TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Compression of Weights of Recurrent Neural Network for Speech Recognition Acceleration in Reconfigurable Hardware (FPGA)

Author:
Alexandros POUPAKIS

Thesis Committee:
Prof. Apostolos DOLLAS
Prof. Aggelos BLETSAS
Prof. Michail LAGOUDAKIS

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

February 22, 2023

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Compression of Weights of Recurrent Neural Network for Speech Recognition Acceleration in Reconfigurable Hardware (FPGA)

by Alexandros POUPAKIS

Over the last decades, advances in machine learning and neural networks have been unprecedented, with ever more sophisticated models trickling down the mainstream and forming the backbone of products and services we use every day. While cutting edge research in this field has expanded the realm of what is feasible, the learning superiority of deep neural networks is largely attributed to their size. Hardware acceleration of deep learning inference is necessary, for such models to be practically deployable. However, as model size increases rapidly, the available memory bandwidth on massively parallel computing platforms such as FPGAs is outpaced, constituting a bottleneck for scalability. This study addresses the problem of compressing deep neural network weights for inference acceleration on FPGAs. The DeepSpeech2 model for Speech Recognition is trained and used as a case study for weight pruning and quantization. The pièce de résistance of this thesis is the development of a novel compression method suitable for quantized weights, which is tested on the sparse matrices of DeepSpeech2. This method generates a tree of overlapping symbol sequences and uses it to encode the data with mathematically decodable, variable length codes. Importantly, our compression method inherently allows one to coarsely select the decompression throughput. Lastly, the decompressor's architecture is designed and a general model for its resource cost in UltraScale FPGAs is created. When compared against various LZ77-based decompressors in literature, our decompressor consumes more than an order of magnitude fewer logic resources, while being capable of the same or higher throughput.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Apostolos Dollas, for his support and guidance during my studies and this thesis. His continuing encouragement and grant of absolute freedom for scientific inquiry have been momentous to the work presented in this thesis and to areas explored, yet not included. On a personal level, he has been compassionate, empathetic, and sharing of his own experiences, exemplifying the qualities of a mentor. Collaborating with him has been as much a delight as it has been an inspiration, and I am grateful for his guidance and admiring of his character.

Furthermore, I would like to thank my thesis committee, Prof. Aggelos Bletsas and Prof. Michail Lagoudakis, for their time and evaluation of my work. The review and comments of Prof. Bletsas on the first paper from this thesis are much appreciated.

Additionally, I would like to thank Dr. Gregory Tsagkatakis and Dr. Christos Kozanitis, members of the CARV team at FORTH, for our many helpful discussions, their guidance, expertise, and comments on my work. I am especially thankful to Dr. Kozanitis for providing access to FORTH servers, on which the neural network was trained. I am also grateful to Mr. Kostas Kokkinos, who was the daily contact point at FORTH during the model's training and who ran the experiments on the server.

Most importantly, I would like to express my deepest thanks to my family and friends, who have been by my side and supported me in every step along the way. Your friendship is invaluable to me, and I hope I am as deserving of your appreciation as you all are of mine. I am grateful to have you in my life and, in recognition that no man is an island, I thank you all.

Alexandros Poupakis,
Chania, 2023

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ANN	**A**rtificial **N**eural **N**etwork
API	**A**pplication **P**rogramming **I**nterface
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
AVX	**A**dvanced **V**ector E**X**tentions
BRAM	**B**lock **R**andom **A**ccess **M**emory
BRNN	**B**idirectional **R**ecurrent **N**eural **N**etworks
CCE	**C**ategorical **C**ross **E**ntropy
CDF	**C**umulative **D**istribution **F**unction
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
CSC	**C**ompressed **S**parse **C**olumn
CSR	**C**ompressed **S**parse **R**ow
CTC	**C**onnectionist **T**emporal **C**lassification
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DAG	**D**irected **A**cyclical **G**raph
DDR	**D**ouble **D**ata **R**ate memory
DNN	**D**eep **N**eural **N**etwork
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
FC	**F**ully **C**onnected
FF	**F**lip **F**lop
FIFO	**F**irst **I**n **F**irst **O**ut buffer
FNN	**F**eedforward **N**eural **N**etwork
FORTH	**FO**undation for **R**esearch and **T**echnology **H**ellas
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FRNN	**F**ully **R**ecurrent **N**eural **N**etwork
GMP	**G**lobal **M**agnitude **P**runing
GPGPU	**G**eneral **P**urpose **G**raphics **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
GRU	**G**ated **R**ecurrent **U**nit

HBM	**H**igh **B**andwidth **M**emory
IC	**I**ntegrated **C**ircuit
LAS	**L**isten, **A**ttend and **S**pell
LMP	**L**ayer-wise **B**ased **P**runing
LM	**L**anguage **M**odel
LSTM	**L**ong-Short **T**erm **M**emory
LUT	**L**ook **U**p **T**able
MAC	**M**ultiply **AC**cumulate
MBP	**M**agnitude **B**ased **P**runing
MLP	**M**ulti**L**ayer **P**erceptron
ML	**M**achine **L**earning
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
MSE	**M**ean **S**quared **E**rror
MUX	**MU**ltiple**X**er
PATH	**P**enalty **A**rranged **T**ree **H**ierarchy
PDF	**P**robability **D**istribution **F**unction
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
RNN	**R**ecurrent **N**eural **N**etwork
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SSE	**S**treaming **S**IMD **E**xtensions
TPU	**T**ensor **P**rocessing **U**nit
USD	**U**nited **S**tates **D**ollar

Dedicated to my family and friends…

Chapter 1

Introduction

1.1 Motivation

The era of big data ushered in by the digital age has presented the scientific community with unique challenges in processing and analyzing the extraordinary amount of data generated on a daily basis. According to a Domo report, the 5 billion worldwide internet users in 2022 created, copied, and consumed an estimated 97 zettabytes in one year, which amounts to 265 billion GB every single day. Extracting information and knowledge from these vast amounts of data is crucial to maintain and advance all aspects of life supported by technology.

Deep neural networks have proven time and again their unparalleled capabilities in dealing with data at these scales and tackling increasingly difficult problems, with superhuman results. The most recent landmark achievements in the field of Artificial Intelligence, powered by deep neural networks, include AlphaFold, which is the highest-accuracy protein folding predictor, Tesla's Autopilot, providing driver assistance, and ChatGPT, possibly the most advanced chatbot in existence, which is even capable of producing software code when prompted with programming questions.

However, cutting-edge capabilities come at a significant computational cost, with model sizes increasing rapidly to meet our ever higher expectations. CPUs simply neither parallel nor power-efficient to keep up with the computation needs of neural networks. On the other hand, GPUs provide excellent parallel and power efficient computing capabilities, but their hard IPs do not allow for custom computing pipelines tailored to each application.

Reconfigurable computing platforms, such as FPGAs, offer hardware customizability which, sadly but inherently, yields much lower clock speeds and

reduced power efficiency compared to GPUs. Yet, the advantages of reconfigurability and custom architectures are too useful to overlook, which is why much research is focused on developing fast and efficient neural network inference accelerators on FPGAs.

Surprisingly enough, a roadblock in the progress of inference accelerators is the memory bandwidth. While FPGAs do provide massive parallelism and on-chip memory is split up into hundreds or thousands of modules which can all be accessed in parallel, external memory is the limiting factor. As the weight matrices of even mid-range neural networks do not fit in on-chip memory, they must be stored in external RAM which is orders of magnitude slower and more powerhungry. Thus, addressing the memory problem of neural network inference accelerators is of paramount importance to their success.

Much research exists in literature regarding the compression of neural networks. Network compression focuses on techniques which reduce the number of weights necessary for a given network, thereby reducing its computational cost and the memory footprint of said weights on external memory. While these techniques are highly beneficial in alleviating the bandwidth problem, very few works extend past them and adopt data compression methods as well, to further reduce the footprint of weight matrices.

This work, in recognition of the potential of data compression in inference accelerators, extends past network compression techniques and addresses weight matrix compression specifically for reconfigurable hardware. Target devices and applications include any machine learning inference accelerator on reconfigurable hardware, from high-performance computing systems like datacenters, to low-end, low-power, and low resource utilization inference on edge devices.

1.2 Scientific Contributions

This thesis develops a novel compression method addressing the needs of quantized weight matrices in inference applications. It is suitable for compressing static files in applications where compression is needed only once, while decompression is needed continuously. The method splits the data into constant-length sequences, which due to the probability distribution of quantized weight magnitudes, will exhibit significant overlap. This overlap

is exploited to map the sequences onto a tree whose structure has been specifically designed to provide mathematically determinable regions of different codeword lengths. The resultant variable-length encodings are necessary to achieve reasonable compression ratios, while the mathematical decodability of the penalty regions allows for fast and low-cost decoding. More importantly, the method inherently provides the ability to select the decompression throughput, provided the sequences have sufficient length to support it. Our method provides higher throughput compared to Huffman coding and lower decoding complexity compared to LZ77-based methods.

The hardware architecture for the decompressor is also designed and evaluated for implementation in FPGAs. By using the inherent properties of the method itself, as well as our in-depth understanding of the target FPGA's architecture, we were able to design a decompression pipeline with low logic resource utilization, high clock speeds, and high throughput. A generalized model to calculate the resource cost of the decompressor's implementation for any configuration is also provided, and, through it, we show our decompressor requires an order of magnitude fewer logic resources compared to LZ77-based decompressors. The structures used in the architecture appeal to common characteristics of FPGAs, not a specific platform, for the methodology to be relevant in the long run.

Training of the DeepSpeech2 speech recognition neural network is carried out to provide us with a case-study model. Robustness analysis is performed on the trained model to examine the effects and limits of post-training network compression techniques. The factors used to evaluate these methods were the degradation of the model's accuracy, the reduction of the number of weights and their binary representation, and the increase of inference complexity. The pruning and quantization applied to the network yield the sparse matrices used in the evaluation of our compression method.

The quantitative contributions are summed up below.

- 7x reduction of GRU kernels' memory footprint due to network compression
- a further 1.6x reduction due to weight compression using our method
- a total of 11x reduction in memory, and therefore bandwidth, compared to the baseline model

- overall file sizes within 9.5% of the entropy limit for the given weight matrices
- low-cost decompressor architecture, on the order of 1K LUTs
- constant decoding duration per packet
- 2x the decompression throughput for 25% more LUTs using the forked decoder pipeline
- 4-65x lower logic resource utilization compared to other decompressors for the same hypothetical throughput

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** The theoretical background of Machine Learning, specifically of various Neural Network types, and Data Compression is described.
- **Chapter 3 - Related Work:** The literature around automatic speech recognition and neural network compression is presented, as well as the available frameworks and platforms for neural network development and deployment.
- **Chapter 4 - Training and Robustness Analysis:** The DeepSpeech2 model is explained in-depth and trained. Then, robustness analysis is performed, wherein the model's weights are pruned and quantized to reduce their memory footprint.
- **Chapter 5 - Compression:** The main contribution of this thesis, namely the novel compression method suitable for quantized weights and FPGA implementation, is developed and analyzed. Also, the sparse quantized weight matrices of DeepSpeech2 are compressed with this method and the results are compared against the entropy limit.
- **Chapter 6 - Hardware Architecture:** The hardware architecture for our decompressor is designed and explained. A model for the architecture's resource utilization is also developed, and our decompressor is compared with the literature, using said model.
- **Chapter 7 - Conclusions and Future Work:** This thesis is being concluded and directions for future research are provided.

Chapter 2

Theoretical Background

This chapter presents the necessary theoretical background on Neural Networks and Data Compression.

2.1 Machine Learning

Machine Learning (ML) is a term coined by Arthur Samuel in 1959 [1] and refers to the subfield of Artificial Intelligence (AI) which employs sophisticated computational models that adapt to a given dataset. ML models generate predictions or decisions based on the input stimuli and training algorithms gradually adjust the models, so their outputs approach some ground truth or improve some metric.

Described as "Software 2.0" by Andrej Karpathy [2], ML differs from traditional software, or "Software 1.0", in many respects, most notable of which is the fact that models are not explicitly programmed to perform the desired task. Instead, their behavior is data-driven and the task of adapting that behavior is automated, requiring minimal to no human interaction. Thus, ML models are notably versatile and able to tackle highly complex problems, which would otherwise be out of reach with traditional software methodologies.

In order to perform a given task, these models must first be trained. At the time of writing, the most widespread types of ML paradigms, with respect to what is being learned, are the following:

- **Supervised Learning**: A "labeled" dataset is provided, which consists of inputs and their respective outputs. The model tries to learn the relationship between the given inputs and outputs and generalize that behavior to stimuli not included in the dataset, i.e. predict their output.

- **Unsupervised Learning**: An "unlabeled" dataset is provided, containing only the inputs. The model's goal is to separate the inputs into groups, called "clusters" and assign new stimuli to some cluster.
- **Reinforcement Learning**: Unlike previous paradigms, this one does not rely on datasets. Instead, a set of rules evaluate the model's actions in some environment and the goal is for the model to exhibit behavior that maximizes the evaluation score.

There are various model types, differing in complexity, capabilities, and application domain. The most notable of these types are the following:

- **Decision Trees**
- **Support Vector Machines**
- **Bayesian Networks**
- **Artificial Neural Networks**

An ode to the importance of ML as a discipline is its success and massive adoption in industry. For 2021, Fortune Business Insights reports the global ML market was valued at USD 15.44 billion and is expected to reach USD 209.9 billion by 2029. Some industries adopting the technology include, but are not limited to, Information Technologies, Banking, Automotive, Healthcare, and Advertisement [3].

2.2 Artificial Neural Networks

As the name suggests, Artificial Neural Networks (ANNs) mimic the natural neural networks of the brain. Given that natural brains are exceptionally efficient and capable of multimodal pattern recognition, the 21st century is a race of innovation towards replicating and harnessing those capabilities.

The building block of any neural network is the neuron and its artificial counterpart is called perceptron. Both structures accept input signals, process them and then transmit the output further down the network. Figures 2.1 and 2.2 illustrate the two structures and their resemblance. Equation (2.1) is the mathematical equivalent of Figure 2.2.

$$\text{output} = f\left(\sum_{i=1}^{n} x_i \cdot w_i\right) \tag{2.1}$$

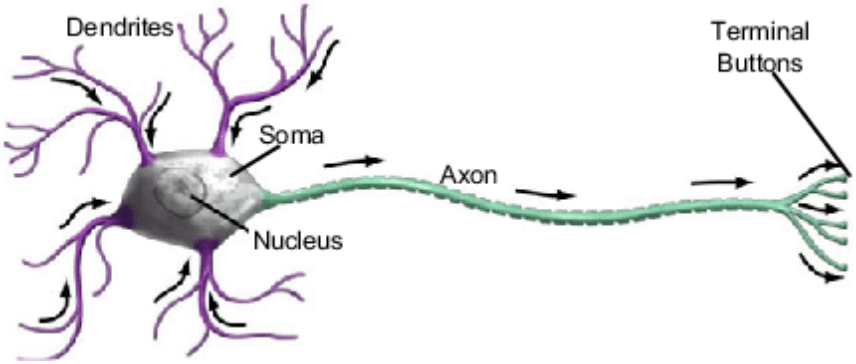


FIGURE 2.1: Biological neuron and signal flow (source: URL)

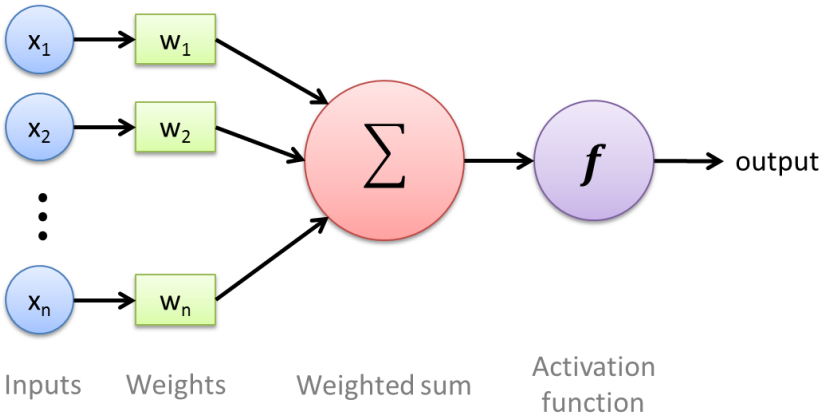


FIGURE 2.2: Artificial neuron - Perceptron

The components that - mostly - define a Perceptron are its weights and the activation function. The weights, or parameters, must be "learned" for the specific network and dataset. The activation function provides a non-linearity between the input and the output, which is crucial for any non-trivial problem. Without activation functions, every ANN would simply perform a linear transformation of the input data, regardless of the network's size. There are various activation functions, some of which are illustrated in Figure 2.3.

Of course, natural brains - and the human one in particular - are not a homogenous network of neurons. Instead, it is a network of subnetworks, each tuned to the specific task they aim to solve. While humanity is far from understanding the intricacies of natural brains, it is widely accepted scientific knowledge that tasks as familiar as object recognition are handled hierarchically by multiple cascaded networks.

To achieve the desired variety, many types of ANNs have been proposed

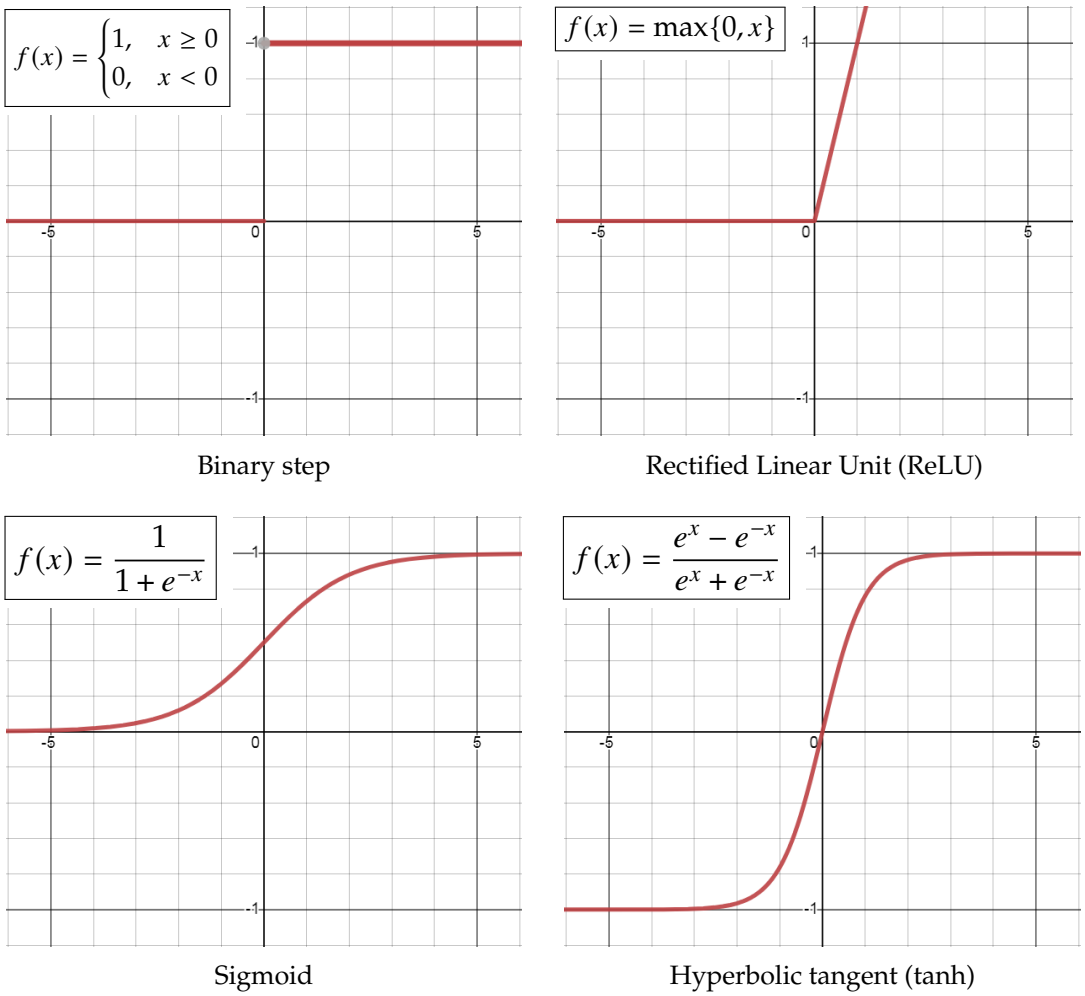


FIGURE 2.3: Activation functions

over the years, each of them targeting a specific subset of pattern recognition applications. The most relevant network types are described in the following sections.

2.2.1 Feedforward Neural Networks

Networks which do not contain feedback loops and, hence, can be represented as Directed Acyclical Graphs (DAGs), are called Feedforward Neural Networks (FNNs). In these networks, the output is independent of time and is only a function of the inputs.

Interestingly enough, multilayer FNNs have been proven to be universal approximators [4]. Any continuous function of finite dimensionality can be approximated by FNNs of arbitrary width, i.e. networks with one hidden layer with an arbitrary number of neurons, or arbitrary depth, i.e. networks

with an arbitrary number of hidden layers, each with a limited number of neurons.

Multilayer Perceptron

The simplest ANN is created by "vertically" stacking perceptrons to create a layer and then "horizontally" stacking layers to create the full network, called Multilayer Perceptron (MLP). Each neuron of one layer is connected to every neuron on the next layer. These networks are alternatively called Fully Connected (FC) or Dense Networks.

These networks are structure agnostic, meaning that they do not assume anything about the target problem and the inputs. As such, they are more general, but harder to train due to the high number of parameters. They are typically used as the final processing step in Deep Neural Network (DNN) architectures.

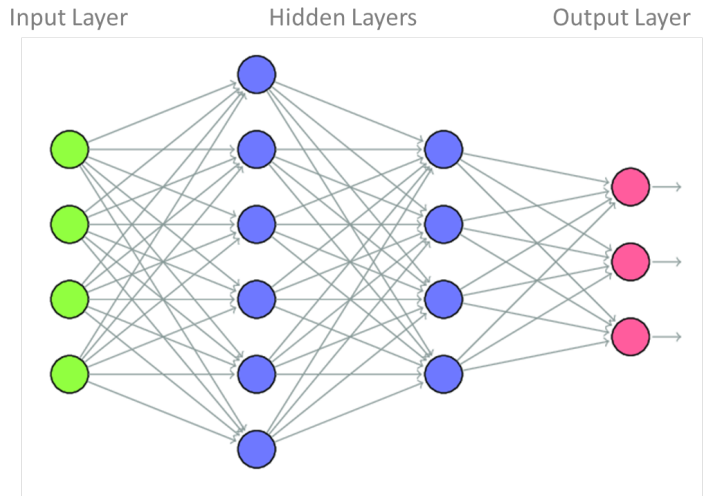


FIGURE 2.4: Multilayer Perceptron Network (source: URL)

By extending equation (2.1), an MLP can be mathematically expressed as

$$\begin{aligned} \mathbf{h}_1 &= f_1\left(\mathbf{x}\mathbf{W}_{ih_1} + \mathbf{b}_{h_1}\right) \\ \mathbf{h}_i &= f_i\left(\mathbf{h}_{i-1}\mathbf{W}_{h_{i-1}h_i} + \mathbf{b}_{h_i}\right), \qquad 1 \leq i \leq k \\ \mathbf{o} &= g\left(\mathbf{h}_k\mathbf{W}_{h_k o} + \mathbf{b}_o\right) \end{aligned} \tag{2.2}$$

where $\mathbf{x} \in \mathbb{R}^{1\times n}$ is the input vector, $\mathbf{h}_i$ are the hidden layer activation vectors, and $\mathbf{o} \in \mathbb{R}^{1\times m}$ is the output vector. Matrices $\mathbf{W}_{ih_1}, \mathbf{W}_{h_{i-1}h_i}, \mathbf{W}_{h_k o}$ hold the

weights between the designated layers and the vectors $\mathbf{b}_{h_i}$ and $\mathbf{b}_o$ are called biases and facilitate the network's learning.

Convolutional Neural Networks

Drawing inspiration from the receptive fields in the visual cortex, Convolutional Neural Networks (CNNs) can exploit hierarchical patterns in the input data, by extracting increasingly more complex features. This is achieved by applying filters - or kernels - on the data, via convolution. The result of this operation, namely the convolution layer, is a set of feature maps which can be downsampled and filtered again.

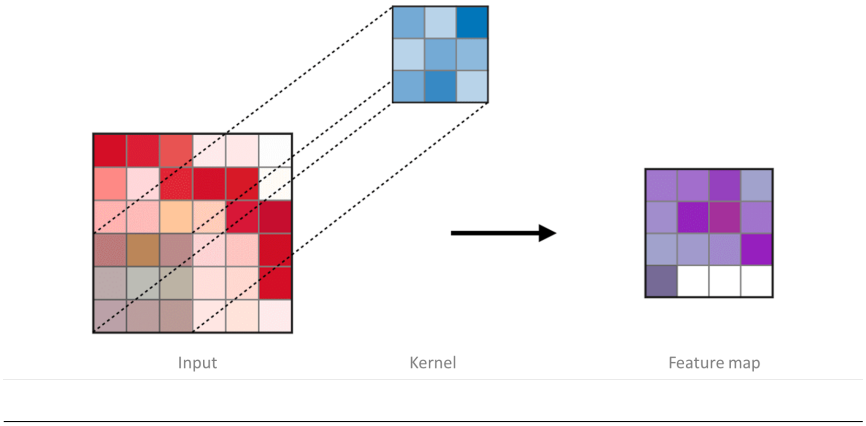


FIGURE 2.5: CNN kernel (source: URL)

Filters are tensors that sample part of the input data, across all feature maps - or channels - and output the weighted sum after it's fed through an activation function. Essentially, filters are a special case of MLPs, with a single output unit and no hidden neurons. The purpose of each filter is to learn some feature of the input data through its weights.

Succeeding the convolution layer is - typically - a pooling layer, which downsamples the feature maps for a further data size reduction. The chosen pooling filter is applied separately on each channel, so that the resultant tensor has the same number of channels as the starting tensor, but of smaller size. Common pooling filters are the *max* and *average* operations, where the max and average value inside the kernel is chosen respectively.

The convolution and pooling layers comprise the feature extraction section of a CNN architecture, which is followed by the classification section. The classification subnetwork is a fully connected network with one output neuron per classification category, representing the network's confidence of the

given category existing in the original image. Figure 2.6 illustrates a conventional CNN architecture.

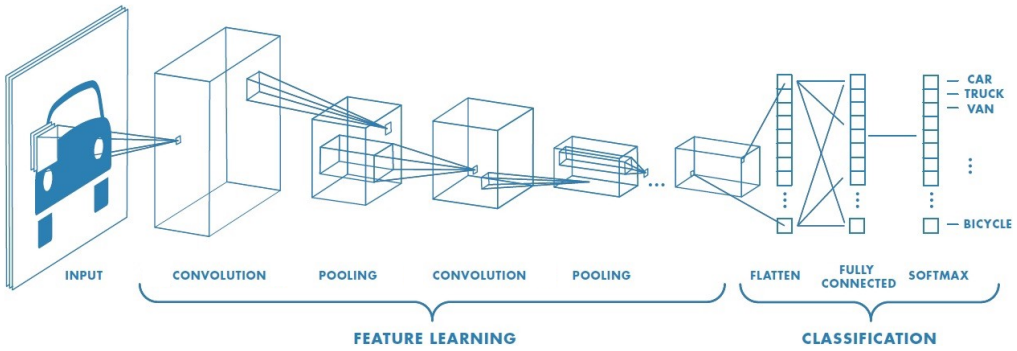


FIGURE 2.6: CNN architecture example (source: URL)

2.2.2 Recurrent Neural Networks

While feedforward architectures excel in processing fixed size data, many applications require the processing of time series of variable, unknown, or infinite length. In such cases, feedforward architectures are simply not applicable, since the input data size must be constant and bounded by the current computational capabilities. Instead, Recurrent Neural Networks (RNNs) are used, which can process time series of any length, with a fixed network size.

To achieve this, RNNs - as the name implies - contain a feedback loop. If we regard the input to a Network as an entry of a time series at a discrete time step, then the notion of time - in relative or absolute terms - can be transferred to the Network's outputs. Therefore, RNNs can coherently be described as networks where some form of the output at any time step is used to process the input at the succeeding time step. In other words, RNNs have an internal state, which affects and is affected by the processing of the time series' data. Note, however, that the state is relevant only during each distinct time series and is reset to some initial state between time series.

This simple feature makes RNNs suitable for tackling problems expressible as timeseries, such as machine translation, time series prediction, speech recognition, sentiment analysis, handwriting recognition, music composition, and many more. Indeed, RNNs have improved these fields, notably speech recognition [5], machine translation [6], video prediction [7] and are being used in cutting-edge industry applications, as in Google voice search [8, 9]. The most notable variants of RNNs are discussed below.

Fully Recurrent Neural Networks

Analogously to MLPs, Fully Recurrent Neural Networks (FRNNs) are the simplest and most general type of RNNs. Their simplicity is attributed to their minimal mathematical complexity and their generality on the fact that other RNN types can be represented by an FRNN of appropriate size and with appropriate weight matrices. These characteristics, however, are also the reasons why other RNNs outperform FRNNs.

The mathematical description of an FRNN cell is quite similar to that of the MLP with a single hidden layer and is presented in equation (2.3). The cell is illustrated in figure 2.7. Note that a cell contains an entire hidden layer and - perhaps counterintuitively - is not just a single neuron. Of course, multiple hidden recurrent layers can be added by stacking cells vertically.

$$\begin{aligned} \mathbf{h}_t &= f\left(\mathbf{x}_t \mathbf{W}_{xh} + \mathbf{h}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h\right) \\ \mathbf{o}_t &= g\left(\mathbf{h}_t \mathbf{W}_{ho} + \mathbf{b}_o\right) \end{aligned} \tag{2.3}$$

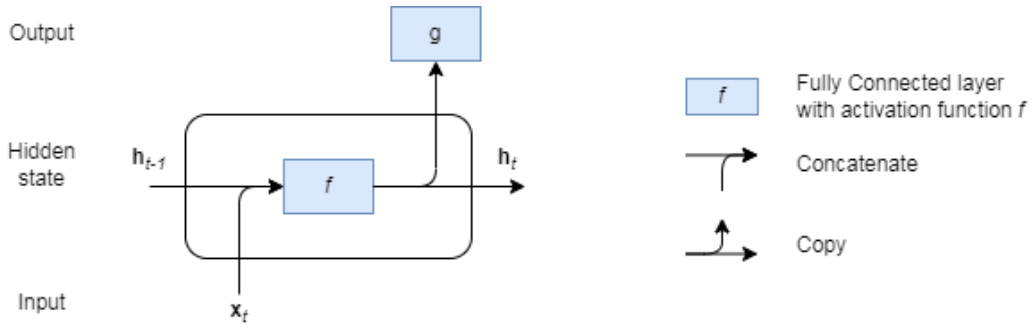


FIGURE 2.7: FRNN cell and minimal network

A recurrent theme in these networks - no pun intended - is the shorthand form of the sum of the vector-matrix products for the hidden state update, shown in figure 2.7. Instead of the expanded sum of equation (2.3), the same expression can be compactly written by concatenating the input and hidden vectors, and the two weight matrices, as shown in equation (2.4). The concatenated form has the advantage of a more succinct illustration, compared to the expanded form, which will prove particularly useful later on, in the more complicated cells.

$$\begin{aligned} \mathbf{h}_t &= f\left([\mathbf{x}_t \;\; \mathbf{h}_{t-1}] \begin{bmatrix} \mathbf{W}_{xh} \\ \mathbf{W}_{hh} \end{bmatrix} + \mathbf{b}_h \right) \\ \mathbf{o}_t &= g\left(\mathbf{h}_t \mathbf{W}_{ho} + \mathbf{b}_o \right) \end{aligned} \tag{2.4}$$

These networks, for their simplicity, can handle short-term dependencies, but do not fare nearly as well on long-term dependencies. The reason for this is mainly due to the network's training. While the network learns with the backpropagation algorithm, it corrects its weights based on the difference of the prediction vs the ground truth. However, the more recurrent layers in the architecture and the lengthier the dependencies, the smaller the weight corrections will be since each hidden layer at each time step contributes less and less to the overall output. This is the infamous vanishing gradient problem [10].

Long-Short Term Memory

To directly address this learning deficiency, a new and more complicated recurrent cell was developed in 1997, by Josef Hochreiter and Jürgen Schmidhuber [11]. In addition to the hidden state, the Long-Short Term Memory (LSTM) cell contains a memory component, which is engineered to retain additional information.

The memory component is controlled by three gates. The *output gate* selects which hidden state entries to read. The *input gate* decides when to read data into memory. Lastly, the *forget gate* resets the memory contents. It is evident that this approach is much more detailed and structured than the vanilla FRNN cell, hence the improved learning capabilities.

The internal structure of the LSTM cell is illustrated in figure 2.8, with its mathematical counterpart shown in equation (2.5). Note that σ denotes the sigmoid function and *tanh* the hyperbolic tangent function. As in the FRNN, only the hidden state would be used in a subsequent layer. The memory component does not directly participate in any computation outside its cell.

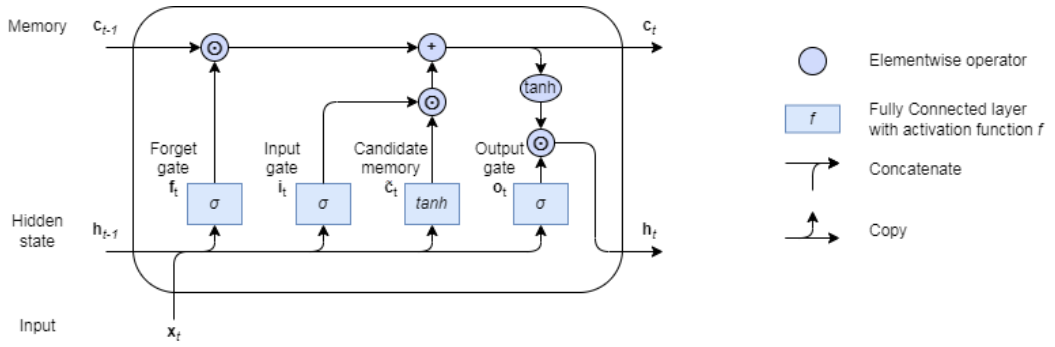


FIGURE 2.8: LSTM cell

$$
\begin{aligned}
\mathbf{i}_t &= \sigma\left(\mathbf{x}_t\mathbf{W}_{xi} + \mathbf{h}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i\right) \\
\mathbf{f}_t &= \sigma\left(\mathbf{x}_t\mathbf{W}_{xf} + \mathbf{h}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f\right) \\
\mathbf{o}_t &= \sigma\left(\mathbf{x}_t\mathbf{W}_{xo} + \mathbf{h}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o\right) \\
\tilde{\mathbf{c}}_t &= \tanh\left(\mathbf{x}_t\mathbf{W}_{xc} + \mathbf{h}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c\right) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh\left(\mathbf{c}_t\right)
\end{aligned}
\tag{2.5}
$$

The vanishing gradient problem is directly addressed by the computation of $\mathbf{c}_t$. If the forget gate is close to 1 and the input gate is close to 0, then the past memory contents, namely $\mathbf{c}_{t-1}$, will be retained. This way, past information is saved through time and thus the network can remember long range dependencies.

Gated Recurrent Unit

The superior performance of the LSTM cell gained increasing popularity, but its heavy computational cost was far less appealing. With the conception of the Gated Recurrent Unit (GRU) [12], comparable performance could be attained at a significantly smaller cost [13].

The GRU cell has one fewer gate compared to the LSTM and their functionality is different. The *reset gate* controls how much of the previous state is remembered, and the *update gate* controls how much of the new state is a copy of the old state. As for the result of the *tanh* fully connected layer, it is called *candidate* because it only incorporates the reset gate.

Similarly to the LSTM, the vanishing gradient problem is addressed in the GRU by the update gate. When $\mathbf{z}_t$ is close to 1, the old state is retained, while

when it is close to 0, the candidate state is passed through. Thus, relevant information can be saved across lengthy subsequences, capturing long term dependencies.

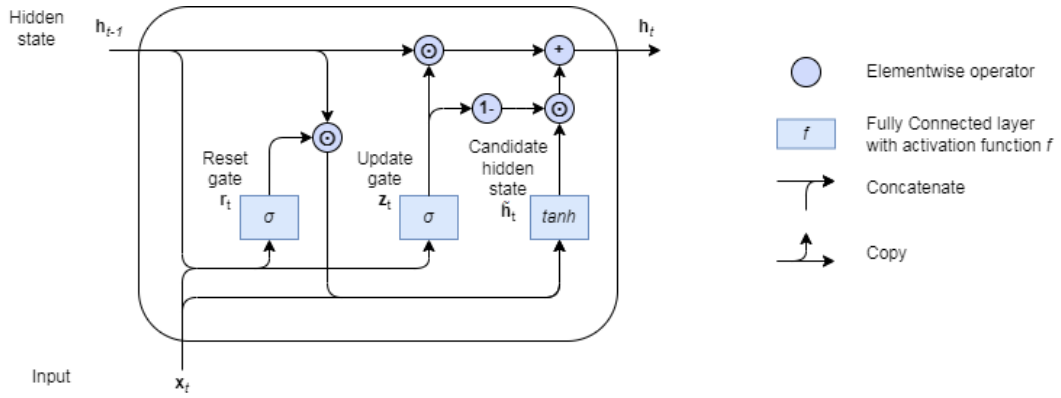


FIGURE 2.9: GRU cell

$$
\begin{aligned}
\mathbf{r}_t &= \sigma\left(\mathbf{x}_t \mathbf{W}_{xr} + \mathbf{h}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r\right) \\
\mathbf{z}_t &= \sigma\left(\mathbf{x}_t \mathbf{W}_{xz} + \mathbf{h}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z\right) \\
\tilde{\mathbf{h}}_t &= \tanh\left(\mathbf{x}_t \mathbf{W}_{xh} + \left(\mathbf{r}_t \odot \mathbf{h}_{t-1}\right) \mathbf{W}_{hh} + \mathbf{b}_h\right) \\
\mathbf{h}_t &= \mathbf{z}_t \odot \mathbf{h}_{t-1} + \left(1 - \mathbf{z}_t\right) \odot \tilde{\mathbf{h}}_t
\end{aligned}
\tag{2.6}
$$

Bidirectional Recurrent Neural Networks

So far, we've explored feedforward networks which learn spatial data dependencies and recurrent networks, which learn both spatial and temporal data dependencies. While the spatial dependencies are direction-agnostic, the temporal dependencies are learned on a physical-time basis, meaning that only past samples affect the current one. One could be forgiven for thinking that this one-directional approach of time is the only choice, but it is not.

In some real-world applications, the "future" is inherently unknown. Stock market prediction, for example, revolves entirely around the fact that we want to predict the price of an asset, in a setting where that price is impossible to know beforehand. In other applications, however, the notion of physical time is less constraining and can be adapted to suit our models. A key example of this is any speech-related application, like speech recognition.

The problem of speech recognition, i.e. converting speech audio to text, is hard for the simple reason that a single letter can be pronounced in various ways, depending on the particular word (e.g. the letter "a" in "act", "ant",

"any") and, inversely, a particular sound can map to different letters (e.g. "end", "and" or "ill", "eel"). Therefore, in such problems, it is extremely useful to know both the preceding and succeeding processed samples of the currently examined sample, so the samples should be processed bidirectionally.

To that end, Bidirectional Recurrent Neural Networks (BRNNs) process their sequence of input samples from both directions, at the same time [14]. This is achieved via dedicated hidden recurrent layers for each direction. Where in traditional networks, a hidden recurrent layer would be comprised of a single RNN cell, in BRNNs it contains two cells, one for each processing direction. The outputs of the two cells are concatenated together and fed to the next layer of the network. The architecture is illustrated in figure 2.10.

An important detail is the dimensionality of the output. In the cells presented so far, the size of the input and hidden state vectors is a design choice and the output, being just the hidden state, has the same size as the hidden state. In BRNNs, the dimensionality of the input and hidden state vectors is - again - a design choice, but the output's size is the sum of the sizes of the forward and backward hidden states. Naturally, as in any deep RNN, when multiple recurrent layers are stacked together, the input size of one layer is entirely determined by the output size of the previous layer.

Generally, with $\mathbf{x}_t \in \mathbb{R}^{1\times n}$, $\overrightarrow{\mathbf{h}}_t \in \mathbb{R}^{1\times h_f}$, and $\overleftarrow{\mathbf{h}}_t \in \mathbb{R}^{1\times h_b}$, the output would be $\mathbf{h}_t \in \mathbb{R}^{1\times h_f+h_b}$. In practice, the forward and backward cells are identical in design parameters and differ only in their weight matrices and bias vectors, which are not shared between directions. So, for $h_f = h_b = h$, the output would be in $\mathbb{R}^{1\times 2h}$. Of course, the input size does not influence the output size, but both n and h define the size of the cell's weights and biases.

2.2.3 Loss function and training algorithms

Having presented the various network types and topologies, we will now explore how neural networks learn their tasks. The applicable machine learning paradigm in this thesis is supervised learning, so the context hence forth will be the existence of a labeled dataset. Thus, the process of learning translates to finding appropriate network parameters (weights and biases) such that the total discrepancy between the predicted outputs and the ground truths of the labeled inputs is minimized across the entire dataset.

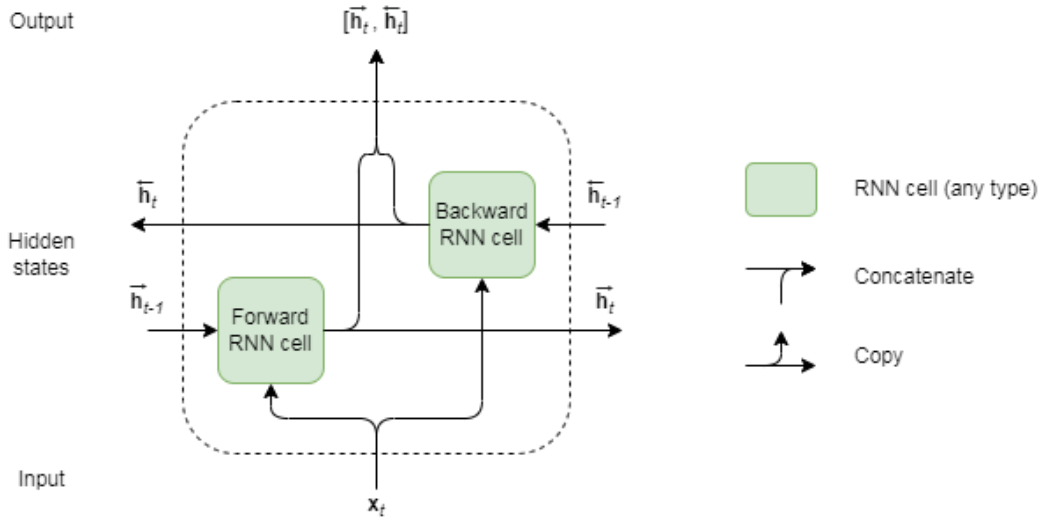


FIGURE 2.10: BRNN hidden layer

Since it is not possible to analytically solve for the network's parameters, learning is approached iteratively, updating the parameters in every iteration and gradually reducing the overall discrepancy. To achieve this, three components are needed: (i) a loss function to calculate the discrepancy, (ii) a way to determine how much each individual parameter contributes to the loss function, and (iii) a parameter update rule which minimizes the loss function.

The later two components will be briefly discussed first. The contribution of each parameter to the loss function is most commonly determined by the backpropagation algorithm. This algorithm uses the rules of calculus to compute the gradient of the loss function with respect to each network parameter. backpropagation is used in feedforward networks, while for recurrent ones a similar algorithm exists, called backpropagation through time [15, 16, 17].

While backpropagation is a fair algorithm, assigning blame to each weight based on its contribution to the overall error and then updating the weights proportionally to that blame, there exists algorithms with a different approach altogether. Perhaps unintuitively, a network can learn, or rather, learn how to learn, by assigning blame proportional to random, fixed values, rather than the actual weights [18]. This idea is extended by the Direct Feedback Alignment and Indirect Feedback Alignment methods [19].

