compared between models as batch sizes are different. Multiplying latency and throughput by the batch size wouldn't work, as performing multiple small batches has higher overhead than a single large batch. Total memory describes the maximum amount of CUDA memory used to test the models. Therefore, this metric does not take into account training memory usage such as activations and gradients.

For the sake of readability, in upcoming results we won't write the number of parameters or model memory usage as they can be inferred from the optimization method used. For pruning, the amount of parameters and model memory won't change unless we use a sparse storage format, in which case model memory usage will be reduced by the sparsity percentage. In quantization, we only change the precision of weights, meaning the effect is on the model memory. Fp16 will half the model memory usage, while Int8 will half Fp16. Tensor factorization methods alter the model layers and reduce the amount of parameters, meaning the number of parameters is directly reduced based on the percentage specified.

After baseline results, we will first start with the accuracy results of optimization methods, as we don't want to use methods that cause high accuracy degradation. Figures 6.1, 6.2, and 6.3 show the accuracy graph of optimization methods for LSTM-TasNet, Conv-TasNet, and DPTNet respectively. In the graphs, we color each optimization method differently and are looking for points that are closest to the top right, meaning they achieve high SI-SDR and SNR.

Tensor decomposition methods were not used on the Conv-TasNet model as it contained too many convolutional layers for the methods to be executed in a reasonable

| Metric \ Dataset | Aishell1Mix | LibriMix | LibriCSS |
|---|---|---|---|
| SI-SDR[db] | -0.494116 | -15.750694 | -53.724457 |
| SNR[db] | -21.255798 | -25.018694 | -56.821930 |
| Latency[ns/batch] | 1,042,926,828.4 | 1,425,516,681 | 12,375,298,454.5 |
| Throughput[batch/s] | 0.994591 | 0.682281 | 0.082434 |
| Parameters | 23,168,513 | 23,168,513 | 26,243,585 |
| Model memory[bytes] | 92,674,052 | 92,674,052 | 104,974,340 |
| Total memory[bytes] | 3,549,108,224 | 719,061,504 | 12,210,176,512 |

Table 6.1: LSTM-TasNet baseline results

| Metric \ Dataset | Aishell1Mix | LibriMix | LibriCSS |
|---|---|---|---|
| SI-SDR[db] | -0.485094 | -0.383385 | -50.609952 |
| SNR[db] | -69.226654 | -69.213966 | -68.760139 |
| Latency[ns/batch] | 357,075,077 | 310,947,963.2 | 3,032,943,748 |
| Throughput[batch/s] | 3.304651 | 4.0358795 | 0.36473475 |
| Parameters | 4,985,394 | 4,985,394 | 5,447,346 |
| Model memory[bytes] | 19,941,576 | 19,941,576 | 21,789,384 |
| Total memory[bytes] | 464,708,096 | 278,207,488 | 1,548,478,464 |

Table 6.2: Conv-TasNet baseline results

| Metric \ Dataset | Aishell1Mix | LibriMix | LibriCSS |
|---|---|---|---|
| SI-SDR[db] | -0.554791 | -15.892398 | -53.429151 |
| SNR[db] | 2.438616 | -22.330877 | -55.249344 |
| Latency[ns/batch] | 964,047,325.9 | 1,515,144,942.1 | 10,506,321,391.6 |
| Throughput[batch/s] | 1.116504 | 0.695022 | 0.096712 |
| Parameters | 2,857,666 | 2,857,666 | 2,882,626 |
| Model memory[bytes] | 11,430,664 | 11,430,664 | 11,530,504 |
| Total memory[bytes] | 4,046,401,536 | 826,960,384 | 6,082,228,224 |

Table 6.3: DPTNet baseline results

time. Additionally, we have excluded CP decomposition from accuracy graphs as they are much worse than the rest of the methods and therefore make the graph unreadable. This is possibly because the gradients in the inserted CP layers are prone to gradient explosion, as one study has found [59]. One possible way to remedy this is to reduce the learning rate while fine-tuning the model. In tables 6.4 and 6.5 we added the values of

CP decomposition for LSTM-TasNet and DPTNet models. Because CP decomposition methods have a substantial decrease in accuracy, we will not be analyzing them further.

In the results, we can see every model responds differently to each dataset. While for Aishell1Mix and LibriMix the optimization methods come close to baseline, in LibriCSS baseline is always better in terms of SI-SDR. This could be because the number of speakers affects the performance of optimization methods. Also, the baseline model is better than optimization methods in terms of SI-SDR, while for SNR, sometimes optimizations exceed the baseline model. This could indicate that SNR is not a good metric of model performance, as we do not expect the optimization methods to exceed the baseline. We can also see the scale ranges of values differ substantially for each dataset and each model. In table 6.6, we extract the accuracy ranges from the graphs. Lower ranges indicate the optimization methods do not have a substantial effect on model accuracy. In our tested models, there is no clear winner in terms of having the lowest range, as all of them have a configuration where they are the best. Note that just having low range does not indicate a good configuration, as we would expect models with higher accuracy to have larger ranges. Intuitively high-accuracy models are less probable than low-accuracy models as they require a specific configuration of model parameters. By introducing optimization methods that affect parameter precision or their amount, we are reducing the pool of available parameter configurations and therefore making high-accuracy models less likely.

| | 10% | | 20% | | 30% | |
|---|---|---|---|---|---|---|
| | SI-SDR | SNR | SI-SDR | SNR | SI-SDR | SNR |
| Aishell1Mix | -37.2551 | -37.2607 | -36.3692 | -69.1230 | -35.3168 | -69.0785 |
| LibriMix | -37.5460 | -69.1290 | -29.4931 | -67.9022 | -35.0663 | -68.7375 |
| LibriCSS | -63.7152 | -69.1728 | -63.5798 | -69.1633 | -63.5385 | -68.0773 |

