POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Weights Compression for Efficient Convolutional Neural Networks Acceleration on FPGA

Supervisors

Prof. Luciano Lavagno

Co-supervisor Giovanni Brignone

Co-supervisor Roberto Bosio

Co-supervisor Teodoro Urso

Candidate

Giovanni Cascone

Academic Year 2024/2025

Abstract

Convolutional Neural Networks (CNNs) have significantly advanced image recognition and computer vision. Their growing size and complexity are driven by the need for higher accuracy and the ability to tackle more complex tasks. **Larger networks** can learn richer and abstract features at various levels, enabling them to recognize not only basic patterns (e.g., edges and textures), but also more complex structures like objects and faces, even in challenging conditions like varying lighting or cluttered environments.

The CNNs are scalable, but bigger networks imply more memory and compute capacity. This often becomes a problem, mainly on **FPGAs**, which have stringent resource restrictions, and this work tries to tackle these constraints for effective implementation.

This thesis addresses the challenge of deploying large Convolutional Neural Networks, such as MobileNet or ResNet-50, on FPGAs, and to overcome the limited on-chip memory capacity, the network **weights** (which constitute the majority of the memory footprint) are first compressed offline using entropy-based techniques and stored in external **DDR**. The compressed weights are then transferred to the FPGA's on-chip BRAM and decompressed in hardware using a dedicated decompressor (implemented in **Vitis HLS**) before being fed directly to the convolutional layers for computation. This approach allows larger neural networks to fit within FPGAs that would otherwise support only smaller models. By reducing the memory footprint of the weights and the required bandwidth between the FPGA and external memory, the method significantly improves system scalability. However, the decompression process introduces challenges, particularly in terms of throughput, which can become a bottleneck for real-time applications.

Various compression techniques were explored from the literature, including pruning, weight clustering, low-rank factorization, arithmetic coding and **Huffman coding**, analyzing their pros and cons. Ultimately, encoding-based methods were chosen as they

provide decent compression ratios while being lossless, ensuring no accuracy loss after decompression. Specifically, **Gzip compression** was used, which employs the **Deflate algorithm** combining LZ77 and dynamic Huffman coding to balance efficiency and feasibility.

Initially, the approach was tested on smaller networks such as ResNet-8, where both compression and decompression were evaluated. This was done for a single layer, but it can be extended to all layers with appropriate parallelization. For larger networks, only the compression of already quantized weights (fixed-point 8-bit) was tested, compressing the weights layer by layer (considering only convolutional layers) and computing a weighted average of the compression factors to determine the overall compression ratio across the entire network. This resulted in an average total compression of approximately **30-35%**.

Future work could focus on improving throughput to further optimize the deployment of large-scale CNNs on FPGAs.

Acknowledgements

Vorrei ringraziare il Professor Lavagno e i Co-Relatori Brignone, Bosio e Urso, i quali mi hanno seguito in questo percorso di tesi che mi ha appassionato molto e mi ha fatto imparare tanto.

Ringrazio soprattutto i miei genitori, mio fratello, i miei nonni e mia zia per il sostegno e l'incoraggiamento che mi hanno sempre dato, per questo percorso ed in ogni momento della mia vita.

Ringrazio Salvatore, insieme abbiamo condiviso momenti di gioia, tante serate e belle esperienze. Un grazie di cuore anche a Giuseppe, Mario e Raffaele, insieme abbiamo condiviso tanti momenti di divertimento.

Vorrei ringraziare i miei amici conosciuti a Torino – Domenico, Yonas, Gabriele, Flavia, Chiara, Edoardo e Vasanth – per la loro compagnia e per aver reso questi due anni così speciali.

Ringrazio anche i miei amici siciliani Sebastiano, Emma, Giada e i miei cugini Damiano e Massimiliano, ci conosciamo da una vita e abbiamo passato una sacco di bei momenti assieme.

Table of Contents

List of Tables x

List of Figures xi

1 Introduction 1

Introduction 1

1.1 Background on CNNs . . . 1

1.1.1 CNN Architecture and Core Components . . . 2

1.1.2 Weights in Convolutional Layers . . . 3

1.2 CNN Architectures . . . 4

1.2.1 The ResNet Architecture . . . 4

1.2.2 MobileNet V1 . . . 6

1.2.3 MobileNet V2 . . . 7

2 The Growth of CNNs 8

2.1 Challenges and Drawbacks . . . 9

2.2 Thesis Goal . . . 11

3 Weight Compression techniques 12

3.1 Pruning . . . 12

3.1.1 Fundamental Pruning Process . . . 12

3.1.2 Factors Influencing Pruning Effectiveness . . . 13

3.1.3 Benefits of Pruning . . . 13

3.1.4 Advanced Pruning Strategies . . . 14

3.1.5 Post-Pruning Optimization and Compression . . . 15

3.2 Weight Clustering . . . 15
3.2.1 Key Concepts of Weight Clustering . . . 15
3.3 Low-Rank Factorization . . . 17
3.4 Entropy-Based Coding . . . 19
3.5 Arithmetic Coding for Weight Compression . . . 20
3.5.1 Principle of Arithmetic Coding . . . 20
3.5.2 Weight Compression with Arithmetic Coding . . . 22
3.5.3 Decoding with Arithmetic Coding . . . 22
3.6 Huffman Coding for Weight Compression . . . 22
3.6.1 Limitations . . . 23

4 Selected Method and Algorithm 25
4.1 How GZIP Compression/Decompression works . . . 26

5 Compression of Large Networks 28
5.1 Compression of the Largest Layer on ResNet-50 . . . 29

6 Implementation 30
6.1 Weights Compression . . . 31
6.2 Loading of Compressed Weights . . . 31
6.3 Loading of compressed Weights from Master AXI to BRAM . . . 32
6.4 Decompression and Streaming for Convolution . . . 33

7 Results on ResNet-8 35

8 Conclusion 36

A Source Codes 37
A.0.1 Code for Weights Compression . . . 37
A.0.2 Loading of Compressed Weights . . . 41
A.0.3 Top-level function ResNet-8 . . . 43
A.0.4 Memory management . . . 45
A.0.5 Weights from DDR to BRAM . . . 47
A.0.6 Decompression and Streaming for Convolution . . . 49

List of Acronyms

Acronym	Description
CNN	Convolutional Neural Network
FPGA	Field-Programmable Gate Array
LUT	Look-Up Table
FF	Flip-Flop
CLB	Configurable Logic Block
DSP	Digital Signal Processing
RAM	Random Access Memory
DRAM	Dynamic RAM
BRAM	Block-RAM
URAM	Ultra-RAM
DDR	Double Data Rate
HLS	High Level Synthesis
AXI	Advanced Extensible Interface
M_AXI	Master-AXI

List of Tables

4.1 Comparison of Weight Compression techniques 25

5.1 Compression factors for each Network 28
5.2 Compression of the Largest Layer in ResNet-50 29

7.1 Results on ResNet-8 35

List of Figures

1.1 Convolutional Neural Network [1] 3

1.2 Shortcut module from ResNet. Note that ReLU following last CONV layer in shortcut is after the addition [1] 5

3.1 Illustration of unstructured pruning and structured pruning. (a)Unstructured pruning, where synapses-unimportant connections, can be pruned in order to sparse the network. Neurons can also be pruned to achieve the same purpose. (b) A typical structured that prunes the filters. Channels of the generated feature maps are reduced accordingly [2] 15

3.2 Weight clustering by scalar quantization (top) and centroids fine-tuning (bottom) [3] 17

3.3 Matrix decomposition vs. tensor decomposition: (a) low-rank matrix decomposition (truncated SVD); (b) low-rank tensor decomposition (Tucker decomposition) [4] 18

3.4 An example of the arithmetic coding process, where the set of data [1.064, 0.395, 1.061, 0.704] is encoded [5] 21

3.5 Bit coding [5] 21

3.6 Example of Huffman tree and Encoding process [6] 23

4.1 Deflate Encoding and Decoding for a symbol sequence with 40 symbols [7] 27

Chapter 1

Introduction

1.1 Background on CNNs

Convolutional Neural Networks (CNNs) are a class of machine learning (ML) models that use specialized convolutional layers to extract features from data. Originally designed to process structured grid-like data such as 1D vectors, 2D matrices, and multidimensional tensors, CNNs have also been adapted for graph-based data processing. They are widely used in various applications, including image recognition, natural language processing, and audio signal analysis.

The fundamental operation in a CNN is **convolution**, a mathematical transformation that applies a weighted sum of input elements using a convolutional kernel (or filter). The objective of a CNN is to learn the optimal filters that maximize the extracted information from input data, which can then be used for predictive tasks. These filters are typically smaller than the input object and are applied over a sliding window across the entire input domain.

A typical CNN architecture consists of a **forward propagation** of the input data through a sequence of convolutional, activation, and pooling operations [1, 8].

1.1.1 CNN Architecture and Core Components

A **convolutional layer** applies a set of convolutional operators (filters) to the input tensor, producing a feature map. This operation can be viewed as a pattern-matching technique, where filters capture specific patterns in the data. In graph-based CNNs, convolutional operations aggregate information from a node and its neighbors using weighted sums.

In image processing, CNNs represent images as a stack of color channels (e.g., RGB), forming a 3D tensor. The network learns filters that detect fundamental features such as edges, textures, and higher-order patterns.

After convolution, a non-linear **activation function** is typically applied. These functions introduce non-linearity into the model, enabling it to learn complex relationships within the data. Common activation functions include the Rectified Linear Unit (ReLU).

Pooling layers reduce the dimensionality of feature maps, making the network more robust to minor input translations and distortions while also reducing computational cost. A commonly used pooling operation is *max-pooling*, which selects the maximum value in each pooling region.

A CNN typically consists of multiple convolutional blocks, with initial layers extracting simple features (such as edges) and deeper layers capturing complex patterns. The output of the final convolutional block is flattened and passed through **fully connected (dense) layers**, which serve as classifiers or predictors.

The learnable parameters of a CNN are primarily the **weights**, which determine how input features are mapped to outputs. CNNs use the **backpropagation** algorithm to adjust these weights by minimizing a loss function that quantifies the error between predictions and true labels. This optimization process is typically performed using gradient-based methods such as Stochastic Gradient Descent (SGD) and its variants (Adam, RMSprop) [1, 8].