As for the parameter update rule, the most basic optimization algorithm is gradient descent. Gradient descent is a first-order iterative algorithm which finds a local minimum of a differentiable function, by following steps along

the direction of the negative gradient. In practice, extended optimizers are used instead of the vanilla gradient descent, due to their superior performance. Such optimizers are Momentum [20], Nesterov accelerated gradient [21], RMSprop, and Adam [21], to name a few.

All of these optimizers are first-order methods, but second-order methods also exist in literature. These methods use the Hessian matrix, or some estimation thereof, to incorporate information about the curvature of the loss function in their update steps, thus converging faster. Some of these methods are the Newton method, Conjugate gradient, Quasi-Newton method, Levenberg-Marquardt algorithm [22]. However, these methods are beyond the scope of this thesis, so we move on to discuss some notable loss functions.

Mean Squared Error

Also known as quadratic or L2 loss, Mean Squared Error (MSE) is one of the simplest and most basic loss functions. This function is used in regression problems where outputs are real-valued numbers. Note that y_i denotes the ground truth of the respective training example and $\hat{y}_i$ denotes the network's prediction.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.7}$$

As in other loss functions, the average sum of the individual deviations is used to reduce the influence of individual training examples on the network's parameters. This technique is called mini-batching and involves the processing of a small batch of n examples through the network and the mean error is used to update the network parameters once per batch, instead of once per example. This results in more efficient and faster convergence - i.e. learning.

Categorical Cross Entropy

In multiclass classification scenarios, the output of the network is a vector of probabilities denoting the confidence of the given input belonging to each class. Categorical Cross Entropy (CCE) evaluates the loss of the entire output vector, across m classes. Similarly as before, y_{ij} denotes the j-th element of the ground truth vector of the i-th example and $\hat{y}_{ij}$ denotes the respective prediction.

$$CCE = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\log(\hat{y}_{ij}) \tag{2.8}$$

Interestingly, since multiclass classification admits only a single class per input, the ground truth vector will exhibit one-hot encoding, whereby a single element will be 1 and all others will be 0. It follows that, regardless of the prediction vector $\hat{y}_i$, the internal sum will be reduced to a single term, i.e. $\log \hat{y}_{ik}$, where k is the index of the non-zero element of y_i.

Connectionist Temporal Classification

The loss functions presented so far address singular outputs, be it scalars or vectors. However, many interesting problems are far more complex and deal with output sequences, where the individual outputs of multiple input samples form the encoded prediction. In these scenarios, a different type of loss function is needed.

Suppose a RNN is tackling the problem of handwriting recognition by taking segments of the image and outputting character probabilities. An example of a series of inputs is shown in figure 2.11, where the red lines show the segments of the image processed individually by the network. For each segment, the network would yield a vector of probabilities for each possible character, and the most likely prediction is also shown below each segment.

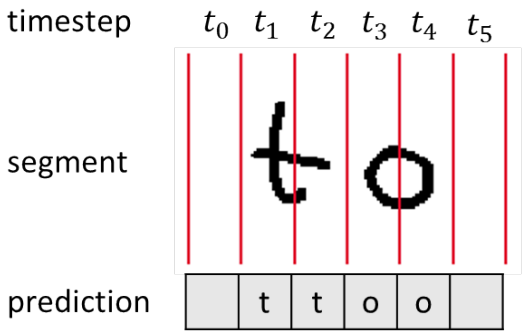


FIGURE 2.11: Handwritten text - input example

The apparent problem is the prediction of the same letter when it spans multiple segments. This cannot be overcome by simply ignoring repeating predictions, because some of them may result from repeating letters in the image (e.g. the word "to" vs "too"). Instead, Connectionist Temporal Classification (CTC) [23] works with the entire prediction matrix and the ground truth text

and calculates the score for any alignment between the two. To distinguish between repeating characters, a new "blank" character is introduced.

The blank character is not to be confused with whitespace. The whitespace character belongs to the set of possible, depicted characters, while the blank character, here denoted by "-", is an artificial separator. Its use is to separate runs of collapsible characters, such as the "oo" in figure 2.11. With this encoding scheme, multiple prediction sequences can map to the same output sequence, such as "-too", "tt-o", "too-" all mapping to the word "to".

Due to the many-to-one relation between the encoded and output sequences, the loss function must take all of them into account. Once the prediction matrix for a given number of timesteps is generated, the value of the CTC function for a ground truth text is computed as the sum of the probability of every possible prediction sequence mapping to the given ground truth text. In turn, the probability of each prediction sequence is given by the product of the probabilities of its components. For our minimal, toy example, an illustration is given in figure 2.12. The path "-too" is shown via the arrows.

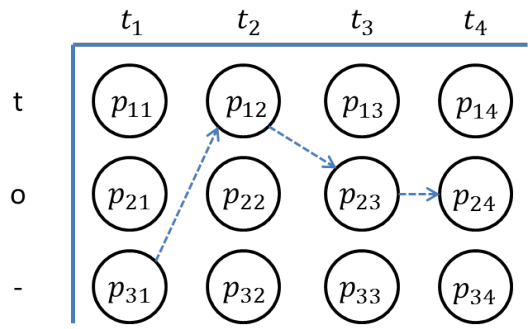


FIGURE 2.12: CTC example

Obviously, a probability matrix for n characters and m timesteps contains n^m paths. Evaluating all of them becomes practically infeasible even for research studies, let alone real world applications. However, this computational cost is circumvented during both training and inference, albeit in different ways. During training, the ground truth text is known, so computing all paths that generate it is sufficiently easy and practical via dynamic programming [23]. But, during inference, a decoding algorithm is needed.

The CTC decoding algorithm attempts to select an output sequence based on some approximation of its probability. There are three well-known decoding algorithms in literature, of increasing complexity and accuracy:

- **Best path decoding / Greedy decoding** [23]: At each timestep, the most likely character is selected.
- **Beam search decoding** [24]: A predefined number of beams is kept at each timestep, denoting the best text candidates. At each timestep, every beam is extended by every possible character, the scores of paths representing the same output text are merged together and the resultant best scoring beams are kept. It is possible to implement a character-pair language model with this decoding algorithm, to further improve its accuracy.
- **Word beam search decoding** [25]: An extension of the vanilla beam search decoding. This algorithm assigns a *word* or *non-word* state to each beam. Transitions between the states happens based on the given extension character, which can be either a word (letter) or non-word (symbols and numbers) character. When in word-state, only characters that will eventually form valid words are allowed, based on a prefix tree. In its simplest form, this algorithm uses only a dictionary for the valid words, but word-level language models can also be incorporated.

2.3 Data Compression

Data compression is the subfield of information theory which attempts to transform information such that the resultant representation is condensed. In our digital world, this process is expressed in terms of binary digits, i.e. bits, but from a theoretical standpoint, any set of elementary units of information can be used as the basis for representing information. This set is called *alphabet* and each information unit is called *symbol*.

Decoupling our understanding of the basics of compression from the underlying binary nature of real-world systems is crucial both for understanding the subject in general and the nuances of this thesis. After all, the binary system is just one of infinite possible alphabets [26]. Abstraction of the alphabets for compression is as fundamental as abstraction of the number basis for arithmetic.

The goal of compression, i.e. reducing the size of a data file, is practically useful because of the inherent, natural relation between fundamental resources and the size of the data. Physical laws dictate that the energy, time, space, and materials needed to store or transmit data increase as the size of the data

increases. Thus, compression directly translates to cost reduction, monetary or otherwise. However, the complexity and requirements of the compression and decompression method must also be taken into account.

Compression is related to machine learning beyond the utilitarian goal of reducing the model's memory footprint. Machine learning models attempt to predict an output given a sequence of inputs. But, a system which can predict the posterior probabilities of a sequence given its history is, by definition, an optimal compressor. Also, an optimal compressor can act as a predictor, by selecting the best symbol given the sequence history. This duality indicates a deeper, fundamental connection between compression and artificial intelligence [27].

Indeed, research has demonstrated that compression can be used to tackle machine learning problems, in various applications, including text mining [28], text categorization [29], and genome analysis [30]. Notably, it has been proven that finding the optimal behavior of a rational agent is equivalent to compressing its observations [31].

2.3.1 Lossless vs lossy compression

Compression methods are categorized - mainly - by the difference of the original data and the decompressed data. Lossless compression includes all schemes for which the decompressed data is exactly the same as the original. Lossy compression includes all other schemes, for which the decompressed data differs from the original. Choosing between lossless or lossy methods comes down to the specific application and its requirements.

Usually, lossy methods are applied in cases where human perception is the only metric for data quality. Image, audio, and video applications are the most common use-cases of lossy compression. In these applications, the inherent information saturation points of the human sensory system can be leveraged, by reducing the quality of the data without significantly affecting our perception of it. Of course, lossy compression is applied to other cases as well, but the underlying rational of the cost-benefit tradeoff is common across all of them. In any case, the major advantage of lossy compression is the significantly reduced file sizes, especially compared to lossless compression.

A landmark statement regarding lossless compression is that no such method can compress all possible input files to a smaller size. In other words, for

every lossless method, there exists inputs for which the resultant file will be larger. Proving this statement is surprisingly easy by contradiction.

Given an original file size of N bits, there are 2^N possible input files. Suppose all of these files can be compressed to K bits, with $K < N$ (K does not have to be the same for every file). There are $\sum_{K=1}^{N-1} 2^K = 2^N - 1$ possible compressed files. But then, at least one compressed file would map to two distinct input files, meaning that the scheme is either irreversible or not lossless. This contradicts our hypothesis, therefore the original statement is true.

Being an active research topic for decades, multiple compression algorithms exist, serving different purposes, catering to different assumptions. Some notable lossless compression algorithms include, but are not limited to:

- Run-Length encoding
- Huffman coding
- Lempel-Ziv-Welch
- Arithmetic coding

Some notable classes of lossy compression algorithms include, but are not limited to:

- Transform coding
- Discrete cosine transform
- Discrete wavelet transform
- Fractal compression

2.3.2 Entropy

Compression methods are not developed for individual data files, but instead are general and suitable for entire classes of data files, which exhibit certain characteristics. The mathematical tool of choice for describing such classes of files is probability theory and their characteristics are expressed as statistical properties via the Probability Distribution Function (PDF).

If we assume a simple, symbol-serial communication system where a source generates symbols based on a given PDF, then this source is mathematically represented by an independent and identically distributed discrete random variable. In 1948, Claude Shannon established a metric to quantify the rate of

produced information per generated sample of the discrete random variable. Entropy is defined as

$$H(X) = -\sum_{x \in \mathcal{X}} P(x) \log P(x) \tag{2.9}$$

where $\mathcal{X}$ is an alphabet and X is the discrete random variable distributed according to $P : \mathcal{X} \to [0, 1]$ [32]. Using base-2 logarithm, entropy represents the average binary codeword length per symbol. For n samples of X, the expected value for the total produced information is $nH(X)$.

A fundamental result, called source coding theorem and proven by Shannon, is that entropy represents the absolute mathematical limit for lossless compression. In other words, no lossless compression algorithm can achieve better compression, i.e. average codeword length, than entropy. For this result to be understood, however, it is crucial to contemplate its assumptions.

The source coding theorem holds for individual symbol encodings, independent and identically distributed symbols, infinite length sequences, and a complete a priori knowledge of the symbols' distribution. If the distribution is partially known and the unknown probabilities are assigned equal value, then the actual compression limit will be greater than or equal to the computed entropy. If any of the theorem's other assumptions are violated, then the actual compression limit will be less than or equal to the computed entropy.

2.3.3 Entropy coding

Methods which attempt to approach the source coding bound are called entropy coders. Their main characteristic is that the expected codeword length of any symbol is greater or equal to the negative logarithm of the probability of that symbol. The two most common such methods are described below.

Huffman coding

Huffman coding [33] is an optimal prefix-free code, meaning that no codeword is the prefix of any other codeword. It works by arranging the symbols on the leaves of a tree, based on their probabilities, and assigning a codeword to each of the symbols based on the path from the root node to the respective leaf node. The encoding process consists of two steps; generating the tree and then replacing every symbol by its codeword. The decoding process

identifies the individual codewords in the encoded message and inversely maps them to their associated symbols, using the generated tree.

Algorihtm 1 is used to generate the Huffman tree for a given alphabet and PDF. Both its space and time complexity is $O(n)$, where n is the cardinality of the alphabet. The encoding process is also of linear time and space complexity, with respect to the number of symbols in the data to be encoded. As for the decoding process shown in algorithm 2, the time complexity is linear to the length of the compressed data.

Algorithm 1: Binary Huffman tree generation

Input : $\mathcal{X}$ – The symbol alphabet
PDF – The probability distribution function
Output: The root of the tree

```
1  Q ← empty min-heap
2  for x ∈ X do
3  |   n ← new node
4  |   n.symbol ← x
5  |   n.weight ← PDF(x)
6  |   INSERT(Q, n)
7  end
8  for i = 1 to |X| − 1 do
9  |   n ← new node
10 |   n.left ← EXTRACTMIN(Q)
11 |   n.right ← EXTRACTMIN(Q)
12 |   n.weight ← n.left.weight + n.right.weight
13 |   INSERT(Q, n)
14 end
15 return EXTRACTMIN(Q)
```

The expected decoding time per symbol is equal to the average codeword length (in bits), which, in turn, is approximately equal to the entropy of the source. Thus, the expected throughput (symbols per iteration) is inversely proportional to entropy.

An example of a Huffman tree and an encoded sequence is illustrated in figure 2.13. This example assumes the alphabet $\{a, b, c, d\}$ and the symbol probabilities 0.4, 0.3, 0.2, 0.1 respectively. Note that the specific bit assignment in the edges of each node is arbitrary. Any assignment is valid, where the edges of each node form an exact cover of the base digits.

Algorithm 2: Huffman decoding

Input : S – The compressed data stream
T – The root of the Huffman tree
Output: The decompressed data stream

```
1  n ← T
2  for i = 1 to |S| do
3      if n.leftBit = S[i] then
4          n ← n.left
5      else if n.rightBit = S[i] then
6          n ← n.right
7      end
8      if n.symbol ≠ ∅ then
9          out n.symbol
10         n ← T
11     end
12 end
```

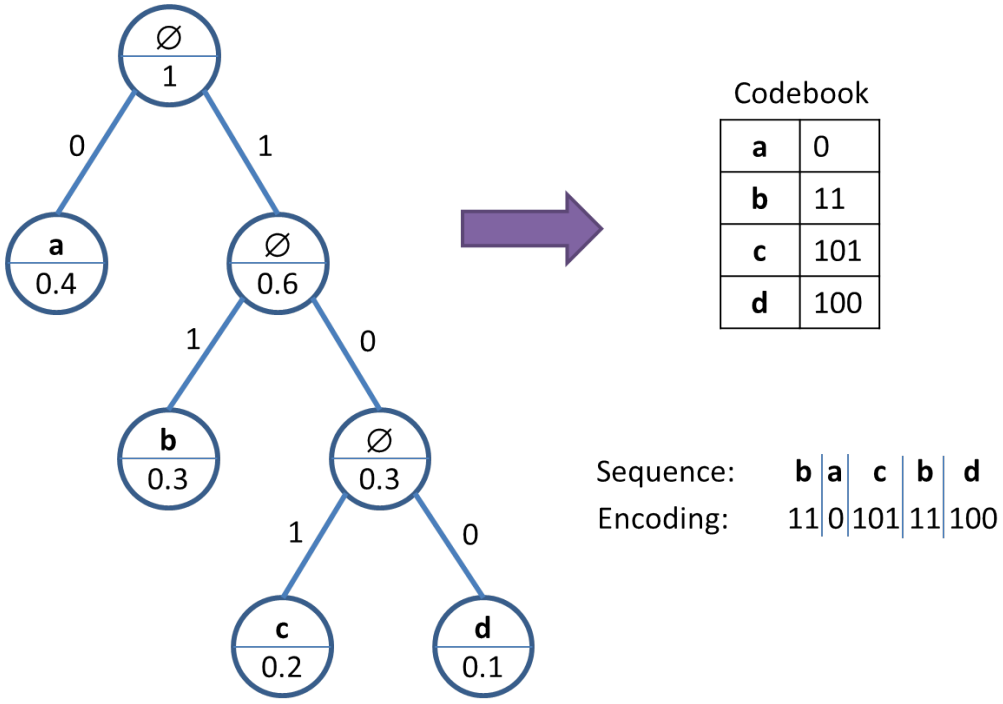


FIGURE 2.13: Huffman tree and sequence encoding example

Arithmetic coding

Instead of encoding each symbol in the data separately, arithmetic coding [34, 35] encodes the entire data file with a single real number in the range $[0, 1)$. For every symbol position in the data, the running range is divided into intervals proportional to each symbol's probability and the range bounds are updated to correspond to the desired symbol interval at each step. Finally, any number is selected within the resultant range. This single number unambiguously represents the entire sequence of symbols, by simultaneously falling within the subinterval of the subdivision corresponding to the given symbol at each position in the sequence.

Algorithm 3 presents the encoding procedure and algorithm 4 presents the decoding procedure. Both of these algorithms are ideal, in the sense that they require infinite-precision arithmetic. There exist, of course, implementations of the method with fixed-precision arithmetic and efficient decoding [36, 37], but their details are well beyond the scope of this chapter.

Arithmetic coding performs better in terms of compression ratios compared to Huffman, achieving results closer to entropy. Huffman allocates an integer number of bits per symbol, thus its codeword length, m, will be within 1 bit of its optimal length, i.e. $m \leq -\log P(s) < m + 1$. Arithmetic coding, on the other hand, allocates $-\log P(s)$ bits, i.e. non-integral number of bits, with the only source of inefficiency being the limited precision arithmetic.

Algorithm 3: Arithmetic encoding

Input : S – The symbol sequence
PDF – The probability distribution function
CDF – The cumulative distribution function for some ordering of the alphabet
Output: A single number encoding the entire sequence S

```
1 L ← 0
2 W ← 1
3 for i = 1 to |S| do
4 |   s ← S[i]
5 |   L ← L + W · CDF(s)
6 |   W ← W · PDF(s)
7 end
8 return SHORTESTBINARYFRACTIONWITHIN(L, L + W)
```

Algorithm 4: Arithmetic decoding

Input : V – The fractional value
N – Original sequence length
PDF – The probability distribution function
CDF – The cumulative distribution function for some ordering of the alphabet
Output: The decompressed data stream

```
1 for i = 1 to N do
2     Determine s such that CDF(s) − PDF(s) ≤ V < CDF(s)
3     L ← CDF(s) − PDF(s)
4     W ← PDF(s)
5     V ← (V − L)/W
6     out s
7 end
```

A visualization of the range division and selection is shown in figure 2.14, assuming the same alphabet, statistics and data of figure 2.13. The fractional value transmitted will be within $[0.50008, 0.5008)$, the shortest of which is 0.50048828125 with the binary representation 0.10000000001.

2.3.4 Dictionary coding

Compressors which replace occurrences of symbol phrases with references to a previous occurrence of the phrase are called dictionary coders. The data structure holding the referenceable phrases is called the dictionary. Based on the type of dictionary, these methods are divided into two categories: static or dynamic. Static-dictionary coders determine the dictionary contents before the encoding process starts and do not change it thereafter. Dynamic-dictionary coders may or may not initialize their dictionaries before the encoding process, but their contents adapt to the encoded data during encoding.

Dynamic coders do not require any knowledge of the statistics of the data stream, nor is the source required to have static statistics. Instead, this class of coders adapt to the data based on some dictionary update policy. Interestingly enough, the update policies work both during encoding and decoding, so the decoder reconstructs the encoder's dictionary at each time step, so both processes work with the same running dictionary. No transmission of the dictionary contents is required with this scheme.

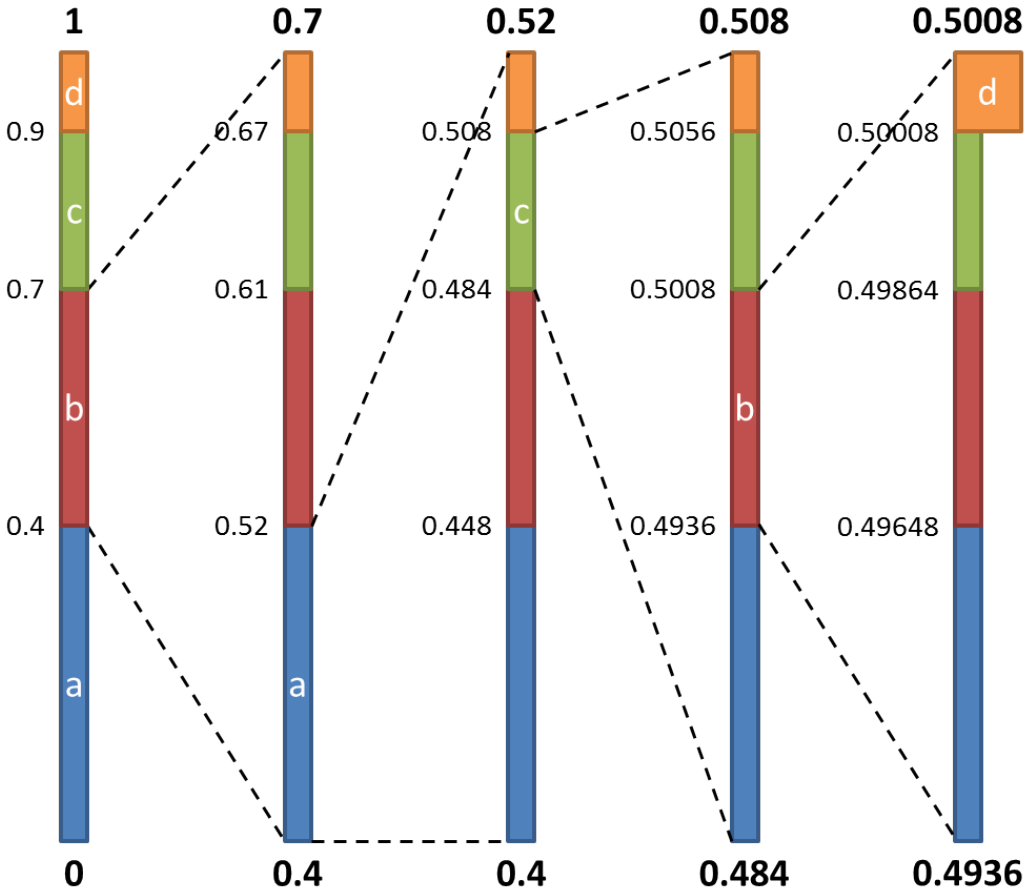


FIGURE 2.14: Arithmetic coding example - Range visualization

Lempel-Ziv coding

Lempel-Ziv is a family of dynamic dictionary coders, namely LZ77 [38] and LZ78 [39]. The latter was developed as an improvement on the former and though the data structures and exact mechanics are different, the dictionaries have shown to be equivalent. These two algorithms are the basis for several other variants, such as LZW [40], LZSS [41], and LZMA. We will briefly discuss details of the original algorithm.

LZ77 maintains a dictionary in the form of a sliding window of the most recently coded data. The algorithm replaces the longest prefix of the remaining data, yet to be "seen" by the sliding window buffer, with a reference of said prefix to the buffer. The reference consists of two integers, the position in the buffer and the length of the prefix. If no such prefix exists, the length is set to 0 and the raw symbol is transmitted in place of the position pointer.

At first, the sliding window buffer of length W is empty, so the first W symbols are transmitted uncompressed. Then, the sliding window is advanced either by the length of the matched prefix at each step, or by 1. The encoding and decoding pseudocodes are presented in algorithms 5 and 6 respectively. Note that for the matched prefix S_{i-p+1}^{i-p+l}, the starting symbol S_{i-p+1} will always be inside B, but the ending symbol S^{i-p+l} can be outside the current buffer bounds.

Algorithm 5: LZ77 encoding

Input : S – The symbol sequence
Output: A stream of pairs (p, l) with the position and length of the matched prefixes

$B \leftarrow$ empty buffer of length W
for $i = 1$ **to** $|B|$ **do**
    $B_i \leftarrow S_i$
    out $S_i, 0$
end
$i \leftarrow W$
while $i < |S|$ **do**
    find $p \in [1, W]$ and maximum l such that $S_{i-p+1}^{i-p+l} = S_{i+1}^{i+l}$
    $i \leftarrow i + l$
    $B \leftarrow S_i^{i+W}$
    out p, l
end

Algorithm 6: LZ77 decoding

Input : X – The sequence of pairs
Output: The decompressed data stream

$B \leftarrow$ empty buffer of length W
foreach $(p, l) \in X$ **do**
    if $l = 0$ **then**
        $B \leftarrow B_2^W \cup \{p\}$
        out p
    else
        for $i = 1$ **to** l **do**
            $B \leftarrow B_2^W \cup \{B_p\}$
            out B_p
        end
    end
end

Chapter 3

Related Work

3.1 Automatic speech recognition architectures

Automatic speech recognition is the process of converting human speech into text. Prior to the advent of Artificial Neural Networks, traditional techniques were used to tackle this task, such as Dynamic Time Warping [42] and Hidden Markov Models [43]. Methods implementing such techniques also focused on manual feature extraction and an ASR system would be comprised of several processing steps, such as pre-processing, feature extraction, classification, acoustic modeling, and language modeling.

Since the late 1980s, ANNs have been successfully used for various steps of ASR systems [44, 45]. However, recent advances in ML and an ever-increasing abundance of computational resources have given rise, since 2014, to a holistic approach to ASR, called end-to-end ASR [46]. A remarkable benefit of this approach is the need for a single dataset to train the entire system, as opposed to multiple datasets to train each individual component.

Given the nature of supervised ML as a data-driven approach, the role of datasets is essential to any such model. For ASR, there exist multiple datasets with labeled recorded speech, in various conditions, such as noise, conversation, reading etc., and several languages. At the time of writing, the most widely-used open-source ASR datasets include, but perhaps are not limited to, the LibriSpeech corpus with 1000 hours of read English text [47], and Common Voice with over 24000 hours of read text across 100 languages and a wide demographic [48, 49]. A commonly used paid dataset is the Wall Street Journal corpus with a few hundred hours of read English text.

Various architectures have been explored to tackle ASR over decades of research. Few notable end-to-end architectures are briefly discussed.

3.1.1 CTC-based architectures

The first attempt

The first RNN trained via CTC for end-to-end ASR was attempted by Alex Graves and Navdeep Jaitly, in 2014 [46]. The network consists of 5 bidirectional LSTM layers. Each layer has 500 hidden units, amounting to a total of $\tilde{2}6.5$ million parameters.

The model was trained on the Wall Street Journal corpus and achieved 8.2% Word Error Rate (WER), compared to the 7.8% WER of the baseline model. However, this performance was attained in combination with a trigram Language Model (LM), for both the RNN and the baseline model. The RNN proved incapable of learning the LM, despite being able to learn the pronunciation and acoustic models.

DeepSpeech2

In 2015, Baidu Research published DeepSpeech2, a DNN architecture for end-to-end speech recognition in English and Mandarin [50]. DeepSpeech2 is not one concrete architecture, but rather a family of architectures containing 1 to 3 CNN layers, followed by 1 to 7 BRNN layers, followed by 1 to 4 FC layers. Figure 3.1 illustrates the general layout of the model.

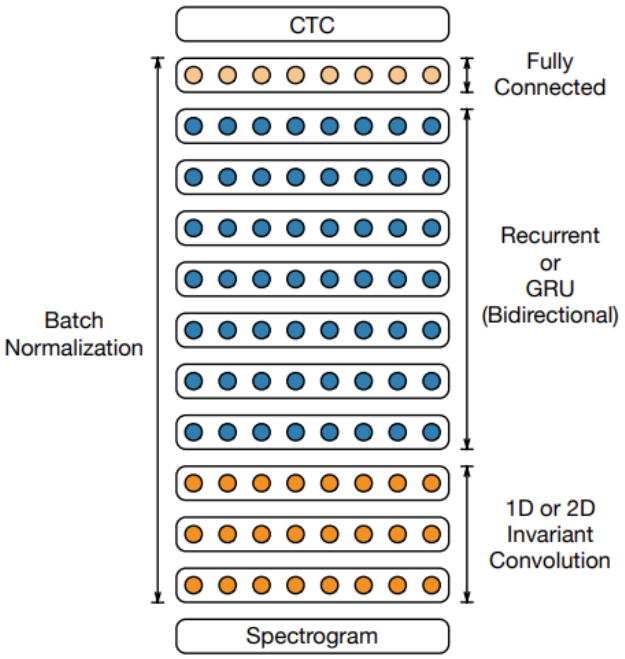


FIGURE 3.1: DeepSpeech2 architecture (source: [50])

The model was trained on a mixture of internal Baidu and public domain datasets, totaling approximately 12000 hours of speech. Experiments were also carried out for several configurations of its architecture, varying the number of layers of each type, using FRNN or GRU cells, and changing the total number of parameters from 18 million up to 100 million.

To provide results comparable to existing literature, the model was tested against non-internal datasets. In these tests, DeepSpeech2 proved to be a significant improvement over its predecessor, and surpassed human-level performance in 3 out of 4 test sets on read speech. However, these advances did not come cheap, computationally speaking, as each model was trained on a server with 8 or 16 NVIDIA Titan X Graphics Processing Units (GPUs) and required 3 to 5 days to complete training.

3.1.2 Attention-based architectures

Traditional NNs learn how to combine each individual input, through their weights, to calculate their output. The attention mechanism was developed in 2015 [51, 52] to select which inputs, based on the particular context, are most relevant and rely on them to generate each output.

LAS

"Listen, Attend and Spell" (LAS) was one of the two attention-based ASR models introduced simultaneously in 2016 [53]. It consists of a pyramidal bidirectional LSTM network with 3 layers, which comprises the "listener" component of the system, while the "attend and spell" component consists of an LSTM transducer. In turn, the transducer is made up of the attention mechanism, 2 LSTM layers, and an MLP. Figure 3.2 illustrates the architecture.

The model was trained on a dataset with 2000 hours of Google Voice searches. It performed comparably to the then-state-of-the-art model, which achieved 8.0% on the clean and 8.9% WER on the noisy test set. LAS, with a LM, achieved 10.3% on the clean and 12.0% WER on the noisy test set.

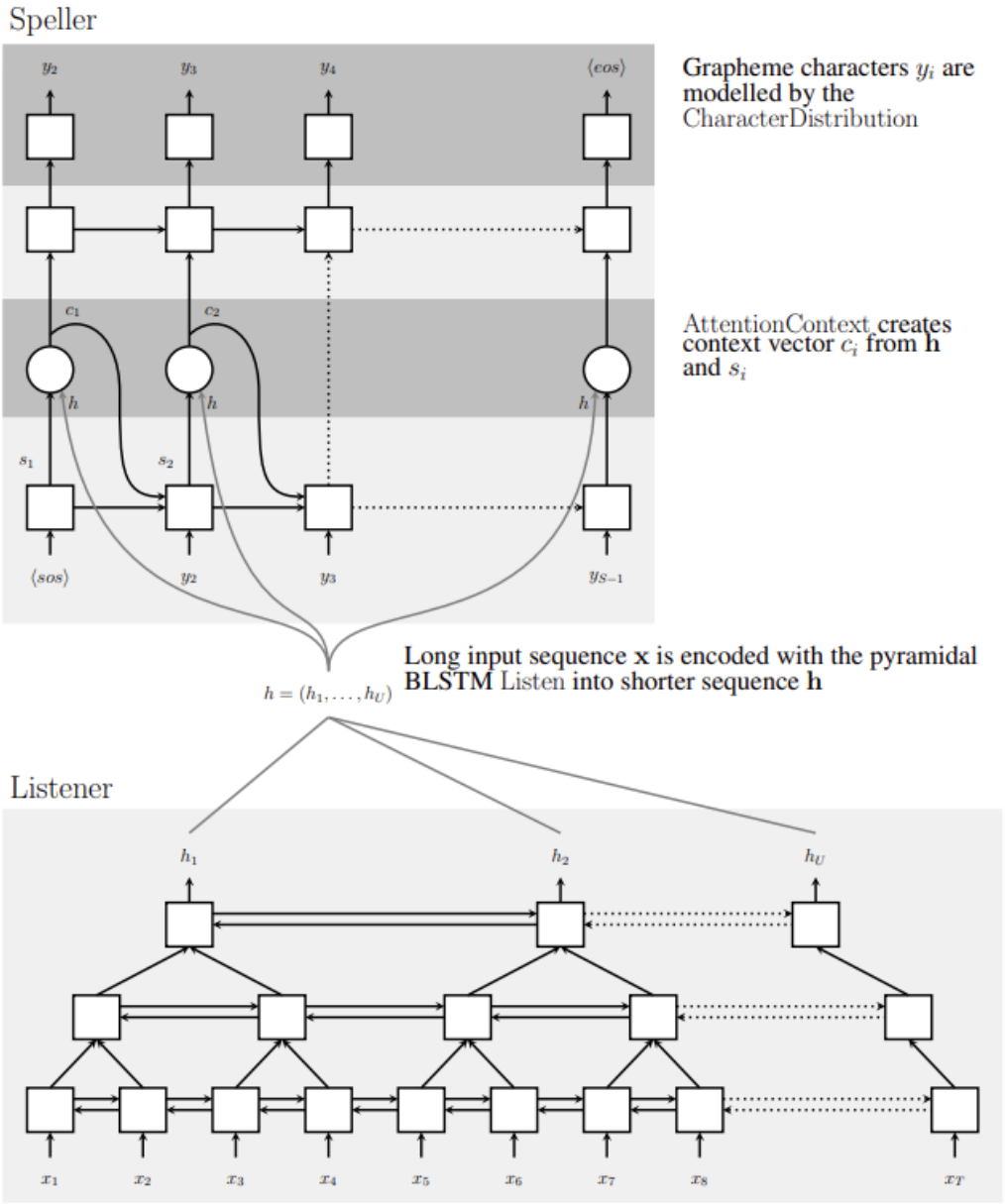


FIGURE 3.2: LAS architecture (source: [53])

3.2 Deep learning software frameworks

Deep learning software frameworks are software structures which facilitate the creation, training, and deployment of deep learning models, through high-level programming interfaces. Such frameworks have become industry standards and the de facto tools of anyone venturing into the field of ML and DNNs.

Many frameworks exist, each with their own sets of features and learning curves. Some of the most popular are described below.

3.2.1 TensorFlow

TensorFlow [54] was developed by Google and was initially released in 2015 [55]. It is written in Python, C++, and CUDA. Its offers Application Programming Interfaces (APIs) in several languages, such as Python, C++, JavaScript, Java, and more. Compared to others, TensorFlow is a lower-level framework and requires more coding, but also provides high configurability and is suitable for a very wide range of platforms, from computing clusters to embedded and edge devices.

3.2.2 Keras

Keras [56] is a high-level library for Python released in 2015 [57]. It runs on top of TensorFlow, Theano, and CNTK. It provides a simple, user-friendly API and requires little code to create prototypes. Naturally, it is less configurable compared to its lower-level counterparts, but also easier to learn.

3.2.3 PyTorch

PyTorch [58] is the ML framework developed by Meta, formerly known as Facebook, and was initially released in 2016 [59]. Like TensorFlow, the framework is written in Python, C++, and CUDA, however it offers APIs only in Python, C++, and Java. PyTorch comes with a slew of pretrained models for vision, audio, and text. Several commercial deep learning applications are built with PyTorch, with Tesla Autopilot being, perhaps, the most successful.

3.3 Hardware platforms

Having discussed various ANNs, DNN architectures, and software frameworks, we now turn to the final link in this chain of research and innovation: the hardware platforms that enable experimentation with and use of DNNs.

Both phases of DNN computation, i.e. training and inference, are computationally demanding. Training is quite often applied on a one-off basis, with DNNs learning their parameters once, usually before being deployed. Inference, on the other hand, is the process of using the model to perform its task on new data and is thus applied more than once, after the model is deployed. Inference requires less computation per iteration compared to training, but it can be difficult to meet latency and through specifications, e.g. in real-time applications.

Existing hardware platforms suitable for DNNs are discussed below, including their advantages and limitations.

3.3.1 CPU

Every device, from desktop computers to edge devices, has a Central Processing Unit (CPU) to operate, with clock speeds, typically, in the gigahertz range. CPUs are the most versatile of the platforms, given their general-purpose design requirements. Thus, they are capable of running any model, regardless of complexity or novelty, with little coding effort. Speedwise, despite their high clock speeds, DNN computations are guaranteed to be slow on CPUs, as the later don't lend themselves to parallelism. Granted, multicore and multithread CPUs are the norm, and Single Instruction Multiple Data (SIMD) instructions via Advanced Vector Extensions (AVX) [60], and Streaming SIMD Extensions (SSE) [61] do provide parallelism capabilities to CPUs that support them, but they are negligible compared to the alternatives.

3.3.2 GPU

GPUs are massively parallel platforms designed to process large blocks of data in parallel. While they were original meant to handle graphics exclusively, they have now become General Purpose Graphics Processing Units (GPGPUs), a form of general purpose stream processors. GPUs have hundreds or, more often, thousands of cores, fed by High Bandwidth Memory (HBM) which can achieve throughput of hundreds of GBs per second. Figure

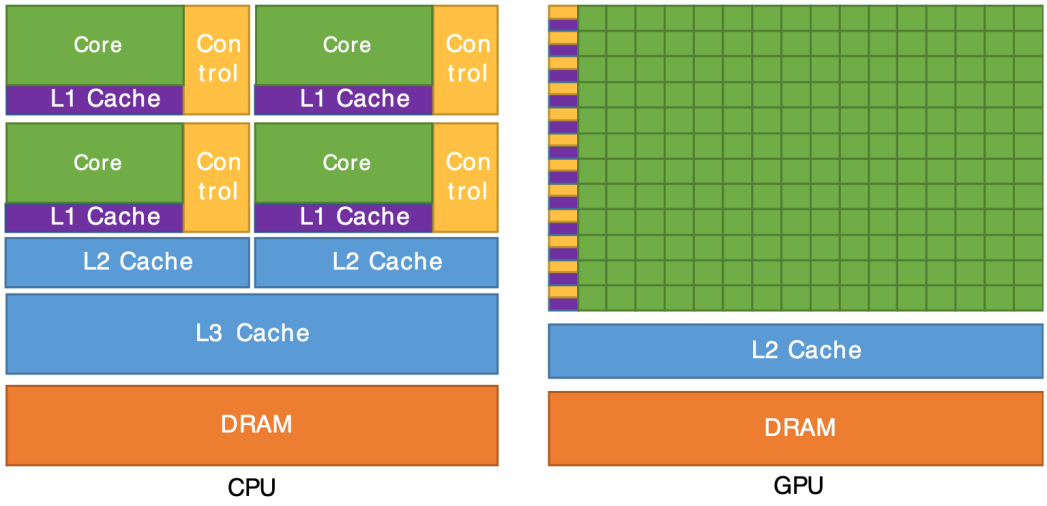


FIGURE 3.3: CPU vs GPU architecture (source: URL)

3.3 illustrates the qualitative difference between CPUs and GPUs, in terms of architecture.

Writing code for GPUs is harder and requires specialized libraries, which often provide highly optimized implementations of commonly used operations or procedures in ML, like matrix multiplications and convolutions. NVIDIA has developed the Compute Unified Device Architecture (CUDA) [62] framework for its GPUs, providing APIs in C, C++, Python, and more. On top of it, NVIDIA built the CUDA Deep Neural Network (cuDNN) [63] library, utilized by all deep learning software frameworks. AMD has developed ROCm [64], the equivalent of CUDA for their own line of GPUs. While ROCm is supported by TensorFlow and PyTorch, it is not as mature as CUDA and does not support all AMD GPUs.

Overall, modern GPUs provide massive parallelism to DNN computations, thus achieving high throughput. But, their latency and energy efficiency is lacking, compared to the other alternatives.

3.3.3 TPU

Tensor Processing Units (TPUs) [65] are Application Specific Integrated Circuits (ASICs) developed by Google. They were introduced in datacenters in

2015 with the purpose of accelerating machine learning workloads. In recognition of the dominance of matrix multiplications, and therefore Multiply-Accumulate (MAC) operations, in the typical computational effort distribution of traditional DNNs, TPUs have been designed with dedicated matrix multiplication units, as shown in figure 3.4.

Google's first generation TPUs run at 700 MHz and achieved 92 TeraOperations Per Second (TOPS), a 70x and 200x increase in TOPS per Watt, compared to a GPU and CPU respectively. Currently on their 4th generation, TPUs have become exceedingly more performant and efficient, compared to all other alternatives.

Their inherent limitation lies in their ASIC nature. TPUs are designed around the current assumptions and adopted practices of ML. Should these assumptions change in the future, the relevance of TPUs, as well as any other such ASIC, rests entirely on the question of compatibility of its architecture with the emerging trends. Therefore, impressive performance notwithstanding, the static and limited scope of the TPU against the rapid innovation and research in ML, along with the significant cost of designing, manufacturing, and maintaining such platforms, amount to considerable technical risk.

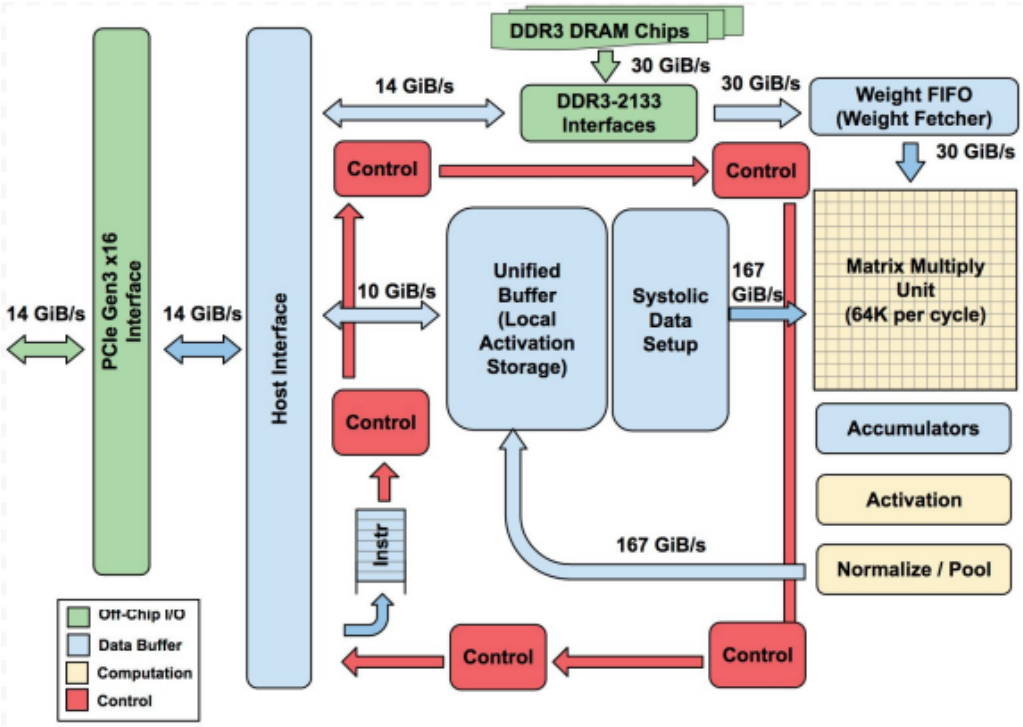


FIGURE 3.4: TPU block diagram (source: URL)

3.3.4 FPGA

Field Programmable Gate Arrays (FPGAs) are Integrated Circuits (ICs) with programmable or reconfigurable logic. The FPGA fabric, i.e. the part of the chip containing the Programmable Logic (PL), can contain various types of resources, such as Flip Flops (FF), Look Up Tables (LUT), Block Random Access Memories (BRAMs), Digital Signal Processor (DSP) blocks, and more. FPGAs can also contain hard processor cores on the same chip, which comprise the chip's Processing System (PS).

FPGAs typically run in the few hundred megahertz clock speeds, but they make up for it with the custom, fine-tuned hardware architecture for each individual application. Of course, they are not easy to use, requiring significant expertise and time to develop the custom architectures. Their reconfigurability makes them suitable for rapidly changing fields, like ML. They are highly energy efficient and can achieve low latency.

In both memory and computation intensive applications, such as ML, FPGAs can be memory bottlenecked, despite the slew of BRAMs. The reason is that the total capacity of on-chip RAM is low, in the very few megabytes range. To mitigate that, most FPGA boards provide dedicated Dynamic Random Access Memory (DRAM) chips, which connect with the FPGA, but their bandwidth is limiting. At that point, the solution is not to rely on technology, but on carefully engineered implementations that require less of their external memory modules.