Table 6.4: LSTM-TasNet CP accuracy

| | 10% | | 20% | | 30% | |
|---|---|---|---|---|---|---|
| | SI-SDR | SNR | SI-SDR | SNR | SI-SDR | SNR |
| Aishell1Mix | -32.4208 | -41.4395 | -34.0540 | -41.7094 | -30.5297 | -40.5451 |
| LibriMix | -36.1560 | -61.3936 | -32.9784 | -59.9193 | -34.6262 | -56.3932 |
| LibriCSS | -63.6869 | -63.5381 | -63.4647 | -63.4828 | -62.8411 | -63.0791 |

Table 6.5: DPTNet CP accuracy

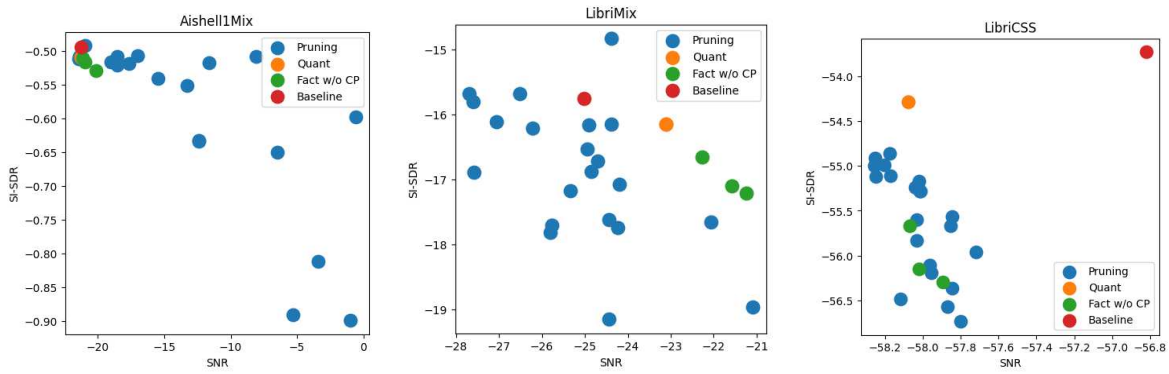


**Figure 6.1:** LSTM-TasNet optimizations accuracy

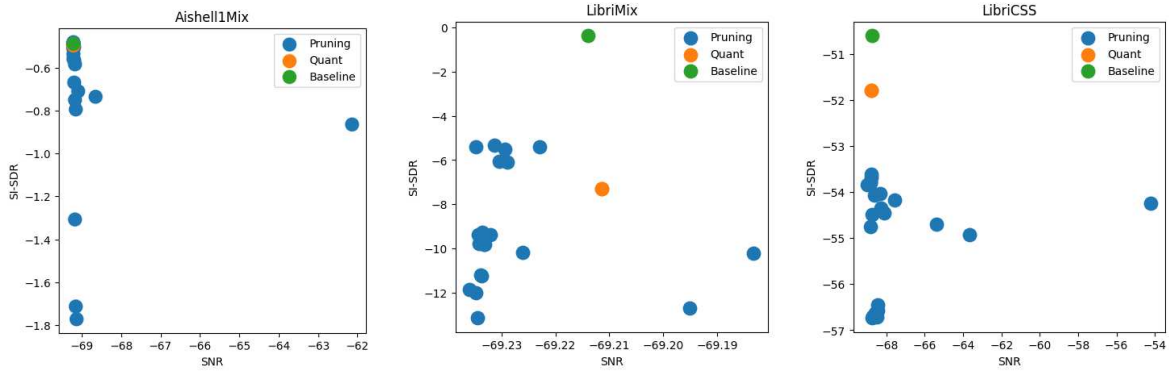


**Figure 6.2:** Conv-TasNet optimizations accuracy

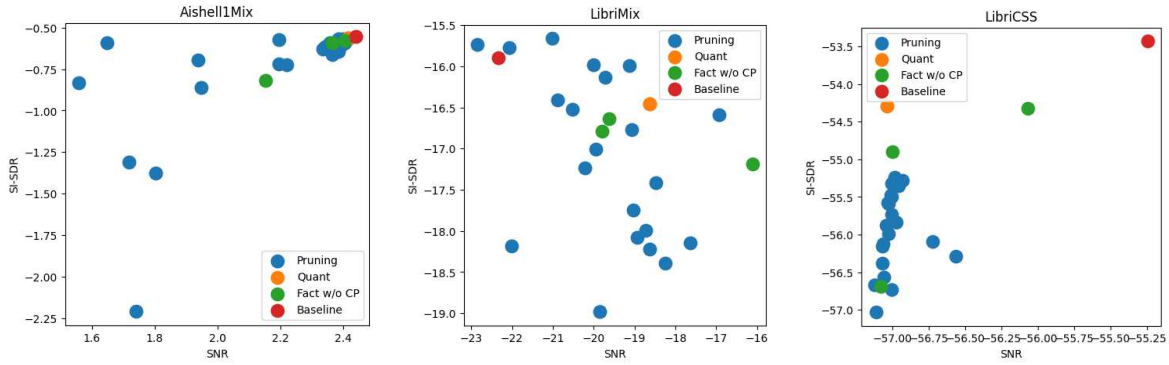


**Figure 6.3:** DPTNet optimizations accuracy

| | Aishell1Mix | | LibriMix | | LibriCSS | |
|---|---|---|---|---|---|---|
| | SI-SDR | SNR | SI-SDR | SNR | SI-SDR | SNR |
| LSTM-TasNet | 0.40696 | 20.833 | 4.3197 | 6.5886 | 2.4544 | 0.53728 |
| Conv-TasNet | 1.2912 | 7.0745 | 7.8005 | 0.052749 | 4.9404 | 14.791 |
| DPTNet | 1.6437 | 0.85631 | 3.3194 | 6.7283 | 2.7374 | 1.0503 |