Figure 1.1 illustrates the functioning and all the stages in CNNs:

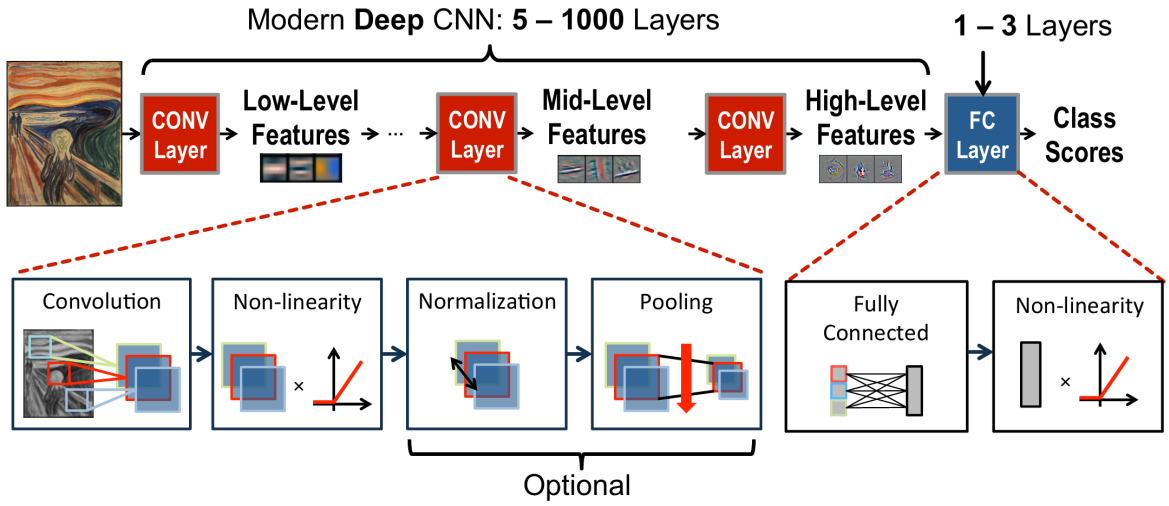


Figure 1.1: Convolutional Neural Network [1]

1.1.2 Weights in Convolutional Layers

In convolutional layers:

- Each **filter** contains a small set of learnable weights that are convolved over the entire input.
- Filters are designed to detect specific patterns, such as edges or textures.
- **Weight sharing** ensures that the same set of weights is used across all spatial locations, reducing the number of learnable parameters and making the network more translation-invariant.

In fully connected layers:

- Each **neuron** has a weight for every neuron in the previous layer.
- These weights determine the strength of connections between neurons.
- The number of weights grows significantly with the number of neurons, increasing the computational complexity of the model.

The choice of **weight initialization** affects training speed and stability: random initialization assigns small random values to weights, Xavier/Glorot initialization considers the number of input and output neurons to maintain variance stability, while He initialization is optimized for ReLU activation functions.

Large models with excessive weights risk **overfitting**, where they memorize training data instead of generalizing to new data.

L1 and L2 regularization add penalties to the loss function based on weight magnitude, discouraging excessive weight values.

Dropout randomly disables neurons during training to force the model to learn robust features [1, 8].

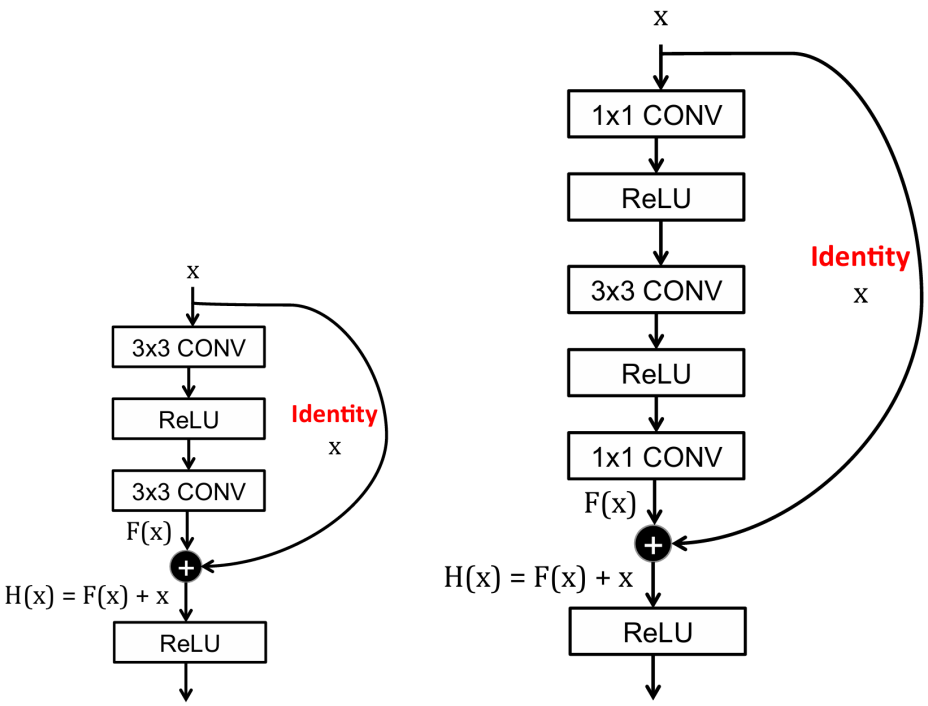
1.2 CNN Architectures

Since the compression results and tests were conducted on architectures like MobileNet and ResNet, it is useful to provide an overview of these models and understand their underlying mechanisms.

1.2.1 The ResNet Architecture

Residual Network (ResNet) employs **residual connections** to achieve greater depth—often exceeding 34 layers. ResNet was a groundbreaking entry in the ImageNet Challenge, being the first deep neural network to surpass human-level accuracy with a top-5 error rate of less than 5%.

A significant challenge in training deep networks, such as ResNet, is the **vanishing gradient** problem. As the error is backpropagated through the layers, the gradients can shrink, hindering the ability to update the weights in earlier layers effectively. To address this issue, ResNet introduces a unique **shortcut module** that incorporates identity connections, allowing the output from the earlier layers to bypass one or more weight layers (i.e., CONV layers) as shown in the figure 1.2:



(a) Without bottleneck (b) With bottleneck

Figure 1.2: Shortcut module from ResNet. Note that ReLU following last CONV layer in shortcut is after the addition [1]

This **"skip connection"** enables the network to learn residual mappings rather than the original unreferenced function, represented mathematically as:

$$F(x) = H(x) + x \tag{1.1}$$

In this formulation, $H(x)$ represents the residual mapping that the network aims to learn, while x is the input to the weight layers. Initially, when training begins, the residual function $F(x)$ can be set to zero, resulting in the identity connection being utilized. Gradually, as training progresses, the forward connection through the weight layer becomes more prominent, allowing the model to learn richer representations. This approach is somewhat analogous to the mechanisms found in Long Short-Term Memory (LSTM) networks used for sequential data, where information can persist through time steps.

Additionally, ResNet employs a **"bottleneck"** architecture to reduce the number of weight parameters effectively. Instead of using a straightforward stack of convolutional

layers, the architecture uses 1x1 filters to compress the dimensionality of the feature maps before applying a 3x3 convolution, followed again by 1x1 filters to restore the dimensionality. This leads to a more efficient model that significantly reduces computational complexity while maintaining performance [1].

1.2.2 MobileNet V1

MobileNet V1 represents a class of efficient convolutional neural networks designed specifically for mobile and embedded vision applications. The main goal was to create lightweight models with low latency while maintaining reasonable accuracy. The key innovation of MobileNet V1 is the introduction of **depthwise separable convolutions**. These convolutions factorize a standard convolution into two lighter operations:

- A **depthwise convolution**, which applies a single spatial filter to each input channel.
- A **pointwise convolution** (a 1x1 convolution), which linearly combines the outputs of the depthwise convolution to create new features.

This decomposition significantly reduces computational cost and the number of parameters compared to standard convolutions. MobileNet V1 introduces two global hyperparameters to adjust the model size and speed based on the specific constraints of the application:

- **Width Multiplier** (α): This multiplier uniformly scales the number of channels in each layer of the network. A value of $\alpha < 1$ creates "thinner" models, with an approximate reduction in computational cost and parameters of α^2. Typical values are 1, 0.75, 0.5, and 0.25.
- **Resolution Multiplier** (ρ): This multiplier reduces the spatial resolution of the input and all internal network representations. In practice, it is set by choosing a lower input resolution (e.g., 224, 192, 160, 128). The computational cost reduction is proportional to ρ^2.

The architecture of MobileNet V1 primarily consists of a sequence of depthwise separable convolutional layers, preceded by a standard convolution in the first layer. Each convolutional layer (both depthwise and pointwise) is followed by Batch Normalization (BN) and a ReLU non-linearity. Spatial downsampling is performed via stride convolutions in the depthwise layers and the first layer. The network ends with an average pooling layer and a fully connected layer for classification.

Experiments demonstrate that MobileNet V1 achieves a good balance between accuracy, model size, and computational speed on ImageNet and in various applications such as object detection, fine-grained classification, and face attribute analysis. The use of depthwise separable convolutions significantly reduces the number of operations and parameters with minimal accuracy loss [9].

1.2.3 MobileNet V2

MobileNet V2 was later introduced to further improve the performance of MobileNet V1 while maintaining computational efficiency. The main innovations in MobileNet V2 include:

Unlike traditional residual blocks, which have thinner bottleneck layers in the intermediate stages, the **inverted residual blocks** in MobileNet V2 first expand the number of channels using a 1x1 convolution, apply a depthwise convolution, and then project back to a lower number of channels using another linear (without ReLU) 1x1 convolution. This design allows the network to work with richer representations and better leverage non-linearity.

Using a linear activation in the last 1x1 layer of the inverted residual block is crucial. It has been observed that applying non-linearity after a significant channel reduction can result in excessive information loss. The **linear bottleneck** aims to preserve information.

Thanks to these modifications, MobileNet V2 generally achieves higher accuracy than MobileNet V1 with a similar or lower number of parameters and computational operations [9].