3.4 Neural network compression

DNNs become increasing bigger, in terms of total parameters, as better performance is required from them on tasks of ever-increasing complexity. As a result, the models become exceedingly expensive to store and use. While performant servers can, at least for the time being, handle these models, the less capable FPGAs, Internet-of-Things (IoT) and mobile devices are severely limited, making it perhaps prohibitive to deploy the majority of models on them.

For this reason, several works in literature have emerged over the years on the topic of ANN compression. Unfortunately, the term "compression" can be misleading, especially taking into account the context of section 2.3, and

may even be a misnomer. In this context, compression seeks to remove unnecessary information from the model, minimally impacting its accuracy, to reduce its computational and storage requirements. In other words, the original data are edited, to remove or reuse certain elements. This is fundamentally different from the information-theoretic data compression described in section 2.3, where an intermediate representation is created, which must be decompressed prior to use.

Many ANN compression methods exist, such as weight sharing, pruning, quantization, low rank decomposition, and knowledge distillation [66, 67]. Depending on the specific method, it can be applied before [68], during [69], or after training [70] and all methods, except one, target the given model. Knowledge distillation, however, attempts to use the given model to help train a smaller one, which will in turn be used for deployment [71]. For completeness purposes, it should be noted that low rank decomposition methods are exempt from the previous disambiguation of "compression", as they do, in fact, constitute a compression method.

The most relevant categories of ANN compression methods are briefly discussed below.

3.4.1 Pruning

Pruning is the process of removing components in the network which do not contribute much to its accuracy and are thus unimportant. Pruning can be structured or unstructured, based on the domain of its application. Structured pruning removes entire neighborhoods of weights, thus preserving matrix density, while unstructured pruning removes individual weights and the matrices become sparse [66]. Some structured techniques are the following:

- **Channel pruning**: Reduces the number of channels in each layer of the network [72]. Effectively, it reduces the number of filters of preceding layers [73], since the number of filters in one layer is equal to the number of channels in the next. This approach is specific to CNNs.
- **Filter pruning**: Reduces the number of filters in the network [74]. This approach is also specific to CNNs. While mathematical equivalence between filter and channel pruning exists, their difference lies in the algorithms and optimization techniques used to select which filters or channels, respectively, to prune.

- **Layer pruning**: Removes selected layers from the network. This can achieve ultra-high DNN compression, at the expense of high accuracy degradation, due to the semantic structural deterioration of the network [67, 75]. This technique is applicable to FCs, CNNs, and RNNs.

Unstructured pruning can be applied to all networks and is further subcategorized based on the policy used to select weights. Magnitude-based pruning (MBP) selects weights based on their magnitude, either via a threshold [76], or a percentage of weights closest to zero. MBP can also benefit greatly by editing the loss function to include regularization terms, which tend to reduce the overall magnitude of weights and thus facilitate MBP [77]. Lastly, the policy can depend on the loss function's sensitivity to each weight, therefore opting to prune weights with low sensitivity [78].

Structured pruning is beneficial for every computing platform, as it reduces model size and computations with no overhead, making it a highly suitable method to accelerate DNN computations [73]. Unstructured pruning has been shown in literature to degrade throughput if the resultant matrices are not sparse enough [79, 80], because of its incompatibility with the CPU and GPU designs, which favor contiguous data processing. It is thus generally incompatible with CPUs and GPUs, but may be beneficial in FPGAs and ASICs. Still, however, it has been shown to be outperformed by structured pruning [73].

3.4.2 Quantization

In the vast amount of cases, DNN weights are desired to be real numbers, however, due to the finite precision arithmetic of every computing platform, weights are represented by the next best option, floating-point numbers. Typically, 32-bit floats are used and given the inherent support of CPU and GPU pipelines for this data type, there is little incentive to change it. However, TPUs and ASICs benefit tremendously from less memory intensive representations.

Quantization is the process of shrinking the weight domain, i.e. reducing the number of distinct values the weights can take. This can be done with or without reducing precision. To reduce precision, each weight's value is represented by fewer bits [81]. In the case of quantization without precision

reduction, a subset of the original domain is selected and, through a codebook, weight values can be indirectly referenced, again using fewer bits, but without affecting the underlying representation [82].

It should be noted that quantization is applicable to every DNN and is not limited to the network's parameters. Most works quantize, via reduced representation, the weights and neuron activations [66], with some quantizing also the gradient values [83].

3.5 The FPGA perspective

Traditional computing platforms, i.e. CPUs and GPUs, benefit greatly from the mature ecosystem of DNN frameworks. Unfortunately, these frameworks do not natively support FPGAs and therefore the significant advantages of reconfigurable hardware customization cannot be so easily leveraged. For this purpose, researchers and the FPGA community must rely on a different set of tools.

3.5.1 CHaiDNN

Xilinx's CHaiDNN is an open-source DNN library released in 2018 for accelerating inference on the company's Zynq UltraScale+ family of Multi-Processor Systems on Chip (MPSoCs) [84]. Its support for hardware acceleration is limited to CNNs and other relevant layers, but unsupported layers can be custom added, albeit they will run as software.

The library is designed for 6-bit and 8-bit fixed point data types, both for weights and activations. The conversion from single-precision floating-point model to fixed-point representation is done by the library, using one of two supported modes.

3.5.2 Vitis AI

Vitis AI is a development platform from Xilinx aimed at facilitating AI inference applications on its platforms, released in 2019 [85]. It supports mainstream deep learning frameworks, such as TensorFlow and PyTorch, as well as a range of DNNs, such as CNNs, RNNs, and Transformers. Apart from the tool's development kit and FPGA overlay, there is also a comprehensive set of ML models for common tasks, such as Natural Language Processing, Optical Character Recognition, etc.

The platform consists of several tools, as seen on figure 3.5, such as the AI Compiler, AI Quantizer, AI Optimizer, and AI profiler. The Optimizer prunes and finetunes the network, the Quantizer performs parameter and activation quantization, the Compiler maps the model to highly optimized instructions, and the Profiler analyzes the efficiency and utilization of the implementation. By all accounts, Vitis AI is a landmark design suite which enables and significantly democratizes the intersection of reconfigurable hardware and ML.

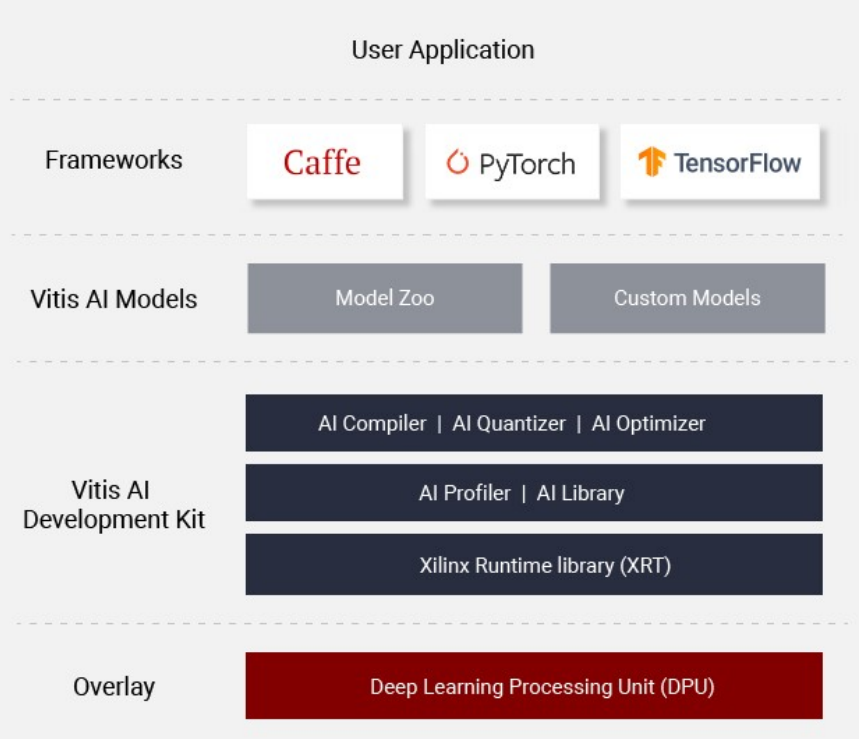


FIGURE 3.5: Vitis AI stack (source: URL)

3.5.3 NVIDIA Deep Learning Accelerator

NVIDIA Deep Learning Accelerator (NVDLA) is an open-source architecture aiming to standardize the design of deep learning inference accelerators [86]. It was released in 2017 and targets NVIDIA supported solutions. NVDLA's architecture is customizable, modular, and scalable, with configurations suitable for FPGAs and ASICs. Testament to its customizability is its support for various data types, from binary and 4-bit integers, to 64-bit floating-point.

3.6 Thesis approach

This thesis aims to develop a compression method suitable for compressing sparse and quantized neural network weight matrices. The goal for this method is to be used in FPGA accelerators of neural network inference, so decompression should also allow for a high-throughput hardware implementation with low resource utilization. Also, as inference accelerators can be deployed on FPGAs of varying capabilities, from edge to high performance computing, it would be highly beneficial for our method to have design-specified throughput. Robustness analysis is carried out to explore the effects of network compression methods on model accuracy, and to initially reduce its memory footprint prior to compressing its weights.

Chapter 4

Training and Robustness Analysis

The goal of this chapter is to quantitatively evaluate network compression techniques and explore possible opportunities for a performant hardware implementation. Training, pruning, and quantization are performed in this thesis for DeepSpeech2 [50], which will be used as an open-source case-study model, along with the relevant dataset. Training will be performed once and network compression will be applied after.

4.1 Model overview

4.1.1 TensorFlow model implementation

Prior to any experiments and code review, an implementation of the model was needed in a familiar deep learning framework. Three GitHub repositories were found implementing the model, PaddlePaddle [87], NVIDIA's OpenSeq2Seq [88], and TensorFlow's Models [89].

PaddlePaddle is an open-source machine learning framework from Baidu Research [90], i.e. the company behind DeepSpeech2. Naturally, a model implemented on this framework suffers from two major disadvantages: the framework itself is new to the end-user, and the majority of discussions on the repository page are carried out in Mandarin. Thus, this particular repository and the framework altogether were deemed unsuitable for this thesis' needs.

Next, NVIDIA's OpenSeq2Seq repository was based on the familiar TensorFlow framework, and provided a small documentation page specifically for the model of interest. However, the code was centered around configuration files which described the model in a high-level, easily reconfigurable manner. This production-level, reusable code, while othertimes preferred,

was bloated enough to impede straightforward understanding of the specific model's code.

Lastly, TensorFlow's Models repository provided a very simple, straightforward and minimally complicated implementation of DeepSpeech2. Its concise code, spread across three files and a bash script to run the model, along with its minimal library dependencies was a perfect fit.

4.1.2 DeepSpeech2 variant

The specific DeepSpeech2 variant chosen for this thesis contains 2 CNN layers, 5 BRNN layers with GRU cells, and 1 FC layer. Also, 6 batch normalization layers are used, each placed between pairs of the 7 total CNN, GRU, FC layers. This model contains approximately 62 million parameters and can be thought of as a combination of the median number of CNN layers, RNN layers, and total number of parameters, of the variants tested in the original paper [50]. As a case-study model, the median architecture approach strikes a reasonable balance between computational intensity during experimentation and generality among the DeepSpeech2 variants.

Input tensor

The model accepts spectrograms of power-normalized audio clips as inputs. The size of dimensions of this 4D tensor vary, based on the audio clip's length, but its shape is of the form (B, T, F, C), where B is the batch size, T is the number of time slices of the clip, F is the number of features, and C is the number of channels. In this case, B is determined based on the available memory of the hardware platform, T is specific to each audio file.

The other dimensions have constant size, with F being 161, and C being 1. These values are directly related to the LibriSpeech dataset, which is recorded at 16 KHz and monophonic sound. To reduce the size of the input tensor, the spectrogram is not provided as-is, for every frequency, but is rather computed for a number of frequency bins. That number is precisely F, which, given the maximum Nyquist frequency of 8 KHz for the 16 KHz sampling rate, yields bins with frequency width of 50 Hz. As for C, monophonic sound, as the name implies, only has one channel.

Each audio file is split into time slices. The defining characteristics for this process are the size of each slice, i.e. the window size, and the stride of consecutive windows. In this particular implementation, the window is 20 ms

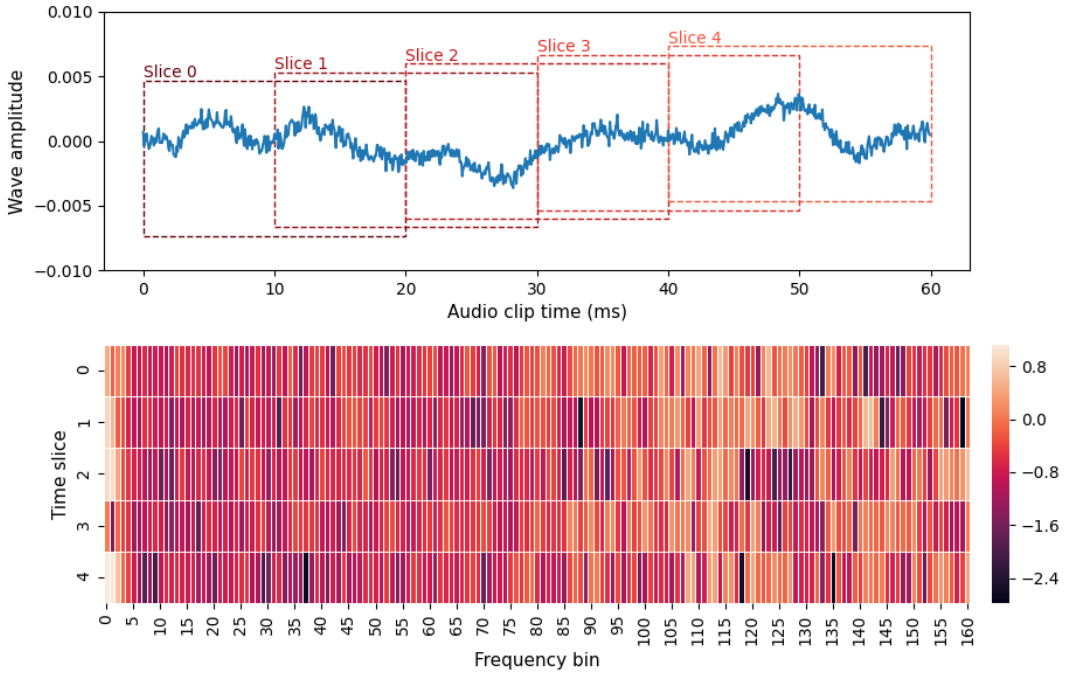


FIGURE 4.1: Audio clip converted to normalized spectrogram input tensor - Example showing the first 5 time slices of a clip from LibriSpeech

wide, with a stride of 10 ms. Thus, if t is the total duration of an audio clip, in milliseconds, then T is computed via equation 4.1. Figure 4.1 illustrates part of a single audio clip and its normalized-spectrogram time-slice representation.

$$T = \left\lfloor \frac{t - \text{window}}{\text{stride}} \right\rfloor + 1 \tag{4.1}$$

Convolutional layers

The first 2D CNN layer is configured with 32 filters of kernel dimensions $(41, 11)$ and a stride of $(2, 2)$. The input tensor is extended with zero elements, via "same padding", a technique which extends the input tensor of a CNN so its kernels, if strided by 1, would produce a tensor of the same size as the input tensor. Of course, since the kernels have a stride value of 2 along both convolved axes, the output tensor will be half the width and height of the input tensor, but with 32 times the channels.

The second 2D CNN layer also has 32 filters, with kernel dimensions of $(21, 11)$ and a stride of $(2, 1)$. "Same padding" is also applied to this layer's input. Both CNN layers use the ReLU6 activation function, shown in equation

4.2, which is a magnitude-capped version of ReLU, as presented in 2.3.

$$\text{ReLU6}(x) = \min(\max(0,\ x),\ 6) \tag{4.2}$$

Note that the description of CNN kernels by their width and height is common in literature, but can be misleading to the uninitiated. As discussed in chapter 2.2.1, CNN filters are applied across all channels. Therefore, in 2D convolutions, filter kernels are 3D tensors, with the last dimension having size equal to the number of channels. Nonetheless, because the last dimension's existence and size are implied, they are omitted for brevity.

Altogether, tensor information and number of parameters, per CNN layer, are presented in table 4.1.

TABLE 4.1: CNN information per layer

Layer	Input size	Kernel size	Stride	Output size	Parameters
CNN 1	$(B, T, 161, 1)$	$(41, 11, 1)$	$(2, 2)$	$\left(B, \lceil \frac{T}{2} \rceil, 81, 32\right)$	14432
CNN 2	$\left(B, \lceil \frac{T}{2} \rceil, 81, 32\right)$	$(21, 11, 1)$	$(2, 1)$	$\left(B, \lceil \frac{T}{4} \rceil, 81, 32\right)$	236544

Recurrent layers

Following the CNN layers, there are 5 BRNN layers with GRU cells. All of them have 800 hidden units and use the hyperbolic tangent activation function for the candidate hidden state computation. The input to these layers is a 3D tensor with shape (B, T, D), where B and T are, as before, the batch size and the number of time slices, and D is the feature depth of each time slice.

Possible points of confusion are the relationship between the actual values of the tensor size for the CNN and GRU layers, as well as the semantic difference between D and F. First, the batch size is a constant number across all layers of the network. While it can change between different batches, it does not change within the network for any given batch. The time slices, however, do change across some layers. Due to the stride of the convolution filters of the CNN, the amount of time slices changes in the CNN layers, while the GRU layers do not affect it. Finally, each time slice, at any point in the network up to the FC layer, contains features describing, in some way, said time slice. While, abstractly, F and D denote the quantity of similar constructs, the distinction is made to avoid confusing their values.

Since the output of the CNN is a 4D tensor, it is transformed into a 3D tensor, before being fed to the GRUs, by flattening the last two dimensions. Thus, (B, T, F, C) becomes $(B, T, F \cdot C)$, i.e. $D = F \cdot C$.

The GRU cell in each layer processes time slices sequentially and outputs its hidden state for each time slice. This means that for each example in the batch, a vector of length D will be fed through the cell, of each direction, T times and the concatenated T hidden states will comprise the layer's output. This output will proceed to the next GRU layer, where the process will start over.

To understand the dimensionality and size of each component of the GRU cell, equations 2.6 are contemplated. For simplicity, the batch size in this analysis is assumed to be 1, but this will later be amended. Assume x_t is a vector of length D denoting the features of time slice t, and h_t is the hidden state vector of length H, at time slice t, where H is the specified number of hidden units of the cell. Using the simple rules of matrix addition and multiplication, the dimensions of every component in the GRU cell can be inferred, as shown in table 4.2.

TABLE 4.2: Single GRU cell tensor dimensions

Input/Output		**Reset gate**		**Update gate**		**Candidate state**	
Tensor	**Size**	**Tensor**	**Size**	**Tensor**	**Size**	**Tensor**	**Size**
x_t	$(1, D)$	W_{xr}	(D, H)	W_{xz}	(D, H)	W_{xh}	(D, H)
h_t	$(1, H)$	W_{hr}	(H, H)	W_{hz}	(H, H)	W_{hh}	(H, H)
		b_r	$(1, H)$	b_z	$(1, H)$	b_h	$(1, H)$
		r_t	$(1, H)$	z_t	$(1, H)$	h_t	$(1, H)$

The total parameters of a single GRU cell are computed as the sum of the number of elements of the weight and bias tensors. For each gate or the candidate state, the number of parameters is $D \cdot H + H^2 + H$, thus, overall, there are

$$\text{GRU}_{\text{parameters}} = 3(D \cdot H + H^2 + H) \tag{4.3}$$

per GRU cell. Per bidirectional GRU layer, which contains two GRU cells, the total number of parameters is twice that of equation 4.3.

Each example in the batch size is processed independently of the others and yields different, independent state vectors. Thus, without any other changes, the input vector for each time slice can be a matrix of size (B, D), which would yield a hidden state matrix of size (B, H). The mathematics are still the same, albeit with B times more operations and a succinct representation.

Lastly, since the recurrent layers are bidirectional, the output of each GRU layer is the concatenated hidden state matrices of the two GRU cells. As such, its size will be $(B, 2H)$. Altogether, the high-level tensor information per GRU layer is presented in table 4.3, in the context of what is generated by the preceding CNN layer.

TABLE 4.3: Bidirectional GRU information per layer

Layer	Input tensor	State tensor (per cell)	Output tensor	Parameters
GRU 1	$(B, \lceil \frac{T}{4} \rceil, 2592)$	$(B, \lceil \frac{T}{4} \rceil, 800)$	$(B, \lceil \frac{T}{4} \rceil, 1600)$	16286400
GRU 2	$(B, \lceil \frac{T}{4} \rceil, 1600)$	$(B, \lceil \frac{T}{4} \rceil, 800)$	$(B, \lceil \frac{T}{4} \rceil, 1600)$	11524800
GRU 3	$(B, \lceil \frac{T}{4} \rceil, 1600)$	$(B, \lceil \frac{T}{4} \rceil, 800)$	$(B, \lceil \frac{T}{4} \rceil, 1600)$	11524800
GRU 4	$(B, \lceil \frac{T}{4} \rceil, 1600)$	$(B, \lceil \frac{T}{4} \rceil, 800)$	$(B, \lceil \frac{T}{4} \rceil, 1600)$	11524800
GRU 5	$(B, \lceil \frac{T}{4} \rceil, 1600)$	$(B, \lceil \frac{T}{4} \rceil, 800)$	$(B, \lceil \frac{T}{4} \rceil, 1600)$	11524800

Dense layer

The last layer in the network, before decoding, is a single FC layer with 29 output units, a bias neuron and the softmax activation function. Each output unit corresponds to one symbol of the network's alphabet. This alphabet consists of the lowercase English letters, as well as the space, apostrophe, and blank characters. The blank character does not represent any character in written English, but is used as the CTC separator discussed in chapter 2.2.3.

The softmax function takes a vector in $\mathbb{R}^N$ and transforms in into a probability vector, i.e. a vector of values in the interval $[0, 1]$ with unit sum. It is defined as

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \qquad \forall i \in [1, N] \subset \mathbb{Z} \tag{4.4}$$

and in the case at hand, assigns a probability to each symbol in the alphabet, transforming each time slice into a probability vector for each symbol.

Again, as in the GRUs, the FC layer is applied per time slice and per example in the batch. Its tensor information is presented in table 4.4. The output of this layer is fed into a greedy decoder.

TABLE 4.4: FC layer information

Layer	Input tensor	Output tensor	Parameters
FC	$(B, \lceil \frac{T}{4} \rceil, 1600)$	$(B, \lceil \frac{T}{4} \rceil, 29)$	46429

4.2 Training

4.2.1 Hardware

The model is non-trivial to train and requires powerful hardware. The authors of DeepSpeech2 used 8 or 16 Titan X GPUs to train their models [50]. Still, the model took between 3 and 5 days to complete training. Unfortunately, the available hardware to this author is a single GTX 980 GPU, which provides a laughable 4 GB of GDDR5 memory and approximately 5 TFLOPS of computing power, compared to 12 GB of GDDR5 memory and 6.7 TFLOPS for a single Titan X.

The authors of DeepSpeech2 used a batch size of 64 examples per GPU, while the 980 can sustain only low, single-digit batch sizes. Naturally, training such a model on this hardware is simply practically infeasible. Consequently, a server with 2 GTX 2080 Ti GPUs was kindly provided by the Foundation for Research and Technology - Hellas (FORTH), on which to train the model. For comparison, the 2080 Ti provides 11 GB of GDDR6 memory and 14.2 TFLOPS.

4.2.2 Distributed learning

In order to take advantage of multiple GPUs, the traditional ANN learning scheme must be adapted to accommodate them. With a single processing unit, e.g. a GPU, the model forward-processes a batch of the training data, computes the loss against the ground truth and the gradient of that loss per parameter, and, finally, updates the parameters based on the gradient with a backward pass. But, with data-parallel distributed learning, the batch is

evenly split among the available processing units, the model replica on each unit forward-processes its slice of the batch, the loss results are aggregated, and the gradient update information is sent back to all units.

A very high-level and abstract diagram of distributed learning is illustrated in Figure 4.2. Note that while the input and output for each batch slice is specific to each processing unit, the update information is common and exact copies are shared to all units. Therefore, the model replicas are all updated in the same way and at the same time. This is known as synchronous data-parallel distributed learning.

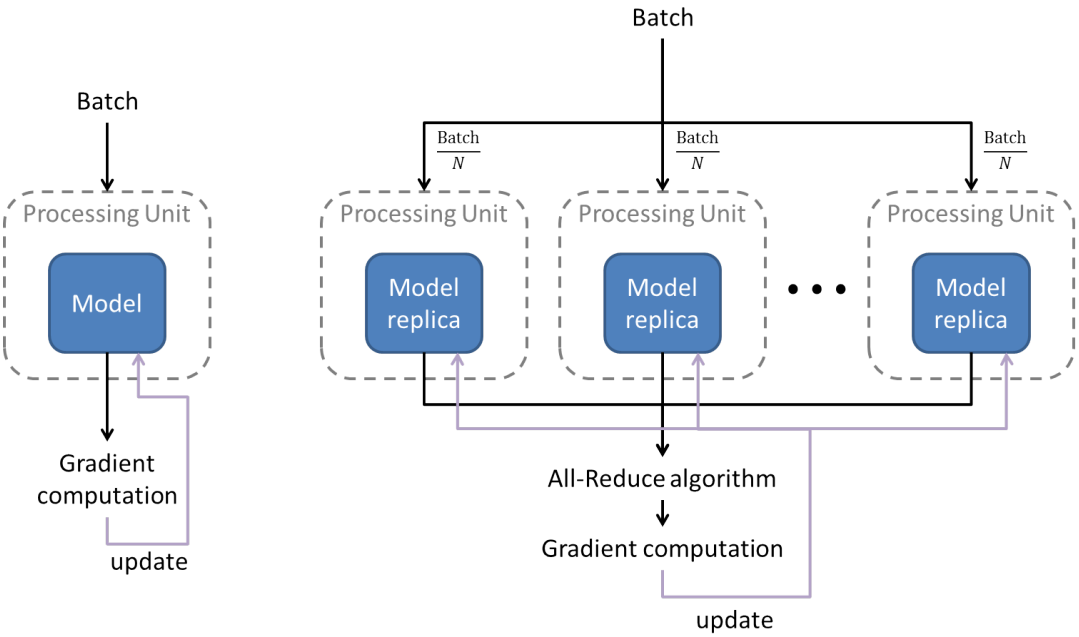


FIGURE 4.2: Single-unit (left) vs data-parallel distributed learning (right) diagram

4.2.3 Training parameters

Despite the capabilities of the 2080 Ti GPUs, training speech models on hundreds of hours of recorded speech takes a significant amount of time. For this reason, a subset of LibriSpeech was used, the statistics of which are presented in table 4.5.

The batch size was set to 32 per GPU, or 64 in total. Since the training set consists of 104014 files, the batch size divides the dataset into 1625 distinct batches. The processing of each batch and subsequent model update constitutes a single training step. The lapse of all training steps constitutes a

TABLE 4.5: Statistics of datasets used for training

Description	Set	Files	Hours	Clip duration (seconds)		
				Min	Median	Max
Training	train-clean-360	104014	363.5	1	14	30
Validation	dev-clean	2703	5.5	1.5	6	32.5

training epoch. Training consists of multiple epochs, i.e. run-throughs of the entire dataset, and in this case the number of epochs was set to 20, same as in the original paper [50]. Lastly, the learning rate was set to the constant value $5 \cdot 10^{-4}$, which falls within the recommended range of $[10^{-4}, 6 \cdot 10^{-4}]$ described in the paper.

4.2.4 Results

Training the model with the parameters described in the previous section yields the results of figure 4.3. The loss is reported once every 100 steps, while the CER and WER metrics refer to the entire validation set, during the model's evaluation at the end of each epoch. The total training duration was 36.5 hours.

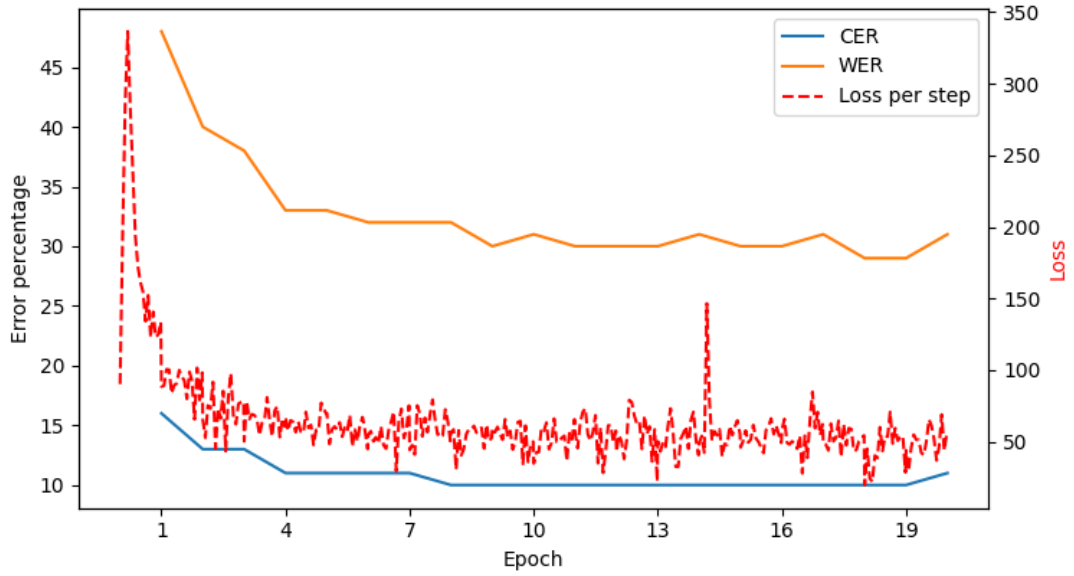


FIGURE 4.3: Training results - WER, CER, and loss

It is evident from the loss plot that albeit the model improves at the beginning, during the first or even second epoch, it proceeds to exhibit erratic behavior instead of converging. This plot is a textbook example of too big a learning rate, a hypothesis indirectly supported by the DeepSpeech2 paper.

The authors use learning rate annealing, i.e. they decrease the learning rate as an exponential function of the number of elapsed epochs, based on equation 4.5

$$\eta_{n+1} = \eta_0 \cdot r^{-n} \tag{4.5}$$

where η_0 is the starting learning rate, r is the annealing factor, and n is the epoch number. In [50], the authors use 1.2 as the annealing factor.

The need for an exponentially decaying learning rate is indicative of the root cause behind the unstable behavior of the loss plot. To mitigate this phenomenon, either a smaller learning rate or - better yet - a similar annealing policy should be adopted.

As for the CER and WER, their high values might seem concerning at first, but they are reasonable and consistent with literature. The high error rates are the result of a combination of factors, namely the small training dataset, the greedy decoder instead of a beam search decoder, the absence of a language model and, of course, the model's hindered ability to learn due to the high learning rate. However, the resultant CER and WER, of 10.9% and 31.5% respectively, are to be expected, since a similar model with 68 million parameters achieves a WER of 29.23% on a 120-hour training set of clean speech [50]. Therefore, given our model's deficiencies and the 360-hour training set, the resultant performance is well within informed expectations.

Naturally, a re-run of the training would be warranted to try and mitigate the learning rate issue. Importantly though, real-world limiting factors come into play, which can be easily underestimated. In this case, the collaboration with FORTH to train the model on their hardware, while invaluable, also introduced some roadblocks. These were the necessary setting-up time and troubleshooting of migrating the code from one machine to another, communicating with FORTH personnel on how to execute, debug, and change it, schedule training iterations, upload the model's checkpoint files etc. It is easy to overlook these issues as mundane, but they separate theory from application and the whole process ended up consuming 3 months in total.

Furthermore, the model's actual performance is of little interest to this thesis, assuming the appropriate practices are followed for its training. The focus is on the final learned parameters, on which the various methods and research are applied. As these parameters are the result of valid training procedures

on valid datasets, the resultant statistical properties are also assumed to be valid, in the context of ML models. Thus, since the quantitative affirmation of our error rate in [50] further validates the procedure followed, the fact that the rates themselves are not impressive on their own right is inconsequential.

4.3 Robustness Analysis

Having obtained the trained model, the aim now is to compress it. This will be achieved via pruning unimportant weights and quantizing the remaining - important - ones. Both processes affect the resultant model's error rate, which will be used as the metric to compare various network compression configurations.

This analysis, however, will not be applied to the entire network. Aggregating the information in tables 4.1, 4.3, and 4.4, the distribution of total parameters per type of layer is obtained. This information is shown in table 4.6 and it is clear that focusing explicitly on the GRUs is a sensible decision, since they make up the overwhelming majority of the model's parameters and thus its computational and memory cost.

TABLE 4.6: Parameters per layer type

Total parameters	**CNN**	**GRU**	**FC**
62683005	250976	62385600	46429
100%	0.40%	99.53%	0.07%

Furthermore, the parameters within a GRU cell are subcategorized into gate and candidate parameters, each of which contains kernels - or weights - and biases. Copying from table 4.2, table 4.7 illustrates each subcategory of parameters.

TABLE 4.7: GRU cell parameter categories

Input/Output		**Reset gate**		**Update gate**		**Candidate state**	
Tensor	**Size**	**Tensor**	**Size**	**Tensor**	**Size**	**Tensor**	**Size**
x_t	$(1, D)$	W_{xr}	(D, H)	W_{xz}	(D, H)	W_{xh}	(D, H)
h_t	$(1, H)$	W_{hr}	(H, H)	W_{hz}	(H, H)	W_{hh}	(H, H)
		Gate kernels				Cand. kernels	
		b_r	$(1, H)$	b_z	$(1, H)$	b_h	$(1, H)$
		Gate biases				Cand. bias	
		r_t	$(1, H)$	z_t	$(1, H)$	h_t	$(1, H)$

The parameter distribution per category and layer is presented in table 4.8. Again, the overwhelming majority of parameters is concentrated in the kernels, thus it is sensible from a cost-benefit standpoint to focus the analysis on the kernels and ignore the biases altogether. For the remainder of the thesis, only the GRU kernels are taken into account, as the rest of the model is comparatively inexpensive by all accounts.

TABLE 4.8: GRU parameter distribution per category and layer

Layer	Gate		Candidate	
	Kernel	Bias	Kernel	Bias
GRU 1	10854400	3200	5427200	1600
GRU 2	7680000	3200	3840000	1600
GRU 3	7680000	3200	3840000	1600
GRU 4	7680000	3200	3840000	1600
GRU 5	7680000	3200	3840000	1600
Total	41574400	16000	20787200	8000
	66.64%	0.03%	33.32%	0.01%

4.3.1 Pruning

Firstly, the weights are inspected to determine their magnitude distribution, as shown in figure 4.4. Each distribution contains the respective kernels of all GRU layers. Since both distributions are qualitatively similar, the same methods can be applied to each kernel category, perhaps with quantitative differences.

The method of choice here is magnitude pruning, since it is easy to implement and suitable for bell shaped weight distributions. The goal is to determine a working magnitude domain, of the form in equation 4.6. Each weight is modified in accordance with equation 4.7, whereby a weight with absolute magnitude less than e is zeroed out, a weight with absolute magnitude greater than m is clipped, and all other weights are unchanged.

$$\mathbb{W} = [-m, -e] \cup [e, m] \tag{4.6}$$

$$w \leftarrow \begin{cases} 0 & \text{if } |w| < e \\ \text{sign}(w) \cdot m & \text{if } |w| > m \\ w & \text{otherwise} \end{cases} \tag{4.7}$$

Since the distribution peak is around zero, this method of pruning will remove a considerable amount of weights for relatively small values of e. As for the domain bounds, varying m does nothing towards pruning. Instead, this parameter, as well as its effects on the final error rate, are endured due to the upcoming quantization step. The narrower the domain, the higher the quantization resolution for a given number of quantization levels. Thus, while we seek to minimize m during pruning, its benefits will only be relevant during quantization.

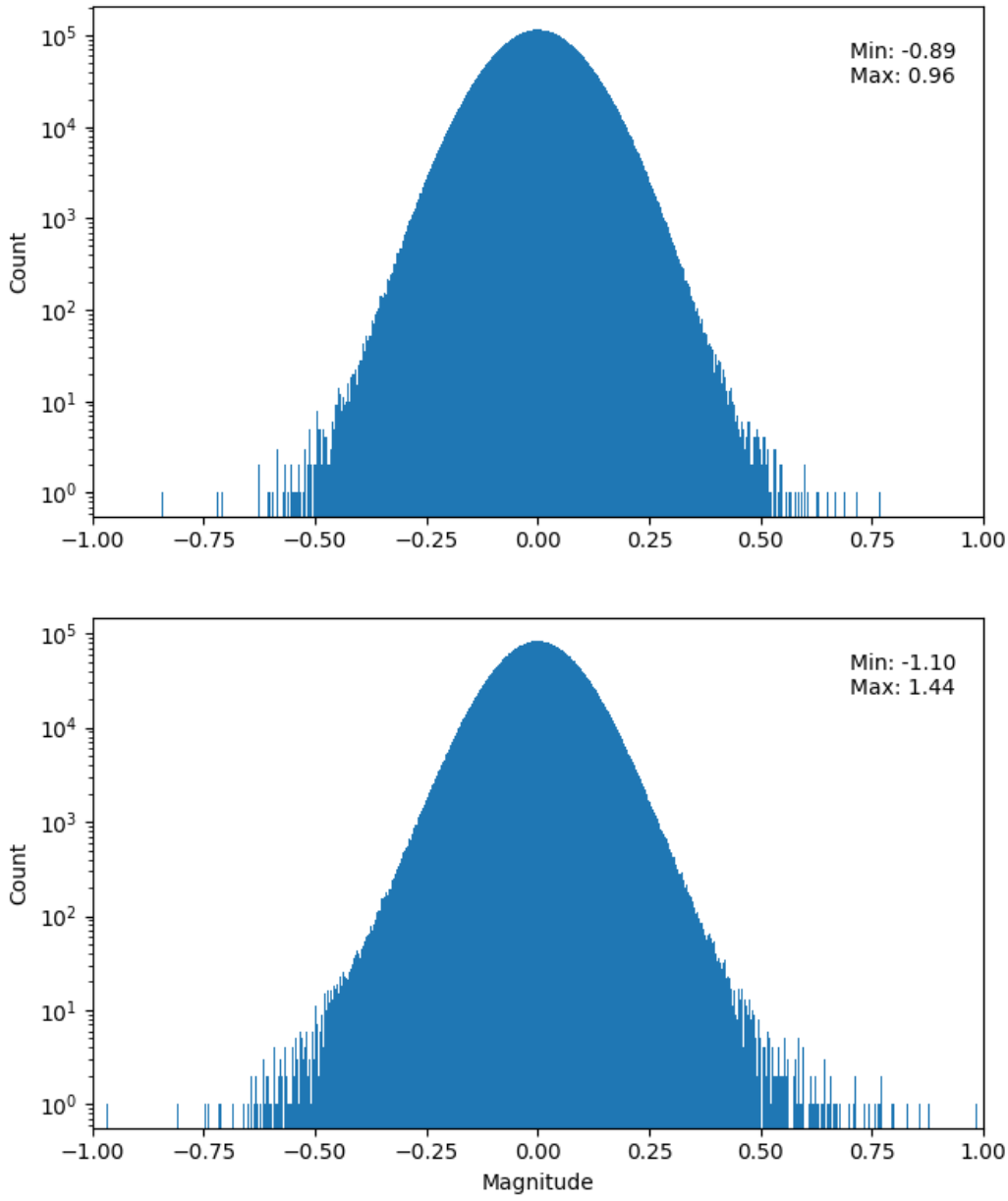


FIGURE 4.4: Weight distribution for gate (top) and candidate (bottom) kernels

Finally, despite the similar statistical properties of the gate and candidate kernels, in this chapter they will be studied separately. The reason is their functional difference in the GRU cell. As these kernels fulfill different roles and are used in conjunction with different activation functions, they are deemed sufficiently dissimilar to warrant separate analysis. Therefore, each kernel category is assigned its own domain and treated separately from the other, regardless of pruning strategy.

Global Magnitude Pruning

The first strategy we explore uses the same weight domain across all GRU layers and is therefore called Global Magnitude Pruning (GMP). The dimensional simplicity of selecting two values, the parameters m and e of equation 4.6, notwithstanding, the issue remains as to how these values are selected.

For every data point in the performance evaluation tests, the entire DeepSpeech2 model needs to go through the entire validation set. This computation takes approximately 7 minutes to complete, so the search space for the pruning parameters needs to be well-thought-out. To that end, a coarse search for e was done first, spanning several orders of magnitude. Then, a finer search followed, examining the subdivisions between the two most interesting orders of magnitude. As for the domain bounds, a few commonsense values for m were selected based on figure 4.4.

Initially, the individual effect on performance of pruning each category was investigated. In other words, how much the model's performance was degraded by pruning only one category of weights, without changing anything else on the baseline model. The results are shown in figure 4.5 and 4.6 for the gate and candidate weights respectively. Note that the baseline indicates the model's performance as obtained after training, prior to any modifications. Also, for every value of e the resultant sparsity percentage is provided as well. Sparsity denotes the percentage of weights that have been zeroed out.

These two figures illustrate the model's tolerance to reducing the weight domain bounds. The fact that we can reduce m quite a bit without any added performance degradation means that the overall domain width is reduced essentially "for free". Moreover, by comparing figure 4.5 to 4.6, it is evident that the candidate kernels are less tolerant to changes in m, compared to the gate kernels.