Table 6.6: Accuracy ranges without CP decomposition

Next we will look at only the pruning optimization accuracies. Figures 6.4, 6.5, and 6.6 show pruning accuracies for all the tested models. In the graph, each sparsity is col-

ored differently, and each point has an annotation for the type of pruning method used. For readability, the names of methods are compressed, where R is random, GC is gradient change, L1/L2 is L1-/L2-norm, S is structured, and U is unstructured. Because most of our configurations use pruning methods, the graphs haven't changed much. For Aishell1Mix, we can clearly see how higher sparsities have lower accuracy. Most of the lower accuracy methods are structured and do not contain the gradient change method. Another pattern we can observe is that configurations with higher median accuracies separate sparsities, while those with low medians lack structure. This is the reason we want models with high accuracy. The change in model accuracy due to optimization methods will be much lower if our model already has low accuracy, which will prevent us from analyzing the behaviour of those methods. Throughout all graphs, four pruning methods always performed the worst at high sparsity: random unstructured, random structured, l1-norm structured, and l2-norm structured. While these methods outperformed gradient change in low sparsities, the gradient change methods seem to work better than others for higher sparsities.

Quantization and tensor factorization will be shown in a single graph as there are only 5 configurations considering we excluded CP decomposition. Figures 6.7, 6.8, and 6.9 show quantization and tensor factorization methods for LSTM-TasNet and DPTNet, while for Conv-TasNet we only show quantization methods. For tensor decomposition, the results are similar to pruning, but because we are using only one method, they are more pronounced. We can see that the bigger the reduction in the number of parameters, the higher the accuracy drop. On the quantization side, 8-bit integer and 16-bit floating point are performing almost exactly the same. In almost all cases (except DPTNet LibriCSS), quantization methods are as good or better than tensor decomposition methods in terms of accuracy.

Because we are not using sparse storage formats in pruning optimization methods, there should be no change in latency or throughput for those models. We also removed CP from the considered methods because of its effect on accuracy. This leaves us with quantization optimizations and Tucker decomposition. Before we analyze model performance, we need to determine the possible variation of latency and throughput due to the changing environment. In table 6.7, we calculated the mean and standard deviation for