Chapter 2

The Growth of CNNs

CNNs have become increasingly larger (deeper, with more layers and parameters) for several reasons, leading to significant advantages:

Greater Availability of Data: The exponential growth of data in various fields allows for the training of more complex and extensive CNN models. Larger networks can better leverage large amounts of data to learn intricate patterns.

Increased Computational Power: GPUs and specialized hardware accelerators enable the training and execution of very deep CNN architectures that would have been impractical in the past. This increased computational power is essential for handling the complexity of larger networks.

Algorithmic Innovations: Developments like residual connections (ResNet) and Inception modules (GoogLeNet) have overcome difficulties in training very deep networks, such as the vanishing gradient problem. These innovations allow for the construction of larger and more effective networks. Increasing the depth of the network tends to provide higher accuracy. Larger networks, with more layers, can learn more hierarchical and complex representations of data.

Greater Capacity to Learn Complex Functions: Larger neural networks, with a greater number of parameters, have a higher capacity to approximate complex nonlinear functions that map inputs to outputs. This is especially crucial for challenging tasks that require a nuanced understanding of the data.

Learning More Abstract and Invariant Features: Deeper layers in a CNN can learn higher-level and more abstract features, which are often more invariant to variations

in the input (such as changes in position, scale, lighting in images). This ability to abstract improves the robustness of the model.

Better Performance on Complex Tasks: Tasks that require a deep semantic understanding of data (e.g., object recognition in complex contexts, visual natural language understanding) often benefit greatly from the representational capacity of larger networks. The increased depth allows the model to break down complex problems into a hierarchy of simpler subproblems.

Potential for More Effective Transfer Learning: Very large pre-trained networks on massive datasets (such as ImageNet) learn a wide range of general features. These networks can then be effectively used for transfer learning on more specific tasks with smaller datasets. The richness of features learned in larger networks can lead to better starting points for training on new tasks.

In summary, larger CNNs, made possible by data, computational power, and algorithmic innovations, offer the advantages of greater accuracy, increased learning capacity, learning more abstract features, better performance on complex tasks, and facilitate more effective transfer learning [1, 8].

2.1 Challenges and Drawbacks

Larger Convolutional Neural Networks, while offering benefits in terms of accuracy and feature representation, also come with several significant drawbacks:

High Memory Consumption:

Larger networks have a greater number of weights. This high number of parameters requires a large amount of memory to store, both during training and inference.

Also, fully-connected (FC) layers require a significant amount of storage. Larger networks often have more or larger FC layers, further increasing memory consumption.

Training has greater storage requirements than inference, as intermediate outputs of the network must be stored for backpropagation. Larger networks generate more intermediate outputs, exacerbating the issue.

The large amount of memory required can make it difficult to implement large-scale models on devices with limited resources, such as embedded or mobile devices. Additionally, it

demands more GPU or RAM memory for training, potentially increasing hardware costs.

Higher Computational Cost:

DNNs have high computational complexity. Training CNNs, in particular, requires performing a large number of convolution operations.

Larger networks have more layers and more filters per layer, resulting in a significantly larger number of multiply-accumulate operations (MACs).

The process of learning operators (filters) in CNNs can be computationally expensive, requiring sophisticated optimization algorithms. Larger networks have more parameters to optimize, increasing the computational cost of training.

The high computational cost leads to longer training times, which can take hours or even days, even when using powerful hardware. It also requires greater computational power, resulting in higher energy costs and potential cooling issues in data centers. Inference with very large networks can also require substantial computational capacity to achieve real-time results.

Overfitting Risk:

One of the biggest drawbacks of DNNs is their need for large datasets to prevent overfitting during training. Larger networks, having a greater number of parameters, have a higher capacity to "memorize" training data, including noise and specificities of the training dataset that do not generalize well to new data.

Overfitting occurs when a model learns the training data too well, achieving high performance on it but performing poorly on unseen data. Larger networks, with their increased capacity, are more susceptible to overfitting, especially when the amount of training data is not sufficiently large compared to the number of parameters in the model.

To mitigate overfitting in large networks, regularization techniques such as dropout, weight decay, and batch normalization are often required. However, these techniques can add further complexity to the training process and may not always fully resolve the issue, especially when the dataset is small relative to the model's capacity.

In summary, while larger CNNs can offer increased accuracy and the ability to learn complex features, they suffer from high memory consumption, higher computational costs during both training and inference, and a greater risk of overfitting, especially when the training data is limited. These disadvantages pose challenges for deploying very large mod-

els in resource-constrained environments and necessitate careful consideration of model size and complexity relative to the available data and computational resources [1, 8].

2.2 Thesis Goal

The objective of this thesis is to **reduce memory consumption** by **compressing the weights** of CNNs, thereby enabling larger networks to fit within the same chip. By applying advanced weight compression techniques, the goal is to maintain or even improve the model's accuracy while significantly reducing its memory footprint. This allows for the deployment of more complex and deeper models on resource-constrained devices, such as embedded systems or mobile platforms, without compromising performance.

Chapter 3

Weight Compression techniques

Research has been conducted on weight compression techniques available in the literature, and below is a description of the most important and interesting methods that are applicable to our case.

3.1 Pruning

Neural network pruning is a technique employed to reduce the storage and computational requirements of deep neural networks without significantly affecting their accuracy. This is crucial for deploying these computationally and memory-intensive models on embedded systems with limited resources. The underlying issue is that deep neural networks often have a large number of **redundant parameters**, leading to inefficiencies [3, 10, 11].

3.1.1 Fundamental Pruning Process

The fundamental pruning process typically involves a three-step method:

1. **Train Connectivity:** The network is initially trained using standard methods to learn which connections (weights) are important for its task. The focus here is on identifying significant connections rather than achieving final weight values.

2. **Prune Connections:** After training, connections with weights below a certain threshold are removed from the network. This transforms a dense, fully connected

network into a sparse network, effectively learning the network topology. The threshold can be determined based on various criteria, such as a quality parameter multiplied by the standard deviation of the weights in a layer.

3. **Train Weights:** The resulting sparse network is then retrained to fine-tune the weights of the remaining connections. This step is critical because using the pruned network without retraining leads to a significant drop in accuracy. Retraining allows the remaining connections to compensate for the removed ones.

This pruning and retraining cycle can be repeated iteratively to achieve even greater reductions in network complexity. Iterative pruning can lead to higher compression rates compared to a single aggressive pruning step [3, 10, 11].

3.1.2 Factors Influencing Pruning Effectiveness

The effectiveness of pruning is influenced by several factors:

- **Regularization:** Using L2 regularization during the initial training generally yields the best pruning results, leading to better accuracy after retraining compared to L1 regularization. L1 regularization tends to push more parameters towards zero initially, which is beneficial before retraining but not as optimal afterward.
- **Dropout:** Since pruning reduces the model's capacity, the dropout rate may need to be adjusted during retraining, often being reduced proportionally to the decrease in connections.
- **Layer Sensitivity:** Different layers in a neural network exhibit varying sensitivity to pruning. Convolutional layers are typically more sensitive than fully connected layers, with the first convolutional layer often being the most sensitive. This layer-specific sensitivity can guide the pruning thresholds applied to each layer.

3.1.3 Benefits of Pruning

Pruning offers significant benefits:

- **Reduced Number of Parameters:** Studies on ImageNet have shown remarkable reductions in parameters for various networks like AlexNet or ResNet-50.
- **Lower Computational Cost:** Pruning reduces the number of floating-point operations (FLOPs) required for inference, leading to faster computation.
- **Improved Energy Efficiency:** Memory access, especially to off-chip DRAM, is a major energy consumer. By reducing model size, pruning can enable storing weights on-chip (SRAM), which is significantly more energy-efficient.
- **Facilitated Deployment on Mobile Devices:** Smaller model sizes and reduced computational demands make it more feasible to deploy complex neural networks on resource-constrained mobile devices.

3.1.4 Advanced Pruning Strategies

Beyond magnitude-based pruning, other pruning strategies exist [3, 10, 11]:

- **Structured Pruning:** Instead of individual connections, structured pruning removes entire filters, channels, or neurons. This results in more regular weight matrices, which can be advantageous for hardware implementations. Techniques like Alternating Direction Method of Multipliers (ADMM) are used to achieve structured sparsity.
- **Importance-Based Pruning:** Methods like Optimal Brain Damage (OBD) and Optimal Brain Surgeon (OBS) use the Hessian of the loss function to identify and remove the least important connections. While potentially more accurate, they can be computationally expensive.
- **Neuron Agglomerative Clustering (NAC):** This method groups and merges similar neurons within the same layer based on the similarity of their weights and biases, reducing redundancy without introducing sparsity. Agglomerative clustering algorithms are used to determine neuron similarity.

Figure 3.1 shows how structured and unstructured pruning work:

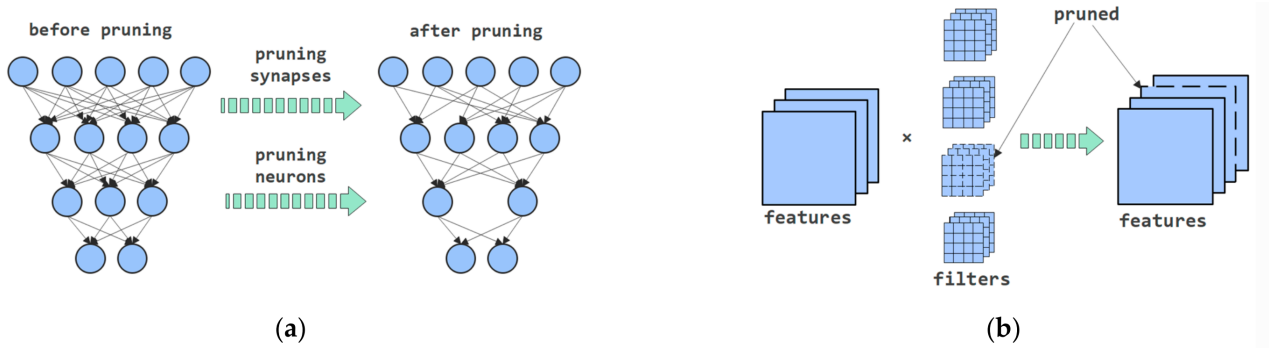


Figure 3.1: Illustration of unstructured pruning and structured pruning. (a)Unstructured pruning, where synapses-unimportant connections, can be pruned in order to sparse the network. Neurons can also be pruned to achieve the same purpose. (b) A typical structured that prunes the filters. Channels of the generated feature maps are reduced accordingly [2]

3.1.5 Post-Pruning Optimization and Compression

After pruning, post-processing techniques can further enhance compression and accuracy. Network Purification and Unused Path Removal (P-RM), for example, optimize structuredly pruned models after ADMM by removing redundant weights and unused paths.