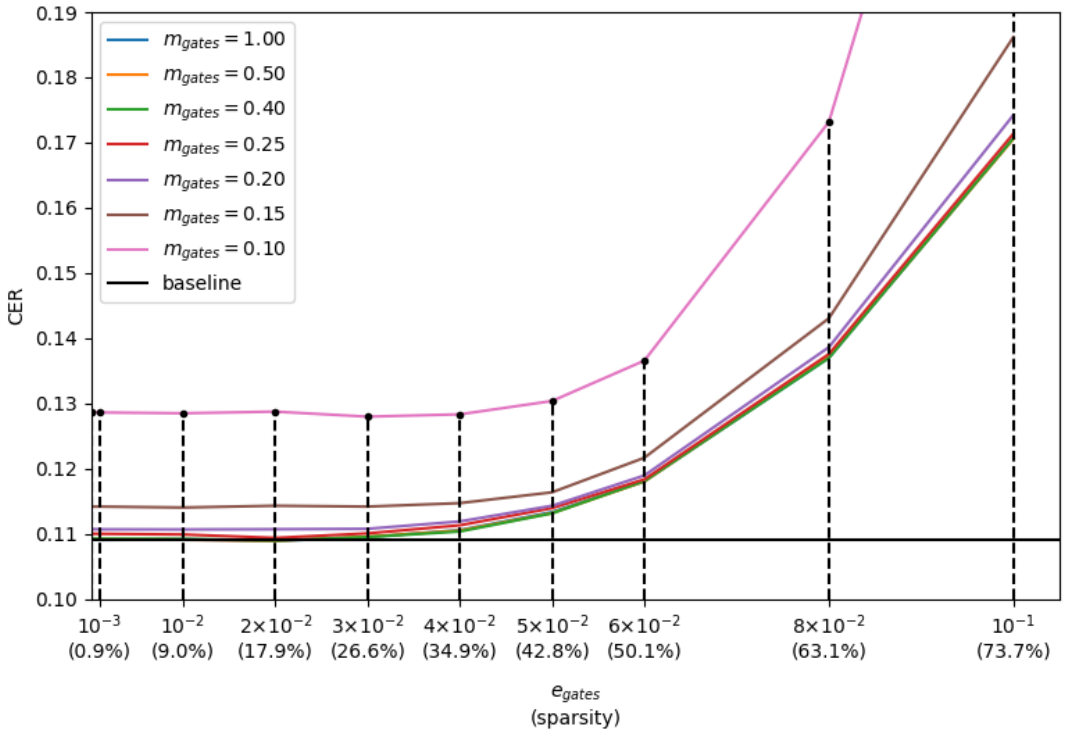


FIGURE 4.5: Gate kernels global pruning performance evaluation

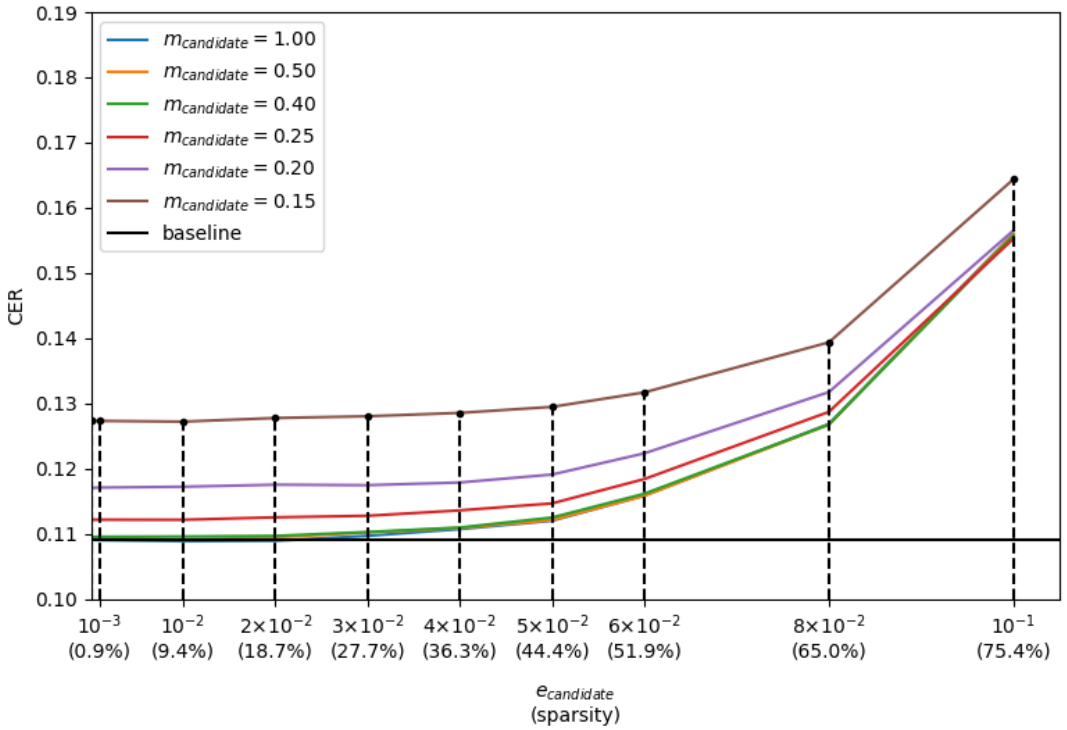


FIGURE 4.6: Candidate kernels global pruning performance evaluation

As for e, around 25% sparsity can be achieved in both cases with minimal performance degradation, while 50% sparsity can be achieved with a relatively small performance degradation. Inversely to m, candidate kernels are more tolerant to changes in e, compared to gate kernels. The observed differences between the two categories validate the decision to study the kernels individually and treat them as functionally different components.

Next, the same investigation is carried out, jointly for gate and candidate kernels. Naturally, the search space cannot be the Cartesian product of the search spaces of the previous experiments, due to the resultant quadratic expansion. Instead, we select a few combinations to narrow down the search space.

For m_{gates}, the values 0.20 and 0.25 are chosen, as the minimum values achieving the least accuracy degradation. The reasoning behind choosing both values, instead of only 0.20, is that gate kernels are more sensitive to m compared to candidate kernels, so it is interesting to examine differences between the two values. For $m_{candidate}$, 0.40 is chosen as the minimum value achieving the least degradation. As for e, values in the interval $[10^{-2},\ 6\times10^{-2}]$ are chosen for both categories.

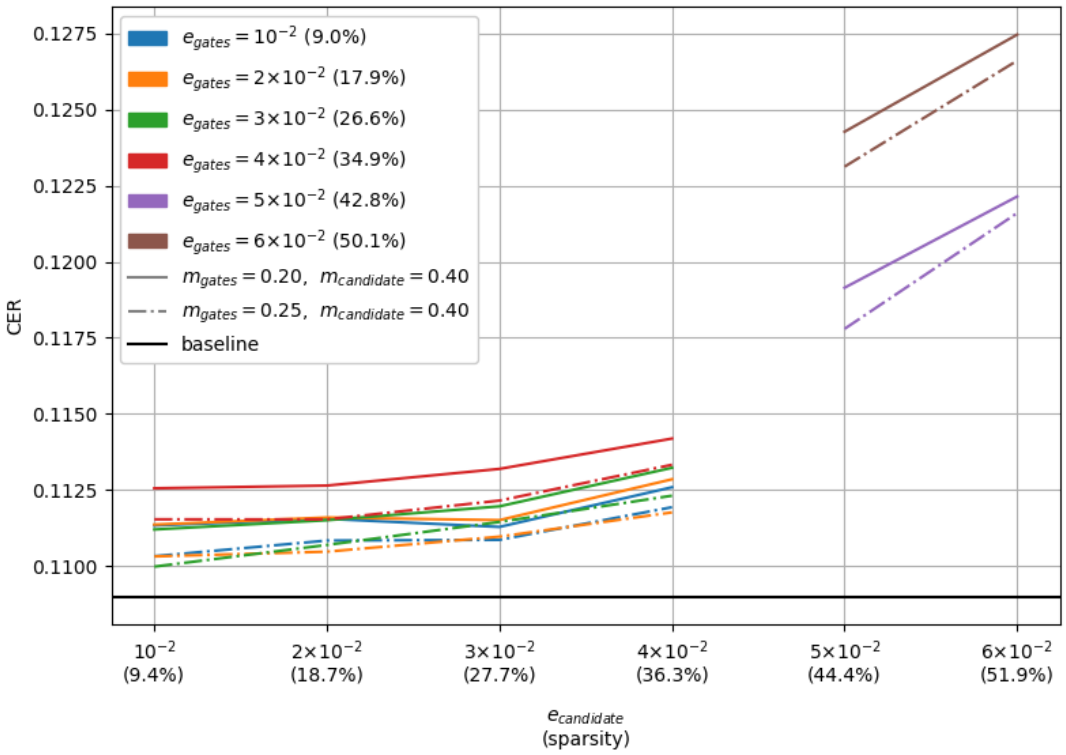


FIGURE 4.7: Joint global pruning performance evaluation

The results of the joint performance evaluation are shown in figure 4.7. The relative separation between the two m_{gates} values is mostly constant across experiments, at approximately 0.1 CER percentage points. Overall, we observe a - mostly - additive behavior in the accuracy degradation, whereby the total error rate degradation is approximately the sum of the individual degradation due to the gate and candidate pruning. Thus, keeping in mind that quantization will further increase the error rate, so far the tradeoff between error rate and sparsity is not at all impressive.

Layer-wise Magnitude Pruning

In an effort to improve sparsity and lower the incurred error rate penalty, a Layer-wise Magnitude Pruning (LMP) scheme is explored. With this scheme, each weight category has one domain per GRU layer. Specifically, the domain bounds will be common across layers, as there is no incentive to specify them individually, so only e needs to be specified per layer. Therefore, instead of a scalar value, e will hence be a vector, where the i-th element corresponds to the i-th layer.

The problem of deciding the search space has become considerably more complex, due to the dimensionality increase. A major observation which drastically reduces it has to do with the feature semantics of each layer in a DNN. Shallow layers extract and process lower level features compared to layers deeper in the architecture, which exact and process higher level features. If a network is overpruned in layers deep in the architecture, its ability to synthesize high level features, close to the output layer, is crippled. On the other hand, highly pruned shallow layers will influence the final output less, as high level features are synthesized by many lower level ones, which can be of poorer quality. Consequently, pruning intensity should be decreasing as a function of the layer index.

The joint LMP results are presented in figure 4.8. Inspecting the plots relating to 48.7% and 48.9% gate sparsity, as well as those of 50.3% and 50.6%, we see that pruning the last two layers less yields consistently better CER, compared to e vectors of very similar overall sparsity and more pruning on these layers. In other words, the 48.7% and 50.3% gate sparsity lines perform better than the 48.9% and 50.6% lines respectively, because the former shift the pruning intensity away from the last two layers. This fact is in accordance with our previous argument regarding the layer-wise decreasing pruning intensity.

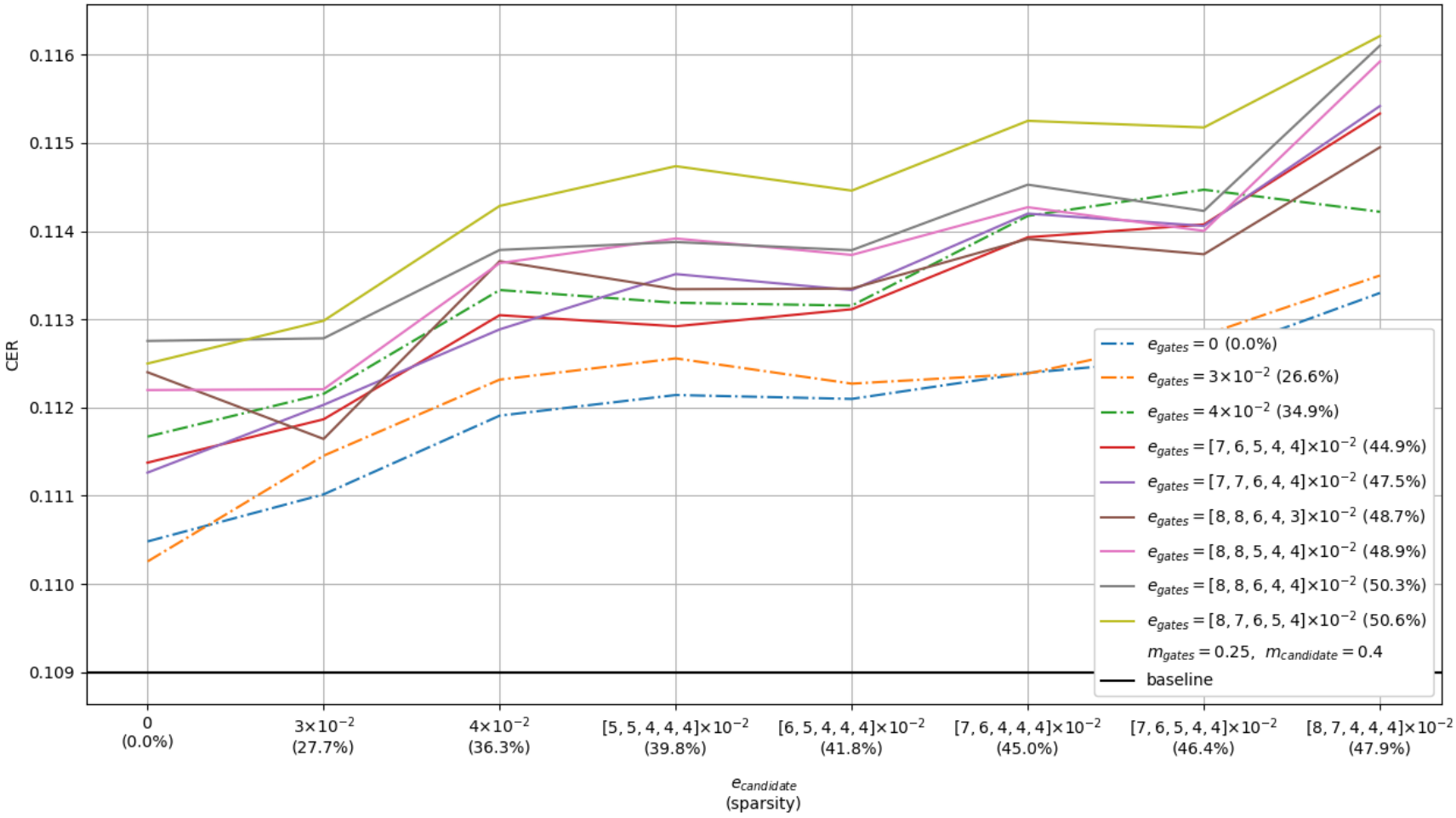


FIGURE 4.8: Joint layer-wise pruning performance evaluation

Another important observation is the improvement of the accuracy-sparsity tradeoff due to LMP. Apart from comparing figures 4.7 and 4.8, the improvement is demonstrated within figure 4.8. For the gates, the LMP lines are close to or better than the GMP line of 34.9% sparsity, except for the 50.6% line. As for the candidate, the slopes after the 36.3% candidate sparsity and up to 46.4% are relatively small, with an overall CER increase of less than 0.1 percentage point in every case. Consequently, LMP indeed yields better results compared to GMP.

Based on figure 4.8, a good tradeoff between accuracy and overall sparsity is achieved for $e_{gates} = [8, 8, 6, 4, 4] \times 10^{-2}$ and $e_{candidate} = [7, 6, 5, 4, 4] \times 10^{-2}$. For these vectors, the resultant weight distributions per layer are shown in figure 4.9. The achieved sparsity per layer is also annotated.

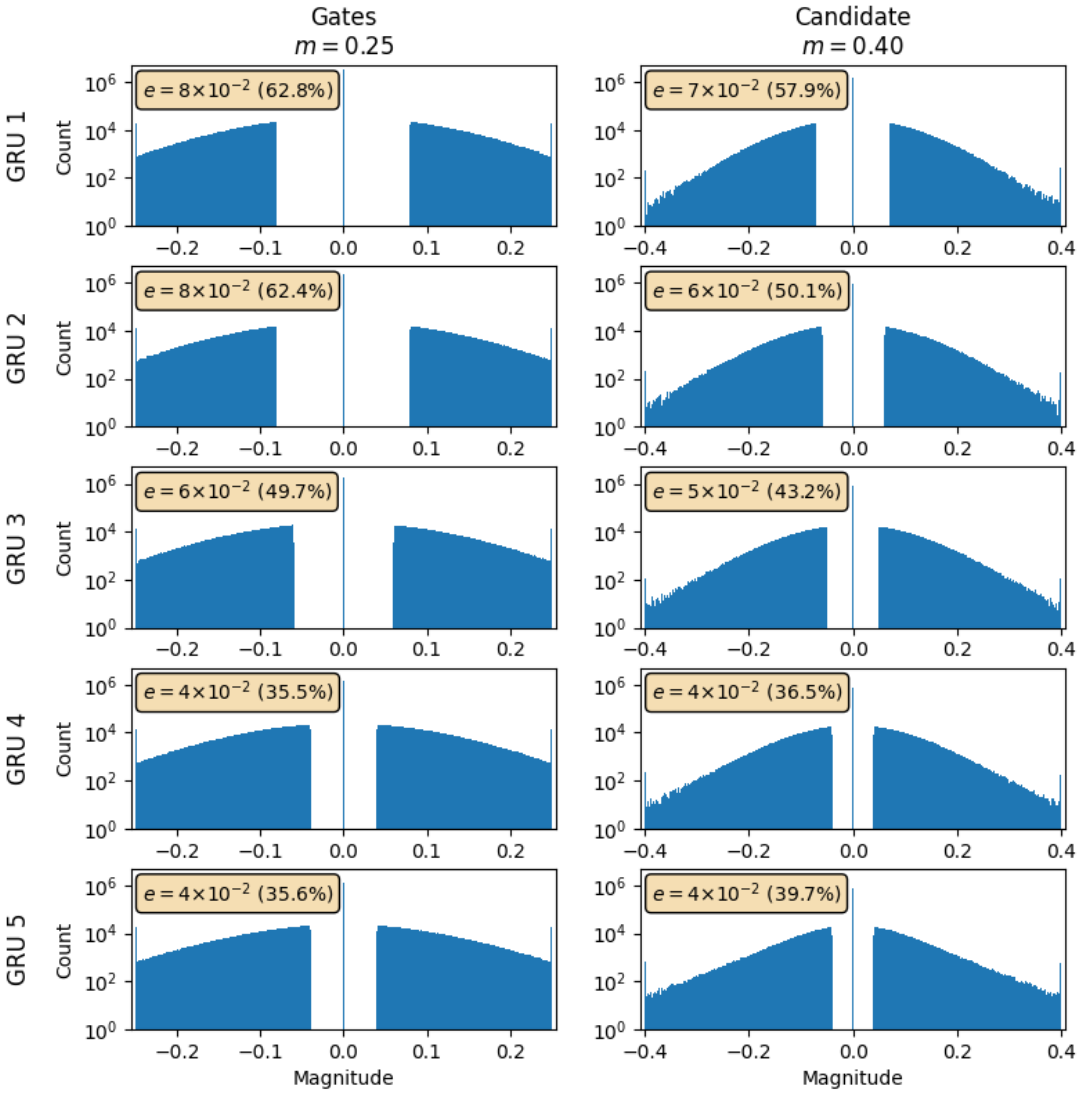


FIGURE 4.9: Pruned weight distribution

4.3.2 Quantization

Similarly to pruning, the effects of weight quantization are comprehensively studied. The experiment suite includes both pruned and non-pruned data points. For the non-pruned cases, there are two quantization bins representing zero, such that the quantization levels are always fully symmetric. Since the number of bins is a power of 2, if only a single bin were used for zero, the remaining odd number of bins would be impossible to split evenly between the positive and negative intervals. Therefore, in order to ensure comparability across experiments, the symmetric bin policy is adopted.

To avoid any confusion, we clarify the semantic difference of a zero value in pruned and non-pruned settings. In non-pruned settings, a weight with zero magnitude is in no way dissimilar to any other weight, in that computations containing it need to be performed. On the other hand, the quintessence of pruning is the outright removal of the zeroed out weights, such that computations containing them are not performed at all. Consequently, the zeroed out weights in pruned settings need not and should not be represented by any quantization level.

Uniform quantization

A quantizer with very simple implementation in software and hardware is the uniform dead-zone quantizer. This quantizer maintains even spacing between the quantization levels and is symmetric around zero. The dead-zone is particularly useful for quantization in conjunction with pruning, since the quantization bins can be positioned to span the weight domain exclusively. An example of such a quantizer is illustrated in 4.10, with the 3-bit encodings of the quantization levels also annotated.

Formally, a uniform dead-zone quantizer covering the domain of equation 4.6 is defined by

$$Q(x) = \text{sign}(x) \cdot \left(\Delta \left\lfloor \frac{|x| - e}{\Delta} \right\rfloor + e \right) \qquad x \in [-m, -e] \cup [e, m] \tag{4.8}$$

where Δ is the quantization step size defined as

$$\Delta = \frac{m - e}{2^{n-1} - 1} \tag{4.9}$$

for a quantizer with n bits.

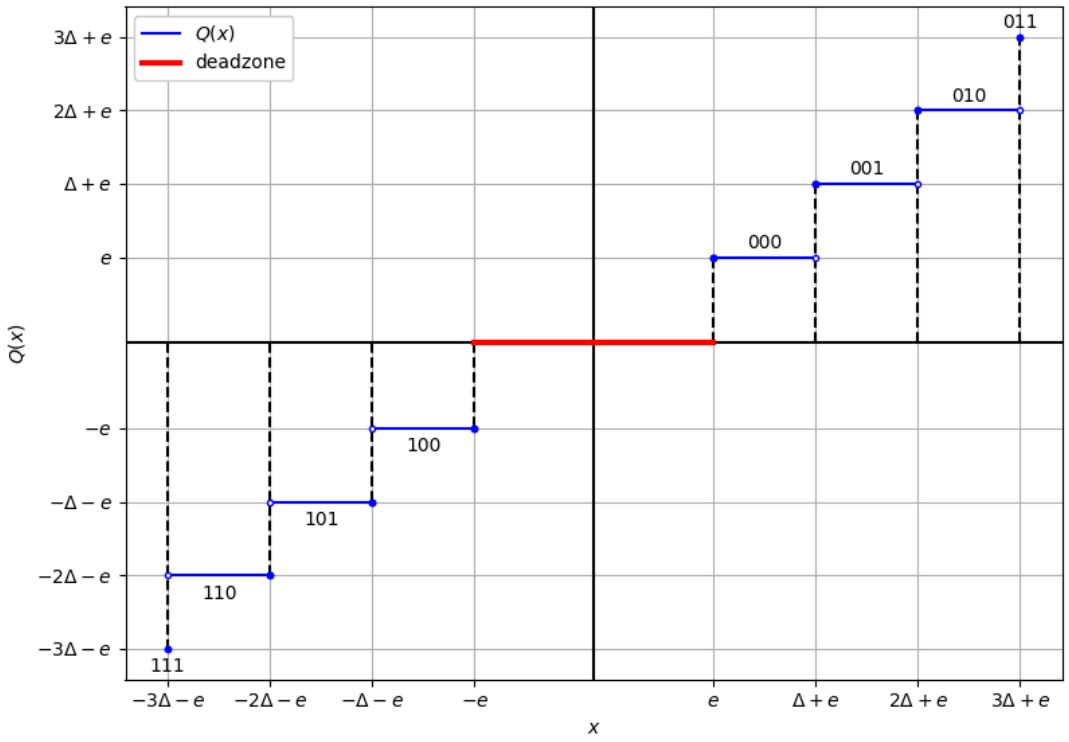


FIGURE 4.10: Uniform dead-zone quantizer example

The quantizer's classification rule, i.e. a mapping of the input to integers representing the quantization levels is given by 4.10 and the reconstruction rule, i.e. the mapping from the index to the quantization level is given by 4.11.

$$k = \text{sign}(x) \cdot \max\left(0, \left\lfloor \frac{|x| - e}{\Delta} \right\rfloor + 1\right) \qquad x \in \mathbb{W} \tag{4.10}$$

$$y_k = \Delta \cdot (|k| - 1) + \text{sign}(k) \cdot e \qquad |k| \in \mathbb{Z} \cup [1, 2^{n-1}] \tag{4.11}$$

The fact that the edge bins of the quantizer include only a single value, instead of a proper interval, is no accident, but a fully conscious design decision. Since quantization will be used in conjunction with pruning, the edge bins will include the clipped weights. Therefore, to assign proper intervals to the edge bins would be equivalent to reducing the weight domain bounds, i.e. m. From an experimentation standpoint, this equivalence would be entirely useless and confusing, so it is avoided altogether by having the edge bins represent singular values, equal to the domain bounds.

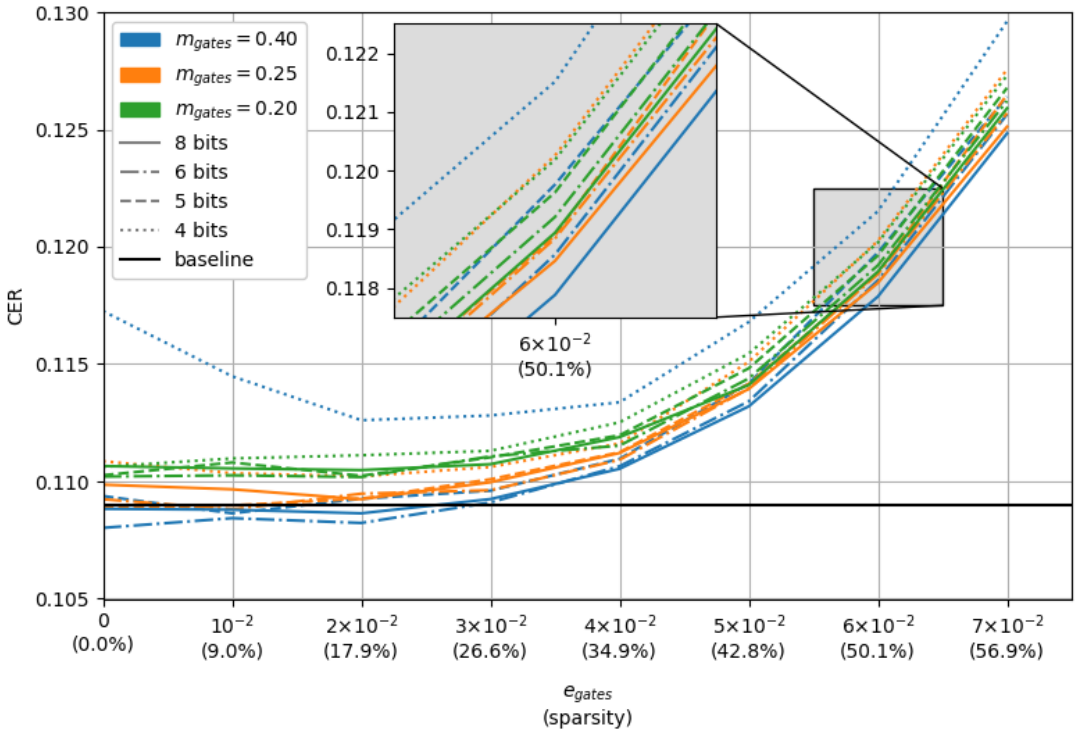


FIGURE 4.11: Gate kernels uniform quantization performance evaluation

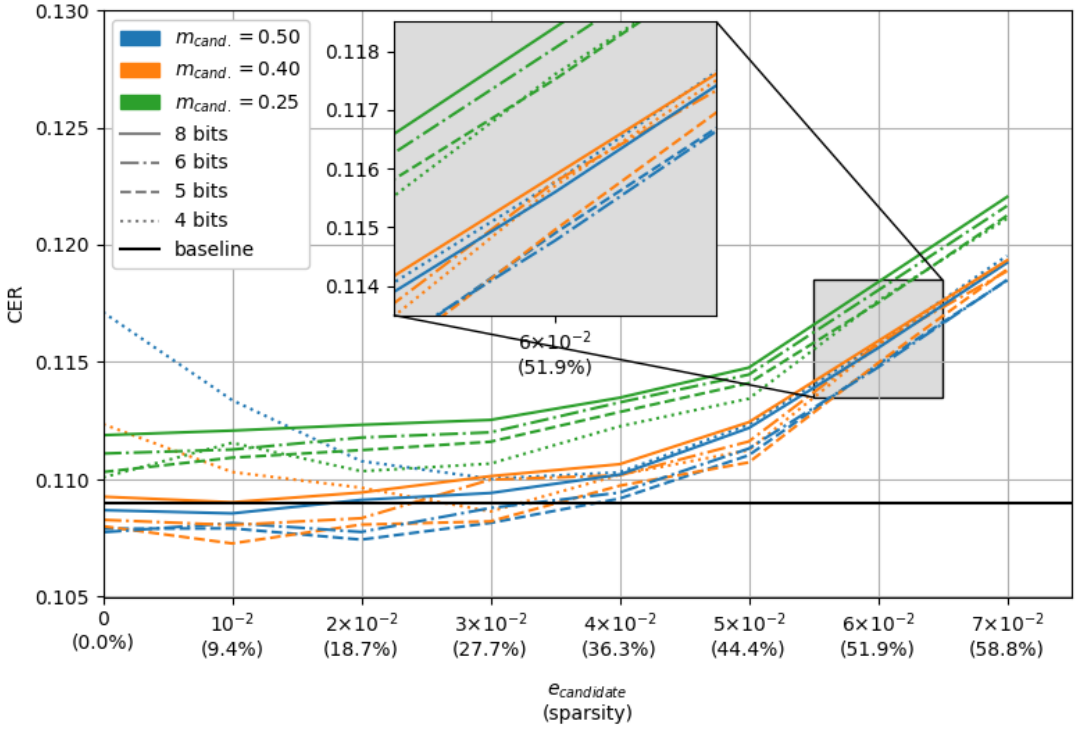


FIGURE 4.12: Candidate kernels uniform quantization performance evaluation

The effects of applying the dead-zone quantizer separately on the gate and candidate kernels are shown in figure 4.11 and 4.12 respectively. Overall, the spread for the various bit widths is relatively small, which indicates that the network is impressively resistant to a significant weight granularity decrease.

In both cases, the 4-bit variants of higher m values exhibit a bowl shape. This is most pronounced for zero e. The cause is rather interesting and is tightly coupled with Δ. Recall that when no pruning is applied, the quantizer does not have a dead-zone, rather it represents zero twice. Therefore, for zero pruning, the quantizer reduces all weights within $[-\Delta, \Delta]$ to zero. This is equivalent to pruning with $e = \Delta$. Note, however, that overall, quantizing unpruned weights and weights pruned with $e = \Delta$ are not exactly equivalent, due to the different number of bins representing non-zero values.

We will quantitatively elaborate for an example case, that of $m_{gates} = 0.40$, $e_{gates} = 0$ and $n = 4$ bits. By equation 4.9, we get $\Delta = 5.7 \times 10^{-2}$. The respective CER value is 11.7% which is similar to the CER value for $e_{gates} = 5 \times 10^{-2}$, all other parameters being equal. This effect is the result of a combination of factors, namely the curves in figures 4.5 and 4.6, the quantizer design of figure 4.10, and the choices for m and n yielding a Δ value in the "steep" region of the pruning performance evaluation figures (i.e. $\Delta > 5 \times 10^{-2}$).

Of course, while this explanation is sufficient to describe the CER behavior for a zero e, it is not immediately applicable to the $e = 10^{-2}$ cases, where the effect is again present. Indeed, in this case the weights in the smallest bins by absolute magnitude are quantized to 0.01, not 0, but since Δ is quite large, a significant proportion of weights are reduced by magnitude to a value very close to zero. As such, their significance is essentially diminished, comparably to the $e = 0$ case, hence the effect is still present, albeit less pronounced.

Lastly, the gap between the $m = 0.25$ curves compared to the rest of the curves in figure 4.12 is consistent with the findings in 4.6. However, the proportional relationship of the CER and the number of quantization bits generally observed in figure 4.12, for a given m, is counterintuitive and not present in figure 4.11. Unfortunately, no explanation has been found for this behavior.

Moving on, the results of the joint uniform quantization performance evaluation with GMP are illustrated in figure 4.13. Quite surprisingly, these results are, generally, as good as those of joint GMP without quantization in figure 4.7, despite the low bit width. For $n_{gates} = 5$, the results are marginally better compared to joint GMP without quantization, while for $n_{gates} = 4$ the results

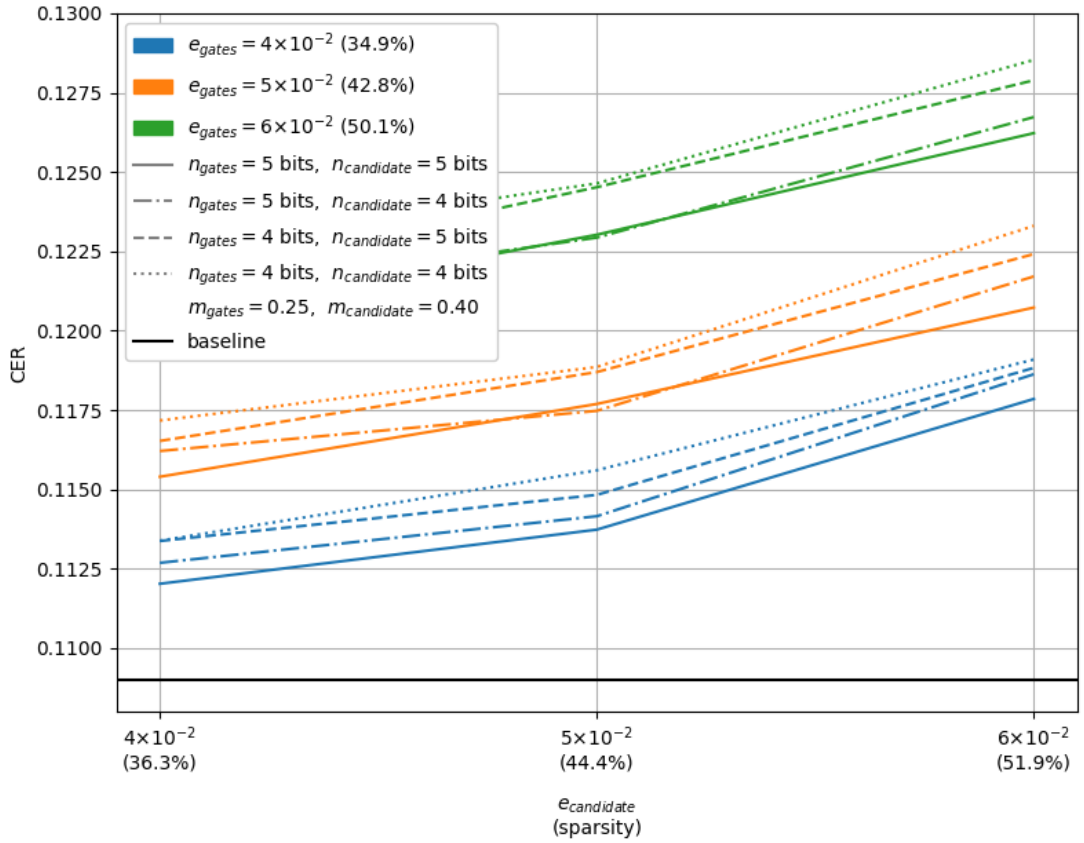


FIGURE 4.13: Joint uniform quantization performance evaluation with global pruning

are slightly worse. Notably, we observe the error rate to be more sensitive to changes in n_{gates} versus $n_{candidate}$.

Lastly, a similar quantization evaluation is performed with layer-wise pruning and its results are shown in figure 4.14. Again, as in figure 4.8, the set of curves corresponding to 48.7% and 50.3% gate kernel sparsity perform consistently better, mostly by wide margins, compared to the curves corresponding to 48.9% and 50.6% sparsity respectively, despite their insignificant difference in the sparsity percentage. Moreover, points on the 47.9% candidate kernel sparsity mark show a significant performance degradation compared to the 46.4% sparsity points, while, again in agreement with figure 4.8, the 46.4% candidate sparsity marks a local minimum for most curves.

Most importantly, layer-wise pruning and quantization vastly outperforms global pruning and quantization. This is a direct conclusion of comparing similar points in figures 4.13 and 4.14. Furthermore, the best accuracy-sparsity tradeoffs are on the 46.4% candidate sparsity.

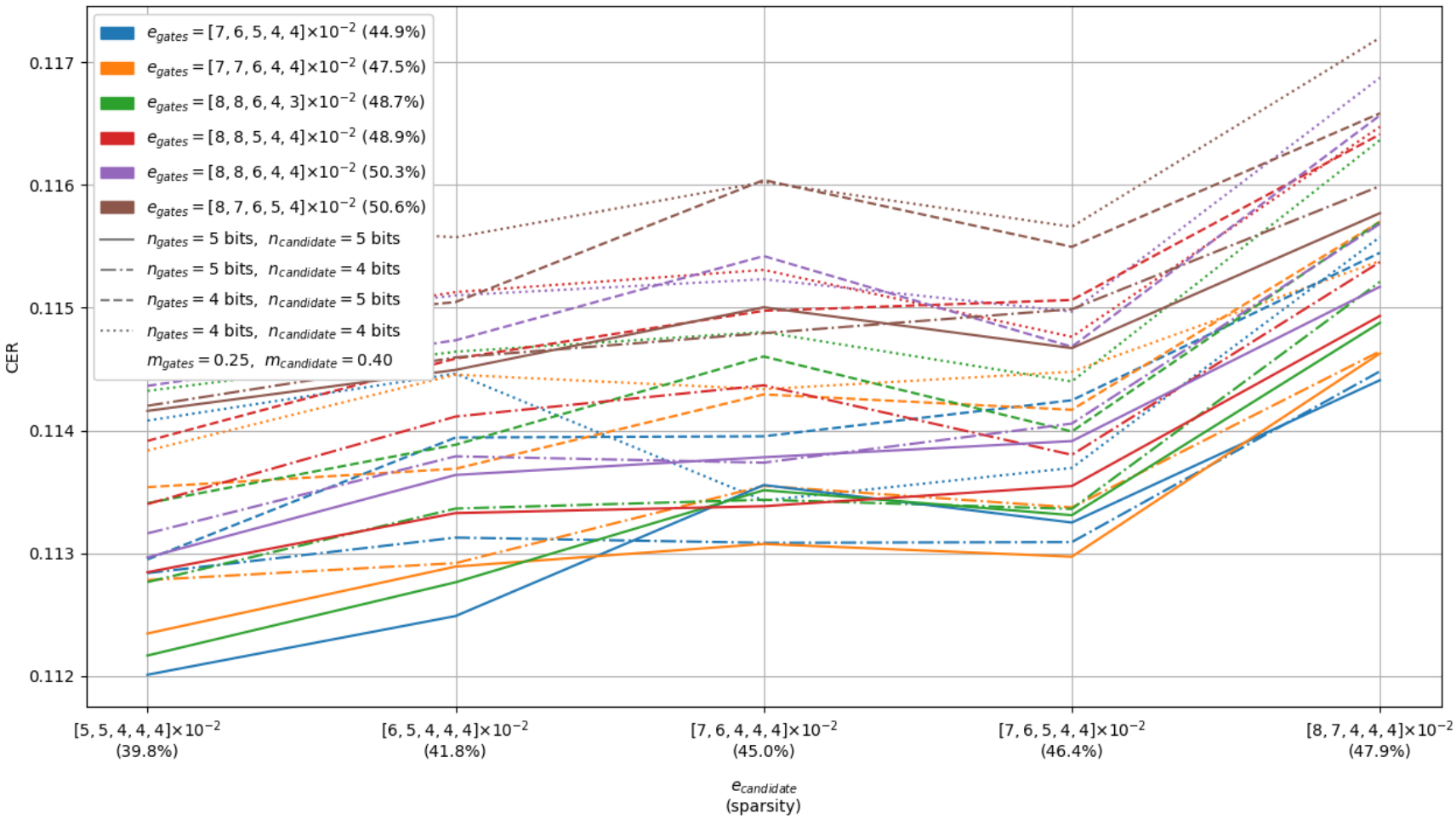


FIGURE 4.14: Joint uniform quantization performance evaluation with layer-wise pruning

Similarly to figure 4.9, the resultant weight distributions per layer are shown in figure 4.15. The same pruning vectors are used as in figure 4.9, since they still achieve a good tradeoff between accuracy and sparsity, and for consistency's sake. For both categories, 4 bits are used in the quantizer.

Note, again, that zero is not represented by any quantization level for pruned weights. However, in figure 4.15, as in 4.9, the zeros are added in the histograms for completeness and comparability purposes. Also, mostly on the gate histograms, the edge levels corresponding to $\pm m$ magnitude are very thin, being single points, and with counts very close to the adjacent bins, they are not always clearly visible.

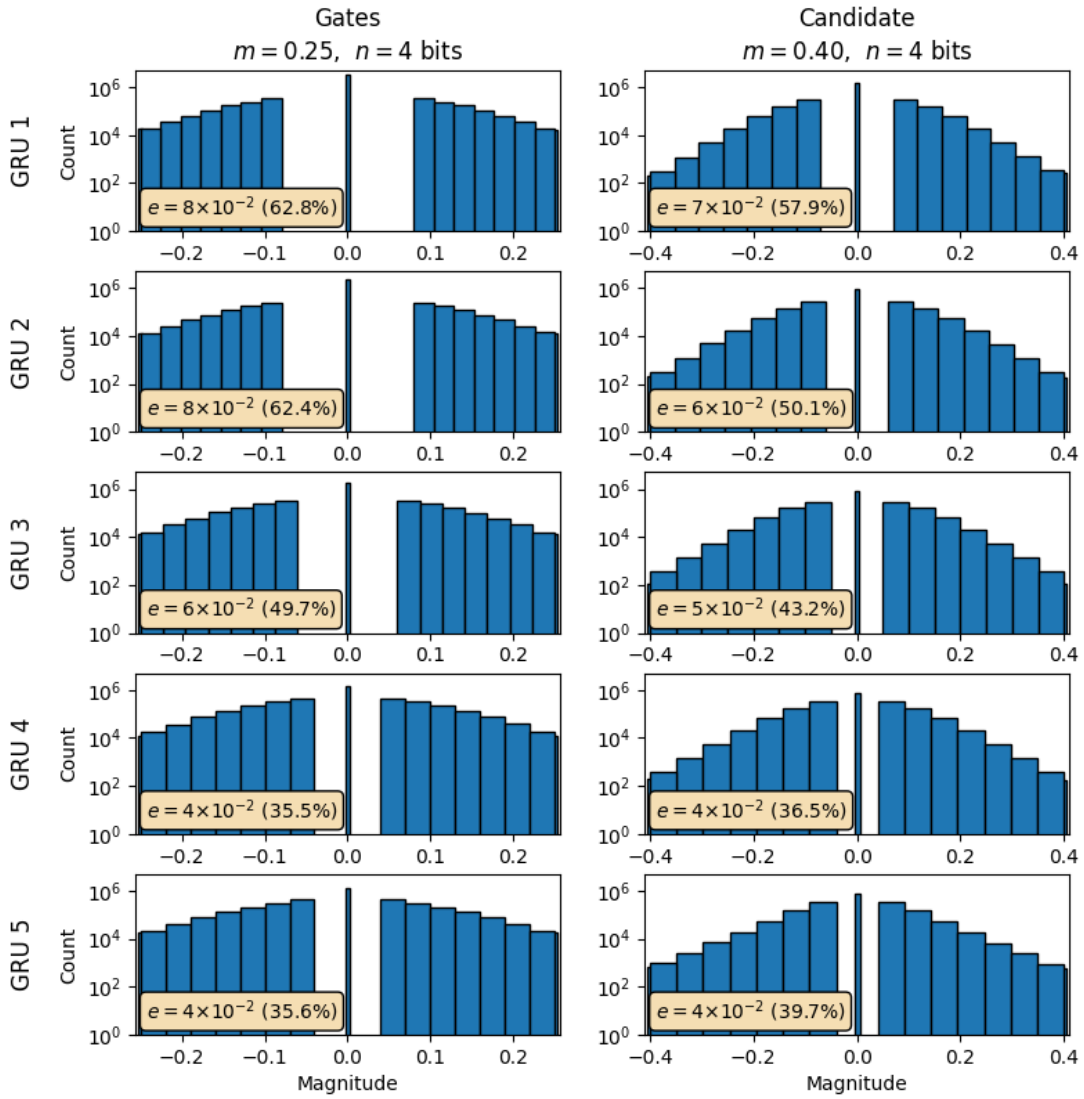


FIGURE 4.15: Pruned and quantized weight distribution

Global K-means quantization

The uniform quantizer of the previous section was selected as the starting point of the quantization analysis due to its simplicity. However, there is no guarantee or indeed assumption regarding its suitability in the case at hand. As a result, we now explore a fundamentally different quantization scheme.

K-means clustering is an algorithm, in the unsupervised learning family, which aims to partition the given data into a predefined number of clusters, by minimizing the within-cluster variance of the data. Each cluster is represented by the centroid, i.e. the mean of the data points assigned to the cluster.

With K-means quantization, the magnitude of all weights is set to their respective cluster's centroid. In other words, our quantizer quantizes each weight to the closest centroid. Formally, if $\mathbf{C} = \{c_k \mid k \in [1, K]\}$ is the set of all K centroids determined by the K-means algorithm, the K-means quantizer is defined by equation 4.12. The classification and reconstruction rules are shown in equations 4.13 and 4.14.

$$Q(x) = \arg\min_{c_k \in \mathbf{C}} ||x - c_k|| \qquad x \in \mathbb{W} \tag{4.12}$$

$$k = \arg\min_{k \in [1,K]} ||x - c_k|| \qquad x \in \mathbb{W} \tag{4.13}$$

$$y_k = c_k \qquad k \in \mathbb{Z} \cup [1, K] \tag{4.14}$$

Here, the clustering algorithm is run globally, by aggregating the weights of all layers for each category. Since the experiment suites of the previous sections were quite comprehensive, in this section we speed up the process by investigating only the joint performance evaluation with layer-wise pruning, around the area of interest as determined via figure 4.14.

The results are presented in figure 4.16, where $K = 2^n$ clusters were used per kernel category. Comparing them to the corresponding curves in figure 4.14, it is evident that this particular approach performs consistently worse compared to uniform quantization. Unsurprisingly, possible reasons for the poor performance might be the global application scope of the method.