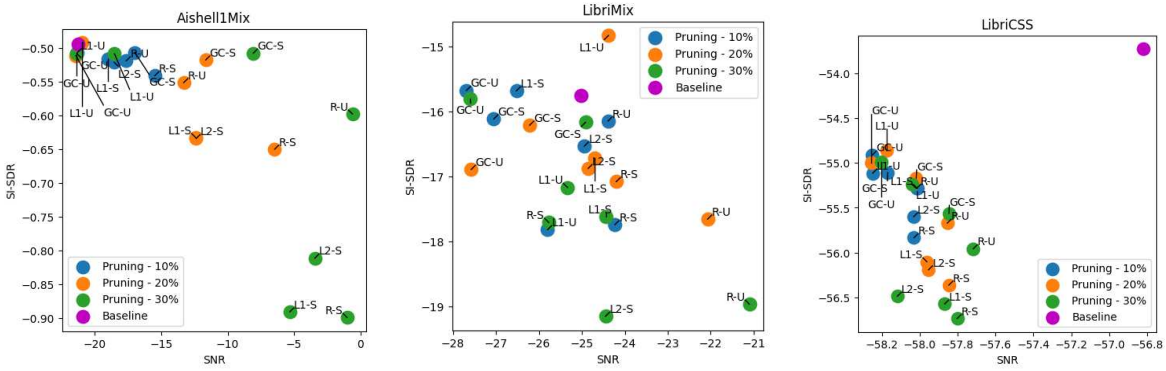


**Figure 6.4:** LSTM-TasNet pruning accuracy

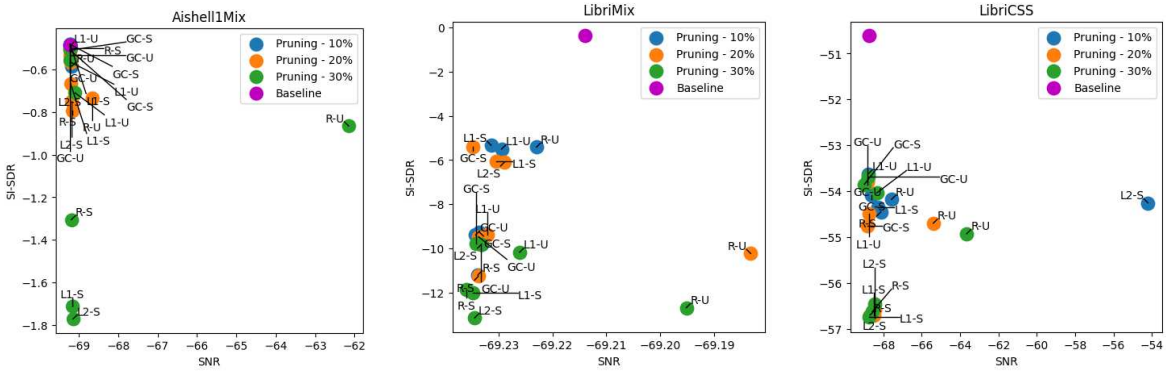


**Figure 6.5:** Conv-TasNet pruning accuracy

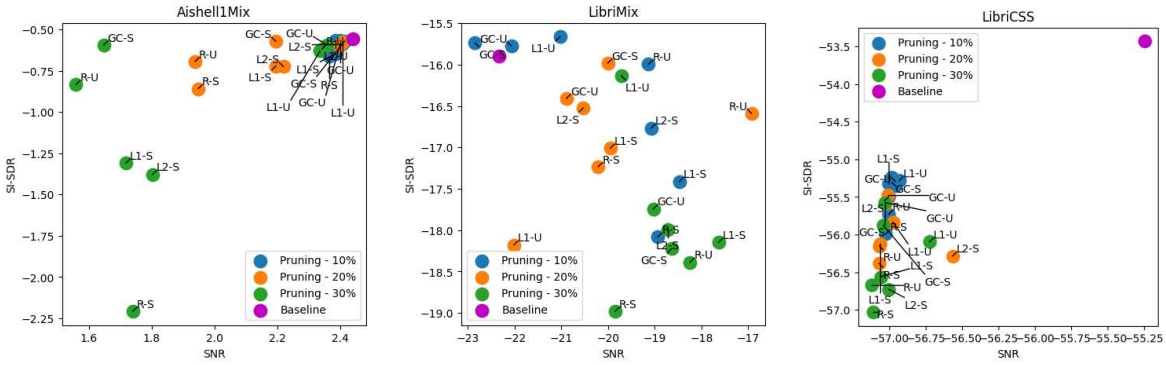


**Figure 6.6:** DPTNet pruning accuracy

all models in each dataset using the baseline and pruning results. From the table, we can see the standard deviation is only one or two magnitudes less than the mean. In figures 6.10, 6.11, and 6.12, we show the optimization performance for all the models in all the datasets. For almost all models, the optimization methods have higher throughput and lower latency than the baseline model. There are two exceptions: Conv-TasNet LibriMix and DPTNet LibriCSS. The first exception is most likely caused by implementation differences between quantization and baseline model structure. The second exception is

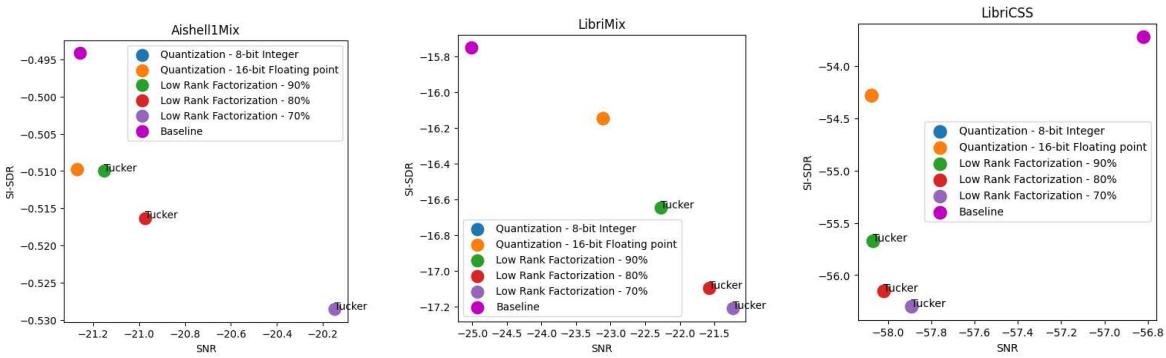


**Figure 6.7:** LSTM-TasNet quantization and factorization accuracy

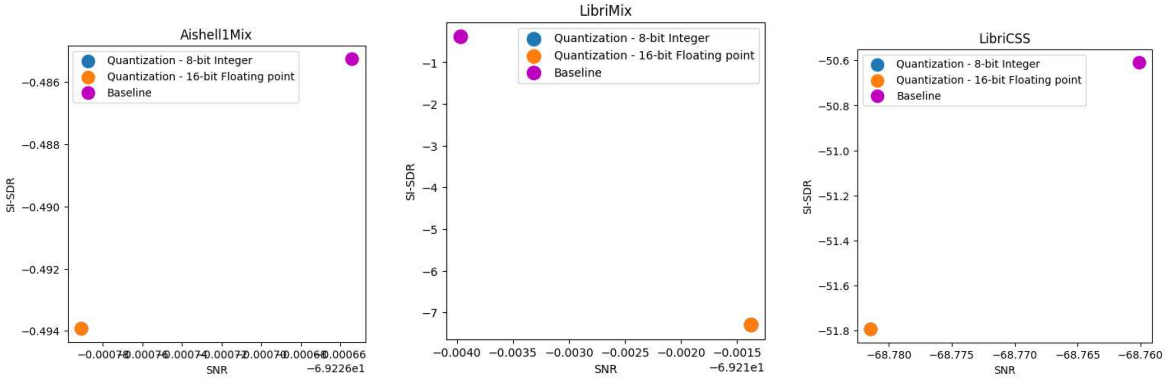


**Figure 6.8:** Conv-TasNet quantization accuracy

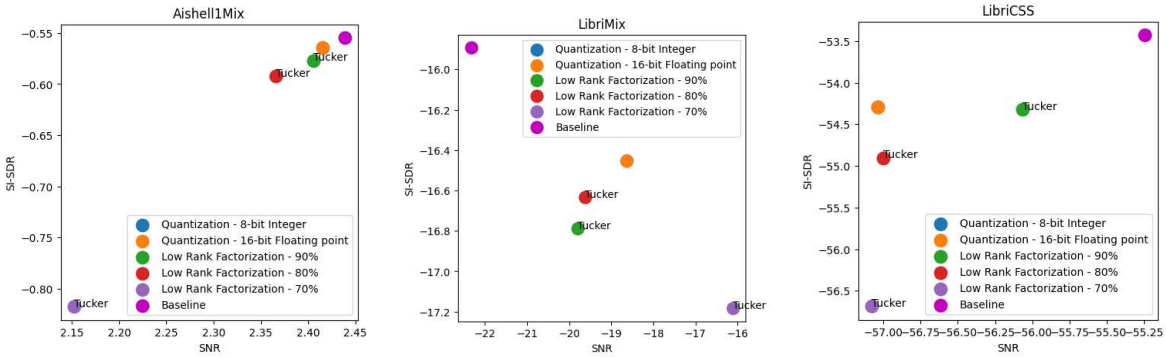


**Figure 6.9:** DPTNet quantization and factorization accuracy

not really an exception as the measurements are close together, where the differences are only 0.7 seconds for latency and 0.007 batches per second for throughput. Comparing tensor decomposition and quantization performance, no method stands out as they all have comparative performance results. This is unexpected, as models with a lower number of parameters should be faster, but the speed gain seems to be negligent.