Pruning is often used in conjunction with other compression techniques like weight quantization and Huffman coding to achieve even higher compression rates: this combined approach can significantly reduce the storage requirements of neural networks.

3.2 Weight Clustering

Weight clustering is a deep neural network compression technique that aims to reduce parameter (weight) redundancy by grouping similar weights into **clusters** and representing them with a smaller number of distinct values, typically the cluster **centroids**.

3.2.1 Key Concepts of Weight Clustering

Use of K-Means Clustering: A common approach to weight clustering involves the k-means clustering algorithm. This algorithm groups weights into k clusters, where k is a

predefined number of clusters, ensuring that weights within each cluster are more similar to each other than to those in other clusters.

Centroids as Shared Weights: Once clustering is performed, the weights within each cluster are replaced by the centroid (mean) of the cluster. Instead of storing many individual weight values, only the centroids are stored, and each connection stores an index pointing to the shared weight table (the centroids).

Feed-Forward and Back-Propagation Phases: During the feed-forward phase, the effective weight of a connection is obtained by referencing its index in the shared weight table. During back-propagation, the gradient for each shared weight is computed and used to update the shared weight.

Centroid Initialization: The way centroids are initialized can influence the performance of the compressed model. Various initialization methods are mentioned in the sources, including linear initialization, which seems to perform better after clustering and fine-tuning.

Fine-Tuning: After the clustering phase, fine-tuning the network with clustered weights is often necessary to recover any accuracy loss due to compression.

Neuron Agglomerative Clustering (NAC):
It is important to note that another compression technique, Neuron Agglomerative Clustering (NAC), groups similar neurons within the same layer based on the similarity of their weights and biases, merging them into a cluster centroid. While this is not strictly "weight clustering" in the sense of grouping individual weight values, NAC is a related technique that exploits redundancy at a higher level (neurons) to compress the network.

The image 3.2 illustrates the process of weight clustering:

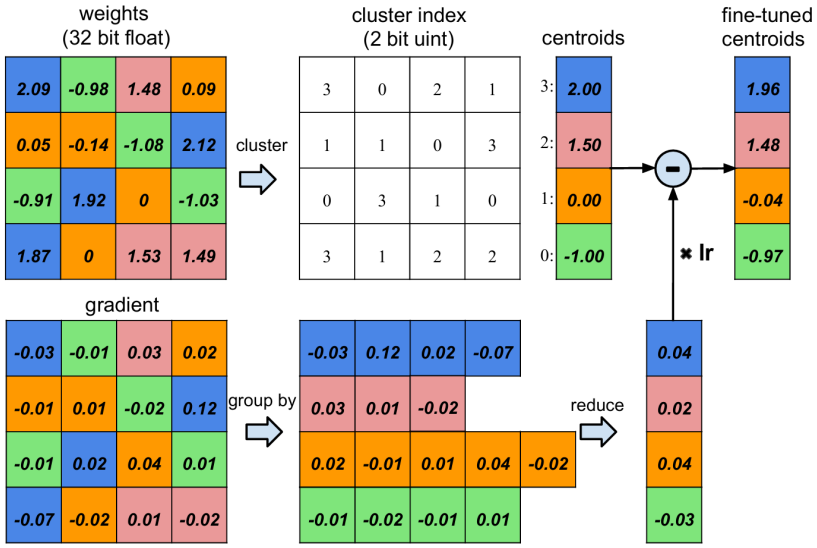


Figure 3.2: Weight clustering by scalar quantization (top) and centroids fine-tuning (bottom) [3]

In summary, weight clustering is an effective technique for reducing the size of neural network models by grouping similar weights and representing them with a smaller set of shared values (the cluster centroids). This leads to lower memory requirements and potentially improved computational efficiency [3, 12, 13].

3.3 Low-Rank Factorization

Low-rank factorization is a technique used to compress neural networks by reducing the redundancy in their weight tensors or matrices. This is achieved by decomposing the original high-rank weight parameters into a product of two or more smaller, **lower-rank matrices** or tensors. The rank of this factorization is a crucial factor, as a lower rank leads to a higher compression ratio but potentially a greater loss in model performance. Here are some key aspects and methods of low-rank factorization:

- **Singular Value Decomposition (SVD) and Truncated SVD (TSVD)**: SVD is a powerful matrix decomposition method that expresses a matrix as a product of three matrices: a left singular matrix (U), a diagonal singular value matrix (Σ), and a right singular matrix transpose (V^T). TSVD is an approximation obtained by keeping only the largest k singular values and their corresponding singular vectors, resulting in a lower-rank representation. TSVD can decompose a weight matrix

W into USV^T. It is considered one of the best methods for restoring the original matrix.

- **CUR Decomposition**: This method selects a few important rows (R) and columns (C) from the original matrix (W) and approximates it using a low-dimensional matrix (U) derived from their intersection, such that $W \approx CUR$. While potentially sub-optimal in approximation compared to TSVD, CUR decomposition has the advantage of preserving the original properties and interpretability of the selected rows and columns.

- **Tucker Decomposition (including Tucker-2)**: This is a higher-order tensor decomposition technique used to factorize weight tensors (especially convolutional kernels) into smaller core tensors and factor matrices. Tucker-2 is a specific format used in low-rank training for convolutional layers. For a convolutional layer $W \in \mathbb{R}^{C_{in} \times C_{out} \times K \times K}$, it can be represented in a Tucker-2 format.

Figure 3.3 shows how matrix and tensor decomposition work:

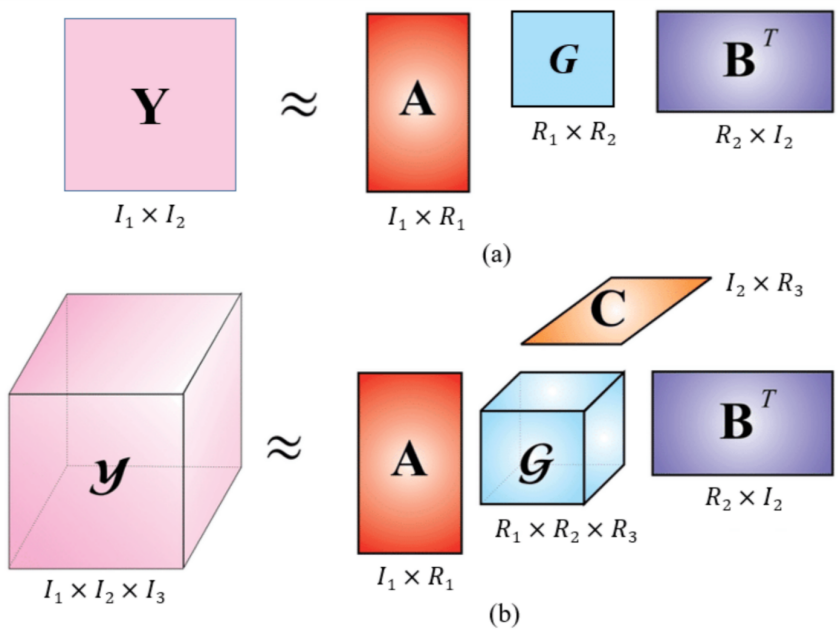


Figure 3.3: Matrix decomposition vs. tensor decomposition: (a) low-rank matrix decomposition (truncated SVD); (b) low-rank tensor decomposition (Tucker decomposition) [4]

Low-rank factorization can be applied to both the four-dimensional weight tensors of convolutional layers and the two-dimensional weight matrices of fully connected layers. For fully connected layers with non-high order, TSVD is often used for compression [14, 15].

Low-Rank Compression vs. Low-Rank Training:

- *Low-Rank Compression (post-training)* involves first training a full-rank model and then applying low-rank factorization to its weight parameters to obtain a compressed model, often followed by fine-tuning.
- *Low-Rank Training* aims to train compact, low-rank models directly from scratch by maintaining a low-rank structure throughout the training process. Methods like Efficient Low-Rank Training (ELRT) utilize low-tensor-rank formats and can incorporate techniques like orthogonality regularization to improve the performance of these directly trained compact models.

In summary, low-rank factorization encompasses various matrix and tensor decomposition techniques aimed at compressing neural networks by approximating their weight parameters with lower-rank representations. This can be done after training a full-rank model or by directly training a low-rank model from the outset, offering effective ways to achieve model compactness and efficiency.

3.4 Entropy-Based Coding

Entropy-based coding is a widely adopted data compression technique that utilizes **variable-length encoding** based on the probability of occurrence of a given symbol in a data stream. Depending on the symbol's probability, it assigns different code lengths for encoding the data stream. Two of the most commonly used entropy-based coding schemes are **Huffman coding** and **arithmetic coding**.

Huffman coding generates a binary **Huffman tree**, which represents the encoding for each symbol based on the probability of occurrence. To encode a symbol using Huffman coding, a tree traversal is performed starting from the root node until the desired symbol

is located in the tree. In contrast, arithmetic coding encodes the data by mapping a stream of symbols into the real number space [0, 1) [16].

3.5 Arithmetic Coding for Weight Compression

Arithmetic coding is an entropy-based data compression technique that maps a sequence of symbols (or data) into the real number space between 0 and 1. Unlike Huffman coding, which uses variable-length binary codewords for individual symbols, arithmetic coding represents an entire sequence of symbols as a single number [16].

3.5.1 Principle of Arithmetic Coding

Arithmetic coding [5] compresses a data sequence by subdividing the interval [0, 1) into smaller sub-ranges based on the probability of each symbol's occurrence. For a given sequence of symbols, the algorithm continuously narrows the range in [0, 1) that corresponds to the sequence by multiplying the current range by the probability of the next symbol. Each symbol's sub-range is determined by its **cumulative probability distribution**.