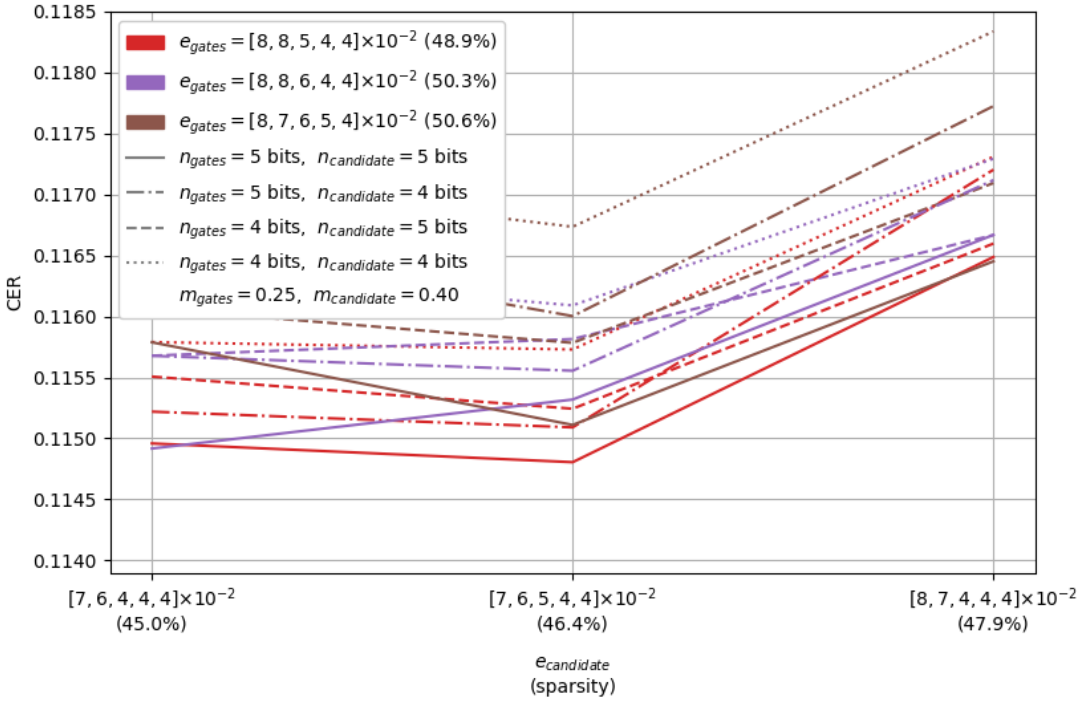


FIGURE 4.16: Joint global K-means quantization performance evaluation with layer-wise pruning

Layer-wise K-means quantization

In an attempt to improve the results of K-means quantization, we now investigate its layer-wise application. However, an additional condition is imposed. The centroids need to be symmetric around zero. The reason for this restriction is the subsequent reduction in hardware resources necessary to implement the reconstruction rule. Namely, the lookup tables storing the centroids per layer and category can be half their original size, due to the imposed symmetry.

To enforce the symmetry, the algorithm is simply run on the absolute weight magnitudes and the resultant centroids are expanded to cover both real number half-lines. As the weight distributions per layer and category are almost symmetric themselves, we can safely assume that the imposed centroid symmetry will not affect the error rate in any significant way.

The results of layer-wise symmetric K-means quantization are presented in figure 4.17. While there is an undeniable improvement over global K-means, due to the layer-wise scope, these results are still lacking compared to uniform quantization in figure 4.14. This is attributed to the fact that the centroids are the cluster centers and, as such, cannot be the values $\pm e$ and $\pm m$.

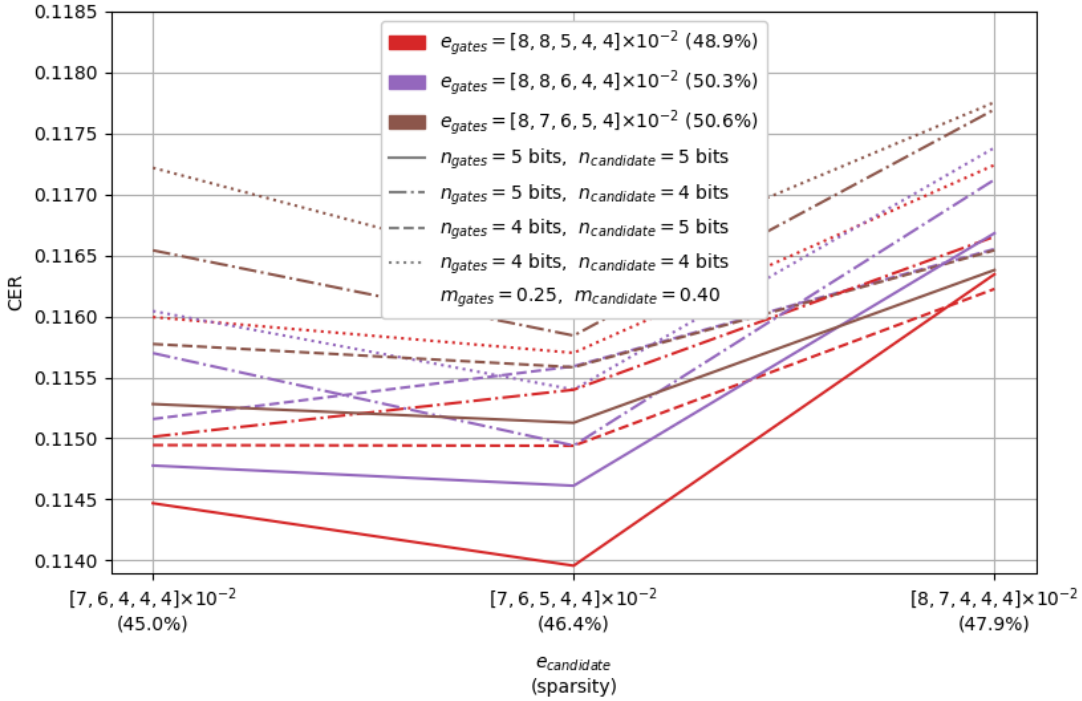


FIGURE 4.17: Joint layer-wise symmetric K-means quantization performance evaluation with layer-wise pruning

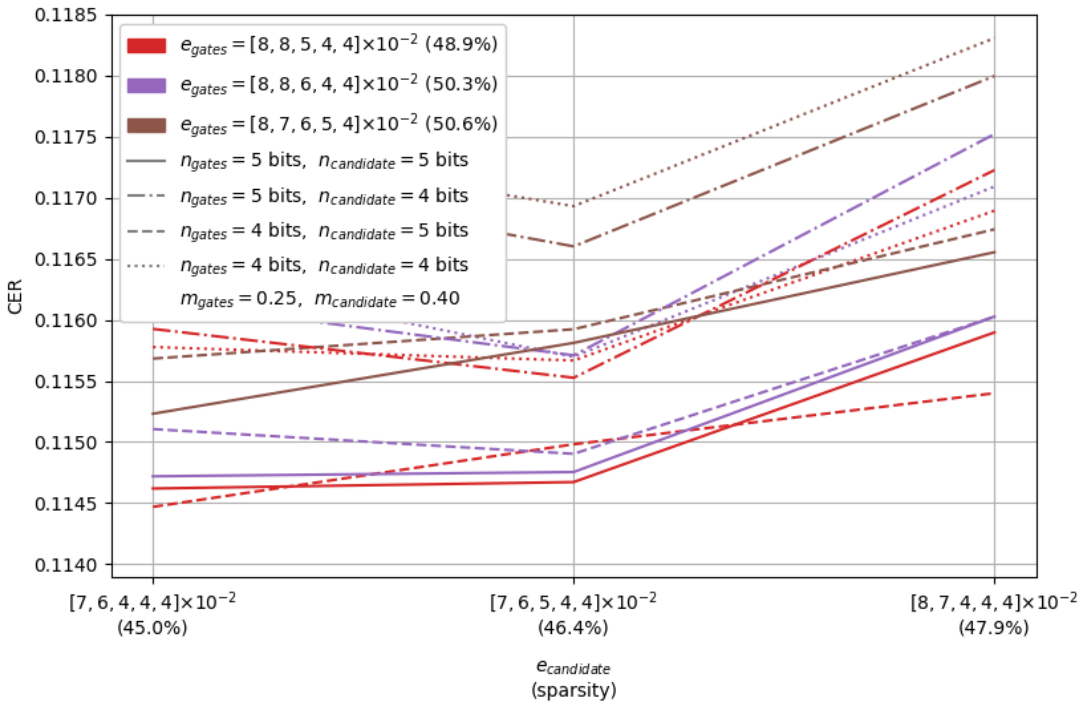


FIGURE 4.18: Joint layer-wise symmetric anchored K-means quantization performance evaluation with layer-wise pruning

Lastly, we try to further improve the K-means results by forcing two centroids to be on $\pm e$. This is achieved by adding two artificial weights to the data supplied to the algorithm, whose magnitudes are $\pm e$ but with the difference that they are assigned infinite clustering weight, while the actual weights have a unit clustering weight. In weighted clustering, the algorithm tries to minimize the weighted sum of the within-cluster variance, so the clustering weight is a measure of influence each data point exerts on their respective centroid. Thus, adding the two artificial points to the data, we essentially anchor two centroids at $\pm e$.

The results of this modification are illustrated in figure 4.18. Compared to 4.17, some curves are better while others are worse. Effort notwithstanding, uniform quantization still outperforms K-means pruning.

Ordinarily, further investigation would be carried out to determine if anchoring centroids only at $\pm m$, and jointly at $\pm e$ and $\pm m$ would improve the K-means results. However, given the extent of the analysis so far and the consistent superiority of the uniform quantizer, such an effort would not be practically beneficial. Even if K-means quantization yields the same results as uniform quantization, the latter has, by design, additional implementation benefits. The reconstruction rule of the uniform quantizer can be implemented either mathematically or via lookup table, while K-means requires a lookup table. Therefore, for this particular problem, interest in K-means has dwindled.

4.3.3 Final results

Having gone through the entire robustness analysis, we conclude this chapter by selecting a satisfactory combination of pruning and quantization parameters. As has already been established, our options will be drawn from the 46.4% candidate sparsity pool and with the uniform quantizer applied. Since there are no specific CER or WER requirements, our notion of a satisfactory tradeoff will be somewhat arbitrary, in terms of error rate.

Consulting figure 4.14, we will compile a list of contenders. Due to the absence of CER requirements, we opt for the two highest gate sparsities, since gate kernels contribute 66.6% of total GRU parameters, as shown in table 4.8. Of these options, we immediately eliminate those with very similar CER, but longer bit widths. The remaining options are presented in table 4.9 with additional relevant details.

TABLE 4.9: Comparison of pruning and quantization final contenders

Option	Bits and sparsity		Overall sparsity	Memory (MB)	CER	WER
	Gates	Candidate				
1	4, 50.3%	4, 46.4%	49.0%	15.9	11.49%	33.40%
2	4, 50.3%	5, 46.4%	49.0%	17.3	11.46%	32.92%
3	5, 50.3%	4, 46.4%	49.0%	18.5	11.40%	32.95%
4	4, 50.6%	4, 46.4%	49.2%	15.8	11.56%	33.57%

The memory footprint per category is computed as the number of bits of the quantizer multiplied by the number of total remaining weights after pruning. The overall memory footprint is the sum of the footprints per category. Of course, this computation is extremely reductive, as it does not account in any way for the spatial information necessary to encode the actual matrices. In other words, some form of spatial information is required to disambiguate the actual position of each weight inside their respective matrix. The described computation, and thus the memory information on table 4.9, take into account only the remaining, raw weights, in the form of quantization indices, after pruning and quantization.

The first option to eliminate is 4, since it has the highest CER and WER, and is only 100 KB less than option 1. The remaining options exhibit an inverse relation between memory and CER. Taking into account the broader context in which the robustness analysis was performed, namely the hardware implementation of DeepSpeech2, the option of choice is the first one, due to its smaller memory footprint. As this amount of data can fit on the dedicated on-chip memories of only the absolute highest-end FPGAs, for any practically accessible FPGA the weights have to be stored on external memory modules. Data transfers from external memories are orders of magnitude more expensive, energy- and latency-wise, compared to on-chip computations. Therefore, cutting down the external memory requirements as much as possible is a priority, due to which we are led to choose option number 1.

Overall, in this section, we significantly reduced the memory footprint of the GRU weight kernels, via pruning and quantization. This was achieved without retraining the model and with great care not to degrade the model's accuracy significantly. We should also note, to avoid any ambiguity, that the quantization applied reduces the resolution of weight magnitudes, but not their precision, or the precision of computations on the entire model.

The benefits of the pruning and quantization methods notwithstanding, they have introduced new issues regarding the implementation. A major issue left purposefully undiscussed is the encoding of the sparse matrices. Several such encodings exist in literature, e.g. Compressed Sparse Row (CSR) [91] or Compressed Sparse Column (CSC) [92], and the answer to this problem will impact the full memory footprint of the sparse matrices. However, the sparse matrix encoding is closely related to the chosen sparse matrix compression scheme, both of which are discussed in the next chapter.

Chapter 5

Compression

In this chapter presents the main contribution of this thesis, which is the development of a novel entropy-based compression method, suitable for hardware implementation in machine learning inference applications. The method achieves - to a degree - design-specified decoding throughput and memory footprint by trading off compression ratio, while its mathematically decodable codes will allow for fast FPGA decompressors.

5.1 Motivation

The aim of pruning and quantizing the GRU weight matrices was to lessen the storage and computational burden of the network. The achieved 49.0% sparsity might seem significant, but is actually surprisingly low for what would be considered an equitable tradeoff between the increased encoding and hardware complexity due to sparse matrices versus their memory and computational benefits.

Sparse matrices introduce inherent representation overheads, due to their irregular structure. For such low sparsities, the encoding overhead outweighs the actual information stored, i.e. the quantized weights. It is therefore prudent to compress the kernels with information theoretic approaches, in order to alleviate the overhead.

For this particular case, the sparse matrices will be compressed once, outside the hardware implementation, and decompressed and used constantly. Therefore, given the nature of the application, the compression method must be efficient in terms of decompression complexity, throughput, compression ratio and decoder memory footprint.

Entropy-based compression is one of the fundamental classes of compression techniques, with the most notable general-purpose technique being Huffman Coding [33]. While its compression ratio is generally close to the entropy limit, its memory footprint and decompression throughput are less impressive. For an alphabet of n symbols, Huffman's space complexity is $O(n)$ and the decoding time complexity is $O(d)$, where d is the depth of the tree.

Much work has been done to achieve the above space complexity and improve the time complexity, by efficiently implementing the Huffman tree. In [93], they split the tree information across three tables and achieve a memory footprint of $5n$. The bound is reduced in [94] to $\approx 1.6n$, by exploiting features of the single-side grown Huffman tree to condense the memory representation. In all cases however, the space complexity is at least $O(n)$.

As for the decoding speed, [95] uses canonical codes to reduce the decoding time from a single cycle per bit in [33], to a "few operations per symbol". A variant has also been proposed in [96] which utilizes quaternary instead of binary trees and brings the decoding time down to $O(\log_4 n)$. Lastly, [97] achieves a decoding complexity that is independent of n, d or the codeword length. In all cases, the decompression throughput is less than a single symbol per cycle.

A notable mention is [98], where they propose an $O(1)$ decoding time complexity by parallelizing the process across d processors. Still, for high performance applications, the single symbol per cycle and linear space complexity limits can be prohibitive, to say nothing of the processor count.

The compression scheme would need to have constant decoding time, low resource utilization and high decompression throughput, to feed the matrix multiplication pipelines. Implementation variants of Huffman compression fail to meet all the requirements, so a novel method is explored.

5.2 Proposed Method

Let $\mathcal{Y}$ be the base-symbol finite alphabet of the source X that outputs a string $x_0x_1x_2...$ of base symbols, with $x_i \in \mathcal{Y}$. Similarly to [95], there always exists a mapping from $\mathcal{Y}$ to $\mathcal{Z} = \{0...|\mathcal{Y}| - 1\}$, such that $y_i \mapsto i$. In other words, we can always represent the symbols of $\mathcal{Y}$ with integer indices. Hence, the term "symbol" will refer to these indices. The output of X can now be written as $z_0z_1z_2...$ with $z_i \in \mathcal{Z}$.

We define an L-sequence as a sequence of L consecutive symbols in the output of X, starting at integer multiples of L. I.e., L-sequences are all sequences of the form $z_{\alpha L}\ z_{\alpha L+1}\ \ldots\ z_{(\alpha+1)L-1}$ with $\alpha \in \mathbb{N}$. We assume that the frequency distribution of distinct L-sequences is known or computable.

Penalty Arranged Tree Hierarchy (PATH) compression is an easily decodable, variable-length encoding which compresses these L-sequences. The L-sequences are mapped onto a tree – of any degree – such that each node contains a single symbol and that there exists, for "most" L-sequences, a continuous path from a node towards the root that generates them. Thus, any L-sequence present in the tree is fully encoded by the index of a starting node that generates it. Unmapped sequences will have to be fetched from memory uncompressed.

For this scheme, the following questions need to be answered: What properties should the tree exhibit? How do we assign symbols to nodes? What is an efficient set of addressing modes? Each of these points are discussed below.

5.2.1 Data Structure

Assuming independent and identically distributed symbols that follow a non-uniform frequency distribution, we reasonably expect the most frequent L-sequences to exhibit significant overlap, since high-frequency subsequences will be part of high-frequency L-sequences. Therefore, it is desirable that our data structure be able to translate spatial proximity into numerical (node addresses) proximity since we can then implement a caching mechanism to further reduce file sizes. In layman's terms, the interdomain proximity requirement means that sequences close to each other on the tree should have small differences in their nodes' addresses while sequences far from each other should have larger address differences.

The second property for our tree stems directly from Shannon-Fano coding [32, 99] and Huffman coding [33]. It is necessary for higher-frequency sequences to have shorter encodings. Consequently, the tree needs to be divided into penalty regions, to provide variable-length encodings. However, these regions need to be mathematically, on-the-fly decodable, as opposed to arbitrary in the case of Huffman. Easy decodability for the penalty regions will be a source of compression ratio inefficiency, but will ultimately allow for fast decoding.

A tree formula found to satisfy the interdomain proximity requirement and provide penalty regions is the following

$$children(n) := \begin{cases} \{2n, n+2\} & \text{if } n \text{ is odd} \\ \{2n\} & \text{if } n \text{ is even} \end{cases} \tag{5.1}$$

and its inverse

$$parent(n) := \begin{cases} n-2 & \text{if } n \text{ is odd} \\ n/2 & \text{if } n \text{ is even} \end{cases} \tag{5.2}$$

and the resultant tree for an address width, N, of 4 bits is presented in figure 5.1.

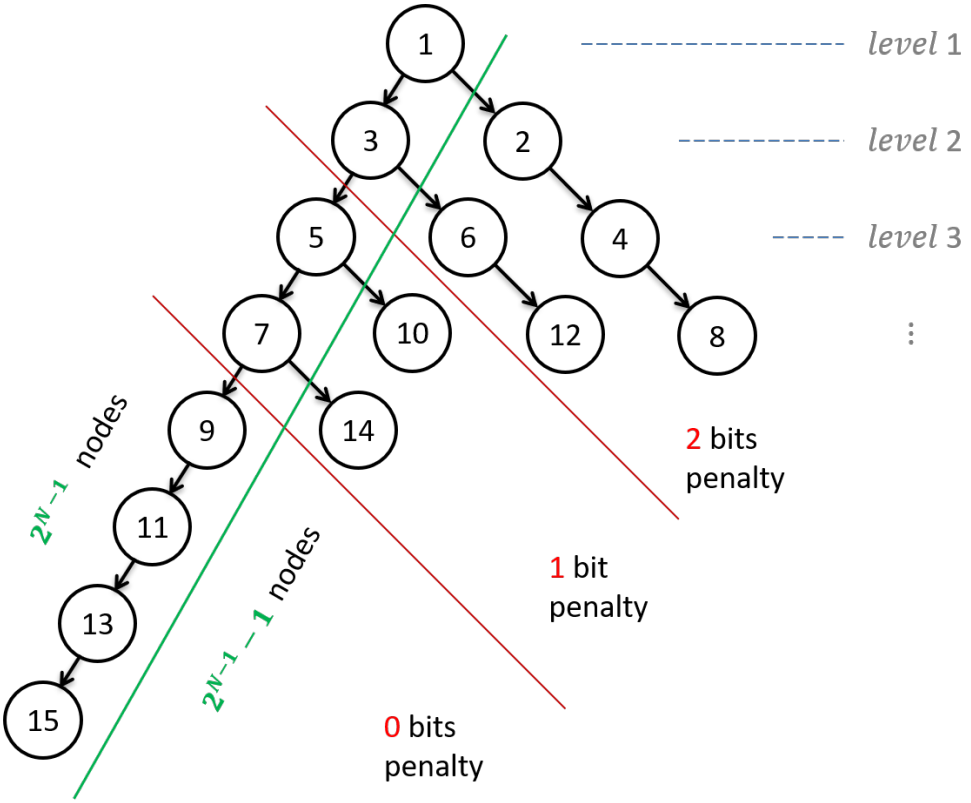


FIGURE 5.1: Structure and penalty regions of proposed tree for 4-bit addresses

The odd numbers form a primary branch, from which all the even nodes branch out. From a given primary node, we can address any node in the right branch (including the primary node) with some extra, "penalty" bits. A penalty group consists of all nodes in the tree that require the same number of penalty bits to be addressed, given their primary node.

The primary branch contains 2^{N-1} elements, creating

$$PGs := \lceil \log_2 N \rceil + 1 \tag{5.3}$$

penalty groups (PGs). The proof of equation 5.3 is derived as follows: node 1 can be doubled $N-1$ times, thus requiring $\lceil \log_2 N \rceil$ penalty bits to address all N nodes, node $2^{N-1}+1$ cannot be doubled, thus requiring zero penalty bits. Due to the continuity of numbers and of the log function, all penalty widths in the range $[0, \lceil \log_2 N \rceil]$ will appear.

The contained nodes per penalty group (NPPG) are

$$NPPG(k) := \sum_{i=\lfloor 2^{k-1} \rfloor}^{\min(2^k-1, N-1)} (i+1)\lceil 2^{N-i-2} \rceil \tag{5.4}$$

where $k \in [0, \lceil \log_2 N \rceil]$ is the number of penalty bits. The summation bounds denote the minimum and maximum number of times an odd node can be doubled in the given penalty group. In the upper bound's min function, $N-1$ becomes relevant only for the highest penalty group (the one at the root) and only when N is not a power of 2. The term $(i+1)$ in the summation element denotes the total nodes in a right branch of i even nodes (i doublings of the odd root), including the branch's root. The term $\lceil 2^{N-i-2} \rceil$ denotes the total right branches of length i. We arrive at this expression by finding how many odd nodes can be doubled exactly i times, as follows:

$$i \in [0, N-1] \quad \text{(the times doubled)} \tag{5.5}$$

$$m := 2x+1,\ x \in \mathbb{N} \quad \text{(the odd node)} \tag{5.6}$$

$$2^i \cdot m < 2^N \leq 2^{i+1} \cdot m \implies 2^{N-i-1} \leq m < 2^{N-i} \tag{5.7}$$

$$\begin{aligned} (5.7) &\overset{(5.6)}{\implies} \quad 2^{N-i-2} - \frac{1}{2} \leq x < 2^{N-i-1} - \frac{1}{2} \\ &\overset{(5.5)}{\implies} \quad \lfloor 2^{N-i-2} \rfloor \leq x < 2^{N-i-1} \\ &\implies \text{true for } \lceil 2^{N-i-2} \rceil \, x\text{s} \quad \text{(odd nodes)} \end{aligned}$$

The starting (odd) node of each penalty group is given by the function

$$SNPG(k) := \begin{cases} 1 & \text{if } k = \lceil \log_2 N \rceil \\ 2^{N-2^k} + 1 & \text{if } 0 \leq k < \lceil \log_2 N \rceil \end{cases} \tag{5.8}$$

since 2^k denotes the maximum possible number of nodes in a right branch (including its root) in PG-k, thus the sum of leading and trailing zeros in the N-digit binary representation of a PG-k node's address will be at most $2^k - 1$. Since the starting node is the smallest odd node that satisfies this property, all the zeros will be leading, hence the power $N - 2^k$.

With one symbol per node and $L \leq N + 1$, there are

$$|\mathcal{P}^L| := 2^N - 1 - \sum_{i=1}^{L-1} i = 2^N - 1 - \frac{L(L-1)}{2} \tag{5.9}$$

distinct, upwards (towards the root) L-paths and thus the tree can hold, at most, the same number of L-sequences. Equation 5.9 is derived by the simple observation that nodes with $level \leq N$ form a triangle. Every other node outside the triangle can form an upwards L-path, thus the total number of upwards L-paths is given by equation 5.9.

The proposed data structure has two advantages over Huffman trees, namely reduced memory requirements and closed-formula traversal. By storing information in every node, our data structure requires 2^N nodes storing 2^N symbols to represent $\approx 2^N$ sequences. A Huffman tree, on the other hand, would require 2^{N+1} nodes, since only leaf nodes can encode data, storing $2^N \cdot L$ symbols. As for the tree traversal, the ancestor function 5.2 does not require look-ups in memory, unlike Huffman, leading to faster decoding.

5.2.2 Symbol Assignment

Given the L-sequences and their frequencies (appearances), symbols need to be assigned to the tree's nodes, such that "most" of the given L-sequences are generated by an upwards path. A sequence $S := s_1 s_2 ... s_L$, where s_i are the individual symbols, exists in the tree iff a path starting from a node n and recursively traversing its parent, until L symbols are output, yields S, with the starting node n generating the first symbol, s_1 and so on.

The assignment should minimize the total memory footprint of the compressed file. The two components of this goal are: *a*) map as many sequences as possible on the tree and *b*) place sequences into the penalty groups based

on their frequency. But these subgoals are competing with each other. The former can be approximately solved as the Shortest Common Superstring problem [100] with no guarantees for the relative position of the sequences, while the latter can be solved by selecting sequences from a limited window of frequencies per penalty group, perhaps with suboptimal overlap. In either case, the solution will be approximate.

The proposed algorithm 7 attempts to balance the two subgoals, with a heuristic greedy approach. The assignment process is run per penalty group, starting from the lowest one (PG-0). After an initial sequence has been placed on the top of the current penalty group, its prefix will become the suffix of the next sequence. The algorithm tries to find the most suitable symbol to assign to one of the child nodes, based on two heuristic scores:

1. the position evaluation function
2. the frequency of the generated sequence in the data

The algorithm is greedy because it selects symbols that maximize the sequence overlap, p, and the heuristic score, when multiple symbols are available for a given overlap value.

The first score, the position evaluation function, compares the number of children of the given node to the size of the $symbol\_list$ of the given new_suffix. Leaf nodes should be assigned sequences whose new_suffix is not in pSD. Nodes with a single child should not be assigned sequences whose new_suffix is not in pSD and nodes with two children should be assigned sequences whose new_suffix contains at least 2 symbols in the list. These rules of thumb are intuitive in the context of overlap maximization. The function is presented in table 5.1.

TABLE 5.1: Position evaluation function (n, new_suffix)

len($pSD[new\_suffix].symbol\_list$)	$n.children$		
	0	1	2
0	1	-1	-1
1	-1	0	0
≥ 2	-1	0	1

The second score acts as a tie-breaker to the first score and measures the appearances in the data of the L-sequence, produced by the candidate symbol.

Algorithm 7: Symbol-to-Node Assignment

Input : L – Length of sequences
U – Dictionary of the L-sequences and their appearance count
T – Tree, all nodes' symbols intialized to $\emptyset$
Output: an assigned version of T

```
1  pSD ← {p–suffix : count, symbol_list} ;          // The length-p Suffix Dict ∀s ∈ U, ∀p ∈ [1, L − 1]
2  for k ∈ [0, ⌈log2 N⌉] do
3      fringe ← initialize_ending_at(SNPG(k), U) ;     // Init PG-k at SNPG(k) and return children
4      p ← L − 1 ;                                          // Amount of overlap of sequences
5      while fringe ≠ ∅ and U ≠ ∅ do
6          best ← (−∞, −∞) ;                                                  // Best score vector
7          for n ∈ fringe do
8              suffix ← n.parent.sequence(p) ;                  // p-sequence after n (n's p-suffix)
9              if suffix ∉ pSD then
10                 continue;
11             end
12             best ← (−∞, −∞);
13             for symbol ∈ pSD[suffix].symbol_list do
14                 n.set_symbol(symbol) ;                              // Temporarily assign the node
15                 new_suffix ← n.sequence(min(L − 1, p + 1)) ;      // Extract the new (next) suffix
16                 score1 ← position_evaluation(n, new_suffix) ;          // Evaluate symbol scores
17                 score2 ← U[n.sequence(L)].appearances;
18                 n.set_symbol(∅) ;                                  // Revert temporary changes
19                 if (score1, score2) > best then                          // Tie-breaking comparison
20                     best ← (score1, score2);
21                     best_symbol ← symbol;
22                 end
23             end
24             if best > (−∞, −∞) then
25                 n.set_symbol(best_symbol) ;                        // Assign symbol with best score
26                 if n.sequence(L) ∈ U then
27                     U.remove(n.sequence(L)) ; // Remove L-seq. from U. Decrement pSD count for
28                     pSD.update(n.sequence(L)) ; // all its suffixes and update their symbol_list
29                 end
30                 fringe.remove(n);
31                 fringe.append_front({c ∈ n.children | c.address ≠ SNPG(k + 1)}) ; // LIFO exploration
32                 break;
33             end
34         end
35         if best = (−∞, −∞) then                          // If no node in fringe could have p overlap
36             p ← p − 1 ;                                    // Decrement p to look for less overlap
37             if p = 0 then                                // Couldn't find any overlapping sequence
38                 fringe ← initialize_ending_at(fringe, U) ;   // Init. all fringe nodes and return
39                 p ← L − 1 ;                              // all their children. Set p to max overlap
40             end
41         else
42             p ← min(L − 1, p + 1) ;                        // Increment p to look for more overlap
43         end
44     end
45 end
```

To facilitate score evaluation, a dictionary of suffixes is built from the data. It contains an entry for every length-p suffix in the data, defined as $S^p := s_{L-p+1}s_{L-p}...s_L$, where $p \in [1, L-1]$. Each entry stores the number of distinct L-sequences for which S^p is their p-suffix and a list of the (unique) symbols immediately preceding the suffix, at position $L-p$. When the dictionary is updated with pSD.update(S), symbol s_1 is removed from the $symbol\_list$ of S^{L-1} and its $count$ is decremented. Entries with a zero count get deleted from the dictionary. The rest $S^p, p \in [1, L-2]$ are also updated by decrementing their $count$ and removing any symbols, s, from their lists, if $s \cup S^p \notin pSD$.

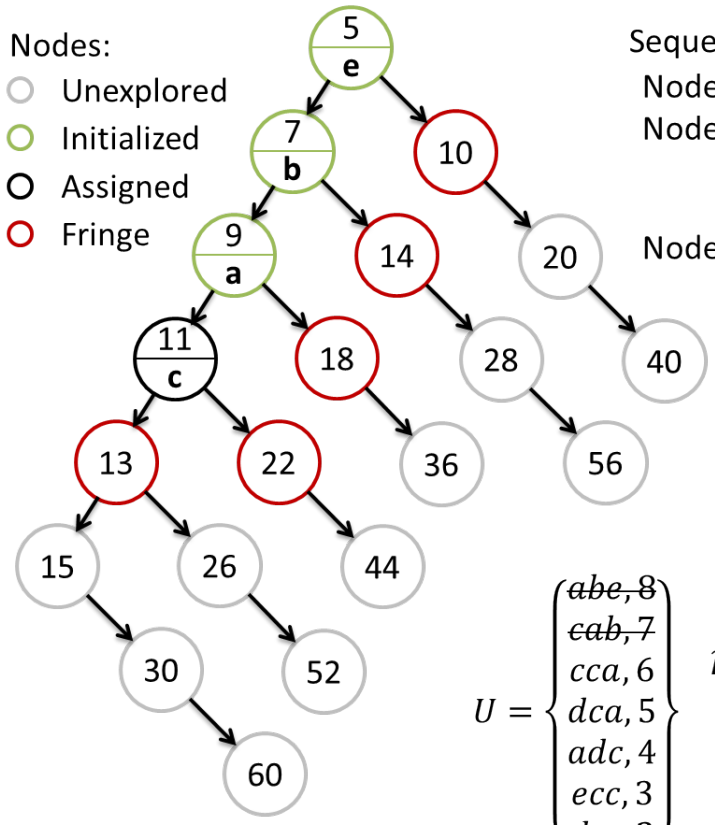
Figure 5.2 provides a snapshot of some of the steps of the assignment process for a penalty group. In conjunction with algorithm 7 and it's explanation, it should provide further insight on how the algorithm works.

For the penalty-group initialization, the most frequent of the remaining L-sequences is selected and placed on the tree such that its last symbol is assigned at the given node. Odd-indexed children of the given node are preferred, since the odd nodes form the main branch. All uninitialized children of all initialized nodes form the starting $fringe$.

The $fringe$ is a list containing every unassigned node in the tree whose parent node is already assigned. At any step, only nodes in the $fringe$ may be assigned a symbol and then the $fringe$ is updated to contain the relevant nodes.

Nodes in the $fringe$ that cannot be assigned any symbol, such that the resultant L-sequence exhibits any overlap with any other remaining L-sequence, are initialized. Again, the new $fringe$ is the set of all uninitialized children of all initialized nodes during the initialization procedure.

Algorithm 7 is of time complexity $O(E_f|A|2^N)$, where $|A|$ is the size of the symbol alphabet, $2^N - 1$ is the number of nodes in the tree and E_f is the expected number of nodes checked before a successful assignment. Implementing the $fringe$ as an array of lists (not shown for brevity), where each list corresponds to an overlap value, speeds up the algorithm, bringing its complexity down to $O(|A|2^N)$. Note that the complexity of creating the pSD dictionary is not taken into account, which is $O(L|U|)$.



Sequences:

Node 9: $S_9 = a\,b\,e$

Node 11: $S_{11} = c\,a\,b$

Node 22: $S_{22} = \underbrace{s_{22}\,c}_{new\_suffix}\,\overbrace{a}^{suffix}$

$$U = \begin{Bmatrix} \sout{abe, 8} \\ \sout{cab, 7} \\ cca, 6 \\ dca, 5 \\ adc, 4 \\ ecc, 3 \\ dcc, 2 \end{Bmatrix} \quad pSD = \begin{Bmatrix} \sout{be, 1, [a]} \\ \sout{ab, 1, [c]} \\ ca, 2, [c, d] \\ dc, 1, [a] \\ cc, 2, [e, d] \\ a, 2, [c] \\ \sout{b, 1, [a]} \\ c, 3, [c, d] \\ \sout{e, 1, [b]} \end{Bmatrix}$$

$$fringe = [13, 22, 18, 14, 10]$$

Assignment steps:

Node	Symbols & Score (for $p = L - 1$)	
13	c: (1, 6)	d: (0, 5)
15	e: (-1, 3)	d: (0, 2)
30	a: (-1, 4)	
60	$ad \notin pSD$: skipped	
26	e: (-1, 3)	
52	$ec \notin pSD$: skipped	
60	$ad \notin pSD$: skipped	
22	d: (0, 5)	

After assigning "c" to node 13:

$$U = \begin{Bmatrix} dca, 5 \\ adc, 4 \\ ecc, 3 \\ dcc, 2 \end{Bmatrix} \quad pSD = \begin{Bmatrix} ca, 1, [d] \\ dc, 1, [a] \\ cc, 2, [e, d] \\ a, 1, [c] \\ c, 3, [c, d] \end{Bmatrix}$$

$$fringe = [15, 26, 22, 18, 14, 10]$$

FIGURE 5.2: Example of the symbol assignment process for PG-2 of a $N = 6$ tree

5.2.3 Generalization

So far, for simplification and explanatory purposes, we have allowed for a single symbol per node. To generalize the method, we can extend the scheme to multiple symbols per node, with each node containing 2^M *ordered* symbols, where $M \geq 0$ is a design parameter. By ordered we mean that the symbols are arranged in a tuple and their positions cannot be changed, i.e. permutations of the tuple are not equivalent. While the generalization works for any number of symbols per node, we restrict it to powers of 2, to maximize indexing efficiency.

To utilize the multiple symbols, an L-sequence is redefined as the sequence starting at node n, symbol m and ending after L symbols, i.e.

$$S := s_m^n s_{m+1}^n ... s_{2^M-1}^n s_0^{n.parent} s_1^{n.parent} ... s_j^{n...parent}$$

where s_i^n denotes the i-th symbol of n. Due to the offset at the starting node, paths that generate L-sequences no longer visit a constant number of nodes, but their length (number of visited *nodes*) depends on that offset and is either $\lceil \frac{L}{2^M} \rceil$ or $\lceil \frac{L}{2^M} \rceil + 1$.

Algorithm 7 can still be used for the generalized case, if the 2^M-symbols-per-node tree is reduced to a single-symbol-per-node tree. The transformation is simple and requires that each node n with 2^M symbols is replaced by a list of 2^M nodes, each with a single symbol. The parent of n connects to one end of the list and its children to the other. An illustration of this transformation is shown in figure 5.3.

While the rest of the equations hold for 2^M outputs, equation 5.9 and its assumptions do not. With 2^M ouputs per node and $L \leq 2^M N + 1$, the number of L-sequences the tree can hold is:

$$\begin{aligned} |\mathcal{P}_M^L| &:= 2^M \left(2^N - 1\right) - \sum_{i=1}^{L-1} \left\lceil \frac{i}{2^M} \right\rceil \\ &= 2^M \left(2^N - 1\right) - (\ell + 1) \left(L - 1 - 2^{M-1}\ell\right) \end{aligned} \tag{5.10}$$

which is easily derived from the expanded multi-output tree and where $\ell = \left\lfloor \frac{L-1}{2^M} \right\rfloor$.

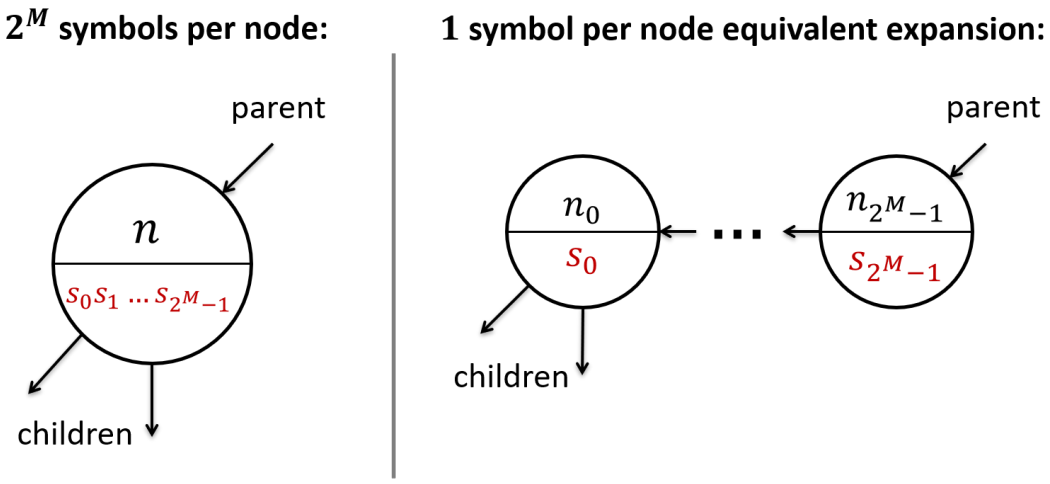


FIGURE 5.3: Multi-symbol node expansion to single-symbol equivalent

This method of generating sequences by traversing a tree can, in principle (and without the penalty groups), work for any tree. For example, the perfect binary tree [101] with $children(n) = \{2n, 2n+1\}$ was considered, as well as the Collatz tree [102, 103]. However, neither satisfy the inter-domain proximity requirement and were thus discarded, due to their inability to concentrate high-frequency L-sequences in some address neighborhood. This point is illustrated in figure 5.4. Note that in all three cases the same parameters (N and M) were used, with the same algorithm (algorithm 7 modified to ignore penalty groups on the Collatz and Perfect binary trees) on the same data.

5.2.4 Addressing Modes

The structural properties of the tree as well as the distribution of its sequences' appearances can be leveraged to minimize the memory footprint of the compressed data. The addressing modes must have variable length and be easily decodable.

The proposed tree contains a main branch consisting of the odd numbers in the range $[1, 2^N - 1]$. Thus, $N-1$ bits can be used to indicate the odd node (primary address). This information is sufficient for PG-0, but not for any other penalty group, if an even node needs to be addressed. In the latter case, k extra bits are sufficient in PG-k to shift (multiply by 2^i) the primary address and index any node in that group. Therefore, for any node in the tree, $N-1+k$ bits are needed, with k being a mathematical function of the primary address.

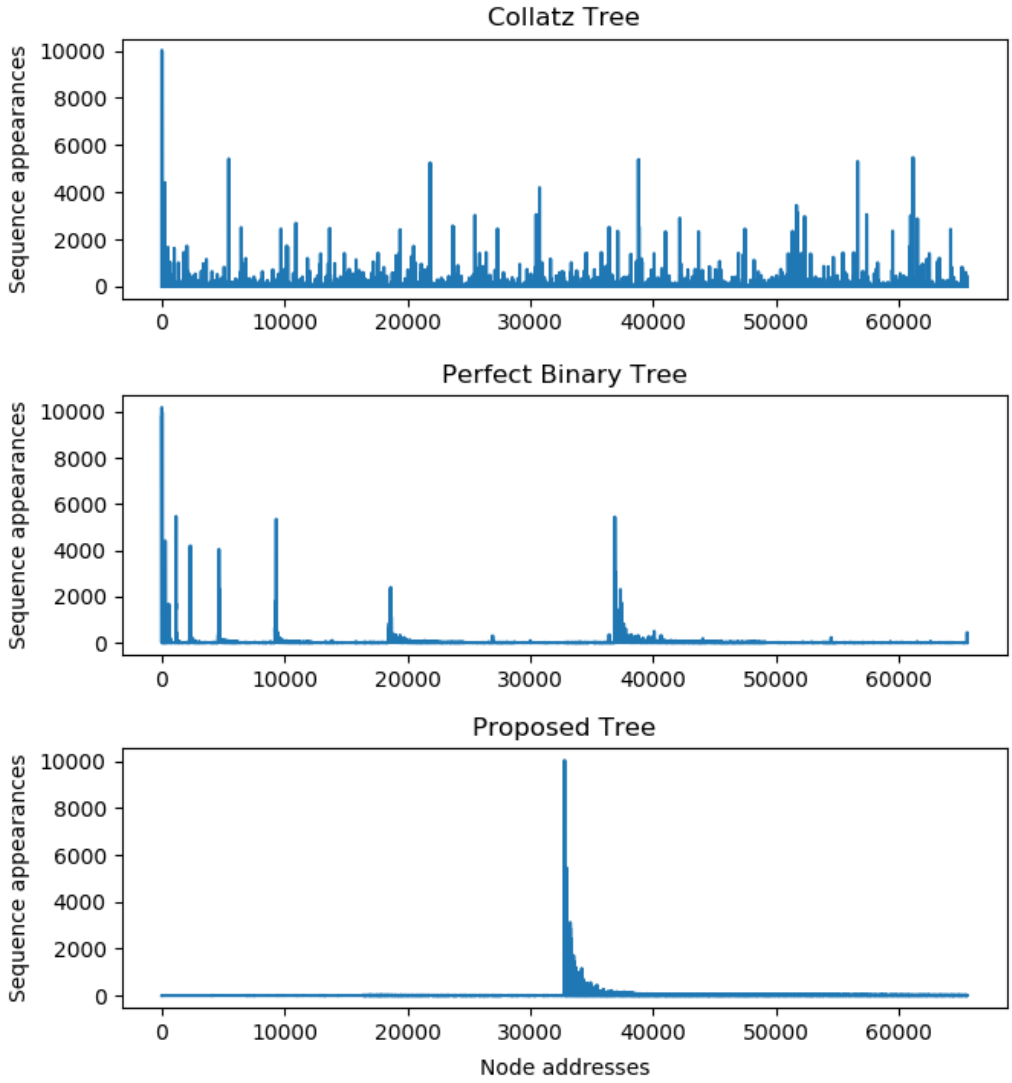


FIGURE 5.4: Distribution of node sequence appearances vs its address for each tree

Fortunately,

$$k = \lceil log_2(P_{LZ} + 1) \rceil \tag{5.11}$$

where P_{LZ} is the number of leading zeros of the primary address. While that relationship looks complex, due to the small values of k and N, it becomes very efficient as a carefully implemented Boolean function.