While the results are not definitive, there are a few takeaways. In higher sparsity pruning, random, l1-norm, and l2-norm structured methods did not perform as well and

| | | Latency | | Throughput | |
|---|---|---|---|---|---|
| | | Mean | Std | Mean | Std |
| LSTM-TasNet | Aishell1Mix | 9.66927e8 | 5.52746e7 | 1.06006 | 5.36619e-2 |
| | LibriMix | 1.17543e9 | 2.11066e7 | 8.43605e-1 | 1.96107e-2 |
| | LibriCSS | 1.07480e10 | 2.37264e8 | 9.33370e-2 | 2.10463e-3 |
| Conv-TasNet | Aishell1Mix | 2.88155e8 | 9.77031e6 | 4.27277 | 1.49526e-1 |
| | LibriMix | 2.92663e8 | 2.89972e7 | 3.50815e | 1.71752e-1 |
| | LibriCSS | 2.68844e9 | 2.50997e7 | 3.78454e-1 | 2.40725e-3 |
| DPTNet | Aishell1Mix | 8.92666e8 | 5.60578e6 | 1.20919 | 5.44137e-3 |
| | LibriMix | 1.17284e9 | 3.72547e7 | 8.47294e-1 | 3.16489e-2 |
| | LibriCSS | 1.06686e10 | 2.56276e8 | 9.43626e-2 | 2.23814e-3 |

Table 6.7: Model latency and throughput variation

had a significant drop in accuracy. Gradient change methods seemed to handle higher sparsities better while dropping behind in lower. We did not notice a significant difference between structured and unstructured pruning methods except in higher sparsities where unstructured were better. In quantization methods, there was practically no difference in model accuracy between the methods, while 8-bit integer was slightly more performant than 16-bit floating point. For low-rank factorization methods, we had CP decomposition, which did much worse than all other optimization methods, possibly because of the learning rate was too high. On the other hand, Tucker decomposition performed slightly worse than quantization still but had accuracy results comparable to other optimization methods while providing increases in throughput and reductions in latency. Much of the results did not show a structure in the data, which is possibly because models did not achieve high enough accuracy. For example, we expected higher sparsity pruning methods to be worse than lower sparsity. While this was true for Aishell1Mix, it was not for the other two datasets. We believe much higher accuracy models can be produced if we train the model with SI-SDR and PIT where the PIT is minimized instead of maximized.

While there were significant steps taken in order to ensure the reliability of testing, we were using a publicly available cluster for which we don't have total control and thus were not in control of all the resulting variables. Therefore, for future work, more testing should be attempted on more reliable equipment. While our chosen models have many differences between them, we only used time-domain models due to their simplicity. It