Figure 3.4 illustrates the encoding process. At the start, the entire range [0, 1) is assigned to the first symbol based on its probability. As more symbols are processed, the range is subdivided further, converging towards a specific point that represents the entire sequence. The final number chosen within the sub-range is converted to a binary representation, which is the compressed output.

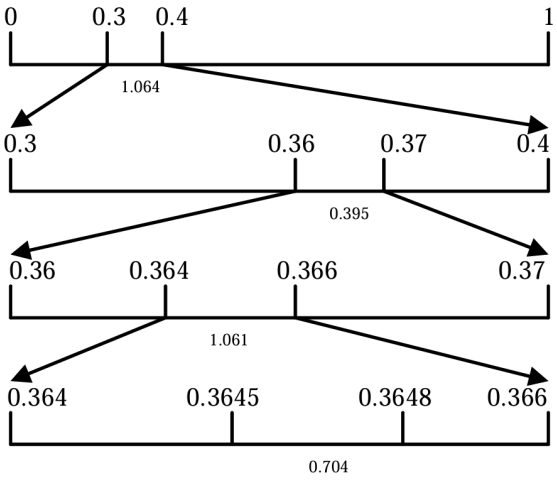


Figure 3.4: An example of the arithmetic coding process, where the set of data [1.064, 0.395, 1.061, 0.704] is encoded [5]

As shown in 3.5, in the final output probability interval, the left and right boundaries are converted to **binary**, and the final compression result is extracted from this interval, a process referred to as **Bit encoding**. The decoder is responsible for reconstructing the original data from the compressed result. To achieve this, it requires both the first character of the original data and the compression result as input. Using an iterative approach, the decoder restores the original data. The probability distribution plays a critical role in this process, as it determines the entropy, enhances the efficiency of the extracted interval, and significantly impacts compression performance. Adaptive arithmetic coding further improves this by generating a coding interval that is closer to the ideal.

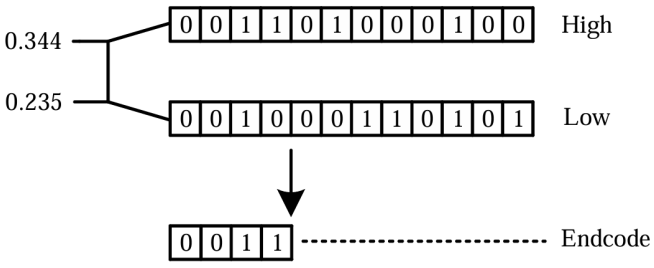


Figure 3.5: Bit coding [5]

3.5.2 Weight Compression with Arithmetic Coding

In the context of CNN inference on edge devices, Arithmetic coding is applied to further compress the quantized weight data.

The process starts with calculating the probability distribution of each weight value. This probability is used to guide the subdivision of the interval [0, 1), ensuring that more probable weight values are encoded with shorter bit sequences. The result is a **compressed bitstream** that represents the entire weight dataset.

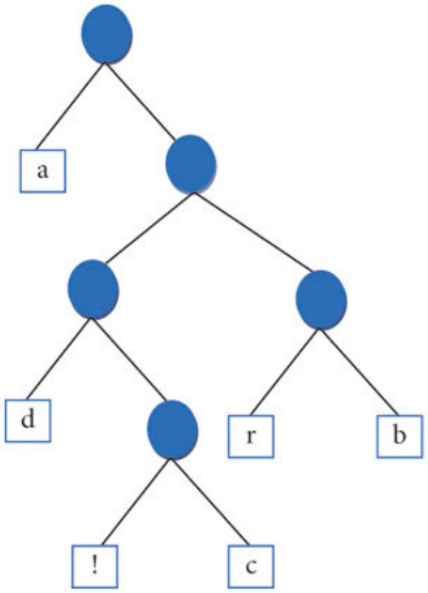
3.5.3 Decoding with Arithmetic Coding

To perform inference on the edge device, the compressed weights need to be decoded. The decoding process mirrors the encoding process, but in reverse. The decoder reads the compressed bitstream and progressively refines the range until the original sequence of weights is reconstructed. A hardware-based decoder, is implemented to ensure efficient, low-latency weight decoding during CNN inference, with minimal overhead in terms of power and time.

3.6 Huffman Coding for Weight Compression

The process of Huffman coding involves two main steps: building a Huffman tree and encoding the data. The algorithm begins by calculating the **frequency** of each symbol in the data. Symbols are then placed in a priority queue based on their frequency. The two symbols with the lowest frequencies are repeatedly extracted from the queue and merged into a new node, which becomes their parent in the **Huffman tree**. This process continues until a single tree is formed, where each leaf node represents a symbol from the original data.

The path from the root of the tree to each leaf determines the code for the corresponding symbol, with left branches typically representing a binary '0' and right branches representing a '1'. Once the tree is built, the data can be encoded by replacing each symbol with its corresponding variable-length binary code [17, 18].
Figure 3.6 illustrates the process of building a Huffman tree for a simple set of symbols.



Char	Encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

Figure 3.6: Example of Huffman tree and Encoding process [6]

Huffman coding is particularly effective when the weight distribution is non-uniform, meaning it has high variance. When some symbols (weights) appear much more frequently than others, Huffman coding can efficiently assign shorter codes to high-frequency weights and longer codes to less common ones, leading to better compression rates.

Huffman coding is a **lossless compression** technique, meaning that the original data can be perfectly reconstructed during decompression. Decompression is performed by traversing the Huffman tree from the root to a leaf node based on the sequence of binary bits in the encoded data. Since the encoding process ensures that no code is a prefix of another (prefix-free property), the decoding process is unambiguous and efficient. This property makes Huffman coding particularly effective in applications where exact reconstruction of the original data is required.

3.6.1 Limitations

While Huffman coding is efficient for many use cases, it has some limitations:

- **Fixed Length Issues**: Huffman coding is limited by the fact that it must encode each symbol with an integer number of bits, leading to inefficiencies for symbols with non-integer optimal bit lengths.

- **Suboptimal Compression Ratios**: As Huffman coding assigns each symbol a distinct code, the compression ratio may be suboptimal when the probability distribution of the symbols is highly skewed.

Chapter 4

Selected Method and Algorithm

This table compares the techniques based on **compression ratio** and **accuracy loss** founded in literature. Among all these techniques, **Huffman coding** was chosen because it is **lossless**, meaning there is no loss of information. Additionally, it avoids complications found in other methods, such as the irregular memory access of pruning, the precision loss of weight clustering, and the approximation errors of low-rank factorization. Moreover, Huffman coding provides good compression ratios for weights already quantized to 8-bit, which will be the ones targeted for our case.

	Compression ratio	**Accuracy loss**	**Network**
Pruning	$\sim 2x$–$3x$ (FP32)	$< 1\%$	ResNet-50
Weight Clustering (128 clusters)	$\sim 5x$ (FP32)	$\sim 1\%$	ResNet-18, ResNet-50
Low-Rank Factorization	$\sim 4x$ (FP32)	$\sim 1\%$	ResNet-32
Huffman Coding	$\sim 1.2x - 2x$ (INT8)	0% (no loss)	AlexNet, ResNet

Table 4.1: Comparison of Weight Compression techniques

Huffman coding is implemented using the **Gzip** format. Gzip has been compared with other formats such as Zstd, Snappy, LZ4, and others. However, it provided the best trade-off between compression ratio and speed, measured in terms of throughput.

4.1 How GZIP Compression/Decompression works

The **Deflate** algorithm [19] is a **lossless** data compression technique that combines two main methods: **LZ77** for replacing repeated patterns and **Huffman coding** for reducing redundancy through entropy encoding. This combination allows Deflate to achieve a good balance between efficiency and speed in both compression and decompression. It is widely used in formats such as Gzip, PNG, and ZIP and is also employed in network protocols like HTTP for compressing transmitted data.

The Deflate compression process is based on independent data blocks, each of which can be processed separately. The first step in compression is tokenization using LZ77, which transforms the data into a sequence of symbols consisting of literals and distance-length pairs. When a repetition is detected, it is replaced with a reference to the previous position and the length of the duplicated segment, thus **reducing redundancy** without the need to build explicit dictionaries.

After this step, the resulting sequence is further compressed using Huffman coding. This method assigns variable-length binary codes to symbols based on their frequency of occurrence, ensuring that more common symbols have shorter codes while less common symbols have longer ones. Depending on the type of compression chosen, either a static Huffman table, which remains the same for all blocks, or a dynamic Huffman table, which is built specifically for each block to achieve more efficient compression, can be used.

A file compressed with Deflate consists of a series of blocks, each of which can be uncompressed, compressed with static Huffman coding, or compressed with dynamic Huffman coding. Uncompressed blocks are written without any processing and include a header specifying the data length and a checksum for integrity verification. Blocks with **static Huffman coding** use a predefined table, simplifying decompression but potentially being less efficient if the frequency distribution of symbols is irregular. Blocks with **dynamic Huffman coding**, on the other hand, include their own Huffman table, ensuring more efficient compression but slightly increasing the output size.

Dynamic Huffman Coding is particularly advantageous in our case because the weights are sent as a **data stream**, and the algorithm adapts in real-time to changes in symbol frequencies, unlike classical Huffman Coding, which uses a fixed tree, therefore has been

chosen over the static one.