To take advantage of the concentrated high-frequency sequences at the start of PG-0, as shown in figure 5.4, we define an elite addressing window of size 2^W, starting at $SNPG(0)$. In this window, nodes require W bits to be addressed, as well as an extra bit for the elite window flag. The purpose of this window is to further shorten the memory footprint of high-frequency

L-sequences, which is naturally achieved for $W+1 < N-1$. The process for finding the optimal value for W is described in section 5.3.3.

Lastly, since the scheme cannot guarantee the presence of all wanted sequences in the tree, an addressing mode is necessary for unmapped sequences. In the first mode, $N-1$ zeros would correspond to node 1. By assigning this code to indicate an unmapped sequence, we are only sacrificing $N-L+1$ sequences (since we cannot address node 1 and thus, its right branch) to gain this functionality. Of course, any primary address could be assigned for this mode (e.g. a node in PG-0 that does not produce a wanted sequence), but $N-1$ zeros are assigned for simplicity.

The addressing modes are summarized on table 5.2 in decreasing decoding priority. Field widths are not proportional to their visual lengths, but comparative size and position characteristics are preserved. The elite flag is set to "1" only for the elite mode.

TABLE 5.2: Addressing modes, packet fields and lengths

Priority	Description	Data packet fields & lengths
1	Elite	ELITE FLAG (1) | ELITE ADDRESS (W) | SYMBOL OFFSET (M)
2	Unmapped	ELITE FLAG (1) | 00...00 ($N-1$) | RAW SEQUENCE DATA ($L * SYMBOL\_BITS$)
3	Regular	ELITE FLAG (1) | PRIMARY ADDRESS ($N-1$) | SYMBOL OFFSET (M) | ADDRESS SHIFT ($k(PR.ADDR.)$)

Algorithm 8 presents the address decoding and decompression pseudocode. Initialization and implementation details have been omitted for brevity, while the pseudocode represents the body of an infinite loop (circuit). The rules of hardware concurrency and sequential logic apply. For unmapped sequences, the symbols are stored in and sequentially output from a raw symbol buffer (RSB), while for mapped sequences, the node symbols are retrieved from memory (MEM). Superscript t denotes data used in the current decompression cycle (iterations until a data packet is decompressed and all it's symbols have been output) and superscript $t+T$ denotes data pending use in the next decompression cycle. Signal "rem" denotes the remaining symbols to be output from the last node and "SB" is shorthand for symbol bits.

Algorithm 8: Decoding & decompression pseudocode

Input : buf – Stream of bits appended to a buffer
Output: out – Output symbols (up to 2^M symbols)
$valid$ – Which outputs are valid (2^M bit mask)

```
// Address decoding
if i = path_length then                                   // Prev. path ending
    if buf[0 ... N − 1] = 0 then                          // Unmapped
        RSB^{t+T} ←buf[N ... N + L · SB − 1] ;
        addr^{t+T} ← 0 ;
        offs^{t+T} ← 0 ;
        shift^{t+T} ← 0 ;
        rem^{t+T} ← L  mod 2^M ;
        buf.remove(N + L · SB) ;
    else
        if buf[0] = 1 then                                // Elite
            addr^{t+T} ← 2^{N−1} + 1 + 2· buf[1 ... W] ;
            offs^{t+T} ← buf[W + 1 ... W + M] ;
            shift^{t+T} ← 0 ;
            buf.remove(1 + W + M) ;
        else                                              // Regular
            addr^{t+T} ← 2·buf[1 ... N − 1] + 1 ;
            offs^{t+T} ← buf[N ... N + M − 1] ;
            k ← penalty(buf[1 ... N − 1]) ;
            shift^{t+T} ← buf[N + M ... N + M + k − 1] ;
            buf.remove(N + M + k) ;
        end
        rem^{t+T} ← (L+ offs^{t+T} − 2^M)  mod 2^M ;
    end
end

// Tree traversal & symbol output
if i = 0 then                                             // First node
    valid ← 2^{offs^t} − 1 ;
    i ← i + 1 ;
else if i = path_length then                              // Last node
    valid ← (2^{rem^t} − 1) ≪ (2^M − rem^t) ;
    i ← 0 ;
    path_length ← ⌈(L+offs^t)/2^M⌉ − 1 ;                  // Get new data
    RSB^t ← RSB^{t+T} ;
    addr^t ← addr^{t+T} ≪ shift^{t+T} ;
    offs^t ← offs^{t+T} ;
    rem^t ← rem^{t+T} ;
else                                                      // Intermediate node
    valid ← 2^{M+1} − 1 ;
    i ← i + 1 ;
end

if addr^t = 0 then
    out ← RSB[2^M · i · SB ... 2^M · (i + 1) · SB − 1] ;
else
    out ← MEM(addr^t) ;
    addr^t ← parent(addr^t) ;
end
```

5.3 Scheme Analysis

5.3.1 Mapping efficiency

We define the mapping efficiency of an assigned tree as the ratio of wanted L-sequences the tree can generate to the total number of L-sequences the tree can hold:

$$\eta := \frac{|U_b| - |U_a|}{|\mathcal{P}_M^L|} \approx \frac{|U_b| - |U_a|}{2^{N+M}} \tag{5.12}$$

where $|U_b|$ is the number of unique L-sequences in U before algorithm 7 run and $|U_a|$ is the number of sequences remaining in U after the algorithm finished. This metric, together with the total frequency of sequences in U_a, indicate the fitness of this compression scheme *and* the assignment algorithm to the given data.

While the global efficiency metric provides an overview for the entire tree, it weighs all nodes (and their sequences) equally. However, it is more important for low-penalty groups to be "tightly packed" than for high-penalty groups. Moreover, based on the symbol distribution assumptions made in the beginning, low-frequency sequences will exhibit less overlap than high-frequency ones. Thus, we define the penalty-group approximate efficiency as

$$\widetilde{\eta}(k) := \frac{|U_{b_k}| - |U_{a_k}|}{2^M \cdot NPPG(k)} \tag{5.13}$$

where $|U_{b_k}|$ and $|U_{a_k}|$ are the number of L-sequences in U before and after the assignment of PG-k respectively.

5.3.2 Memory footprint

For a given file, assigned tree and total appearances of all sequences in each group, the compressed data memory footrprint (in bits) will be

$$\begin{aligned} \mathcal{D}_C = {} & (1 + W + M)A_e + (N + M)A_{0\_ne} \\ & + \sum_{k=1}^{PGs-1} (N + M + k)A_k \\ & + (N + L \cdot SB)A_u \end{aligned} \tag{5.14}$$

where A_e are the total appearances of all sequences in the elite window, A_{0_ne} are the total appearances of sequences in PG-0 that are *not* in the elite window, A_k are the total appearances in PG-k and A_u are the total appearances

of all unmapped sequences.

The memory footprint of the tree is given by

$$\mathcal{M}_T = 2^{N+M} \cdot SB \tag{5.15}$$

in cases where memory modules of arbitrary width can be created (e.g. FPGA BRAMs). Otherwise, the footprint will depend on the exact memory mapping, the memory's width and the bits per symbol.

5.3.3 Optimal elite window size

Equation 5.14 provides a condition for the optimal value of the elite window width, W. In an assigned tree, $A_e + A_{0\_ne}$ (corresponding to a window width W) will be constant since it denotes the total appearances of all sequences in PG-0. Let A'_e and A'_{0_ne} correspond to width $W+1$. The following relations will hold

$$A_e + A_{0\_ne} = A'_e + A'_{0\_ne} \tag{5.16}$$

$$A_e < A'_e \tag{5.17}$$

Thus, from equation 5.14, increasing W by 1 decreases $\mathcal{D}^C$ iff

$$(a+1)A'_e + b \cdot A'_{0\_ne} < a \cdot A_e + b \cdot A_{0\_ne} \iff \tag{5.18}$$

$$\frac{A'_e}{A'_e - A_e} < N - W - 1 \tag{5.19}$$

where $a = 1 + W + M$ and $b = N + M$. Therefore, W should be increased as long as condition 5.19 holds.

5.3.4 Throughput & input bandwidth

In the context of concurrency and a single decoder-decompressor pair, packet decoding for mapped sequences takes at least $\lceil \frac{L}{2^M} \rceil$ cycles to complete, while an L-sequence is being generated from the previous packet, as shown in algorithm 8. Thus, the design parameters can be tuned to allow for sufficient decoding time (based on the technology and implementation) and possibly allow for more than one decompressor for a single decoder.

The expected number of cycles to decompress a single packet is given by

$$E_D = \sum_{m=0}^{2^M-1} \left\lceil \frac{L+m}{2^M} \right\rceil Pr(OFFS = m) \tag{5.20}$$

where $Pr(OFFS = m)$ is the probability of the symbol offset having the value m. Further, when the symbol offsets follow a uniform distribution, which is expected since the assignment algorithm does not take the symbol offset into account, equation 5.20 can be simplified to

$$E_D = \frac{L + 2^M - 1}{2^M} \tag{5.21}$$

Assuming an uninterrupted supply of packets and excluding unmapped sequences, the average decompression throughput of a single pair is given by

$$\mathcal{T} = \frac{L}{E_D} \frac{symbols}{cycle} \tag{5.22}$$

or simplified, under the equiprobable-symbol-offset assumption, as

$$\mathcal{T} = \frac{2^M \cdot L}{L + 2^M - 1} \frac{symbols}{cycle} \tag{5.23}$$

To maintain the above throughput and not stall the decompressor - without using buffers - an average input bandwidth of

$$\mathcal{B} = \frac{\mathcal{D}_C}{A_{total}} \frac{bits}{E_D \, cycles} \tag{5.24}$$

must be maintained, where A_{total} is the total appearances of all sequences in the data.

Equation 5.24 shows that the input bandwidth can fluctuate within the E_D-cycles window and the decompression throughput will not be affected, as long as equation 5.24 holds. Therefore, the method itself acts as a buffer, making it inherently tolerant to such fluctuations. This tolerance depends on L and M and is inversely proportional to throughput.

5.4 Sparse matrix representation

CSR [91] and CSC [92] are sparse matrix representation schemes which split all the necessary information of a 2D sparse matrix into three vectors. These formats are similar, with the only difference being that CSR stores information row-wise and CSC stores information column-wise. For CSC, one vector holds the non-zero values in the matrix, another holds the respective row indices of the non-zero values, and the last holds the column indices, pointing to the other two vectors and denoting where each column starts in the encoded data.

A major problem with this approach is that indices are stored in full precision. This is problematic for low sparsities, since the encoding overhead far outweighs the benefits of sparse matrices themselves. Namely, for the GRU kernels, the indices would have lengths on the order of 11 bits, far higher than the 4 bit quantized weights. Therefore, these formats are not suitable for the case at hand.

To alleviate the overhead, differential indices will be used, instead of global ones. Consequently, the bit length of the indices will no longer be determined by the number of rows in the matrices, but by the maximum distance between consecutive non-zero weights. In other words, we are applying run-length encoding on the zeros between consecutive non-zeros.

Thus, the 2D matrices are first flattened column-wise. The zeros are then run-length encoded and the resultant vector is split into two vectors, such that all non-zero weights are stored in one and all run-length-encoded zeros are stored in the other. Thus, the i-th element in the zeros vector corresponds to the number of zeros between the $(i-1)$-th and i-th element in the weights vector. The transformation is illustrated in figure 5.5.

The information contained in the two vectors is sufficient to reconstruct the original matrix, given its number of rows, N_R and columns, N_C. To achieve this, a counter is needed to keep the running sum of the zeros vector's elements, incrementing it by 1 for every weight element. When this sum exceeds the number of rows, N_R is subtracted from it and the column index is incremented.

Representing sparse matrices with this encoding allows for compressing the zeros and weights vectors with PATH compression. The distribution of the quantized weights is known and discussed in chapter 4, and their suitability

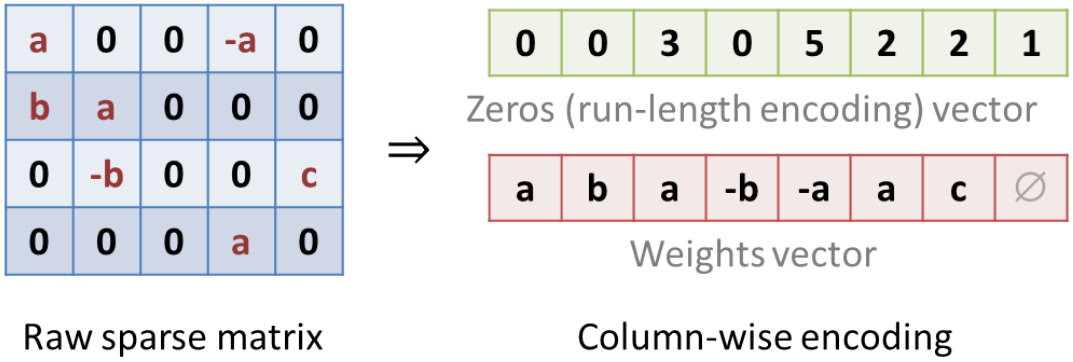


FIGURE 5.5: Sparse matrix encoding example

for PATH compression has been theoretically established. As for the spatial information, it is rather intuitive that creating L-sequences of the global indices of the non-zero weights would yield too many and infrequent sequences, which would make compressing this data with PATH compression infeasible. On the other hand, the proposed encoding is guaranteed to produce sequence distributions at least as suitable as global indexing.

5.5 Experiments

The memory footprint of the encoded sparse matrices are presented in table 5.3, along with other relevant information. Since there are 16 quantization levels, each weight is represented by a 4-bit index to a lookup table with its numeric value. As for the zeros, their run-length encoded values are in the range $[0, 31]$, so they are represented by 5 bits. Memory footprint information is presented per layer, while sparsities denote the average percentages per layer and kernel category. Also, remember that since the GRU is bidirectional, there are two kernels per layer and category.

The first experiment involved the compression of the split and flattened data, as described in figure 5.5, separately for each layer. The compression parameters, common for all layers, were set to $N = 15$, $M = 1$, $W = 10$ for the weights and $N = 16$, $M = 1$, $W = 11$ for the zeros.

The values were chosen based on the following rationale: 2^{N+M} should be *close* to the number of distinct L-sequences in the data (if it's too small, many sequences will be unmapped, incurring a heavy overhead, whereas if it's too big, the addresses will be unnecessarily wide). Then, M can be freely selected to match the desired decompression throughput (here, $M = 1$ was selected

TABLE 5.3: Encoded matrix statistics per layer and type

Layer	Category	Size (each)	Memory (MB)		Sparsity
			Weights	Zeros	
1	Gates	3392×1600	3.147	3.933	63%
	Candidate	3392×800			58%
2	Gates	2400×1600	2.409	3.012	62%
	Candidate	2400×800			50%
3	Gates	2400×1600	3.018	3.773	50%
	Candidate	2400×800			43%
4	Gates	2400×1600	3.691	4.615	35%
	Candidate	2400×800			36%
5	Gates	2400×1600	3.635	4.544	36%
	Candidate	2400×800			39%
Total			**15.900**	**19.877**	

as the smallest value resulting in a nontrivial throughput value) and W is decided through condition 5.19.

Therefore, to get the best results, N and L were jointly decided, by examining which L values yield a count of distinct L-sequences close to a power of 2. The weights had 52K-56K distinct L-sequences and the zeros had 92K-164K, per layer for their respective Ls.

The results are presented on table 5.4. Memory reduction is always computed with respect to the original memory footprints presented in table 5.3. This being an entropy-based lossless compression scheme, Shannon entropy was used as a baseline, where the entropy compressed footprint was calculated as the product of the number of instances of all L-sequences with the total entropy of the distinct L-sequences.

Overall, the method yielded file sizes 5.3% larger than the entropy limit for the weights and 17.2% for the zeros. As for the memory reduction, the weights were reduced by 27% less than the limit and the zeros by 10% less. The penalty-group approximate mapping efficiency for PG-4 was at 0% for the weights because all L-sequences were mapped in previous penalty-groups.

TABLE 5.4: Compression results for signed weights vector and zeros vector

			Proposed method										Entropy limit	
	Layer	L	η	$\widetilde{\eta}(k)$					$\mathcal{T}$ $\left(\frac{\text{symbols}}{\text{cycle}}\right)$	Compr. data (MB)	Memory reduction	Tree (KiB)	Compr. data (MB)	Memory reduction
				0	1	2	3	4						
Weights	1	4	86.5%	99.9%	97.8%	91.5%	50.4%	0.0%	1.60	2.775	11.8%	32	2.637	16.2%
	2	4	79.8%	99.9%	98.0%	85.7%	20.9%	0.0%	1.60	2.119	12.0%	32	2.015	16.4%
	3	4	80.6%	99.9%	97.5%	86.0%	25.7%	0.0%	1.60	2.658	11.9%	32	2.528	16.2%
	4	4	81.2%	99.9%	97.6%	85.6%	29.8%	0.0%	1.60	3.250	11.9%	32	3.084	16.4%
	5	4	83.8%	99.9%	97.2%	87.4%	42.9%	0.0%	1.60	3.209	11.7%	32	3.044	16.3%
	Total									**14.011**	**11.9%**	**160**	**13.308**	**16.3%**
Zeros	1	6	83.1%	99.5%	88.6%	75.7%	66.8%	62.4%	1.72	2.204	44.0%	80	1.908	51.5%
	2	6	72.3%	98.5%	79.1%	60.8%	47.8%	43.5%	1.72	1.538	48.9%	80	1.379	54.2%
	3	7	63.7%	96.5%	67.4%	50.0%	38.0%	34.0%	1.75	1.598	57.6%	80	1.412	62.6%
	4	9	59.0%	92.5%	58.5%	45.5%	37.9%	36.0%	1.80	1.610	65.1%	80	1.325	71.3%
	5	9	61.4%	92.8%	60.9%	48.6%	41.6%	39.7%	1.80	1.666	63.3%	80	1.329	70.7%
	Total									**8.616**	**56.7%**	**400**	**7.353**	**63.0%**

TABLE 5.5: Compression results for unsigned weights vector with signs as encoding overhead

			Proposed method										Entropy limit	
	Layer	L	η	$\widetilde{\eta}(k)$					$\mathcal{T}$ $\left(\frac{\text{symbols}}{\text{cycle}}\right)$	Compr. data (MB)	Memory reduction	Tree (KiB)	Compr. data (MB)	Memory reduction
				0	1	2	3	4						
Weights	1	6	83.5%	99.9%	92.8%	78.2%	58.9%	49.5%	1.72	2.766	12.1%	48	2.618	16.8%
	2	6	74.9%	99.7%	88.8%	67.0%	37.3%	26.1%	1.72	2.103	12.7%	48	2.001	16.9%
	3	6	78.3%	99.9%	91.0%	70.7%	46.3%	34.7%	1.72	2.634	12.7%	48	2.513	16.7%
	4	6	80.0%	99.9%	92.3%	73.5%	49.2%	38.3%	1.72	3.220	12.8%	48	3.068	16.9%
	5	6	82.4%	99.9%	93.1%	76.8%	55.4%	45.3%	1.72	3.191	12.2%	48	3.025	16.8%
	Total									**13.914**	**12.5%**	**240**	**13.224**	**16.8%**

A larger sequence length L is a desirable feature since it contributes to the decompression throughput and the input bandwidth fluctuation tolerance. However, for the weights, setting $L = 5$ results in a rapid increase of the distinct L-sequences, to roughly 300K per layer, requiring a tree 4 times bigger.

To mitigate this cost increase, a second experiment was carried out, in which the weights were stripped off their sign. Because the distribution of single weight values is symmetric around zero, the sign information constitutes a high entropy component of the data. Thus, L sign bits were added before each compressed data packet, as an encoding overhead, carrying the raw sign information for each instance of each unsigned weight L-sequence. Each weight was encoded by the remaining 3 bits for its magnitude index information.

This experiment's parameters were $N = 16$, $M = 1$, $W = 11$ and the number of distinct L-sequences was 100K-118K. The results are presented in table 5.5. The memory footprints for the proposed method and the entropy limit were calculated based on the unsigned L-sequences, with the sign overhead added. Evidently, throughput increased, while the memory footprint *decreased*, compared to table 5.4. The compressed file statistics are slightly better than the previous ones, with a file size 5.2% bigger than the limit and a 25.6% less memory reduction than the limit.

5.6 Conclusion

The proposed novel lossless compression method produces symbol sequences of constant length, by traversing paths of constant length in a tree. The tree structure is designed to provide simple, mathematically-decodable, variable-length encodings among its penalty groups. As such, it lends itself well to hardware implementation on reconfigurable hardware.

By storing the symbol sequences in overlapping fashion, our method requires tree sizes a fraction of those produced by Huffman coding when applied on the sequence alphabet. The constant length of the sequences and the uniform-time decodability of the encodings yields regular decompression latencies and throughput.

The ability to select tradeoffs between file size and tree size, decompression throughput and input bandwidth fluctuation tolerance makes this method

particularly versatile. Additionally, it is shown to combine well with other encoding schemes such as run-length encoding and sign separation.

Combining the results of this and the previous chapter, table 5.6 sums up the reduction of the memory footprint for storing DeepSpeech2's GRU weight kernels. Effort notwithstanding, the 11x reduction in memory is less than what we hoped for, but is nonetheless significant, especially in the context of the entropy limit.

TABLE 5.6: GRU kernel memory reduction

Description	**Methods used**	**Data type**	**Memory (MB)**	**Reduction**
Original	-	32-bit float	249.5	-
Pruned & quantized	LMP+Uniform	4-bit weights 5-bit zeros	35.7	7x
Compressed	LMP+Uniform +PATH	4-bit weights 5-bit zeros	22.5	11x

Chapter 6

Hardware Architecture

This chapter demonstrates and highlights the inherent properties of the hardware architecture of our novel compression method. The decompressor architecture is designed, theoretically analyzed, and shown to require an order of magnitude fewer resources compared to LZ77 decompression accelerators, while achieving better throughput.

6.1 FPGA architecture overview

Despite the work in this chapter being theoretical, and therefore implementing the architecture on a target device being outside the current scope, it is still beneficial to frame the chapter in the context of an FPGA architecture. This context will allow quantitative modeling of our architecture's resource utilization and latency, as well as provide the common basis for comparisons with existing literature.

The chosen FPGA architecture is the Xilinx UltraScale, since it is a typical target architecture in literature and the available platform in the University's MHL Lab is a ZCU102 evaluation kit, featuring a Zynq UltraScale+ MPSoC. Thus, our analysis will be both comparable to existing literature and relevant for a future implementation on the Lab's hardware.

A possible point of confusion is, understandably, the nomenclature. A Xilinx FPGA architecture defines the set of elements it provides and their configurations. A Xilinx FPGA series is defined by the FPGA architecture and the manufacturing process node of the chips. A Xilinx FPGA family is a set of devices in a given FPGA series that, broadly speaking, are characterized by the ratios of the elements they contain.

As such, the UltraScale+ series contain FPGAs of UltraScale architecture, but chips are manufactured on a newer process node. The relevant resources available on the UltraScale architecture are briefly presented, with more information available on the official user guides [104, 105].

6.1.1 Configurable Logic Block

The main resource for implementing combinatorial or sequential circuits in the UltraScale architecture is the Configurable Logic Block (CLB). Each CLB contains one slice, either SLICEL or SLICEM. Each slice contains eight LUTs, one 8-bit carry chain, and sixteen FFs.

The types of slices differ only in the configurations they provide. SLICEL, where the L stands for logic, provides most of the configurations and functionality explained below. SLICEM, where the M stands for memory, is a superset of SLICEL, providing extra configurations for the LUTs.

Look Up Tables

Each LUT can be configured as:

- a 6:1 (6-bit input, 1-bit output) arbitrary Boolean function
- a 5:2 arbitrary Boolean function
- a 3:1 (or lower) and a 2:1 (or lower) arbitrary Boolean functions

The difference between the latter two configurations lies in the inputs. A 5:2 function uses the same inputs for both output bits, while a 3:1 and a 2:1 function can use different inputs per function. The propagation delay per LUT is independent of its configuration.

Wider Boolean functions can be constructed inside each CLB by combining its LUTs via dedicated multiplexers (MUXs). Thus, LUTs in each CLB can be additionally configured as:

- up to four 7:1 arbitrary Boolean functions (using two LUTs each)
- up to two 8:1 arbitrary Boolean functions (using four LUTs each)
- a single 9:1 arbitrary Boolean functions (using eight LUTs)

For even wider Boolean functions, multiple CLBs can be connected together. It is therefore possible to create 10:1 functions with two CLBs (sixteen LUTs), 11:1 with four CLBs etc.

To avoid any confusion, we note that the overall configuration of LUTs in one CLB can be any valid combination of individual configurations. I.e., in a single CLB, it is possible to have a 6:1 function, a 3:1 and a 2:1 function, a 7:1 function, and an 8:1 function at the same time.

Carry chain

Adders and subtractors are some of the most common circuits in most applications, but the propagation delay increases linearly with the operand size. For wide operands, if the circuit were built using only LUTs, the propagation delay would be significant, thus reducing maximum clock frequency.

Instead, the critical path of these circuits, i.e. the path with linear propagation delay, is built into a dedicated carry logic unit inside each CLB. The 8-bit long carry chains of multiple CLBs can be cascaded to form wide adder/subtractor logic. Thus, the carry chains facilitate the implementation of wide and fast adders/subtractors.

Flip flops

The sixteen FFs available per CLB are evenly split among the LUTs, with two FFs corresponding to each LUT. Each FF's input can be individually configured, with both FFs per LUT having the same input options. The available options are:

- each of the two LUT outputs
- the output of the local dedicated multiplexer
- each of the two local carry outputs
- a direct input from the CLB bypassing the LUT

Shift register (SLICEM only)

Each LUT in a SLICEM can be configured as a 32-bit shift register without using the FFs. This shift register shifts by 1 bit per clock cycle. Single-bit data can be written to the shift register's LSB, and any of the 32 bits can be dynamically read, while the MSB provides fixed read access. Smaller shift registers can be implemented by setting the read address to the desired register length.

Multiple 32-bit shift registers can be cascaded to form larger shift registers, up to 256 bits per SLICEMs. Cascading can also extend to multiple SLICEMs, for even larger shift registers.

6.1.2 On-chip memory

The ability to store data physically close to the processing units is paramount in any hardware architecture. For this reason, UltraScale FPGAs provide three types of memory within the PL fabric.

Distributed RAM (SLICEM only)

Each LUT in a SLICEM can be configured as a distributed RAM resource. The effective amount of data stored per LUT varies based on the configuration. Multiple LUTs can be combined to form larger distributed RAMs, with up to 512 bits per SLICEM. Multiple SLICEMs can be combined to form larger distributed RAMs.

Distributed RAM height within a single SLICEM can range from 32 to 512 rows, while the width can range from 1 to 16 bits. Multiple ports are also supported, ranging from single to octal port RAMs. Of these ports, exactly one is a write (and possibly read) port, while the rest are read-only ports. Valid distributed RAM configurations within a SLICEM have a product of their parameters no greater than 512, i.e.

$$\text{ports} \cdot \text{height} \cdot \text{width} \leq 512$$

Block RAM

So far, we have seen the various configurations and elements in CLBs. A completely separate resource available in UltraScale hardware is the block RAM modules. Each block RAM stores up to 36Kb and can be configured either as a single 36Kb RAM, or as two independent 18Kb RAMs. Each BRAM, regardless of size, has two read/write ports and these ports can be configured with independent widths.

Available BRAM widths are 1, 2, 4, 9, 18, 36, or 72 bits, while the BRAM heights are 32K, 16K, 8K, 4K, 2K, 1K, or 512 respectively, for the 36Kb BRAMs. For the 18Kb BRAMs, the heights are halved and the width is up to 36 bits. Multiple BRAMs can be cascaded to form larger memories.

All ports are synchronous, regardless of their operation, i.e. read or write for a given clock cycle. As such, a clock edge is required for a read or write, as the input signal has to go through the input register. Additionally, there are optional output registers for each port, which enhance performance by eliminating routing delays and facilitating pipelining without consuming extra resources. Both the input and the output registers are clocked by the same source, which can be optionally inverted for falling clock edge operation.

Ultra RAM

A far less versatile option in UltraScale hardware is Ultra RAM blocks, each of which provides 288Kb storage. Each block is a dual port, 4K high and 72 bits wide, synchronous memory. Unlike all other on-chip RAMs, Ultra RAM cannot be initialized to user-defined values during FPGA programming. This type of memory is not available to every UltraScale or UltraScale+ device, so it is only included in this chapter for completeness purposes. Ultra RAM is not relevant and will not be discussed in the remainder of this chapter.

6.2 External memory

To complement the limited on-chip memory, FPGA chips can connect to external RAM chips. External memories can provide 3 or more orders of magnitude more memory than the combined on-chip alternatives. Fundamental laws of physics dictate that external memory accesses incur orders of magnitude higher latency and energy consumption, compared to on-chip memory accesses.

Several external memory types exist, but FPGAs are most commonly paired with either DDR or HBM memory. Both are high-density memories, but HBM offers higher bandwidth compared to DDR, while using less power [106]. HBM is found on high-end FPGA platforms, while DDR is far more common.

Both external memories use specialized memory interfaces to connect with the PL. These controllers are hardened onto the FPGA chip and can be configured via the Xilinx tools. Details of the parameters and configurations of external memories are well beyond the scope of this thesis.

6.3 Decompressor architecture

Having presented an overview of the FPGA resources, we now move on to the hardware architecture of the decompression accelerator for our novel compression method. A high-level block diagram for the datapath is illustrated in figure 6.1.

PATH decompression consists of three major parts, namely the compressed data, the tree, and the packet decoding logic. The tree is not modified after construction, i.e. it is static, so it requires only read access. The compressed data are also read-only, but this is a general truth and not specific to our compression method.

As per table 5.2, the compressed data is a series of variable-length packets. These packets are concatenated to form the compressed file, which is placed on DDR. Naturally, the packets are not word-aligned in memory, therefore the decoder must identify the packet bounds and handle cases where a packet crosses word boundaries.

The compressed data files, as per tables 5.5 and 5.4, are cumulatively on the order of tens of MB, which far exceeds the few Mb available in a typical FPGA's on-chip memory, so external memory must be used to store them. The trees must be quickly and efficiently accessible, so they are mapped on cascaded BRAMs.

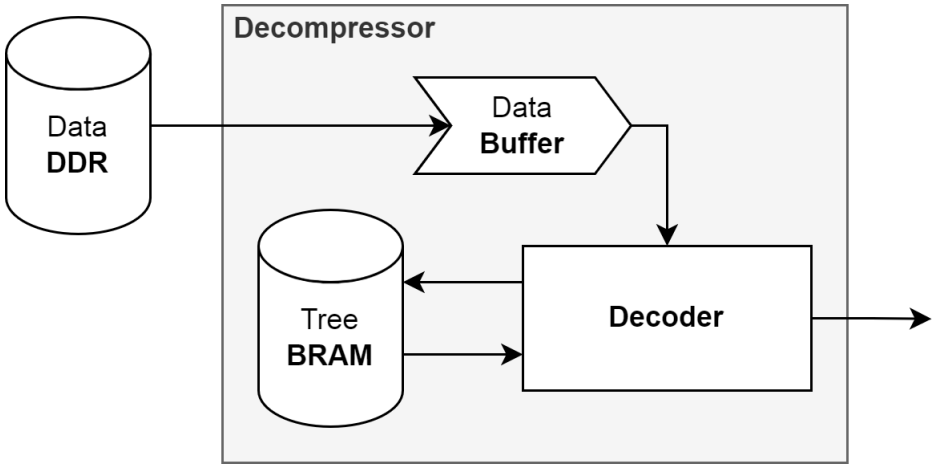


FIGURE 6.1: PATH decompressor high-level block diagram

As a result of using DDR memory, a buffer is required as an intermediary. The DDR will supply data to the decompressor in burst reads, for energy efficiency and speed. These bursts cannot be processed simultaneously upon arrival. Therefore, the First-In-First-Out (FIFO) buffer temporarily holds them, to dispense data on an as-needed basis. The DDR will supply data bursts whenever there is enough empty space in the FIFO to accommodate them. Note that, naturally, a read address bus is necessary to go into the DDR, but it is omitted from figure 6.1. The reason is that data will be sequentially read, so depicting the address bus would cause unnecessary confusion.

The FIFO buffer can be constructed from BRAMs, which contain dedicated logic for implementing FIFOs. The minimum BRAM depth is 512, which is more than sufficient for our FIFO. Details about the width will emerge once we inspect the decoder architecture.

6.4 Decoder architecture

The decoder pseudocode has already been presented in algorithm 8 and while it will the basis for the architecture design, it is, naturally, highly reductive. The most significant details missing are the pipeline stages and the hidden complexity of efficiently manipulating the data buffer.

The data buffer was assumed, for simplicity, in algorithm 8 to be an infinite size bit vector. Of course, such a data structure is detached from the reality of hardware, where the data buffer will be implemented as a FIFO. A more subtle slight, however, concerns the contents of the buffer itself.

6.4.1 Packet layouts

In hardware, we seek to minimize complexity by regularizing the spatial structure of information. Yet, the packets of table 5.2 are inconsistent in their placement of semantically identical fields, namely the symbol offset. The practical side effect of this is the necessity for additional hardware to retrieve the same semantic information from different places, depending on the type of packet. Regardless of the actual cost of this additional hardware, such shortcomings should be mitigated, whenever possible.

A rearrangement of the packet fields of table 5.2 is presented in figure 6.2, along with their sizes and starting indices. Each color represents a semantic

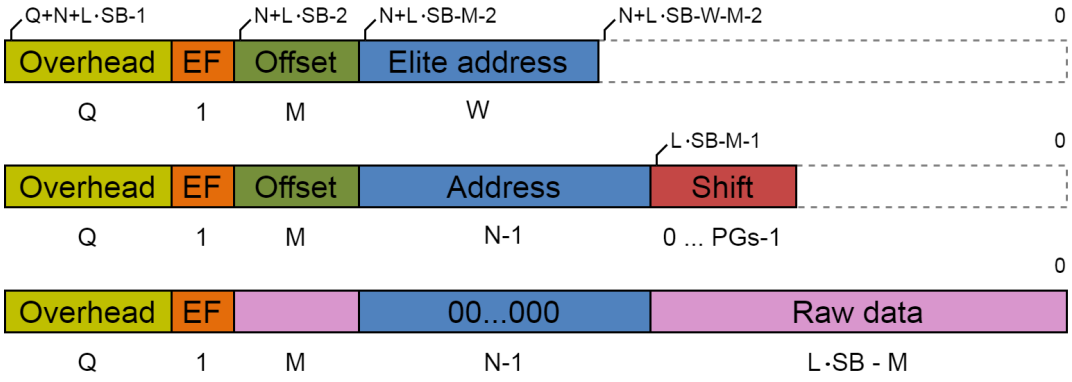


FIGURE 6.2: Rearranged packet fields

group, while the empty dashed fields represent spatially succeeding information that is irrelevant to the packet. The benefit of the rearrangement is obvious, as fields common to multiple packets always start at the same index. Note that by repositioning the offset field, the raw data field had to be split in the unmapped addressing mode since no offset field exists for that mode.

The indices are formulated in the context of the widest packet, with packets being left-aligned. This formulation will facilitate comprehension later on, when the packets are placed into registers. Crucially, however, each packet occupies exactly as many bits as necessary in the DDR and the FIFO. Figure 6.2 is not representative of the DDR or FIFO data layout, because the packets are not, at those points, word-aligned.

The variables in figure 6.2 are all in accordance with section 5.2, except for the overhead length Q. This variable is introduced here for the sake of generality. In chapter 5, the overhead length was L for the weights and 0, i.e. no overhead existed, for the zeros. Throughout the rest of this chapter, Q will be used for generality and is assumed to be any arbitrary natural number.

6.4.2 BRAM organization

Another crucial detail of algorithm 8, which permeates the throughput equations 5.20, 5.21, and 5.23, is the assumption about the organization of the memory storing the tree. Both the algorithm and the equations assume that the memory has a 2^N height and a $2^M \cdot SB$ bits width. In other words, each memory access retrieves exactly one tree node, i.e. the memory is monolithic.

The glaring problem with this approach is the inefficiency of memory accesses, which underutilize their bandwidth when the offset is positive. A positive offset discards a number of starting symbols of the sequence's starting node. Consequently, the first memory access is, in general, underutilized. This is the reason why an L-sequence requires $\lceil \frac{L}{2^M} \rceil$ or $\lceil \frac{L}{2^M} \rceil + 1$ cycles to retrieve from memory, for a zero or positive offset respectively. The variation in decoding time per offset both reduces the average throughput and introduces further irregularity in the instantaneous throughput.

Fortunately, a relatively simple solution exists. Instead of using a monolithic memory, we can create 2^M smaller memories. Each of these slices contains one of the symbols per node, i.e. slice i contains all the i-th symbols of all nodes. Now, each slice can be addressed individually, outputting valid symbols from all slices, in all but the final memory access per L-sequence.

The two memory organization schemes are illustrated in figure 6.3. An important note here is the number of necessary addresses to access the sliced memory. Just two distinct addresses are necessary, that of the current node and its parent, since 2^M adjacent symbols can span at most two adjacent nodes. However, one of the two distinct addresses does need to be supplied to every slice individually.

A side effect of the sliced memory is the need to rotate the output symbols. The relative position of the output symbols is crucial and must match that of the desired L-sequence. In the monolithic approach, it is always guaranteed that the output symbol s_i will precede, in the L-sequence, the output symbol s_{i+1}. In the sliced memory, however, this is not true, as some of the first symbols are retrieved from the parent node, not the current one. Therefore,

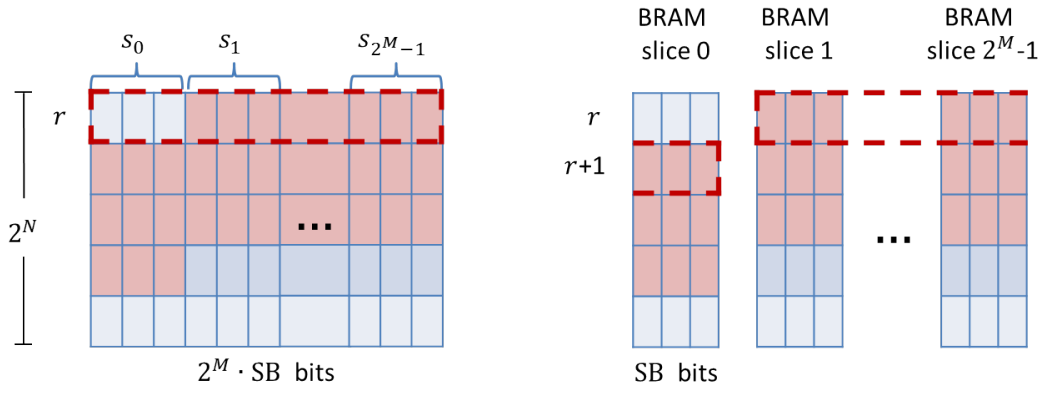


FIGURE 6.3: Monolithic (left) vs sliced (right) memory

we need to rotate the symbols by the offset amount, to ensure the correct output order.

Organizing and accessing the memory in this way homogenizes the expected number of decompression cycles per L-sequence and reduces it to

$$E'_D = \left\lceil \frac{L}{2^M} \right\rceil \tag{6.1}$$

which yields an average decompression throughput of

$$\mathcal{T}' = \frac{L}{E'_D} = \frac{L}{\left\lceil \frac{L}{2^M} \right\rceil} \frac{symbols}{cycle} \tag{6.2}$$

By known identities of the ceiling function, it holds that

$$E'_D = \left\lceil \frac{L}{2^M} \right\rceil = \left\lfloor \frac{L-1}{2^M} \right\rfloor + 1 = \left\lfloor \frac{L+2^M-1}{2^M} \right\rfloor = \lfloor E_D \rfloor \leq E_D$$

so the throughput of equation 6.2 is guaranteed to be at least as good or better than that of equation 5.23. In other words, sliced memory improves the decompression throughput.

6.4.3 Pipeline

With the packet layouts and memory organization in mind, a description of the notation that will be used in the block diagrams is summed up in figure 6.4, and the decoder datapath can now be presented in figure 6.5. The design is, naturally, pipelined to achieve high clock speeds, and each stage is elaborated on below.

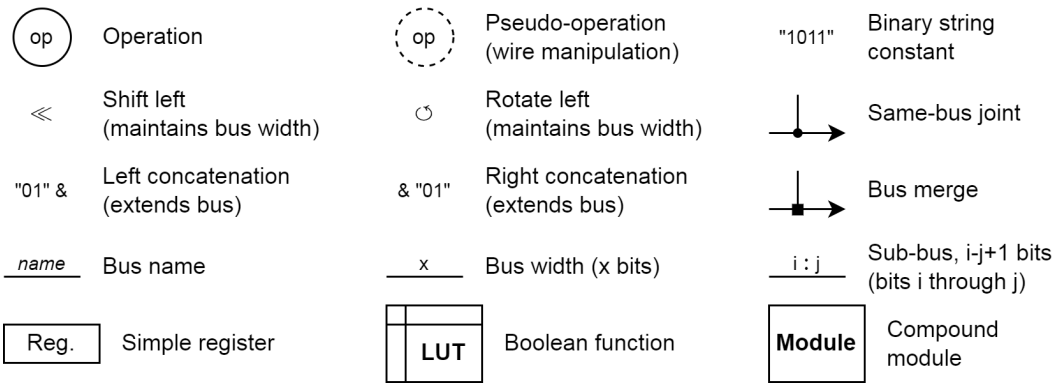


FIGURE 6.4: Block diagram notation

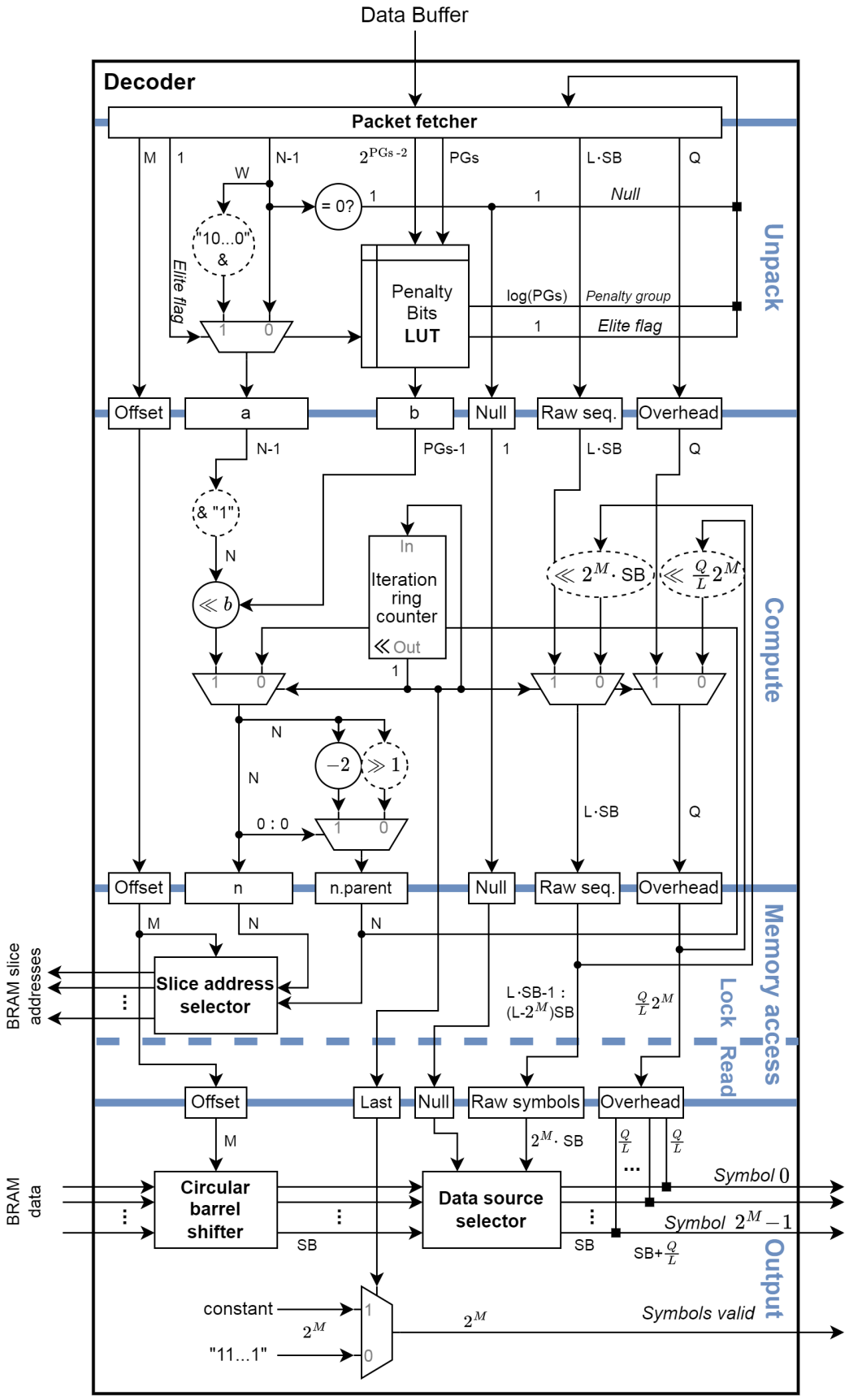


FIGURE 6.5: Decoder datapath

Fetch

The fetch stage is separately presented in figure 6.6 for visualization purposes. The role of this stage is to fetch and align packets from the data buffer. The alignment is necessary for the rest of the datapath to extract the packet fields from their known locations.