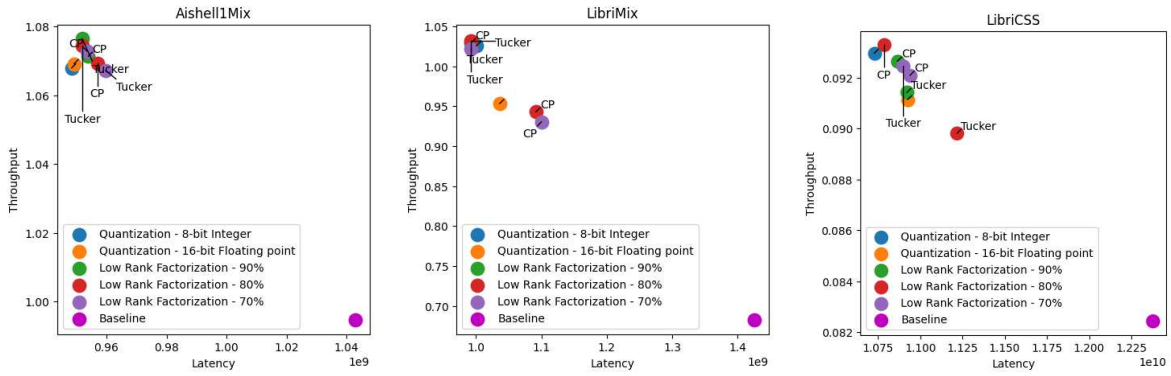


**Figure 6.10:** LSTM-TasNet optimizations performance

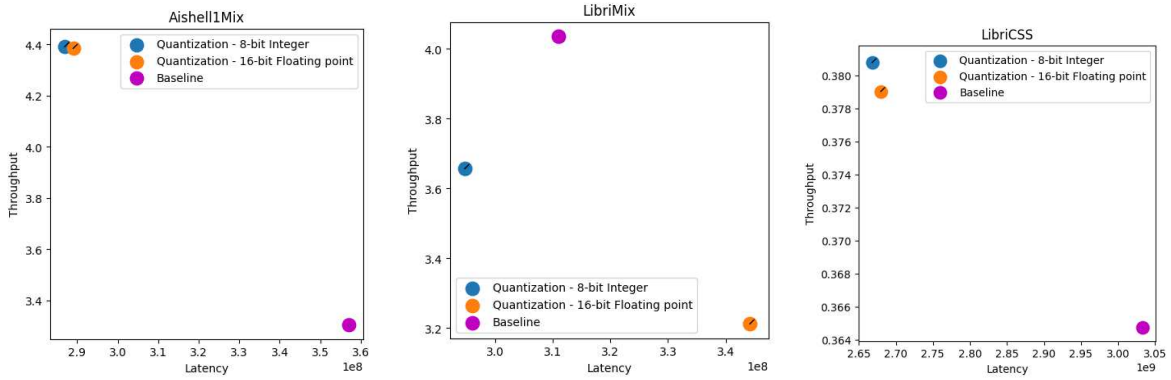


**Figure 6.11:** Conv-TasNet optimizations performance

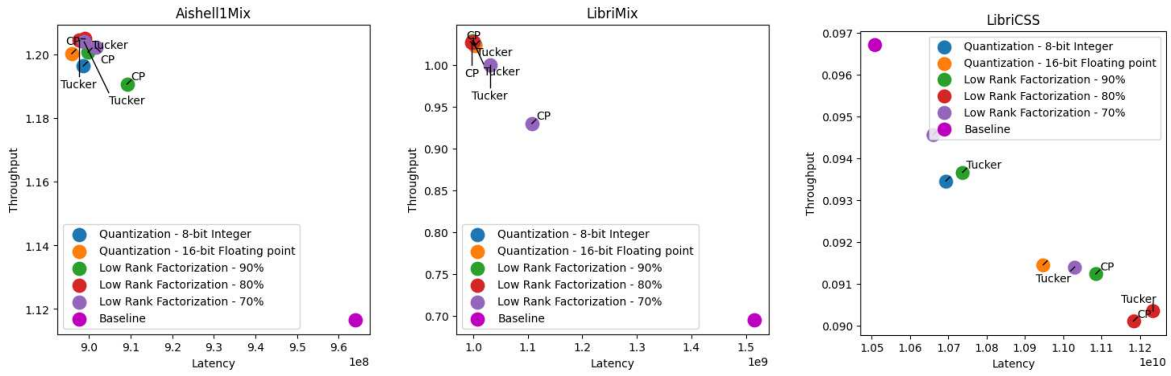


**Figure 6.12:** DPTNet optimizations performance

would prove useful to add frequency-domain models. We have also seen that the current models were not capable of achieving high accuracy in the used datasets. It would be beneficial to test the current models with minimized PIT and also include larger models capable of achieving high accuracy.

While pruning methods have shown great promise, to get throughput and latency improvements, they should be tested with sparse storage formats. For quantization, we used dynamic activation quantization, which contains a small overhead because the ac-

tivations are quantized during run-time. It would prove useful to test ahead-of-time activation PTQ and an even more promising QAT. We also used only two number formats when there are many more available. Additionally, the implementation of the CP and Tucker decomposition should be moved to a low-level language, such as C++. Because we couldn't perform many algorithm iterations, we got worse approximations, and for Conv-TasNet, we couldn't use the decomposition at all. Just using a single optimization method does provide us with faster inference, but to get even greater improvements, many optimization methods could be applied at once. While our library supports everything except faster low-rank factorization, the support is limited to CPUs and therefore requires substantially longer training times, and the inference will also be limited to CPUs.

# 7 Conclusion

Cloud computing is a traditional solution used to handle deep learning on mobile devices. Faced with privacy concerns, we shifted to edge computing in the task of speech separation. Speech is the main method people used to communicate, making speech separation an area of great importance for improving lives. While edge computing solves the privacy problem, we face a new problem: resource-constrained inference. High-accuracy deep learning models in speech separation have prohibitively large amounts of parameters, making edge computing infeasible.

In order to speed up deep learning models, we introduce three optimization methods: quantization, pruning, and low-rank factorization. To test these methods, we explored three different models with diverse architectures on three diverse speech separation datasets. Results have shown that some models were not capable of achieving high accuracy for certain datasets, which prevented us from analyzing them in depth.

We have shown that high sparsity pruning generally performs worse than lower sparsities, but also that certain methods handle higher sparsities better while others are better suited for lower sparsities. Both quantization methods performed similarly in terms of accuracy, while the lower precision method had better speed improvements. Similarly to pruning, low-rank factorization methods performed worse for higher reductions in model parameters. We found CP decomposition to be performing much worse than other optimization methods, possibly because it required lower learning rates on fine-tuning.

While we were unable to test the performance of pruning methods due to implementation constraints, quantization and low-rank factorization methods have shown great improvements in terms of latency and throughput. Finally, we described the problems

we faced, possible solutions, and gave directions for future work.

# References


[1] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation," in *Advances in Neural Information Processing Systems*, vol. 27. Curran Associates, Inc., 2014.

[2] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," Feb. 2016.

[3] J. Chen and X. Ran, "Deep Learning With Edge Computing: A Review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, Aug. 2019. https://doi.org/10.1109/JPROC.2019.2921977

[4] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. Van Gool, "AI Benchmark: All About Deep Learning on Smartphones in 2019," Oct. 2019.

[5] Q. Zhang, M. Zhang, M. Wang, W. Sui, C. Meng, J. Yang, W. Kong, X. Cui, and W. Lin, "Efficient Deep Learning Inference Based on Model Compression," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, pp. 1695–1702.

[6] A. Kumar, A. Balasubramanian, S. Venkataraman, and A. Akella, "Accelerating Deep Learning Inference via Freezing," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[7] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized Convolutional Neural Networks for Mobile Devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4820–4828.

[8] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, “Deep Learning on Mobile and Embedded Devices: State-of-the-art, Challenges, and Future Directions,” *ACM Computing Surveys*, vol. 53, no. 4, pp. 84:1–84:37, Aug. 2020. https://doi.org/10.1145/3398209

[9] Y. Zhou and K. Yang, “Exploring TensorRT to Improve Real-Time Inference for Deep Learning,” in *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. Hainan, China: IEEE, Dec. 2022, pp. 2011–2018. https://doi.org/10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00299

[10] S. S. Ogden and T. Guo, “Characterizing the Deep Neural Networks Inference Performance of Mobile Applications,” Sep. 2019.