Decompression, known as **Inflate**, follows the reverse process. Initially, the block type is read to determine whether it is uncompressed, compressed with static Huffman coding, or compressed with dynamic Huffman coding. If compressed, the symbol sequence is reconstructed by decoding the Huffman encoding. Once the original symbols are obtained, the distance-length references generated by LZ77 are resolved to reconstruct the original data [20]. Here there is an image [7] which shows practically what happens with the DEFLATE/INFLATE algorithm:

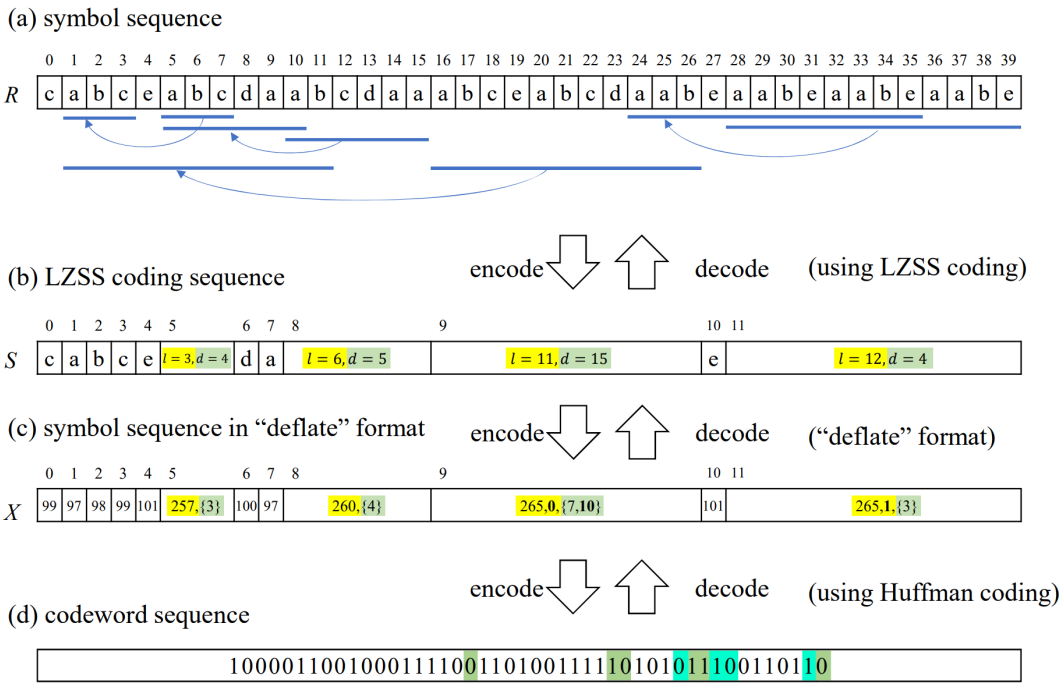


Figure 4.1: Deflate Encoding and Decoding for a symbol sequence with 40 symbols [7]

Deflate is an extremely efficient algorithm because it exploits data redundancy and applies optimized encoding without requiring explicit dictionaries. This makes it suitable for a wide range of applications, from file compression to data transmission in network protocols. Thanks to its combination of LZ77 and Huffman coding, Deflate provides effective compression while maintaining fast decompression, making it ideal for high-performance contexts such as web applications and data storage.

Chapter 5

Compression of Large Networks

Compression factor for each convolutional layer has been computed and then calculated the **weighted average** based on the layer sizes. The results show that ResNet-50 achieved a **37.5%** memory reduction, the highest value among the tested networks, but it was also effective on the other architectures.

The compression is higher on ResNets because they have a higher number of parameters and weight redundancy compared to MobileNets, which, thanks to depthwise separable convolutions and inverted residual blocks, have a smaller number of parameters.

Network	**Compression**
ResNet-20	25.4%
MobileNet v2	24.2%
MobileNet v1	23.7%
ResNet-50	37.5%
ResNet-18	36.3%

Table 5.1: Compression factors for each Network

5.1 Compression of the Largest Layer on ResNet-50

The largest layer from a ResNet-50 was extracted, then was compressed and the memory savings has been evaluated in terms of BRAM usage and percentage reduction. The results showed a memory savings of **34.8%**, which translates to **178 BRAMs saved**, which is a significant improvement for FPGA-based implementations.

	No Compression	**With Compression**
Layer size	2.25MB	1.4MB (compression factor = 1.61)
Total BRAM used	512	334 (including 16 BRAM for the decompressor)
Memory saved	-	800.16 KB (34.8%)

Table 5.2: Compression of the Largest Layer in ResNet-50

Chapter 6

Implementation

In this work, both for the Large Network compression described before and for the Implementation, the **Vitis-HLS (High-Level Synthesis)** tool has been used, which is part of the Xilinx Vitis unified software platform. Vitis HLS allows developers to describe algorithms at a high level using programming languages such as C, C++, or OpenCL, and automatically synthesizes them into RTL (Register Transfer Level) code suitable for FPGA deployment.

This approach is particularly useful in FPGA-based CNN deployments, as it allows for rapid exploration of different hardware architectures, optimizations, and accelerators. Additionally, HLS provides abstractions for complex FPGA optimizations such as pipelining, data partitioning, and interface management, which are crucial for efficiently deploying deep learning models.

The following are the **Device and Network Specifications** for the Implementation:

- **Device:** Xilinx Zynq UltraScale+ FPGA
- **Model:** XCk26-SFVC784-2LV-C
- **BRAM size:** 36 kbit (4608 bytes)
- **URAM size:** 288 kbit (36864 bytes)
- **Neural Network:** ResNet-8
- **Number of weights:** 78052 bytes

6.1 Weights Compression

I took the binary file containing the weights of the entire network and isolated the weights of the **first layer** I was considering. I then saved the binary file containing the weights of the first layer, which are **432 bytes**. This file was then given as input to the **compressor** (remember, the compression process is offline, while only the decompressor will be used in hardware). The compressor applies Huffman coding to these weights, resulting in a compressed binary file as output.

Huffman coding as we said is a lossless encoding technique, meaning that once the data is compressed, it can be decompressed back to its original form without any loss of information.

The resulting compression factor is 1.11, which is obviously very low. However, this is expected because I am compressing a very small layer of a small network. Once this process is implemented for one layer, it can naturally be extended to all layers, keeping in mind the need to properly parallelize the operations for efficiency.

The code listing is provided in Appendix A.0.1.

6.2 Loading of Compressed Weights

In the testbench, I declared a pointer for the compressed weights, which will be used to store the weights dynamically. Then, this pointer was passed to the networkSim function. I allocated **dynamic memory** for both the compressed weights layer and the other layers that are not compressed. To handle the loading of weights, I divided the process into two steps. First, I opened the binary file containing the compressed weights, read its contents, and loaded the data into the memory location that the pointer was referencing, which I had previously declared and passed to the function. Next, I opened the file containing the original weights of the network, skipping the weights of the first layer, read the remaining weights, and then closed the file. Finally, I passed the pointer to the compressed weights to the Resnet8 function, which serves as the **top-level function** for the network simulation.

The code listing is provided in Appendix A.0.2.

6.3 Loading of compressed Weights from Master AXI to BRAM

As shown in the code snippet, I passed the **compressed weights** as a parameter to the top-level function, which is ResNet. Then, I added the **pragma** directive for the **AXI master interface**. This pragma is necessary to establish the communication between the FPGA logic and the external memory (DDR) through the AXI interface, enabling efficient data transfer to and from the FPGA.

Next, I included the pragma directive #pragma HLS interface mode=ap_ctrl_chain port=return. This pragma is used to specify that the function should use a control chain for its return interface, optimizing the data flow and control signals during execution. It ensures that the function interacts properly with the rest of the hardware design, minimizing the overhead of control signal generation.

When the weights are needed to perform the convolution operations, they must be decompressed. So, I pass them to the memory management function, which handles the process. Specifically, the weights will first be loaded from the master AXI interface into the **internal BRAMs**. After that, a stream will be initiated to the **decompressor**. During the decompression phase, the weights will be unpacked and, once decompressed, another stream will carry the decompressed weights directly to the convolution unit. This is the planned workflow.
The code listing is provided in Appendix A.0.3.

In the memory_management function, several important sub-functions are declared, including produce_stream, which generates the **streams** for each layer. These streams are essential for feeding data into the convolution operations. For the layer I am modifying, two specific functions have been created: master_to_comprstream and produce_stream_. The first function, master_to_comprstream, takes the pointer to the compressed weights as input and outputs the compressed weights stream, which is then passed to the decompressor. The second function, produce_stream_, takes the compressed weights stream as input and produces the decompressed weights stream, which directly feeds into the **convolution**.

The reason I chose to create two separate functions instead of one that performs everything is to maintain the **dataflow** design principle. This design ensures that while reading data from memory, the convolution process can simultaneously use the data in parallel. By splitting the operations into two distinct functions, the system can achieve a continuous flow of data between reading, decompressing, and convolution, without blocking or stalling, thus enhancing the overall performance and efficiency of the system.

The #pragma HLS dataflow is already at the top, and this directive is used to enable parallel execution of functions in a design. It allows different stages of the process (e.g., memory reads, computations, and writes) to run concurrently, improving throughput and reducing latency.
The code listing is provided in Appendix A.0.4.

In the master_to_comprstream function, I declared an array to load the compressed weights from DDR to BRAM. The process begins by reading the compressed weights from the master AXI and writing them into the BRAM. Once the weights are loaded into BRAM, I then write the compressed weights from BRAM into a stream, which will be the input for the decompressor. Additionally, I read the BRAM 512 times (in this case, corresponding to c_o_index), so the stream will contain the compressed weights stored in BRAM, repeated 512 times. As a result, the stream will have a total of 512 * 386 = 197,632 elements.

After all the weights have been processed and streamed, the `inStream <<0` line signifies the end of the stream, while `inEos <<1` indicates that the end-of-stream signal is set to 1, marking the conclusion of the data transmission.
The code listing is provided in Appendix A.0.5.

6.4 Decompression and Streaming for Convolution

In this function, the input is the compressed weights stream, which is passed to the decompressor. The decompressor uses Huffman coding to decompress the weights, producing an output stream that contains both the data and validity bits. To separate the data from the validity bits, I process the output stream accordingly, resulting in a stream that contains only the decompressed data.

Since the decompressor outputs the data **sequentially**, I had to reorganize the memory structure. Specifically, I iterated over the BRAM memory multiple times (the outermost loop). Then, I iterated over the number of channels, followed by the window size and parallelism, ensuring that the data was processed in a manner suitable for convolution operations. Ultimately, this results in a stream that feeds directly into the convolution operations.

The code listing is provided in Appendix A.0.6.

Chapter 7

Results on ResNet-8

I performed CSIM, synthesis, cosimulation, and place and route in Vitis HLS to get a real estimate of the allocated resources, the estimated frequency, and the cycle latency. The table shows the results comparing the initial network without and the final network with the layer in DDR and the integrated decompressor. The decompressor uses **2.76%** more memory as we can see.