This stage will need to operate on a FIFO-word level, as it connects directly with the buffer. The FIFO can supply at most one word per cycle and the

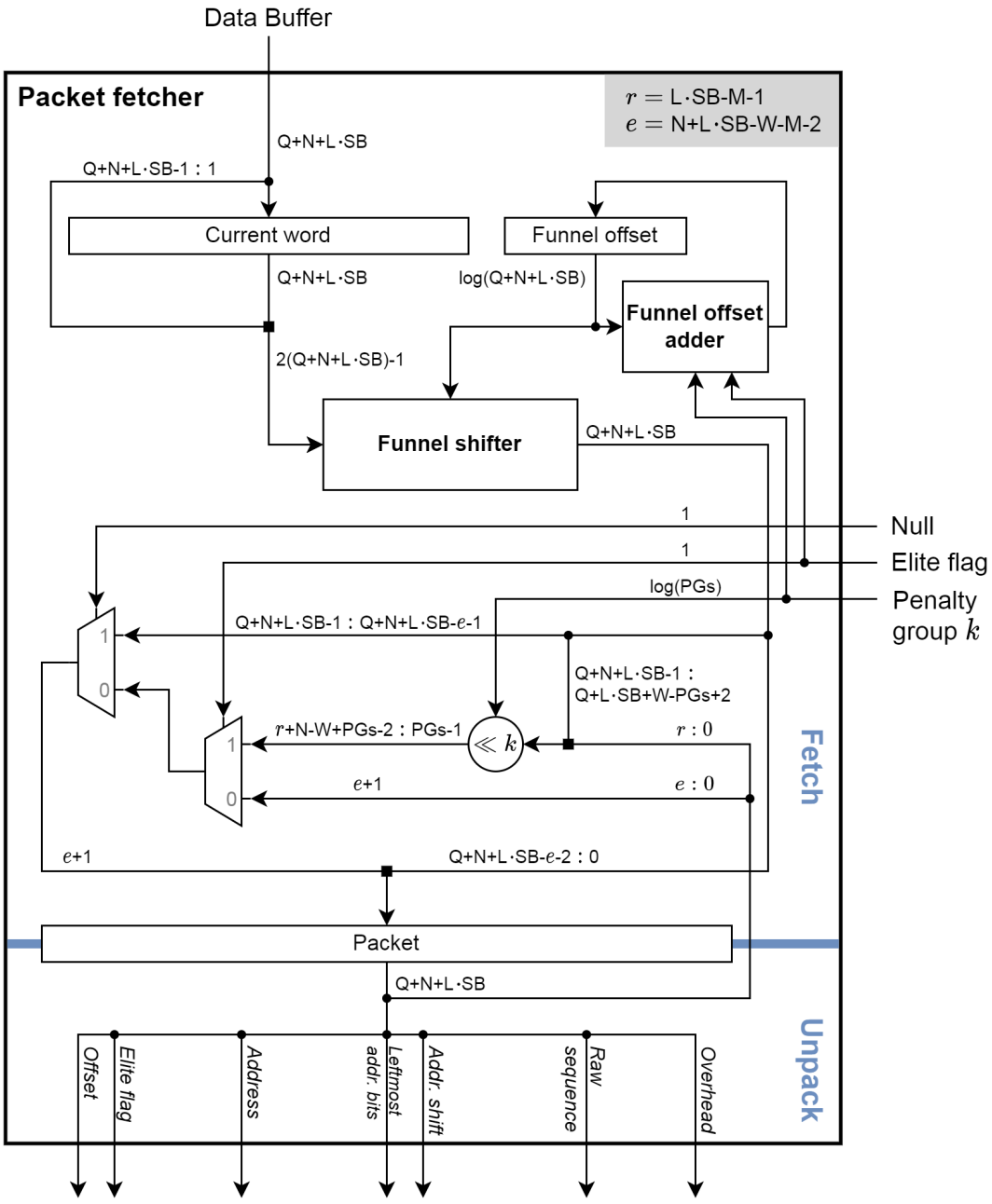


FIGURE 6.6: Packet fetcher

decompressor requires, based on 6.1, at least a single cycle for any packet. Therefore, the FIFO word width must be equal to the width of the largest packet. Thus, the FIFO can supply data quickly enough, even if the design parameters yield a single-cycle decompression duration.

However, the fetch stage cannot rely solely on a single, the topmost, FIFO word, since the packets cross word boundaries. Thus, since any packet can span at most two words, the two topmost words are needed to guarantee the supply of one packet per cycle. Of course, a FIFO provides access only to the topmost word, so an intermediate register is used to hold the current word, and the next word is read directly from the FIFO.

with the help of a counter, a funnel shifter maintains a word-wide window of the next bits in the data that have yet to enter the packet register. This word-wide window is extracted from the current and next FIFO words. The structure of the funnel shifter is generically illustrated in figure 6.7.

Whenever a packet proceeds to the unpacking stage in the pipeline, the contents of the packet register are sufficiently shifted to the left such that all bits pertaining to the packet, now in the unpacking stage, are thrown away, and the next packet is aligned in the register as per figure 6.2. For each packet type, the static length is known and so the shifting is achieved by static wiring. For regular addressing mode packets, the address shift field varies in length, so a variable shifting circuit is necessary, on top of static shifting.

For all packets, regardless of type, the leftmost bits, amounting to a width equal to that of the shortest packet type, will certainly be thrown away. As a result, the rightmost bits, of the same width, will always be replaced by new ones from the funnel shifter, namely the funnel shifter's leftmost bits.

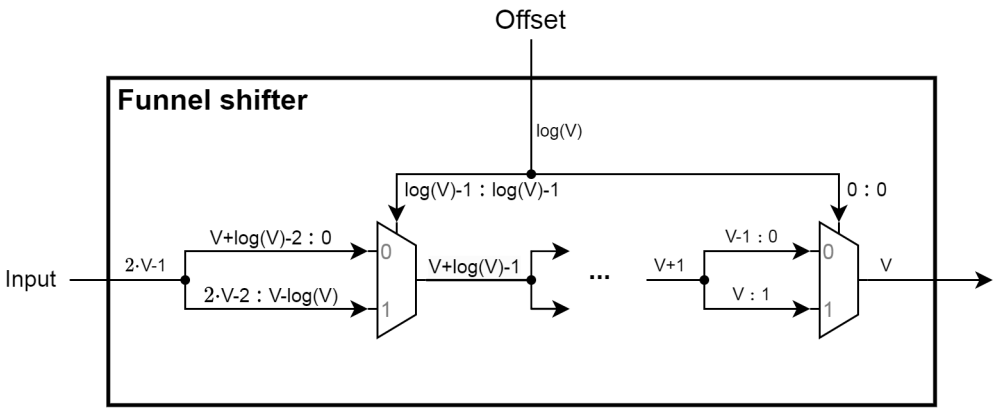


FIGURE 6.7: Generic funnel shifter

Therefore, these bits do not need to go through any additional multiplexing, but can be directly written to the packet register.

A high-level example of the fetch stage is depicted in figure 6.8. For this example, the compression parameters of the unsigned weights of section 5.5 were used. Each color represents an individual packet with the specified length. The funnel window depicts the output of the funnel shifter and its relation to its inputs. Note that when the shift value becomes equal to or greater than the word width, the latter is subtracted from the former.

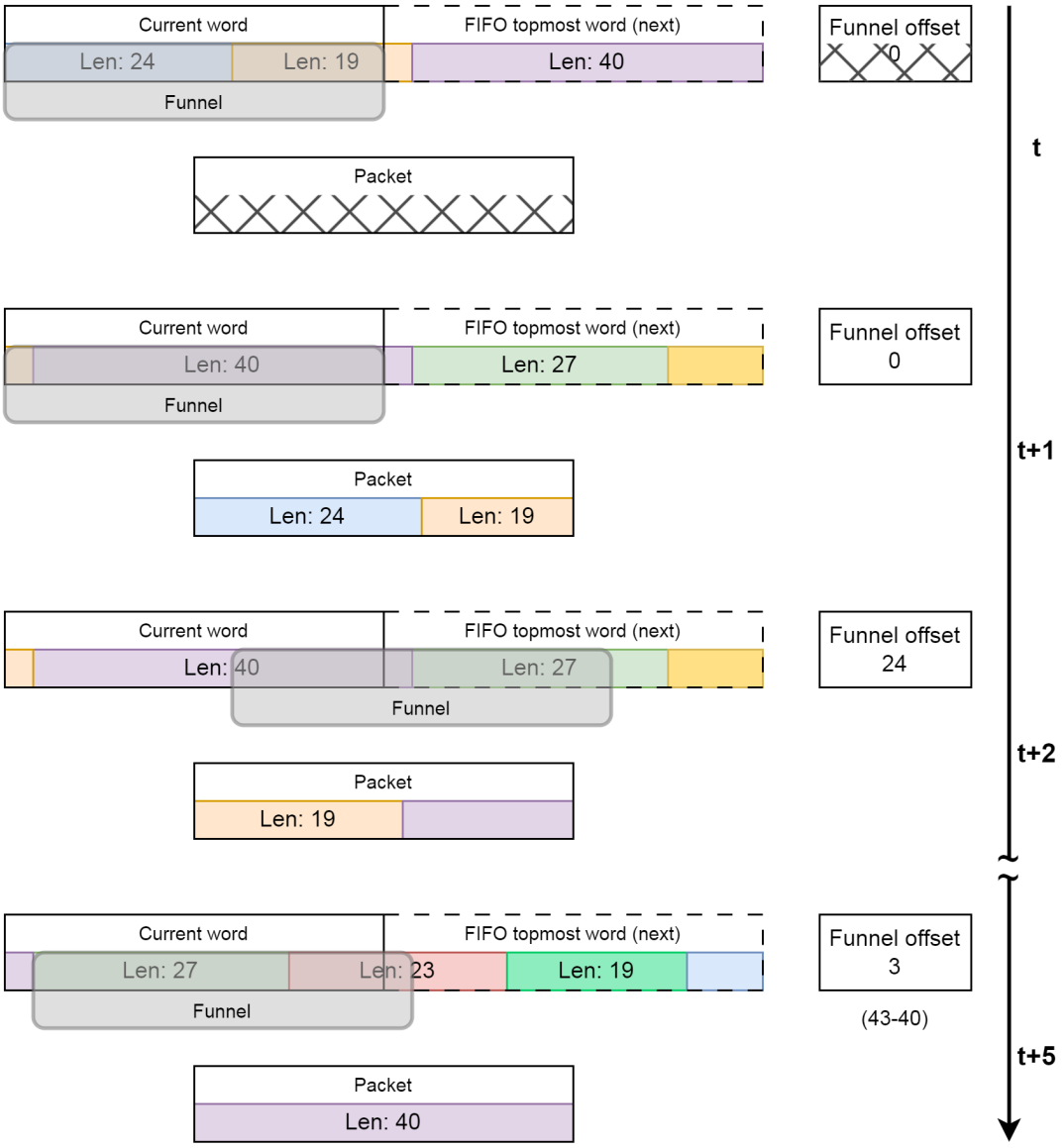


FIGURE 6.8: Fetch stage - high level example

With figure 6.8 providing, at least, an intuitive understanding of the stage in general and the funnel shifter in particular, we now inspect the deceptively simple operation of the funnel offset adder. The sole purpose of this module, which is presented in figure 6.9, is to update the funnel offset, depending on the type of the previously fetched packet. Furthermore, it needs to keep the offset in the range $[0,\ Q + N + L \cdot SB - 1]$, i.e. the valid offset range of the funnel shifter.

We can comprehensively describe the module's desired behavior by equation 6.3, where function f, defined in equation 6.4, is the offset's increment function. The constants ℓ_E, ℓ_R, and ℓ_U denote the constant length of each respective packet type. These constants are defined in figure 6.4 and are derived from the packet layouts of figure 6.2. More precisely, for regular packets, which have variable length, ℓ_R is the minimum packet length.

$$\text{funnel offset}_t := f(\text{funnel offset}_{t-1},\ k_{t-1}) \mod \ell_U \tag{6.3}$$

$$f(x,k) := \begin{cases} x + \ell_E & \text{if elite packet} \\ x + \ell_R + k & \text{if regular packet} \\ x + \ell_U & \text{if unmapped packet} \end{cases} \tag{6.4}$$

A naive design approach would be to construct equation 6.3 structurally from its parts. Since the increment function 6.4 is, at most, a three-operand sum, two binary adders are necessary to implement it. Then, for the modulo operation, a magnitude comparator in the form of a subtractor and a multiplexer are needed, since ℓ_U is generally not a power of two. This approach, while valid, fails to appreciate the maximum potential of the utilized hardware.

An adder utilizing the carry chain logic can select one among several additions to perform, with no extra hardware cost. This benefit is not surprising when one considers the implementation of adders in FPGAs. The inputs to the carry chain logic are supplied by the outputs of the LUTs in the same CLB. Therefore, since these LUTs will be partially used to implement the adder, they can simultaneously act as multiplexers for one of the operands. This simple observation, combined with basic properties of binary arithmetic, led to the funnel offset adder design of figure 6.9.

All operations are in unsigned arithmetic and we note that the only variable operands are the penalty group k and the previous offset value, while all others are constants. The first adder partially implements the function in equation 6.4, since a single carry chain adder can add either two variables or a variable and a constant. The second adder attempts to perform the modulo operation, by subtracting ℓ_U, i.e. the maximum packet length. This adder acts as a comparator of its input versus ℓ_U. Lastly, the final adder performs a correction, adding back ℓ_U if the previous operation underflowed.

The observant reader may notice that figure 6.9 does not address the unmapped packets at all. The reason for this is that by equation 6.3, the funnel offset is not altered for unmapped packets, because the addition of ℓ_U is cancelled by the modulo operator. And while the funnel offset adder will always produce the wrong output for unmapped packets, there is an exceedingly simple way to account for it. Instead of trying to produce the correct result for a value we already have stored in the register, we can simply opt not to overwrite it, by disabling the register. This is part of the minimal control path necessary for the decoder, but simply enough, the funnel offset register should be overwritten if and only if the carry out of the last adder is zero.

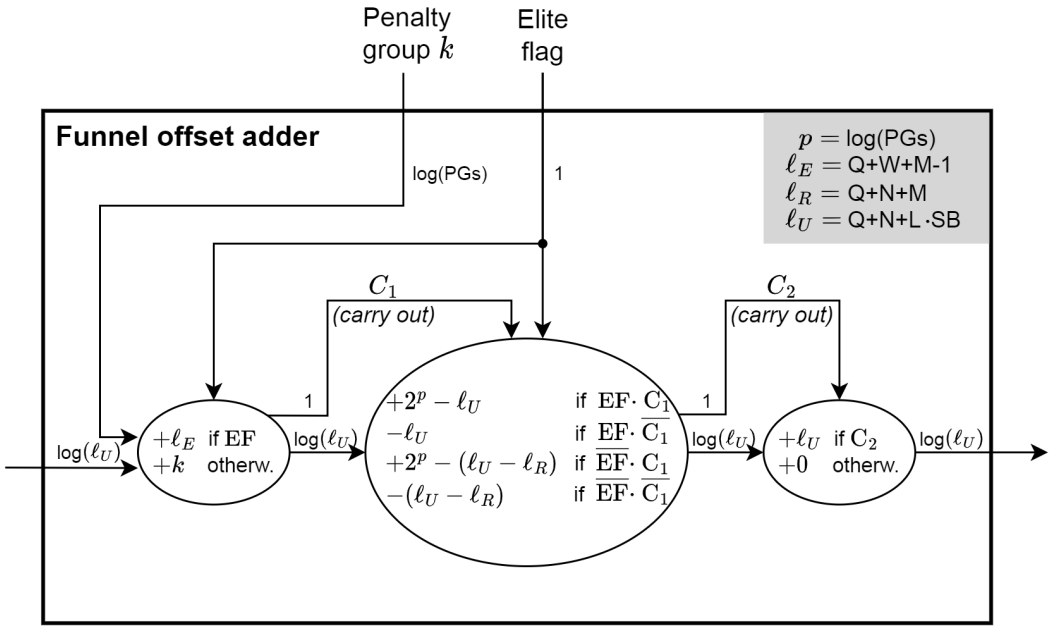


FIGURE 6.9: Funnel offset adder

Unpack

With packets being fetched and aligned in the packet register, we now move on to the unpack stage. The goal of this stage is to split the packet into its component fields and provide a common basis for these fields across packets. The packet layouts in figure 6.2 already offer easy access to all fields except for the penalty bits, which vary in length. Thus, the unpack stage needs to determine the penalty bits, and transform elite addresses into regular primary ones.

Recall from chapter 5, that elite addresses refer to the 2^W topmost nodes in PG-0, and that the addresses in the packets are primary addresses, i.e. they only address nodes in the primary branch. The topmost node in PG-0 is also the group's starting node, i.e. $SNPG(0)$, which has a primary address of 2^{N-2}. So to transform a W bit elite address into a primary $N-1$ bit address, we add $N-W-1$ zeros to the left and set the most significant bit of the result.

As for extracting the penalty bits, the penalty group k first needs to be determined from the primary address via equation 5.11. Generally, the boolean function for producing k from the primary address is illustrated in figure 6.10. Essentially, for the primary address to belong to penalty group k, there must be 2^{k-1} leading zeros, followed by 2^{k-1} bits that cannot all be zero. Observe that for the last penalty group, it is sufficient to check for 2^{PGs-2} leading zeros, i.e. infer the group by the process of elimination. Thus, the leftmost 2^{PGs-2} bits of the primary address are sufficient to determine k.

To keep the propagation delay to a minimum, we do not supply the actual primary address in the case of elite packets. Instead, we include the elite flag

Primary address	k	
1ddddddd...	000...	(0)
01ddddddd...	001...	(1)
001dddddd...	010...	(2)
0001ddddd...	010...	(2)
00001dddd...	011...	(3)
000001ddd...	011...	(3)
0000001dd...	011...	(3)
00000001d...	011...	(3)

FIGURE 6.10: Penalty group truth table

to the penalty group boolean function, outputting the zero penalty group when the flag is set and ignoring it otherwise.

Then, $PGs - 1$ bits from the packet should be shifted right by k. However, shifting by variable offset is far more expensive than bit masking, so a different scheme will be adopted to extract the penalty bits.

We introduce a new rule in the packet specification, that the penalty bits in the regular packet of figure 6.2 will be provided in reverse order, i.e. the least significant bit will be placed at index $L \cdot SB - M - 1$ and so on. Encoding the penalty bits this way provides a static location for each bit significance, even though the encoded numbers will vary in length. Therefore, $PGs - 1$ bits from the packet can now be masked with the help of k, to yield the actual, zero-padded penalty bits, which contain the address shift.

Note that both k and the penalty bits will be wrong in the case of unmapped packets. But, since neither value is used for these packets, we can safely ignore them.

Compute

Moving on to the compute stage, we make use of the unpacked fields to compute the nodes whose symbols make up the compressed L-sequence. For mapped L-sequences, the starting node encoding yields the node address via equation 6.5, where a is the primary address and b is the address shift. This formula follows directly from section 5.2.4 and is also used in the decoding pseudocode in algorithm 8.

$$n = (2a + 1) \ll b = 2^b(2a + 1) \tag{6.5}$$

The linear term is exceedingly simple to implement by appending a set bit to the right of a. This operation is cost-free, as it is achieved by proper wiring of the signal. For the left shift, a barrel shifter is used.

To make full use of the sliced memory scheme, as described in section 6.4.2, we also produce the parent of the current node. The computation is, again, exceedingly simple, and the relevant hardware is a direct implementation of the parent node equation 5.2.

However, to compute the parent, we need to select either the starting node of equation 6.5 as it is computed based on the unpack stage fields, or the node

previously computed by the compute stage. In other words, some type of counter is necessary to use either the L-sequence's starting node, or subsequent ones.

An elegant solution for the counter is to use a LUT as a shift register, and wire it to implement a ring counter. The period of this ring counter will be set to $\lceil \frac{L}{2^M} \rceil$, i.e. the number of cycles needed to decompress a packet. When the single 1 circling the counter reaches the counter's output, it signals the start of a new decompression cycle. Thus, only for the first iteration of the new decompression cycle, we select the address of equation 6.5 as the input to register n.

We emphasize the deliberate omission of a register for the counter's output in the compute stage. In later stages, a signal denoting the last iteration of each decompression cycle will be needed. Since an output of 1 from the ring counter denotes the first iteration of a decompression cycle at time t, it also denotes that the iteration at time $t-1$ was the last of the previous decompression cycle. Therefore, we send the signal backward in time, by omitting a register at the compute stage, to catch up to the last iteration of a decompression cycle in the later stages.

Similar logic to the node address also applies to the raw sequence and overhead registers. On the first iteration of a new decompression cycle, these registers receive their respective packet fields. On subsequent iterations however, the contents of the registers are shifted such that the leftmost bits of these registers always refer to the symbols of the current iteration.

Memory access

The memory access stage provides each memory slice the correct address, from which to retrieve data. In a broader context, this stage prepares the memory to read the first, or next, 2^M of the current encoded L-sequence.

For each slice to receive the correct address, the current node and its parent are fed into the slice address selector presented in figure 6.11, which selects either of the two addresses for each slice, based on the symbol offset. The selection, for some slice i, works based on the rule in equation 6.6.

$$\text{address}_i = \begin{cases} n & \text{if } i \geq \text{offset} \\ n.parent & \text{if } i < \text{offset} \end{cases} \tag{6.6}$$

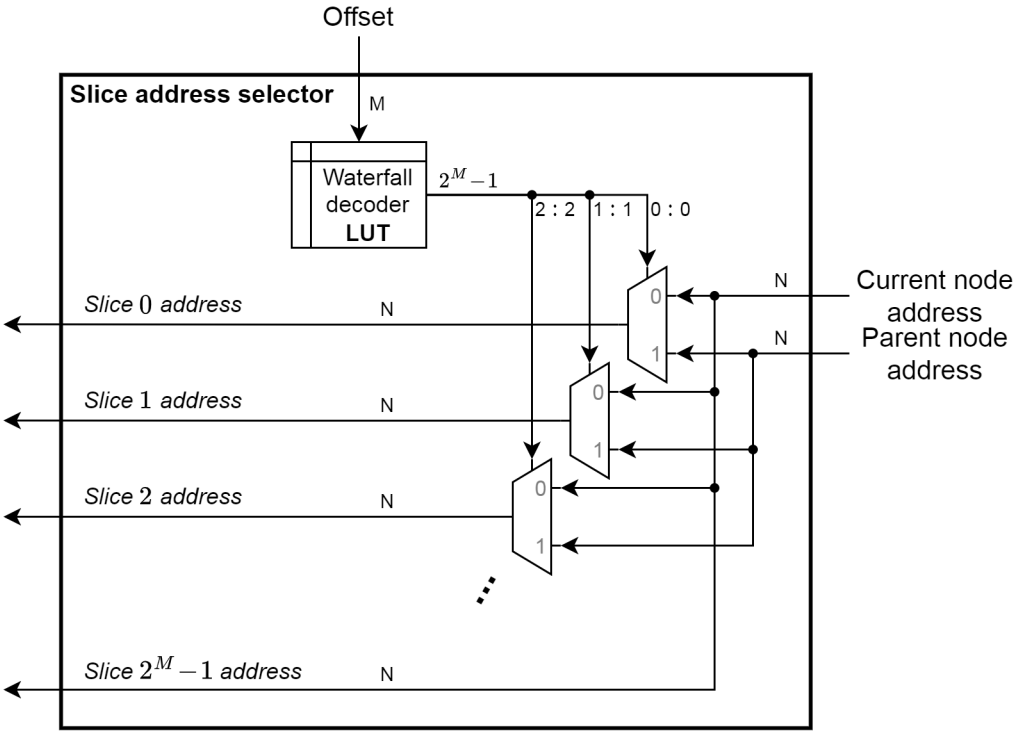


FIGURE 6.11: Slice address selector

To provide the multiplexer's selection bits, a boolean function with M input bits and 2^M output bits is necessary. This function has been named waterfall decoder, for its similarity with the decoder/demultiplexer circuit and the waterfall effect of its truth table, which is presented in figure 6.12.

With the proper addresses being supplied to the slices, a major consideration now is how the BRAM works. The BRAM's logic diagram, provided by Xilinx for the UltraScale architecture, is shown in figure 6.13. The BRAM is synchronous, thus the read address must be stored in the module's register.

	Symbol offset	Selection bits
(0)	...000	00000000...
(1)	...001	10000000...
(2)	...010	11000000...
(3)	...011	11100000...
(4)	...100	11110000...
(5)	...101	11111000...
(6)	...110	11111100...
(7)	...111	11111110...
		bit 0 bit 7

FIGURE 6.12: Waterfall decoder truth table

The configurable option, however, is whether to register the BRAM's output as well, using the module's internal registers.

The advantage of registering the output is the minimization of the propagation delay, and thus higher maximum clock speeds for the entire design. Depending on the size of the tree stored in the BRAMs and the target clock speeds, registering the output may not be necessary. For large trees and high clock speeds, the use of the output register is imperative.

Therefore, in the decoder pipeline of figure 6.5, the memory access stage is optionally split into two substages, the address lock substage and the memory read substage. The former locks the read address to the BRAM address register, while the latter writes the BRAM's asynchronous output to the output register.

Of course, if the memory access stage is indeed split, additional registers between the lock and read substages are necessary for the pipeline to function correctly. Specifically, a register would be needed for every bus that crosses the boundary between the lock and read substages. To avoid any confusion, we underline that no signals shown in figure 6.5 that cross the substage boundary are exempt from this requirement. This note serves to disambiguate the signal denoting the last iteration, which was not registered in the compute stage.

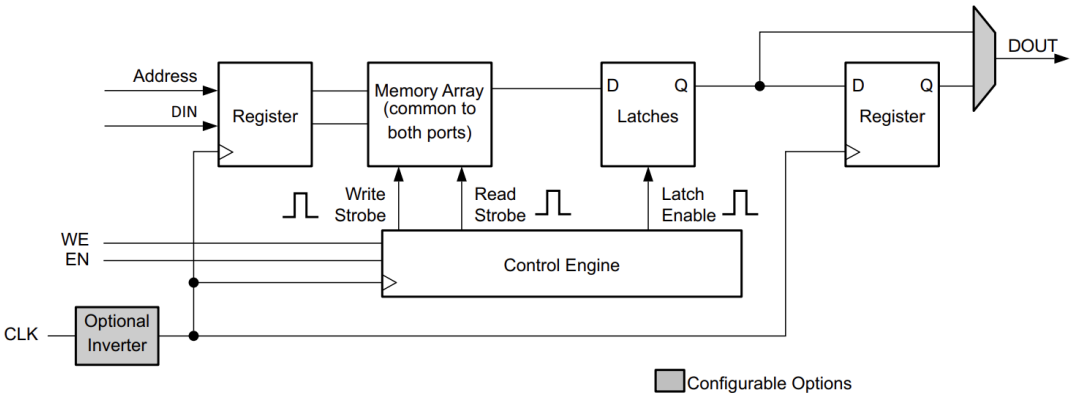


FIGURE 6.13: BRAM logic diagram (source: [105])

Output

Lastly, in the output stage, the symbols read from the sliced memory need to be placed in the correct order, or replaced entirely by the raw symbols, in the case of unmapped sequence packets. Also, this stage needs to produce a

symbol-valid signal for each of the 2^M output symbols, such that the receiving end of these symbols can discern between useful and garbage data.

To reorder the incoming symbols, a circular barrel shifter is used, illustrated in figure 6.14, which operates on the symbol level. This rotation of the symbols by the symbol offset is the inverse of the transformation inherently introduced to the symbols by the efficient sliced memory access. The relative order of the symbols after the circular barrel shifter matches that of the original L-sequence.

For unmapped sequences, the respective packets contain the uncompressed L-sequence, so this stage must output the data in the raw symbols register, instead of the data read from memory. This is a trivial data flow, but the block diagram of the data source selector is nonetheless presented in figure 6.15.

As for the symbol-valid signal, we identify three stage statuses: reset, non-last decompression iteration, and last decompression iteration. The reset status appears for the first time now, and is relevant to the control path, not the datapath. Whenever the output stage is reset, as part of a pipeline reset, all symbol-valid signals become zero. During any decompression iteration other than the last, all 2^M symbols are valid, which is precisely the main benefit of sliced memory, so all symbol-valid signals are asserted. For the last decompression iteration, only the lower $L \mod 2^M$ symbols are valid, so the symbol-valid signal gets a constant value, that of the waterfall decoder in figure 6.12 computed for the input $L \mod 2^M$.

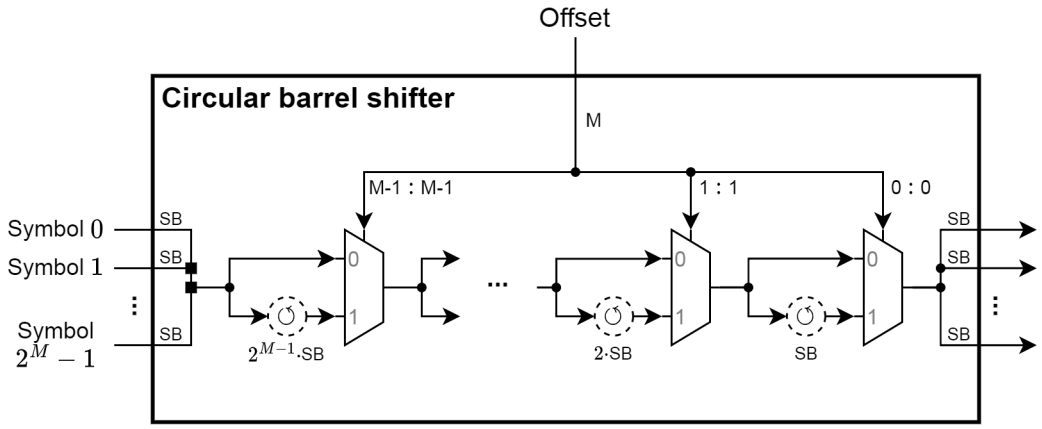


FIGURE 6.14: Circular barrel shifter

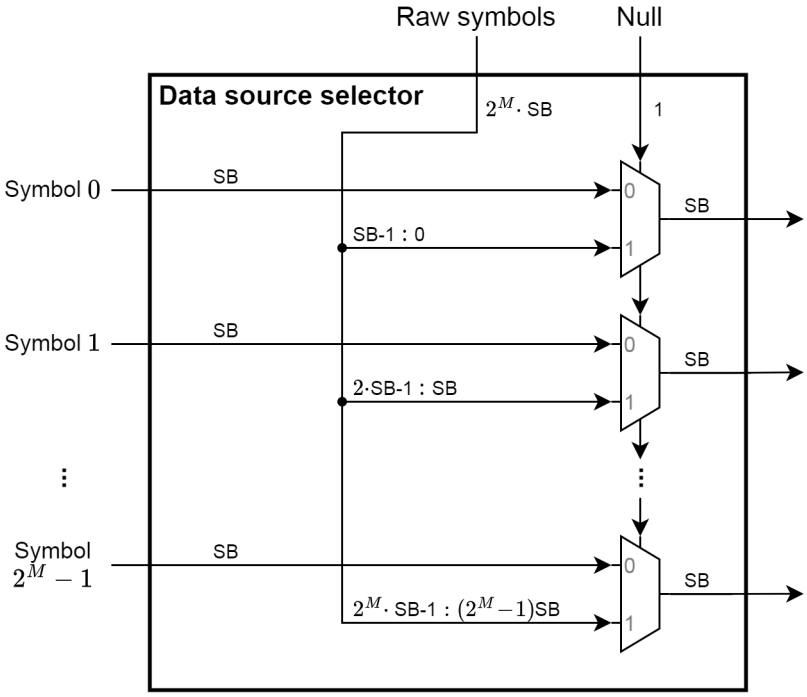


FIGURE 6.15: Data source selector

6.4.4 Resource utilization & propagation delay model

Having discussed the decoder pipeline, we will now model the cost of the decoder in FPGA resources. This model will provide valuable insight into how the architecture scales, as the design parameters increase.

The first step in our analysis is to calculate the cost of the basic building blocks, found throughout the design, namely multiplexers, adders, and barrel shifters. Table 6.1 presents generalized equations for the LUT cost of these circuits. Note that V is a generic variable, not a design parameter. The propagation delay is given in τ, which is the propagation delay of a single LUT. For this table, and all others dealing with resource costs, the ceiling function around the equations is implied in every cell, but is omitted for brevity.

A LUT can implement a 2-bit 2:1 multiplexer, since the MUX inputs are 4 bits total, plus 1 selection bit. As discussed in section 6.1.1, a 5-input LUT can have 2-bit outputs, so the MUX can indeed be implemented in a single LUT. Additionally, a LUT can implement a 1-bit 4:1 MUX, since there are 4 input bits total plus 2 selection bits, and a 6-input LUT has a single output bit. The first implementation yields a denser physical design, while the second implementation can reduce the propagation delay in half in the case of multiple levels of MUXs, as in barrel shifters.

TABLE 6.1: Resource cost and propagation delay of basic circuits

Circuit	Logic levels	LUTs		Propagation delay (in τ)
		Per level	Total	
Multiplexer	1	$\frac{V}{2}$	$\frac{V}{2}$	1
Adder	1	V	V	1
Barrel shifter	$\log(V)$	$\frac{V}{2}$	$\frac{V}{2}\log(V)$	$\log(V)$

However, in modern FPGAs, the delay due to routing is most often larger than that of the logic. Thus, while using 1-bit 4:1 MUXs to merge two barrel shifter levels would, theoretically, halve the propagation delay, we refrain from including this assumption in the equations as it may be entirely unrealistic, due to increased routing delays.

Now that the cost of the basic building blocks is known, we proceed to calculate the cost per module used in our design. This information is presented in table 6.2 and is calculated for the actual buses that connect to each module. The packet fetcher module and the decoder are excluded from table 6.2, as the cost of each pipeline stage will be calculated later.

The equations for the funnel shifter's resource cost require elaboration. There are indeed $\log(\ell_U)$ logic levels of MUXs, but the number of MUXs per level varies. Specifically, there are $\ell_U + i$ MUXs on level i, where $i \in [0, \log(\ell_U) - 1]$.

TABLE 6.2: Resource cost and propagation delay per module

Module	Logic levels	LUTs		Propagation delay (in τ)
		Per level	Total	
Funnel offset adder	3	$\log(\ell_U)$	$3\log(\ell_U)$	3
Funnel shifter	$\log(\ell_U)$	$\frac{\ell_U}{2} + \frac{\log(\ell_U)-1}{4}$	$\left(\frac{\ell_U}{2} + \frac{\log(\ell_U)-1}{4}\right)\log(\ell_U)$	$\log(\ell_U)$
Slice address selector	1	$\frac{N}{2}\left(2^M - 1\right)$	$\frac{N}{2}\left(2^M - 1\right) + 2^{M-1}$	2
Circular barrel shifter	M	$\frac{2^M \cdot SB}{2}$	$M\frac{2^M \cdot SB}{2}$	M
Data source selector	1	$\frac{2^M \cdot SB}{2}$	$\frac{2^M \cdot SB}{2}$	1

Thus, the total number of MUXs is given by expression 6.7, and correspond to half as many LUTs. However, to simplify table 6.2, the funnel shifter cost per level presents the mean cost per level.

$$\sum_{i=0}^{\log(\ell_U)-1} (\ell_U + i) = \left(\ell_U + \frac{\log(\ell_U) - 1}{2}\right) \log(\ell_U) \tag{6.7}$$

As for the slice address selector, the single logic level and the per level cost appearing in table 6.2 refers to the MUXs only. The cost of the waterfall decoder has been added to the LUT total, and the propagation delay also accounts for the waterfall decoder, hence the extra delay.

With the cost of all the individual pieces presented, we now turn to the pipeline itself. The LUT cost per stage and the propagation delay are presented in table 6.3. For each stage, the total LUT cost is given by the sum of the total LUT costs of its components. These sums are not written out for the sake of brevity and readability. However, the propagation delay per stage is written out and refers to each stage's critical path. It is, therefore, not the sum of the stage component's individual delays.

Two of the equations require elaboration. Firstly, the MUX in the unpack stage is depicted in figure 6.5 having $N-1$ bit inputs, and would thus require half as many LUTs to implement. However, because of the concatenation with the "$100\cdots$" binary vector on the second input, a MUX with only W bit inputs is sufficient, as the MUX select bit can be directly used to set or reset the upper bits of the register, thus completing the functionality.

Secondly, the equations for the penalty bits LUT in the unpack stage are a rough estimate. The cost of implementing the truth table of figure 6.10 is approximated by the number of output bits, i.e. $\log(PGs)$, and has a maximum propagation delay of 2τ, for any realistic choices for the design parameters. The other half of the penalty bits LUT, i.e. the masking of the penalty bits, requires exactly $\frac{PGs-1}{2}$ LUTs on a single logic level, for N up to 2^{31}, which is laughably past the realm of reasonable theoretical consideration, let alone practicality. Thus, the LUT cost and propagation delay for this boolean function are the sums of the individual terms, which yield the equations in table 6.3.

Apart from LUTs, the pipeline consumes FFs, the cost of which is presented in table 6.4, per stage. We assume the memory access stage is indeed split

TABLE 6.3: LUT cost and propagation delay per stage

Pipeline stage	Component	Total LUTs	Propagation delay (in τ)
Fetch			$\log(\ell_U) + \log(PGs) + 2$
	Funnel offset adder	$3\log(\ell_U)$	3
	Funnel shifter	$\left(\frac{\ell_U}{2} + \frac{\log(\ell_U)-1}{4}\right)\log(\ell_U)$	$\log(\ell_U)$
	Barrel shifter	$(\ell_U - \ell_E + PGs - 1)\log(PGs)$	$\log(PGs)$
	MUXs	$\ell_U - \ell_E$	2
Unpack			3
	MUX	$\frac{W}{2}$	1
	Comparator	$\frac{N-2}{5}$	$\log_5(N-1)$
	Penalty bits LUT	$\frac{PGs-1}{2} + \log PGs$	3
Compute			$3 + \log(PGs - 1)$
	Barrel shifter	$\frac{N}{2}\log(PGs-1)$	$\log(PGs-1)$
	Subtractor	N	1
	Address MUXs	N	2
	Ring counter	$\frac{L}{32}$	1
	Raw seq. MUX	$\frac{\max(0,\, L-2^M)\cdot SB}{2}$	1
	Overhead MUX	$\frac{Q}{2}$	1
Memory access			2
	Slice address selector	$\frac{N}{2}\left(2^M - 1\right) + 2^{M-1}$	2
Output			$M + 1$
	Circular barrel shifter	$M\frac{2^M \cdot SB}{2}$	M
	Data source selector	$\frac{2^M \cdot SB}{2}$	1
	MUX	2^{M-1}	1

into two substages, as this is the more performant option. Additionally, we also assume the decoder's outputs are registered, as this again is the more performant option, so registers for the output stage have been included.

The model of table 6.3 is, to the best of our abilities, representative of the real-world resource cost of the decoder's implementation, but the reader should keep in mind that it may deviate from the actual resource cost of said implementation. The reasons for this have to do with the inherent inability of a generalized model to account for case-by-case optimizations, the hardware tool features, and practical limitations.

The hardware implementation tools support several optimization settings, be it for area, power consumption, or speed, which directly affect the mapping of logic to resources. Furthermore, when a design is implemented, part of the logic may be duplicated or constructed by shallower but wider LUT trees, in order to meet timing requirements. Naturally, all these factors cannot possibly be taken into account in our model, so some deviation is expected from the actual resource utilization. However, we reasonably expect the tools to do a better job at logic optimization than our theoretical analysis, so our cost model should, generally, provide an upper bound.

To gain some quantitative insight into our model of table 6.3, we plot the cost for several combinations of the parameters. Figure 6.16 presents the decoder's cost in an unconstrained manner, where the design parameters are chosen freely and independently of each other. This plot is useful for observing each parameter's effect on the LUT cost.

TABLE 6.4: FF cost per stage

Pipeline stage	**Substage**	**Total FFs**
Fetch		$2 \cdot \ell_U + \log(\ell_U)$
Unpack		$\ell_U + M + PGs - 1$
Compute		$\ell_U + N + M + 1$
Memory access	Lock	$M + 2^M \left(SB + \frac{Q}{L}\right) + 2$
	Read	$M + 2^M \left(SB + \frac{Q}{L}\right) + 2$
Output		$2^M \left(SB + \frac{Q}{L} + 1\right)$

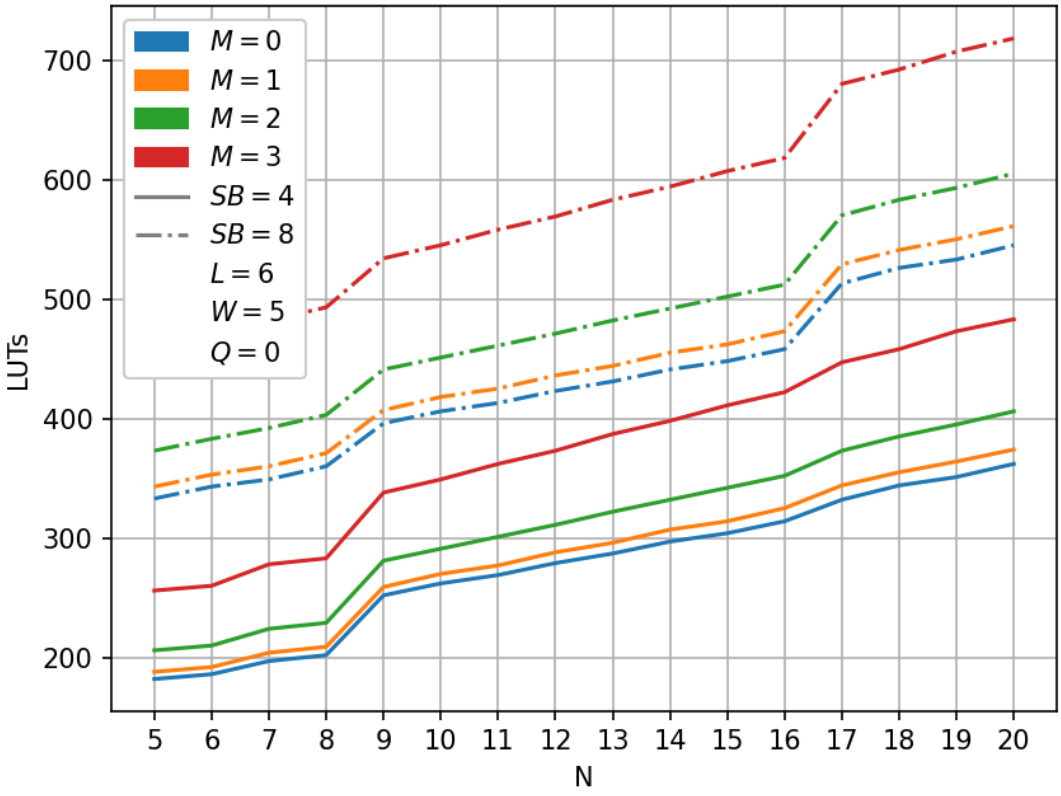


FIGURE 6.16: Decoder's LUT cost for several parameter combinations

Equally interesting is an investigation into the LUT cost as a function of M, but for a given tree. Recall that M is the parameter determining the throughput and a major feature of our compression method is the ability to choose the throughput, via M. But, for a given dataset and L, the tree requires a constant size, and the elite window will need to contain some constant number of symbols. Therefore, when varying M for a given tree, both N and W need to be adjusted, such that $N + M$ and $W + M$ remain constant. In other words, these three parameters are constrained. The LUT cost as a function of M in the constrained case is plotted in figure 6.17.