[11] S. Mittal, “A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform,” *Journal of Systems Architecture*, vol. 97, pp. 428–442, Aug. 2019. https://doi.org/10.1016/j.sysarc.2019.01.011

[12] X. Ma, G. Li, L. Liu, H. Liu, L. Liu, and X. Feng, “Understanding the Runtime Overheads of Deep Learning Inference on Edge Devices,” in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. New York City, NY, USA: IEEE, Sep. 2021, pp. 390–397. https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00061

[13] A. Castelló, S. Barrachina, M. F. Dolz, E. S. Quintana-Ortí, P. S. Juan, and A. E. Tomás, “High performance and energy efficient inference for deep learning on multicore ARM processors using general optimization techniques and BLIS,” *Journal of Systems Architecture*, vol. 125, p. 102459, Apr. 2022. https://doi.org/10.1016/j.sysarc.2022.102459

[14] G. Verma, Y. Gupta, A. M. Malik, and B. Chapman, “Performance Evaluation of

Deep Learning Compilers for Edge Inference," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Portland, OR, USA: IEEE, Jun. 2021, pp. 858–865. https://doi.org/10.1109/IPDPSW52791.2021.00128

[15] M. Božić and M. Horvat, "A Survey of Deep Learning Audio Generation Methods," May 2024. https://doi.org/10.48550/arXiv.2406.00146

[16] G. Menghani, "Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–37, Dec. 2023. https://doi.org/10.1145/3578938

[17] R. G. Pacheco and R. S. Couto, "Inference Time Optimization Using BranchyNet Partitioning," in *2020 IEEE Symposium on Computers and Communications (ISCC)*, Jul. 2020, pp. 1–6. https://doi.org/10.1109/ISCC50000.2020.9219647

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

[19] C. McCarter and N. Dronen, "Look-ups are not (yet) all you need for deep learning inference," Jul. 2022.

[20] V. S. Marco, B. Taylor, Z. Wang, and Y. Elkhatib, "Optimizing Deep Learning Inference on Embedded Systems Through Adaptive Model Selection," *ACM Transactions on Embedded Computing Systems*, vol. 19, no. 1, pp. 1–28, Jan. 2020. https://doi.org/10.1145/3371154

[21] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. Salt Lake City, UT: IEEE, Jun. 2018, pp. 6848–6856. https://doi.org/10.1109/CVPR.2018.00716

[22] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI:

IEEE, Jul. 2017, pp. 6517–6525. https://doi.org/10.1109/CVPR.2017.690

[23] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," Apr. 2017.

[24] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," Nov. 2016.

[25] F. Jia, D. Zhang, T. Cao, S. Jiang, Y. Liu, J. Ren, and Y. Zhang, "CoDL: Efficient CPU-GPU co-execution for deep learning inference on mobile devices," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. Portland Oregon: ACM, Jun. 2022, pp. 209–221. https://doi.org/10.1145/3498361.3538932

[26] T. Zhao, Y. Xie, Y. Wang, J. Cheng, X. Guo, B. Hu, and Y. Chen, "A Survey of Deep Learning on Mobile Devices: Applications, Optimizations, Challenges, and Research Opportunities," *Proceedings of the IEEE*, vol. 110, no. 3, pp. 334–354, Mar. 2022. https://doi.org/10.1109/JPROC.2022.3153408

[27] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, "An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks," *Future Internet*, vol. 12, no. 7, p. 113, Jul. 2020. https://doi.org/10.3390/fi12070113

[28] D. Wang and J. Chen, "Supervised Speech Separation Based on Deep Learning: An Overview," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 26, no. 10, pp. 1702–1726, Oct. 2018. https://doi.org/10.1109/TASLP.2018.2842159

[29] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks," *Journal of Machine Learning Research*, vol. 22, no. 241, pp. 1–124, 2021.

[30] Y. Luo and N. Mesgarani, "Real-time Single-channel Dereverberation and

Separation with Time-domain Audio Separation Network," in *Interspeech 2018*. ISCA, Sep. 2018, pp. 342–346. https://doi.org/10.21437/Interspeech.2018-2290

[31] ——, "Conv-TasNet: Surpassing Ideal Time–Frequency Magnitude Masking for Speech Separation," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 27, no. 8, pp. 1256–1266, Aug. 2019. https://doi.org/10.1109/TASLP.2019.2915167

[32] J. Chen, Q. Mao, and D. Liu, "Dual-Path Transformer Network: Direct Context-Aware Modeling for End-to-End Monaural Speech Separation," Aug. 2020. https://doi.org/10.48550/arXiv.2007.13975

[33] M. van Baalen, A. Kuzmin, S. S. Nair, Y. Ren, E. Mahurin, C. Patel, S. Subramanian, S. Lee, M. Nagel, J. Soriaga, and T. Blankevoort, "FP8 versus INT8 for efficient deep learning inference," Jun. 2023.

[34] M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, and M. Shafique, "Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead," *IEEE Access*, vol. 8, pp. 225 134–225 180, 2020. https://doi.org/10.1109/ACCESS.2020.3039858

[35] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017. https://doi.org/10.1109/JPROC.2017.2761740

[36] M. M. H. Shuvo, S. K. Islam, J. Cheng, and B. I. Morshed, "Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review," *Proceedings of the IEEE*, vol. 111, no. 1, pp. 42–91, Jan. 2023. https://doi.org/10.1109/JPROC.2022.3226481

[37] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation," Apr. 2020.

[38] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," Dec. 2017.

[39] F. Daghero, D. J. Pagliari, and M. Poncino, “Chapter Eight - Energy-efficient deep learning inference on edge devices,” in *Advances in Computers*, ser. Hardware Accelerator Systems for Artificial Intelligence and Machine Learning, S. Kim and G. C. Deka, Eds. Elsevier, Jan. 2021, vol. 122, pp. 247–301. https://doi.org/10.1016/bs.adcom.2020.07.002

[40] A. N. Mazumder, J. Meng, H.-A. Rashid, U. Kallakuri, X. Zhang, J.-S. Seo, and T. Mohsenin, “A Survey on the Optimization of Neural Network Accelerators for Micro-AI On-Device Inference,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 11, no. 4, pp. 532–547, Dec. 2021. https://doi.org/10.1109/JETCAS.2021.3129415

[41] H. Cai, J. Lin, Y. Lin, Z. Liu, H. Tang, H. Wang, L. Zhu, and S. Han, “Enable Deep Learning on Mobile Devices: Methods, Systems, and Applications,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 27, no. 3, pp. 1–50, May 2022. https://doi.org/10.1145/3486618