Since I applied compression only to a single small layer of ResNet-8, the effect on total memory usage was not significant. In this implementation, our main focus was **ensuring** that everything worked correctly and that the decompressor was properly **integrated** into the network. As for compression, it was tested and its effects were more visible on larger networks with more layers and parameters. Regarding the latency, we observed an increase of two orders of magnitude. This means that, while compression reduces memory usage, it introduces a considerable **computational overhead**, which impacts inference speed.

	BRAM	**URAM**	**DSP**	**FF**	**LUT**	**Latency (cc)**
ResNet-8 (w/o decompressor)	186	63	773	63194	70174	85578
ResNet-8 (w/ decompressor)	197	64	773	75569	85300	1716336

Table 7.1: Results on ResNet-8

Chapter 8

Conclusion

In this work, we focused on **memory savings**, achieving compression factors of approximately **30-35%** across all convolutional layers for larger networks. Furthermore, we integrated both a compressor and decompressor into an existing network.

Future work and improvements include integrating the decompressor into larger networks to evaluate whether the compression gain remains consistent or if there is any loss. Another key aspect to address is **optimizing throughput**, which is currently two orders of magnitude higher compared to the original network. Also, Huffman coding could be combined with other compression techniques to achieve better results and improve efficiency.

Appendix A

Source Codes

A.0.1 Code for Weights Compression

```
#include <ap_int.h>
#include <assert.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <string>
#include "cmdlineparser.h"
#include "kernel_stream_utils.hpp"
#include "ap_axi_sdata.h"
#include "hls_stream.h"
#include "zlib_compress.hpp"

#define MULTIPLE_BYTES 8
#define GMEM_DWIDTH 64
#define STRATEGY 0
#define NUM_BLOCKS 8
#define BLOCK_SIZE_IN_KB 32
#define TUSER_DWIDTH 32
```

```
21 typedef ap_axiu<GMEM_DWIDTH, 0, 0, 0> in_dT;
22 typedef ap_axiu<GMEM_DWIDTH, TUSER_DWIDTH, 0, 0> out_dT;
23 typedef ap_axiu<32, 0, 0, 0> size_dT;
24
25 const uint32_t c_size = (GMEM_DWIDTH / 8);
26
27 void gzipcMulticoreStreaming(hls::stream<in_dT>& inStream, hls::
      stream<out_dT>& outStream) {
28 #pragma HLS INTERFACE AXIS port = inStream
29 #pragma HLS INTERFACE AXIS port = outStream
30 #pragma HLS INTERFACE ap_ctrl_none port = return
31
32 #pragma HLS DATAFLOW
33     xf::compression::gzipMulticoreCompressAxiStream<STRATEGY,
          BLOCK_SIZE_IN_KB, NUM_BLOCKS, TUSER_DWIDTH>(inStream,
          outStream);
34 }
35
36 int main() {
37     std::string inputFileName = "layer_432_quantized.bin";
38     std::string outputFileName = "gzipc_resnet386.bin";
39
40     std::fstream inFile;
41     inFile.open(inputFileName.c_str(), std::fstream::binary | std
          ::fstream::in);
42     if (!inFile.is_open()) {
43         std::cout << "Cannot_open_the_input_file!!" <<
              inputFileName << std::endl;
44         exit(0);
45     }
46     std::ofstream outFile;
47     outFile.open(outputFileName.c_str(), std::fstream::binary |
          std::fstream::out);
```

```
48
49      hls::stream<in_dT> inStream("inStream");
50      hls::stream<out_dT> outStream("outStream");
51      hls::stream<size_dT> outSizeStream("outSizeStream");
52
53
54      inFile.seekg(0, std::ios::end);
55      const uint32_t inFileSize = (uint32_t)inFile.tellg();
56      inFile.seekg(0, std::ios::beg);
57
58      auto numItr = 1;
59      uint32_t compressedSize = 0;
60
61      in_dT inData;
62      for (int z = 0; z < numItr; z++) {
63          inFile.seekg(0, std::ios::beg);
64          // Input File back to back
65          for (uint32_t i = 0; i < inFileSize; i += c_size) {
66              ap_uint<GMEM_DWIDTH> v;
67              bool last = false;
68              uint32_t rSize = c_size;
69              if (i + c_size >= inFileSize) {
70                  rSize = inFileSize - i;
71                  last = true;
72              }
73              inFile.read((char*)&v, rSize);
74              inData.data = v;
75              inData.keep = -1;
76              inData.last = false;
77              if (last) {
78                  uint32_t num = 0;
79                  inData.last = true;
80                  for (int b = 0; b < rSize; b++) {
```

```
81                      num |= 1UL << b;
82                  }
83                  inData.keep = num;
84              }
85              inStream << inData;
86          }
87
88          // Compression Call
89          gzipcMulticoreStreaming(inStream, outStream);
90
91          uint32_t byteCounter = 0;
92          // Write file
93          out_dT val;
94          do {
95              val = outStream.read();
96              ap_uint<GMEM_DWIDTH> o = val.data;
97              auto w_size = c_size;
98              if (val.keep != -1) w_size = __builtin_popcount(val.
                    keep);
99              byteCounter += w_size;
100             outFile.write((char*)&o, w_size);
101         } while (!val.last);
102
103         compressedSize = byteCounter;
104     }
105
106     inFile.close();
107     outFile.close();
108
109     // Compression factor computation
110     double compressionFactor = static_cast<double>(inFileSize) /
        static_cast<double>(compressedSize);
```

```
111     std::cout << "Original␣size:␣" << inFileSize << "byte" << std
            ::endl;
112     std::cout << "Compressed_size:" << compressedSize << "byte"
            << std::endl;
113     std::cout << "Compression_factor:" << compressionFactor <<
            std::endl;
114
115     return 0;
116 }
```

Listing A.1: Weights Compression

A.0.2 Loading of Compressed Weights

```
1  std::chrono::duration<double> networkSim(
2    int argc,
3    char** argv,
4    std::string prj_root,
5    const unsigned int n_inp,
6    const unsigned int n_out,
7    const t_in_mem* inp_1,
8    t_out_mem* o_outp1,
9    uint8_t* weights_ddr
10
11   ) {
12 t_weights_st *c_weights;
13 const int c_first_weights_dim = 386;
14 const int c_weights_dim = 78006;
15 const int c_second_weights_dim = 77620;
16
17 posix_memalign((void**)&weights_ddr,4096, 386*sizeof(t_weights_st
       ));
18 posix_memalign((void**)&c_weights, 4096,77620 * sizeof(
       t_weights_st));
```

```
19
20 std::ifstream file_weights("npy/gzipc_resnet386.bin", std::ios::
       binary);
21
22 file_weights.read(reinterpret_cast<char*>(weights_ddr), 386 *
       sizeof(t_weights_st));
23
24 file_weights.close();
25
26 std::ifstream file_weightss(prj_root + "npy/resnet8_weights.bin",
        std::ios::binary);
27
28 file_weightss.seekg(432 * sizeof(t_weights_st), std::ios::beg);
29 file_weightss.read(reinterpret_cast<char*>(c_weights), 77620 *
       sizeof(t_weights_st));
30 file_weightss.close();
31
32                             ...
33
34
35 auto start = std::chrono::high_resolution_clock::now();
36
37   resnet8(
38     c_inp_1_stream,
39     c_weights_stream,
40     c_outp1_stream,
41
42     weights_ddr
43   );
44
45 auto end = std::chrono::high_resolution_clock::now();
46
47                       ...
```

```
48
49 free(c_weights);
50 free(buffer);
51
52 return (end - start);
53
54 }
```

Listing A.2: Compressed Weights Loading and Initialization

A.0.3 Top-level function ResNet-8

```
1  void resnet8(
2    hls::stream<t_inp_1> &i_inp_1,
3    hls::stream<t_weights_stream> &i_data_weights,
4    hls::stream<t_o_outp1> &o_outp1,
5    uint8_t* weights_ddr
6
7    ) {
8
9  #pragma HLS interface m_axi port=weights_ddr depth=386
10 #pragma HLS interface mode=ap_ctrl_chain port=return
11
12   hls::stream<t_net_const_13> s_net_const_13[9];
13   hls::stream<t_net_const_14> s_net_const_14[1];
14   hls::stream<t_net_const_15> s_net_const_15[9];
15
16                        ...
17
18 #pragma HLS interface port=i_data_weights mode=axis
19 #pragma HLS stream variable=s_net_const_13 depth=2 type=fifo
20 #pragma HLS stream variable=s_net_const_14 depth=2 type=fifo
21 #pragma HLS stream variable=s_net_const_15 depth=2 type=fifo
22
```

```
23                          ...
24
25 #pragma HLS interface port=o_outp1 mode=axis
26
27     memory_management (
28     i_data_weights,
29     s_net_const_13,
30     s_net_const_14,
31     s_net_const_15,
32     s_net_const_16,
33     s_net_const_17,
34     s_net_const_18,
35     s_net_const_19,
36     s_net_const_20,
37     s_net_const_21,
38     s_net_const_22,
39     s_net_const_23,
40     s_net_const_24,
41     s_net_const_25,
42     s_net_const_26,
43     s_net_const_27,
44     s_net_const_28,
45     s_net_const_29,
46     s_net_const_30,
47     s_net_const_31,
48     s_net_const_32,
49
50     weights_ddr
51   );
52
53   nn2fpga::produce_stream <
54     ...
55   > (
```

```
56      i_inp_1,
57      s_net_produce_0
58    );
59
60    nn2fpga::shift_op <
61      ...
62    > (
63      s_net_produce_0[0],
64      s_net_produce_0_pre_pad[0],
65      s_net_produce_0_data[0]
66    );
67
68                          ...
69
70  nn2fpga::conv_comp <
71      ...
72    > (
73      s_net_produce_0_compute,
74      s_net_const_13,
75      s_net_const_14,
76      s_net_conv_1,
77    );
78
79                              ...
```