Figures 6.16 and 6.17 illustrate the general characteristics of the model that become evident by contemplating the equations of table 6.3. The parameters N and SB linearly affect the LUT cost, while M has an exponential effect on it, as expected. Notably, the steeper increases on the 8 and 16 mark in figure 6.16 are the result of the penalty groups increasing by one. As for M, we observe from figure 6.17 that it can reach 3 with negligible increase in LUT consumption, while said increase becomes more pronounced for higher values of M. However, we underline the importance of figure 6.17, in that it

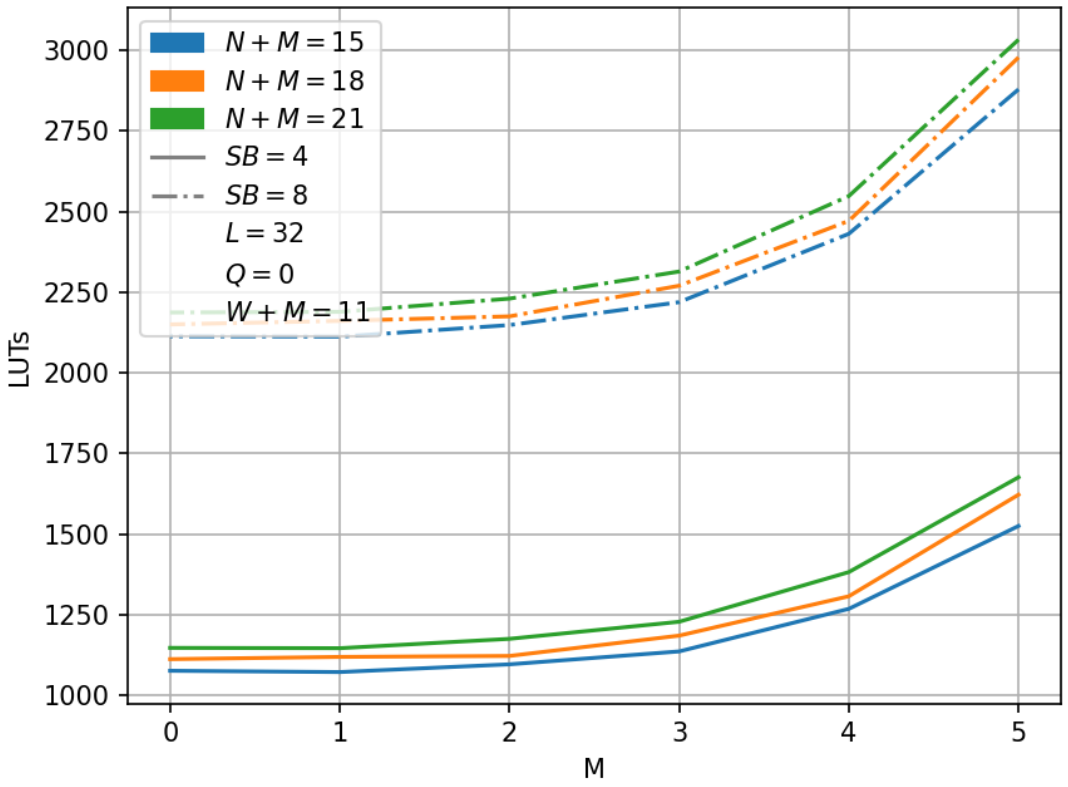


FIGURE 6.17: Decoder's LUT cost with constrained parameters

shows a possible 32-fold increase in the throughput, with less than 50% extra LUTs.

6.4.5 Multiple decoders

So far, the discussion and analysis have focused on a decompressor with a single decoder. The sliced BRAM provides the decoder with parallelized access to up to two nodes per cycle, using a single read port. However, BRAMs, as we have covered, provide two ports. Therefore, each BRAM slice can be independently read from by two decoders, without requiring memory duplication.

Effectively, we can double the decompression throughput, by implementing two decoding modules. Two cases are identified, based on the duration of a single decompression cycle.

Unit decompression duration

When L is no greater than 2^M, equation 6.1 dictates that each packet will be decoded in a single clock cycle, due to sliced memory. As a result, the

entire pipeline is always fully utilized, since no stages halt in waiting of other stages. Therefore, to take advantage of the unused BRAM port, we need to duplicate the entire decoder, possibly along with its buffer, as shown in figure 6.18. Due to duplication, the two decoders will cost twice as much as table 6.3 describes.

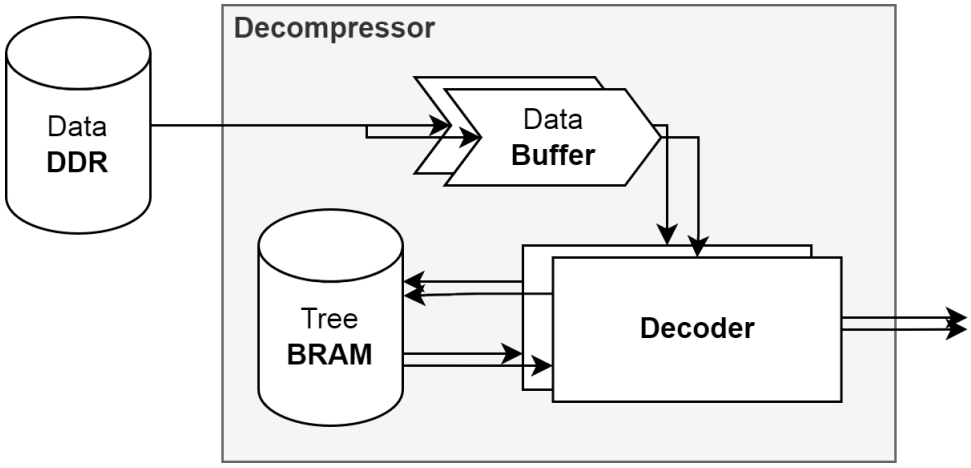


FIGURE 6.18: Decompressor with two decoders diagram

Multi-cycle decompression duration

When L is greater than 2^M, each packet will require at least two clock cycles to decode. Interestingly enough, since our decoder is iterative, the pipeline will be underutilized, due to the fetch and unpack stage having to wait for the decompression iterations to finish. This underutilization is constant, regular, and proportional to the number of clock cycles required per packet.

Even more interesting is the contribution of the fetch stage to the total LUT cost of the decoder. Figure 6.19 illustrates the contribution, per stage, to the decoder's total cost. The constrained parameters in figure 6.19 are in accordance with those in figure 6.17. It is evident that the fetch stage accounts for the overwhelming percentage of the total LUT cost.

Therefore, a far more beneficial strategy, compared to blind logic duplication, is to duplicate only the stages that are fully utilized, and share the underutilized ones amongst the two sub-pipelines. The forked decoder pipeline is presented in figure 6.20 and no additional logic or modification to the stages is necessary. Half of the pipeline has simply been duplicated, and connected directly with the unpack stage without any intermediate logic.

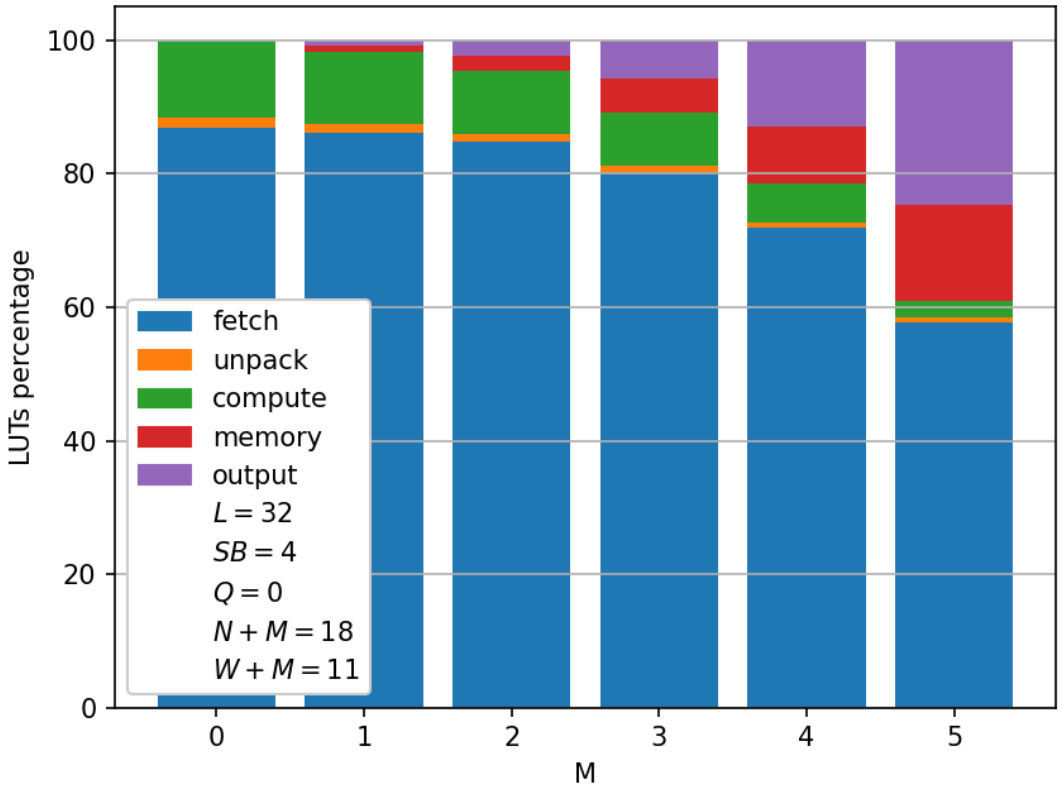


FIGURE 6.19: Contribution of each pipeline stage to the decoder's total LUT cost

Of course, each compute stage must receive data only whenever it needs to, i.e. during the first iteration of each decompression cycle, and not while the other compute stage receives its data. To achieve this, we enable the offset and null registers at the end of the compute stage using the ring counter's output bit. Therefore, every register will receive data from the unpack stage during the first iteration, while no register will receive data from the unpack register during any other iteration. For the offset and null registers, this is achieved by directly enabling and disabling them, while, for the rest of the registers, the MUXs will block inputs from the unpack stage during any iteration other than the first.

Lastly, by resetting one of the two sub-pipelines for an extra cycle during pipeline reset, we offset the decompression iterations of each sub-pipeline by one cycle. This way, no conflict is caused between the two sub-pipelines requiring the unpack stage at the same time. Each sub-pipeline is guaranteed to data from the unpack stage at regular, known, and non-conflicting time steps. In this way, we double the decompression throughput, in exchange for an additional fraction of the entire decoder's cost.

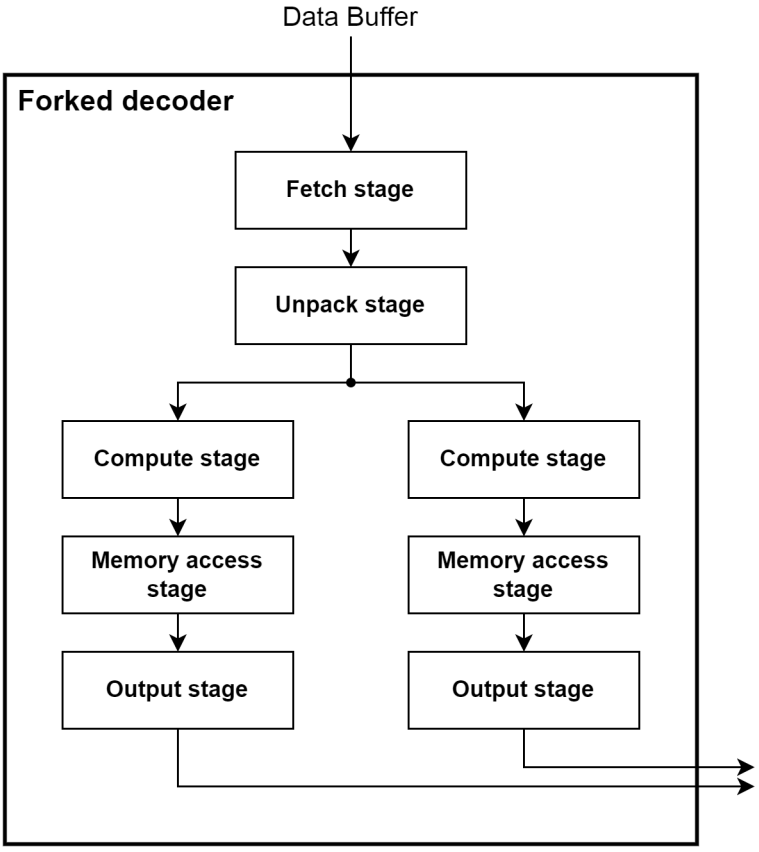


FIGURE 6.20: Forked decoder pipeline

Each sub-pipeline processes a different packet, with consecutive packets being decompressed in parallel. If, depending on the application, the data is needed in the exact order it was encoded, we can simply change the way the data is partitioned into L-sequences. Instead of splitting L consecutive symbols into L-sequences, we split them into sequences interleaved, such that consecutive symbols on same-parity indices make up an L-sequence.

6.5 Results

Using the resource cost model, the cost of implementing decompressors for the GRU weight matrices of section 5.5 can now be calculated. The parameters for the zeros were $N + M = 17$, $W + M = 12$, $SB = 5$, $Q = 0$, and L being layer specific. As for the weights, the parameters were $N + M = 17$, $W + M = 12$, $SB = 3$, $Q = L$, and $L = 6$ for all layers. The resultant decoder costs per layer, along with the throughput, for various values of M, are presented in tables 6.5 and 6.6, for the zeros and weights respectively.

TABLE 6.5: Decoder resource cost and decompression throughput per layer for the zeros

Layer	$M = 0$			$M = 1$			$M = 2$		
	LUTs	**FFs**	$\mathcal{T}$ $\left(\frac{\text{symbols}}{\text{cycle}}\right)$	**LUTs**	**FFs**	$\mathcal{T}$ $\left(\frac{\text{symbols}}{\text{cycle}}\right)$	**LUTs**	**FFs**	$\mathcal{T}$ $\left(\frac{\text{symbols}}{\text{cycle}}\right)$
1	355	237	1.00	350	251	2.00	371	282	3.00
2	355	237	1.00	350	251	2.00	371	282	3.00
3	387	257	1.00	377	271	1.75	405	302	3.50
4	447	297	1.00	437	311	1.80	465	342	3.00
5	447	297	1.00	437	311	1.80	465	342	3.00

As for the BRAM cost, it is exceedingly easy to calculate. Each decompressor requires BRAMs for the tree and the buffer. If we configure the tree's 18Kb BRAMs as 16K x 1 blocks, each slice requires

$$\text{sliceBRAMs} := \frac{2^N}{16K} SB = 2^{N-14} \cdot SB \tag{6.8}$$

18Kb BRAMs, yielding a total of

$$\text{treeBRAMs} := 2^M \cdot \text{sliceBRAMs} = 2^{N+M-14} \cdot SB \tag{6.9}$$

18Kb BRAMs to the store the entire tree in a sliced BRAM topology. This amounts to 40 18Kb BRAMs per layer for the zeros, and 24 18Kb BRAMs per layer for the weights. An additional BRAM is required for the weights and zeros decompressors, to act as the FIFO buffer. However, since each layer is processed sequentially, the buffer can be shared between decompressors of the same category.

TABLE 6.6: Decoder resource cost and decompression throughput per layer for the sign-separated weights

Layer	$M = 0$			$M = 1$			$M = 2$		
	LUTs	**FFs**	$\mathcal{T}$ $\left(\frac{\text{symbols}}{\text{cycle}}\right)$	**LUTs**	**FFs**	$\mathcal{T}$ $\left(\frac{\text{symbols}}{\text{cycle}}\right)$	**LUTs**	**FFs**	$\mathcal{T}$ $\left(\frac{\text{symbols}}{\text{cycle}}\right)$
1	304	210	1.00	297	221	2.00	312	246	3.00
2	304	210	1.00	297	221	2.00	312	246	3.00
3	304	210	1.00	297	221	2.00	312	246	3.00
4	304	210	1.00	297	221	2.00	312	246	3.00
5	304	210	1.00	297	221	2.00	312	246	3.00

Lastly, in tables 6.7 and 6.8, the forked decoder cost and throughput is presented per layer, for the zeros and weights respectively. Comparing the forked decoder to the simple decoder, via tables 6.5 to 6.8, we observe, on average, a 25% increase in LUT and a 40% increase in FF utilization for the forked decoder compared to the simple one. For this little cost increase, we are getting double the throughput in return.

TABLE 6.7: Forked decoder resource cost and decompression throughput per layer for the zeros

Layer	$M = 0$			$M = 1$			$M = 2$		
	LUTs	**FFs**	$\mathcal{T}\left(\frac{\text{symbols}}{\text{cycle}}\right)$	**LUTs**	**FFs**	$\mathcal{T}\left(\frac{\text{symbols}}{\text{cycle}}\right)$	**LUTs**	**FFs**	$\mathcal{T}\left(\frac{\text{symbols}}{\text{cycle}}\right)$
1	434	322	2.00	429	353	4.00	481	417	6.00
2	434	322	2.00	429	353	4.00	481	417	6.00
3	468	347	2.00	459	378	3.50	518	442	7.00
4	533	397	2.00	524	428	3.60	583	492	6.00
5	533	397	2.00	524	428	3.60	583	492	6.00

TABLE 6.8: Forked decoder resource cost and decompression throughput per layer for the sign-separated weights

Layer	$M = 0$			$M = 1$			$M = 2$		
	LUTs	**FFs**	$\mathcal{T}\left(\frac{\text{symbols}}{\text{cycle}}\right)$	**LUTs**	**FFs**	$\mathcal{T}\left(\frac{\text{symbols}}{\text{cycle}}\right)$	**LUTs**	**FFs**	$\mathcal{T}\left(\frac{\text{symbols}}{\text{cycle}}\right)$
1	380	286	2.00	371	311	4.00	411	363	6.00
2	380	286	2.00	371	311	4.00	411	363	6.00
3	380	286	2.00	371	311	4.00	411	363	6.00
4	380	286	2.00	371	311	4.00	411	363	6.00
5	380	286	2.00	371	311	4.00	411	363	6.00

6.6 Comparison with literature

The final consideration in this thesis is the comparison of the hardware design of our novel compression method with existing literature. The superiority of our method, in terms of decompression throughput, compared to various Huffman coding variants has already been established in chapter 5, so we will not delve into the literature on hardware decompressors for Huffman-based compression. Instead, we turn our attention to decompressors for LZ77-based methods.

We underline the goal of this section, which is to compare the hardware architectures of various decompressors, not the compression methods themselves. To enable this comparison, we will try to match the decompression throughput of other architectures with a hypothetical decompressor for our method. The parameters of the hypothetical decompressor will be assigned values that yield a throughput similar to each compared method. Whether the selected parameters are achievable on the datasets each method was tested on is beyond the scope of this comparison, as well as any discussion relating to compression ratios, as these questions relate to the individual datasets and application domains, and are thus addressable on a case-by-case basis.

Furthermore, in striving for fairness of comparisons, we will ignore clock speeds. This decision serves to level the playing field between works with different degrees of implementation optimizations for their architectures. We thus remove the impact of the tool expertise and optimization determination factors from the implementation results reported by each work. Consequently, throughput will be measured in symbols per cycle, not bytes per second.

A side effect of discarding clock speeds from the comparisons is the concealment of qualities in each architecture that directly influence clock speeds. The decompressor architecture for our method inherently provides low propagation delays per stage in the pipeline, which naturally facilitate higher clock speeds. Therefore, by omitting clock speed from the comparison criteria, we underplay our architecture, which may intrinsically allow faster clock speeds compared to other architectures. However, we do so also due to the lack of implementation data for our method and to minimize speculation.

With the above considerations in mind, we proceed to the comparison. Decompressors for two LZ77-based methods will be used, namely Snappy [107] and Deflate [108]. Both are well-established methods in literature, commonly used, and highly credited. Snappy was developed and open-sourced by Google specifically for high throughputs, while Deflate is the core part of the zip and gzip algorithms.

Both methods operate on the byte level, so our hypothetical decoders will have byte-symbols, i.e. $SB = 8$ and $Q = 0$. The hypothetical tree will be set to a reasonable constant size of 64 KiB, i.e. $N + M = 16$. The elite window's effect on the decoder's resource cost is entirely negligible, but for completeness purposes we note that the window is set to contain a constant number of symbols, with $W + M = 12$. Parameter M will be set according to the target

throughput. As for the sequence length, we will use $L = 8$ for M up to, and including, 3, while for higher values of M, we will use $L = 2^M$. This is done to ensure that the specified throughput is met.

For all cases, we assume two hypothetical decoders with full logic duplication. In practice, we would be able to use the forked decoder in all but the highest throughput scenarios, but since it requires a sufficiently large sequence length, we try not to impose too many restrictions with our assumptions. Duplicating the entire decoder logic is always a valid option, which is why it is assumed, albeit skewing the results against our method.

Three Snappy architectures are compared against their hypothetical counterparts of our method, and the results are presented in table 6.9. Evidently, the double-decoder decompressor of our novel method consumes one to two orders of magnitude fewer LUTs, compared to Snappy decompressors. Usage of FFs is also an order of magnitude lower, while the number of BRAMs is still less than the other decompressors, albeit more comparable. However, we do not place too much emphasis on the BRAM counts for our method, as they can vary significantly depending on the sequence length and dataset.

A similar examination is presented in table 6.10 for Deflate compression variants. Again, we observe an order of magnitude difference in our architecture's LUT and FF utilization compared to those of Deflate. The other architectures use fewer BRAMs than ours, except for [114].

TABLE 6.9: Comparison with Snappy decompressor architectures

	Literature				This thesis			
Design	**LUTs**	**FFs**	**BRAMs**	$\mathcal{T}$	**LUTs**	**FFs**	**BRAMs**	$\mathcal{T}$
[109]	15.3K	16.5K	48	3.3	1.1K	0.8K	18	4
[110]	91K	8.9K	32	15	1.4K	1.1K	18	16
[111]	56K		50	30.9	2.8K	2.0K	18	32

TABLE 6.10: Comparison with Deflate decompressor architectures

	Literature				This thesis			
Design	**LUTs**	**FFs**	**BRAMs**	$\mathcal{T}$	**LUTs**	**FFs**	**BRAMs**	$\mathcal{T}$
[112]	4.3K		9	3	1.1K	0.8K	18	4
[113]	15.7K	9.1K	15	2.2	1.1K	0.8K	18	4
[114]	308K	196K	128	62.4	10.2K	5.9K	18	64

We note that each of the compared architectures was implemented in UltraScale FPGAs. Each design's resource cost was condensed into the three resource metrics shown in tables 6.9 and 6.10. For designs that used distributed RAM, its cost was combined with the LUT cost. For designs that used UltraRAMs as well as BRAMs, the BRAM-only equivalent of the combined memory capacity is reported. In every case, BRAM utilization refers to 36Kb BRAM tiles.

In closing, it should not be overlooked that [109] is part of Xilinx's Vitis data compression library, while [112] is a proprietary IP core sold by CAST Inc. Therefore, it comes as no surprise these are the most area efficient architectures in their respective category. The fact that our architecture's cost is so much lower compared to these specific works is humbly regarded by the author as a testament to the quality of the work developed in this thesis.

Chapter 7

Conclusions and Future Work

This chapter sums up and evaluates this thesis' work and provides directions for future research.

7.1 Conclusions

Over the last decades, unprecedented advances in machine learning and neural networks have expanded the realm of what is feasible and produced ever larger and more capable deep neural networks. As model size increases rapidly, it outpaces the available memory bandwidth on parallel computing platforms such as FPGAs, thus bottlenecking inference scalability. Moreover, the increased data traffic from power-hungry external memories, where network weights are typically stored, increases the total memory consumption of inference accelerators.

This thesis aimed to develop a compression method suitable for statically compressing quantized weight matrices. Our method operates on constant-length sequences, which are mapped onto a tree in an overlapping fashion. By exploiting the tree's structure, we organize it into penalty regions, each of which requires a different number of bits for the encoded sequence codes. The tree structure and the encoding scheme were constructed to be easily and mathematically decodable, to facilitate high and selectable decompression throughputs. Testing the method on the sparse GRU weight matrices of DeepSpeech2 reveals that the compression ratio is adequate, despite it being traded off for throughput.

The decompressor was designed with UltraScale FPGAs in mind and its architecture was analyzed to create a generalized model for the resource cost

of its implementation. When compared against various LZ77-based decompressors in literature, our decompressor is calculated to consume, on average, an order of magnitude fewer logic resources, while being capable of delivering the same or higher throughput. The fact that our design uses a fraction of the logic resources used by Xilinx or CAST decompressors is an important achievement for this thesis, and also highlights our novel compression method as promising and warranting further investigation.

7.2 Future Work

The compression method developed in this thesis, along with the architecture design of its FPGA-accelerated decompressor, are the first exploratory steps into a novel and exciting method. We have only scratched the surface of what may be possible with overlapping sequence compression and this work proves the method is, at the very least, worthy of further research. Some ideas for future extensions and further evaluations are listed below.

- Our architecture can be slightly adapted to provide nonfluctuating decompression throughput. In general, the last iteration of a decompression cycle provides lower momentary throughput. In certain scenarios, these fluctuations can hinder pipelines depending on a constant stream of data, such as neural network inference engines. By merging the last iteration of one decompression cycle with the first iteration of the next cycle, throughput would be increased and its fluctuations eliminated.
- A parallel decompressor architecture which works on the sequence level can be developed, decompressing a whole sequence per cycle. Despite our method's inherent capability for symbol-level parallel decompression, sequence-level parallel decompression could provide the throughput of full sequences per cycle, at the exact cost .
- The method can be extended to compress variable length sequences, thus achieving comparable compression ratios, albeit for reduced tree sizes. A significant issue of constant-length sequences is the exponential explosion of the sequence alphabet size as a function of the symbol alphabet size. A scheme where each penalty group compresses sequences of predefined length would drastically cut down on the tree size and, thus, BRAM utilization.

- To alleviate the BRAM utilization cost, an entirely different approach to symbol retrieval is worthy of consideration. For example, we could express node accesses and their returned symbols as a set membership test problem, wherein each full symbol address is checked against the set of symbol addresses belonging to each quantization level to determine the matching symbol. This way, by implicitly representing the tree via probabilistic filters such as Bloom or Cuckoo, BRAM utilization could be lowered.
- A generic decompressor IP core for our method, targeting UltraScale FPGAs, can be developed with Xilinx tools, and the actual resource utilization of various configurations can be compared against our cost model. Aside from ascertaining the accuracy of our resource-cost model, valuable insights would be gained into the maximum achievable clock speeds and the method's energy consumption.
- A cost-benefit analysis should be carried out on integrating weight decompressors into inference accelerators. The GRUs of DeepSpeech2 can be implemented in hardware, with and without weight compression, using our and other compression methods, to determine the real-world effect of weight compression on inference performance.

References

[1] A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229.

[4] K. Hornik, M. Stinchcombe, and H. White. “Multilayer Feedforward Networks Are Universal Approximators”. In: *Neural Netw.* 2.5 (July 1989), pp. 359–366. ISSN: 0893-6080.

[5] H. Sak, Andrew Senior, and F. Beaufays. “Long short-term memory recurrent neural network architectures for large scale acoustic modeling”. In: *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH* (Jan. 2014), pp. 338–342.

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to Sequence Learning with Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014.

[7] Ruben Villegas et al. “High Fidelity Video Prediction with Large Stochastic Recurrent Neural Networks”. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[9] Hasim Sak et al. “Learning acoustic frame labeling for speech recognition with recurrent neural networks”. In: *ICASSP*. 2015, pp. 4280–4284.

[10] S. Hochreiter et al. “Gradient flow in recurrent nets: the difficulty of learning long-term dependencies”. In: *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

[11] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667.

[12] Kyunghyun Cho et al. “On the Properties of Neural Machine Translation: Encoder–Decoder Approaches”. In: *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*.

Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 103–111.

[13] Junyoung Chung et al. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling”. In: (Dec. 2014).

[14] Mike Schuster and Kuldip Paliwal. “Bidirectional recurrent neural networks”. In: *Signal Processing, IEEE Transactions on* 45 (Dec. 1997), pp. 2673–2681.

[15] Michael C. Mozer. “A Focused Backpropagation Algorithm for Temporal Pattern Recognition”. In: *Complex Syst.* 3 (1989).

[16] A. J. Robinson and Frank Fallside. *The Utility Driven Dynamic Error Propagation Network*. Tech. rep. CUED/F-INFENG/TR.1. Cambridge, UK: Engineering Department, Cambridge University, 1987.

[17] Paul J. Werbos. “Generalization of backpropagation with application to a recurrent gas market model”. In: *Neural Networks* 1.4 (1988), pp. 339–356. ISSN: 0893-6080.

[18] Timothy P. Lillicrap et al. “Random feedback weights support learning in deep neural networks”. In: *ArXiv* abs/1411.0247 (2014).

[19] Arild Nøkland. “Direct Feedback Alignment Provides Learning in Deep Neural Networks”. In: (Sept. 2016).

[20] B.T. Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17. ISSN: 0041-5553.

[21] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2015).

[22] Kenneth Levenberg. “A METHOD FOR THE SOLUTION OF CERTAIN NON – LINEAR PROBLEMS IN LEAST SQUARES”. In: *Quarterly of Applied Mathematics* 2 (1944), pp. 164–168.

[23] Alex Graves et al. “Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks”. In: vol. 2006. Jan. 2006, pp. 369–376.

[24] Kyuyeon Hwang and Wonyong Sung. “Character-level incremental speech recognition with recurrent neural networks”. In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2016, pp. 5335–5339.

[25] Harald Scheidl, Stefan Fiel, and Robert Sablatnig. “Word Beam Search: A Connectionist Temporal Classification Decoding Algorithm”. In: *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*. 2018, pp. 253–258.

[26] StÉphane Boucheron, AurÉlien Garivier, and Elisabeth Gassiat. "Coding on Countably Infinite Alphabets". In: *IEEE Transactions on Information Theory* 55.1 (2009), pp. 358–373.

[27] D. Sculley and Carla Brodley. "Compression and Machine Learning: A New Perspective on Feature Space Vectors". In: Apr. 2006, pp. 332–341. ISBN: 0-7695-2545-8.

[28] Ian H. Witten et al. "Text mining: a new frontier for lossless compression". In: *Proceedings DCC'99 Data Compression Conference (Cat. No. PR00096)* (1999), pp. 198–207.

[29] Eibe Frank, Chang Chui, and Ian H. Witten. "Text Categorization Using Compression Models". In: *In Proceedings of DCC-00, IEEE Data Compression Conference, Snowbird, US*. IEEE Computer Society Press, 2000, pp. 200–209.

[30] J. Hagenauer et al. "Genomic analysis using methods from information theory". In: *Information Theory Workshop*. 2004, pp. 55–59.

[31] Marcus Hutter. "Towards a Universal Theory of Artificial Intelligence Based on Algorithmic Probability and Sequential Decisions". In: vol. 2167. Sept. 2001, pp. 226–238. ISBN: 978-3-540-42536-6.

[32] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (July 1948), pp. 379–423. ISSN: 0005-8580.

[33] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (Sept. 1952), pp. 1098–1101. ISSN: 2162-6634.

[34] J. J. Rissanen. "Generalized Kraft Inequality and Arithmetic Coding". In: *IBM Journal of Research and Development* 20.3 (1976), pp. 198–203.

[35] J. Rissanen and G. G. Langdon. "Arithmetic Coding". In: *IBM Journal of Research and Development* 23.2 (1979), pp. 149–162.

[36] Ian H. Witten, Radford M. Neal, and John G. Cleary. "Arithmetic Coding for Data Compression". In: *Commun. ACM* 30.6 (June 1987), pp. 520–540. ISSN: 0001-0782.

[37] Alistair Moffat, Radford M. Neal, and Ian H. Witten. "Arithmetic Coding Revisited". In: *ACM Trans. Inf. Syst.* 16.3 (July 1998), pp. 256–294. ISSN: 1046-8188.

[38] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343.

[39] J. Ziv and A. Lempel. "Compression of individual sequences via variable-rate coding". In: *IEEE Transactions on Information Theory* 24.5 (1978), pp. 530–536.

[40] Welch. "A Technique for High-Performance Data Compression". In: *Computer* 17.6 (1984), pp. 8–19.

[41] James A. Storer and Thomas G. Szymanski. "Data Compression via Textual Substitution". In: *J. ACM* 29.4 (Oct. 1982), pp. 928–951. ISSN: 0004-5411.

[42] H. Sakoe and S. Chiba. "Dynamic programming algorithm optimization for spoken word recognition". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26.1 (1978), pp. 43–49.

[43] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi. "An introduction to the application of the theory of probabilistic functions of a Markov process to automatic speech recognition". In: *The Bell System Technical Journal* 62.4 (1983), pp. 1035–1074.

[44] A. Waibel et al. "Phoneme recognition using time-delay neural networks". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37.3 (1989), pp. 328–339.

[45] Jianxiong Wu and Chorkin Chan. "Isolated word recognition by neural network models with cross-correlation coefficients for speech dynamics". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15.11 (1993), pp. 1174–1185.

[46] Alex Graves and Navdeep Jaitly. "Towards End-To-End Speech Recognition with Recurrent Neural Networks". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 2. Bejing, China: PMLR, June 2014, pp. 1764–1772.

[47] Vassil Panayotov et al. "Librispeech: An ASR corpus based on public domain audio books". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2015, pp. 5206–5210.

[48] R. Ardila et al. "Common Voice: A Massively-Multilingual Speech Corpus". In: *Proceedings of the 12th Conference on Language Resources and Evaluation (LREC 2020)*. 2020, pp. 4211–4215.

[50] Dario Amodei et al. "Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin". In: *ArXiv* abs/1512.02595 (2016).

[51] Dzmitry Bahdanau, Kyunghyun Cho, and Y. Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *ArXiv* 1409 (Sept. 2014).

[52] Thang Luong, Hieu Pham, and Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 1412–1421.

[53] William Chan et al. "Listen, attend and spell: A neural network for large vocabulary conversational speech recognition". In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2016, pp. 4960–4964.

[54] Martin Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *CoRR* abs/1603.04467 (2016). arXiv: 1603.04467.

[65] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *CoRR* abs/1704.04760 (2017). arXiv: 1704.04760.

[66] James O' Neill. "An Overview of Neural Network Compression". In: (2020).

[67] Rahul Mishra, Hari Prabhat Gupta, and Tanima Dutta. "A Survey on Deep Neural Network Compression: Challenges, Overview, and Solutions". In: (2020).

[68] Lucas Liebenwein et al. "Lost in Pruning: The Effects of Pruning Neural Networks beyond Test Accuracy". In: *ArXiv* abs/2103.03014 (2021).

[69] Michael Zhu and Suyog Gupta. "To prune, or not to prune: exploring the efficacy of pruning for model compression". In: *ArXiv* abs/1710.01878 (2018).

[70] Cenk Baykal et al. "SiPPing Neural Networks: Sensitivity-informed Provable Pruning of Neural Networks". In: *ArXiv* abs/1910.05422 (2019).

[71] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. "Distilling the Knowledge in a Neural Network". In: *ArXiv* abs/1503.02531 (2015).

[72] Yihui He, Xiangyu Zhang, and Jian Sun. "Channel Pruning for Accelerating Very Deep Neural Networks". In: *2017 IEEE International Conference on Computer Vision (ICCV)* (2017), pp. 1398–1406.

[73] Xiaolong Ma et al. "Non-Structured DNN Weight Pruning–Is It Beneficial in Any Platform?" In: *IEEE Transactions on Neural Networks and Learning Systems* PP (Mar. 2021), pp. 1–15.

[74] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. "ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression". In: *IEEE*

International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017. IEEE Computer Society, 2017, pp. 5068–5076.

[75] Jagmohan Chauhan et al. "Performance Characterization of Deep Learning Models for Breathing-Based Authentication on Resource-Constrained Devices". In: *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2.4 (Dec. 2018).

[76] M. Hagiwara. "Removal of hidden units and weights for back propagation networks". In: *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*. Vol. 1. 1993, 351–354 vol.1.

[77] Andreas S. Weigend, David E. Rumelhart, and Bernardo A. Huberman. "Generalization by Weight-Elimination with Application to Forecasting". In: *Proceedings of the 3rd International Conference on Neural Information Processing Systems*. NIPS'90. Denver, Colorado: Morgan Kaufmann Publishers Inc., 1990, pp. 875–882. ISBN: 1558601848.

[78] Yann Lecun, John Denker, and Sara Solla. "Optimal Brain Damage". In: vol. 2. Jan. 1989, pp. 598–605.

[79] Wei Wen et al. "Learning Structured Sparsity in Deep Neural Networks". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS'16. Barcelona, Spain: Curran Associates Inc., 2016, pp. 2082–2090. ISBN: 9781510838819.

[80] Jiecao Yu et al. "Scalpel: Customizing DNN pruning to the underlying hardware parallelism". In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017, pp. 548–560.

[81] Itay Hubara et al. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), pp. 6869–6898. ISSN: 1532-4435.

[82] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016.

[83] Tim Dettmers. "8-Bit Approximations for Parallelism in Deep Learning". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016.

[91] Aydin Buluç et al. "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks". In: *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms*

and Architectures. SPAA '09. Calgary, AB, Canada: Association for Computing Machinery, 2009, pp. 233–244. ISBN: 9781605586069.

[92] I. S. Duff, Roger G. Grimes, and John G. Lewis. "Sparse Matrix Test Problems". In: *ACM Trans. Math. Softw.* 15.1 (Mar. 1989), pp. 1–14. ISSN: 0098-3500.

[93] Kuo-Liang Chung and Yih-Kai Lin. "A novel memory-efficient Huffman decoding algorithm and its implementation". In: *Signal Processing* 62.2 (1997), pp. 207–213. ISSN: 0165-1684.

[94] Reza Hashemian. "Condensed Table of Huffman Coding, a New Approach to Efficient Decoding". In: *Communications, IEEE Transactions on* 52 (Feb. 2004), pp. 6–8.

[95] A. Moffat and A. Turpin. "On the implementation of minimum redundancy prefix codes". In: *IEEE Transactions on Communications* 45.10 (Oct. 1997), pp. 1200–1207. ISSN: 1558-0857.

[96] Ahsan Habib and Mohammad Shahidur Rahman. "Balancing decoding speed and memory usage for Huffman codes using quaternary tree". In: *Applied Informatics* 4.1 (Jan. 2017), p. 5. ISSN: 2196-0089.

[97] M. Aggarwal and A. Narayan. "Efficient Huffman decoding". In: *Proceedings 2000 International Conference on Image Processing (Cat. No.00CH37101)*. Vol. 1. Sept. 2000, 936–939 vol.1.

[98] Yih-Kai Lin and Kuo-Liang Chung. "A space-efficient Huffman decoding algorithm and its parallelism". In: *Theor. Comput. Sci.* 246 (Sept. 2000), pp. 227–238.

[99] Robert M. Fano and W. T. Wintringham. "Transmission of Information". In: *Physics Today* 14.12 (Jan. 1961), p. 56.

[100] Jonathan S. Turner. "Approximation algorithms for the shortest common superstring problem". In: *Information and Computation* 83.1 (1989), pp. 1–20. ISSN: 0890-5401.

[101] Yuming Zou and Paul E. Black. *perfect binary tree*. Nov. 2019.

[102] Stefan Andrei, Manfred Kudlek, and Radu Niculescu. "Some results on the Collatz problem". In: *Acta Inf.* 37 (Feb. 2000), pp. 145–160.

[103] Heinz Ebert. *A Graph Theoretical Approach to the Collatz Problem*. 2019. arXiv: 1905.07575 [math.GM].

[106] Joonyoung Kim and Younsu Kim. "HBM: Memory solution for bandwidth-hungry processors". In: *2014 IEEE Hot Chips 26 Symposium (HCS)*. 2014, pp. 1–24.

[108] P. Deutsch. *RFC1951: DEFLATE Compressed Data Format Specification Version 1.3*. USA, 1996.

[110] Yang Qiao, Jian Fang, and H.P. Hofstee. "An FPGA-based Snappy Decompressor-Filter". PhD thesis. Jan. 2018.

[111] Jian Fang et al. "An Efficient High-Throughput LZ77-Based Decompressor in Reconfigurable Logic". In: *Journal of Signal Processing Systems* 92.9 (Sept. 2020), pp. 931–947. ISSN: 1939-8115.

[113] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. "High-Throughput FPGA-Based Hardware Accelerators for Deflate Compression and Decompression Using High-Level Synthesis". In: *IEEE Access* 8 (2020), pp. 62207–62217.

[114] Ruihao Gao et al. "MetaZip: A High-Throughput and Efficient Accelerator for DEFLATE". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC '22. San Francisco, California: Association for Computing Machinery, 2022, pp. 319–324. ISBN: 9781450391429.

External Links

[2] Andrej Karpathy. *Software 2.0*. Nov. 2017. URL: https://karpathy.medium.com/software-2-0-a64152b37c35.

[3] *Market Research Report*. Mar. 2022. URL: https://www.fortunebusinessinsights.com/machine-learning-market-102226.

[8] Hasim Sak et al. *Google voice search: faster and more accurate*. Sept. 2015. URL: https://ai.googleblog.com/2015/09/.

[49] *Mozilla Common Voice Dataset*. URL: https://commonvoice.mozilla.org/en/datasets.

[55] *Tensorflow - Wikipedia*. URL: https://en.wikipedia.org/wiki/TensorFlow.

[56] *Keras - Official website*. URL: https://keras.io/.

[57] *Keras - Wikipedia*. URL: https://en.wikipedia.org/wiki/Keras.

[58] *PyTorch - Official website*. URL: https://pytorch.org/.

[59] *PyTorch - Wikipedia*. URL: https://en.wikipedia.org/wiki/PyTorch.

[60] *AVX - Wikipedia*. URL: https://en.wikipedia.org/wiki/Advanced_Vector_Extensions.

[61] *SSE - Wikipedia*. URL: https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions.

[62] *CUDA - Official page*. URL: https://developer.nvidia.com/cuda-zone.

[63] *cuDNN - Official page*. URL: https://developer.nvidia.com/cudnn.

[64] *ROCm - Official page*. URL: https://www.amd.com/en/graphics/servers-solutions-rocm.

[84] *CHaiDNN - Guthub*. URL: https://github.com/Xilinx/CHaiDNN.

[85] *Vitis AI - Official page*. URL: https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html.

[86] *NVIDIA Deep Learning Accelerator - Official website*. URL: http://nvdla.org/.

[87] *PaddleSpeech - GitHub page*. URL: https://github.com/PaddlePaddle/PaddleSpeech.

[88] *NVIDIA's OpenSeq2Seq - GitHub page*. URL: https://github.com/NVIDIA/OpenSeq2Seq.

[89] *TensorFlow models - GitHub page*. URL: https://github.com/tensorflow/models/tree/master/research/deep_speech.

[90] *Baidu Research - PaddlePaddle blog post*. URL: http://research.baidu.com/Blog/index-view?id=126.

[104] *Xilinx UltraScale Architecture Configurable Logic Block User Guide*. URL: https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb.

[105] *Xilinx UltraScale Architecture Memory Resources User Guide*. URL: https://docs.xilinx.com/v/u/en-US/ug573-ultrascale-memory-resources.

[107] *Snappy compression library*. URL: http://google.github.io/snappy/.

[109] *Xilinx Snappy compression library*. URL: https://xilinx.github.io/Vitis_Libraries/data_compression/2022.1/source/L2/snappy.html.

[112] *CAST Inc. GUNZIP/ZLIB/Inflate decompression core*. URL: https://www.xilinx.com/products/intellectual-property/1-79drsh.html.