[42] F. Hohman, C. Wang, J. Lee, J. Görtler, D. Moritz, J. P. Bigham, Z. Ren, C. Foret, Q. Shan, and X. Zhang, “Talaria: Interactively Optimizing Machine Learning Models for Efficient Inference,” in *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Honolulu HI USA: ACM, May 2024, pp. 1–19. https://doi.org/10.1145/3613904.3642628

[43] Y. Deng, “Deep learning on mobile devices: A review,” in *Mobile Multimedia/Image Processing, Security, and Applications 2019*, S. S. Agaian, S. P. DelMarco, and V. K. Asari, Eds. Baltimore, United States: SPIE, May 2019, p. 11. https://doi.org/10.1117/12.2518469

[44] Q. Qin, J. Ren, J. Yu, L. Gao, H. Wang, J. Zheng, Y. Feng, J. Fang, and Z. Wang, “To Compress, or Not to Compress: Characterizing Deep Learning Model Compression for Embedded Inference,” Oct. 2018.

[45] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, “Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications,” Feb. 2016.

[46] K. Nan, S. Liu, J. Du, and H. Liu, “Deep model compression for mobile platforms: A survey,” *Tsinghua Science and Technology*, vol. 24, no. 6, pp. 677–693, Dec. 2019. https://doi.org/10.26599/TST.2018.9010103

[47] S. Rabanser, O. Shchur, and S. Günnemann, “Introduction to Tensor Decompositions and their Applications in Machine Learning,” Nov. 2017.

[48] T. G. Kolda and B. W. Bader, “Tensor Decompositions and Applications,” *SIAM Review*, vol. 51, no. 3, pp. 455–500, Aug. 2009. https://doi.org/10.1137/07070111X

[49] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, “Tensor Decomposition for Signal Processing and Machine Learning,” *IEEE Transactions on Signal Processing*, vol. 65, no. 13, pp. 3551–3582, Jul. 2017. https://doi.org/10.1109/TSP.2017.2690524

[50] S. Nakajima, M. Sugiyama, S. D. Babacan, and R. Tomioka, “Global Analytic Solution of Fully-observed Variational Bayesian Matrix Factorization.”

[51] J. Cosentino, M. Pariente, S. Cornell, A. Deleforge, and E. Vincent, “LibriMix: An Open-Source Dataset for Generalizable Speech Separation,” May 2020.

[52] Z. Chen, T. Yoshioka, L. Lu, T. Zhou, Z. Meng, Y. Luo, J. Wu, X. Xiao, and J. Li, “Continuous speech separation: Dataset and analysis,” May 2020.

[53] H. Bu, J. Du, X. Na, B. Wu, and H. Zheng, “AISHELL-1: An Open-Source Mandarin Speech Corpus and A Speech Recognition Baseline,” Sep. 2017.

[54] G. Wichern, J. Antognini, M. Flynn, L. R. Zhu, E. McQuinn, D. Crow, E. Manilow, and J. L. Roux, “WHAM!: Extending Speech Separation to Noisy Environments,” Jul. 2019. https://doi.org/10.48550/arXiv.1907.01160

[55] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, “Librispeech: An ASR corpus based on public domain audio books,” in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Apr. 2015, pp. 5206–5210. https://doi.org/10.1109/ICASSP.2015.7178964

[56] D. Yu, M. Kolbæk, Z.-H. Tan, and J. Jensen, “Permutation Invariant Training of Deep Models for Speaker-Independent Multi-talker Speech Separation,” Jan. 2017.

[57] J. L. Roux, S. Wisdom, H. Erdogan, and J. R. Hershey, “SDR – Half-baked or Well Done?” in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2019, pp. 626–630. https://doi.org/10.1109/ICASSP.2019.8683855

[58] Y. Chen, S. Biookaghazadeh, and M. Zhao, “Exploring the capabilities of mobile devices in supporting deep learning,” in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ser. SEC ’19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 127–138. https://doi.org/10.1145/3318216.3363316

[59] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, “Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition,” Apr. 2015. https://doi.org/10.48550/arXiv.1412.6553

# Sažetak


## Razvoj učinkovitog modela dubokog učenja za raspoznavanje govora

Matej Božić

Modeli dubokog učenja postali su temelj u razvoju naprednih algoritama za obradu govora, postižući izuzetne rezultate u točnosti i izražajnosti. Međutim, njihova složenost često ograničava primjenu u stvarnom vremenu gdje su brzina i učinkovitost ključni. Ovaj rad je usmjeren na istraživanje i razvoj modela dubokog učenja koji su optimirani za primjenu u raspoznavanju govora iz zvučnog signala s naglaskom na smanjenje složenosti modela. Cilj je razviti model koji nudi optimalan kompromis između brzine rada, broja parametara za učenje i upotrijebljene memorije, uspoređujući s trenutno zastupljenim rješenjima u ovom području primjene. Posebna pažnja bit će posvećena kreiranju modela s manjim brojem učenih parametara koji zadržavaju visoku učinkovitost uz poboljšanu brzinu rada.



**Ključne riječi:** Raspoznavanje govora; Duboko učenje; Optimizacije; Sparsifikacija; Kvantizacija; Tenzorska dekompozicija

# Abstract


## Development of efficient deep learning models for speech separation

Matej Božić

Deep learning models have become the foundation in the development of advance speech processing algorithms, achieving exceptional results in terms of accuracy and expressiveness. However, their complexity often limits real-time applications where speed and efficiency is key. This work is focused on the exploration and development of deep learning models optimized for speech separation tasks with an emphasis on reducing model complexity. Our goal is to develop a model that offers an optimal compromise between the speed of computation, number of trainable parameters, and memory usage, comparing them with the currently available solutions. Special attention will be dedicated to creating models will smaller number of learned parameters that retain high efficiency with improved execution speed.



**Keywords:** Speech separation; Deep learning; Optimizations; Pruning; Quantization; Tensor decomposition