Listing A.3: Top-Level Function: ResNet8

A.0.4 Memory management

```
1  void memory_management(
2    hls::stream<t_weights_stream> &i_data_weights,
3    hls::stream<t_net_const_13> s_net_const_13[9],
4    hls::stream<t_net_const_14> s_net_const_14[1],
5
```

```
6                               ...
7
8       uint8_t* weights_ddr
9       ) {
10
11                              ...
12
13  typedef ap_uint<IN_BITWIDTH> in_t;
14  hls::stream<in_t> inStream("inStream");
15  hls::stream<bool> inEos("inEos");
16
17    master_to_comprstream <in_t,
18              c_node_const_10_ow,
19              c_node_const_10_oh,
20              c_net_const_13_reuse>
21          (
22      weights_ddr
23     s_net_const_13_init_flag,
24            inEos,
25      inStream
26    );
27
28    produce_stream_ <
29      t_net_const_13,
30      c_node_const_10_ich,
31      c_node_const_10_och,
32      c_node_const_10_ow,
33      c_node_const_10_oh,
34      c_net_const_13_iw,
35      c_net_const_13_ih,
36      c_net_const_13_ops,
37      c_net_const_13_reuse,
38    in_t
```

```
39    > (
40      inStream,
41      inEos,
42      s_net_const_13_init_flag,
43      s_net_const_13
44    );
45
46    nn2fpga::produce_stream <
47      t_net_const_14_st,
48      t_net_const_14_init,
49      t_net_const_14,
50      c_node_const_11_ich,
51      c_node_const_11_och,
52      c_node_const_11_ow,
53      c_node_const_11_oh,
54      c_net_const_14_iw,
55      c_net_const_14_ih,
56      c_net_const_14_ops,
57      c_net_const_14_reuse
58    > (
59      c_net_const_14,
60      s_net_const_14_init,
61      s_net_const_14_init_flag,
62      s_net_const_14
63    );
64
65
66                              ...
```

Listing A.4: Memory management

A.0.5 Weights from DDR to BRAM

```
template <typename in_t,  int OW, int OH, int c_reuse>
void master_to_comprstream(uint8_t* weights_ddr,
                    bool& s_init,
                      hls::stream<bool>& inEos,
                      hls::stream<in_t>& inStream) {

constexpr unsigned c_o_index = OH * OW / c_reuse;
const uint32_t sizeof_in = (IN_BITWIDTH / 8);
uint8_t  weightscompr[386];

for (int i = 0; i < 386; ++i) {

    weightscompr[i] = buffer[i];
}

for (auto s_o_index = 0; s_o_index < c_o_index; s_o_index++) {
    for (uint32_t i = 0; i < 386; i += sizeof_in) {

        in_t x = 0;
        for (uint32_t j = 0; j < sizeof_in; j++) {

       x.range((j + 1) * 8 - 1, j * 8) = weightscompr[i + j];
        }
        inStream << x;
        inEos << 0;
    }
    }
    inStream << 0;
    inEos << 1;
   }
```

Listing A.5: Loading Compressed Weights from BRAM to Stream

A.0.6 Decompression and Streaming for Convolution

```
#include "inflate.hpp"

#define MULTIPLE_BYTES 8
#define LOW_OFFSET 1
#define MAX_OFFSET (32 * 1024)
#define HISTORY_SIZE MAX_OFFSET
#define LL_MODEL false

#define HUFFMAN_TYPE xf::compression::DYNAMIC

#define OUT_BITWIDTH (MULTIPLE_BYTES * 8)


template < typename dout_t, int ICH, int OCH, int OW, int OH,
          int c_fw, int c_fh, int c_ops, int c_reuse,typename
             in_t>
void produce_stream_(hls::stream<in_t>& inStream,
                     hls::stream<bool>& inEos,
                    bool& s_init,
                    hls::stream<dout_t> o_data[c_fh * c_fw]) {
constexpr unsigned FSZ = c_fh * c_fw;
constexpr unsigned c_ch = ICH * OCH / c_ops;
constexpr unsigned c_o_index = OH * OW / c_reuse;

const uint32_t strbSize = (OUT_BITWIDTH / 8);

typedef ap_uint<OUT_BITWIDTH> out_t;

hls::stream<ap_uint<OUT_BITWIDTH + strbSize>> outStream("
   decompressOut");

```

```
30 hls::stream<uint8_t> decompressedStream("decompressedStream");
31
32 const int c_decoderType = (int)HUFFMAN_TYPE;
33
34 xf::compression::details::inflateMultiByteCore<c_decoderType,
      MULTIPLE_BYTES, xf::compression::FileFormat::BOTH, LL_MODEL,
      HISTORY_SIZE>(inStream, inEos, outStream);
35
36 for (ap_uint<OUT_BITWIDTH + strbSize> val = outStream.read(); val
       != 0; val = outStream.read()) {
37         out_t o = val.range(strbSize + OUT_BITWIDTH - 1, strbSize
              );
38         ap_uint<strbSize> strb = val.range(strbSize - 1, 0);
39
40         for (size_t i = 0; i < strbSize; ++i) {
41             if (strb[i]) {
42
43         decompressedStream << o.range((i + 1) * 8 - 1, i * 8);
44
45       }
46         }
47         }
48
49 dout_t s_output;
50 for (auto s_o_index = 0; s_o_index < c_o_index; s_o_index++) {
51 for (auto s_ch = 0; s_ch < c_ch; s_ch++) {
52 #pragma HLS pipeline
53         for (auto s_index = 0; s_index < FSZ; s_index++) {
54    for (auto s_ops = 0; s_ops < c_ops; s_ops++) {
55
56 ap_fixed<8, 4, AP_RND, AP_SAT> data;
57
58     ap_uint<8> byte = decompressedStream.read();
```

```
59                  data.range(7, 0) = byte;
60      s_output[s_ops]=data;
61      }
62
63 o_data[s_index].write(s_output);
64 }} }
65     }
```

Listing A.6: Decompression and Stream Preparation for Convolution

Bibliography

[1] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient processing of deep neural networks: A tutorial and survey. *arXiv*, 1703.09039v2, August 2017. [cs.CV].

[2] Jinghan Wang, Guangyue Li, and Wenzhao Zhang. Combine-net: An improved filter pruning algorithm. *Information*, 12(7), 2021.

[3] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149v5*, Feb 2016. [cs.CV].

[4] Deepak Ghimire, Daehyun Kil, and Seong-hwan Kim. A survey on efficient convolutional neural networks and hardware acceleration. *Electronics*, 11(6):945, 2022.

[5] Zhenfeng Ma, Hongyi Zhu, Zhiqiang He, Yunpeng Lu, and Feng Song. Deep lossless compression algorithm based on arithmetic coding for power data. *Sensors*, 22(14):5331, 2022.

[6] M. Mary Shanthi Rani, P. Chitra, S. Lakshmanan, M. Kalpana Devi, R. Sangeetha, and S. Nithya. Deepcompnet: A novel neural net model compression architecture. *Journal of Computer Science and Applications*, 2022.

[7] Daisuke Takafuji, Koji Nakano, Yasuaki Ito, and Akihiko Kasagi. Acceleration of deflate encoding and decoding with gpu implementations. In *2021 Ninth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 180–186, 2021.

[8] Shengli Jiang, Shiyi Qin, Joshua L. Pulsipher, and Victor M. Zavala. Convolutional neural networks: Basic concepts and applications in manufacturing. *arXiv preprint arXiv:2210.07848v1*, oct 2022. [cs.CV].

[9] Andrew G. Howard, Weijun Wang, Menglong Zhu, Tobias Weyand, Bo Chen, Marco Andreetto, Dmitry Kalenichenko, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861v1*, apr 2017. [cs.CV].

[10] Song Han, John Tran, Jeff Pool, and William J. Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626v3*, Oct 2015. [cs.NE].

[11] Xiaolong Ma, Geng Yuan, Sheng Lin, Zhengang Li, Hao Sun, and Yanzhi Wang. Resnet can be pruned 60: Introducing network purification and unused path removal (p-rm) after weight pruning. *arXiv preprint arXiv:1905.00136v1*, Apr 2019. [cs.LG].

[12] Sanghyun Son, Seungjun Nah, and Kyoung Mu Lee. Clustering convolutional kernels to compress deep neural networks. *arXiv preprint arXiv:1901.03989*, 2019. [cs.CV].

[13] Li-Na Wang, Wenxue Liu, Xiang Liu, Guoqiang Zhong, Partha Pratim Roy, Junyu Dong, and Kaizhu Huang. Compressing deep networks by neuron agglomerative clustering. *MDPI Sensors*, 20(21):6149, Oct 2020. Received: 4 September 2020; Accepted: 16 October 2020; Published: 23 October 2020.

[14] Yang Sui, MiaoYin Yu, Gong JinqiXiao, Huy Phan, and BoYuan. Elrt: Efficient low-rank training for compact convolutional neural networks. *arXiv preprint arXiv:2401.10341v1*, Jan 2024. [cs.CV].

[15] G. Cai, J. Li, X. Liu, Z. Chen, and H. Zhang. Learning and compressing: Low-rank matrix factorization for deep neural network compression. *Applied Sciences*, 13(4):2704, 2023. Academic Editors: Phivos Mylonas, Katia Lida Kermanidis, Manolis Maragoudakis.

[16] Jong Hun Lee, Joonho Kong, and Arslan Munir. Arithmetic coding-based 5-bit weight encoding and hardware decoder for cnn inference in edge devices. *IEEE Access*, 9:166736–166749, 2021.

[17] Alistair Moffat. Huffman coding. *The University of Melbourne*.

[18] Nidhi Dhawale. Implementation of huffman algorithm and study for optimization. *Ramdeobaba College of Engg and Management*, 201x. Mtech VLSI Design Dept. of Electronics, Nagpur-440013, India.

[19] Savan Oswal, Anjali Singh, and Kirthi Kumari. Deflate compression algorithm. *International Journal of Engineering Research and General Science*, 4(1):430, Jan-Feb 2016. B.E. Students, Department of Information Technology, KJ'S Trinity College Of Engineering and Research, Pune, India.

[20] K. Anand, M. Priyadharshini, and K. Priyadharshini. Compression and decompression of files without loss of quality. In *2023 International Conference on Networking and Communications (ICNWC)*, pages 1–6, 2023.