Informatics

A Registry and Benchmarking Tool for Lossy Neural Image Compression

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Dominik Ferrari, BSc
Matrikelnummer 01525826

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Mitwirkung: Dipl.-Ing Alireza Furutanpey, BSc

Wien, 27. Jänner 2025

Dominik Ferrari　　Schahram Dustdar

Informatics

A Registry and Benchmarking Tool for Lossy Neural Image Compression

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Dominik Ferrari, BSc
Registration Number 01525826

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Assistance: Dipl.-Ing Alireza Furutanpey, BSc

Vienna, January 27, 2025

Dominik Ferrari Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Dominik Ferrari, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. Jänner 2025

Dominik Ferrari

Danksagung

Ich möchte mich sowohl bei meinem Betreuer Dr. Schahram Dustdar als auch Dipl.-Ing. Alireza Furutanpey für die Mitwirkung und Bereitstellung dieses Themas bedanken. Ich danke ihnen für die Unterstützung, das Feedback und die Betreuung. Weiters, möchte ich mich bei Alireza und der Distributed Systems Group für die Bereitstellung der Resourcen bedanken.

Ein großer Dank ergeht an Alireza für das detaillierte Feedback, die Einführung in das interessante Thema und laufende Betreuung. Sein Wissen über das Thema ist unschätzbar und seine Worte haben mich immer motiviert.

Meinem Arbeitgeber Manfred Guttmann (NTB Solutions) und dem Team Secure Archives danke ich für die Geduld, das Vertrauen und die Möglichkeit, die Masterarbeit parallel zu meiner beruflichen Tätigkeit schreiben zu können.

Ein besonderer Dank gilt meiner Mutter und meiner Schwester, die mich stets unterstützt haben. Sie haben mich während dieser psychischen und physischen Belastung motiviert.

Zuallerletzt möchte ich mir selbst dafür danken, dass ich den Mut und die Ausdauer hatte, diese Herausforderung anzunehmen.

Acknowledgements

I would like to thank both my supervisor Schahram Dustdar and Alireza Furutanpey, for their assistance and for providing this topic. I am grateful for their support, feedback, and guidance. In addition, I would like to thank Alireza and the Distributed Systems Group for providing the computational resources.

A big thank you to Alireza for the detailed feedback, introduction to this fascinating topic, and ongoing supervision. His knowledge on this topic is priceless, and his words have always motivated me.

I would like to thank my employer Manfred Guttmann (NTB Solutions) and the Secure Archives team for the patience, trust, and for giving me the opportunity to write my master's thesis alongside my professional work.

Special thanks go to my mother and sister, who have always supported me. They provided me with strength, courage, and motivation during this mentally and physically challenging time.

Lastly, I would like to thank myself for having the perseverance to take on this challenge.

Kurzfassung

Neuliche Fortschritte in Deep Learning und Hardware-Beschleunigung führten zur Entstehung von neuen Lösungen in KI-basierter Bildkomprimierung. Mit jeder neuen Lösung wurde es schwieriger, die tatsächlichen Auswirkungen jeder Änderung und getroffenen Entscheidung zu messen.

Wir schlagen eine Zwei-Faktor-Lösung vor. Wir entwickeln eine Klassifizierung, die die grobe Landschaft der KI-unterstützten Bildkomprimierung darstellt. Sie ist in Kernkomponenten eingeteilt wie Quantifizierung, Kontext-Modelle, architektonische Archetypen, hierarchische Prioren und Modelle mit variabler Rate. Wir analysieren die Trends und Weiterentwicklungen, um Forschern und Forscherinnen den Einstieg und Weiterforschung zu erleichtern.

Zusätzlich entwickeln wir ein Benchmark-Werkzeug, um die KI-Modelle zu trainieren und zu testen. Unsere Konfigurationen sind standardisiert und leicht änderbar. Der Test-Prozess misst und vergleicht Variationen der Konfiguration unter der Verwendung von verschiedenen Datensätzen und Metriken. Wir evaluieren jede Konfiguration mit den CLIC- und Kodak-Datensätzen. Wir messen die R-D Leistung, LPIPS, PSNR, MS-SSIM, Kodierungslatenz und den Kodierungsdurchsatz. Wir beobachten die Auswirkung der getroffennen Design-Entscheidungen wie die Änderung der Non-Linearity-Kernkomponente. Während die GSDN-Aktivierungsfunktion in der Non-Linearity-Kernkomponente in einem kleinen Netzwerk eine schlechtere Leistung erzielt hat, war die Leistung die Beste in größeren Netzwerken. Interessanterweise waren Konfigurationen, die größere Netzwerke definiert haben, zwar langsamer, haben aber eine bessere R-D Leistung für den gleichen Kompressionsfaktor erzielt.

Abstract

Recent advancements in deep learning and hardware accelerators led to the appearance of many novel solutions in learned image compression. With each novel solution, measuring the impact of fine-grained decisions gets increasingly more challenging.

We propose a two-fold solution. We develop a taxonomy to define the landscape of lossy learned image compression broadly. We classify advancements into core components such as quantization, context models, architectural archetypes, hierarchical priors, and variable rate models. We analyze the trends and advancements to aid novices and seasoned researchers in showing the focus points of current research.

In addition, we develop a benchmarking tool to train and test models. Our standardized configurations are highly customizable by defining variations. The testing pipeline compares variations with multiple datasets and metrics. We evaluate each configuration on the CLIC and Kodak datasets. We measure the R-D performance, LPIPS, PSNR, MS-SSIM, encoding and decoding latency, and encoding and decoding throughput. We observe the impact of fine-grained design decisions, such as changing the non-linearity block. While the GSDN activation function within the non-linearity block performs worse in small networks, its best performance is achieved in deeper networks. Interestingly, configurations with deeper networks performed slower yet achieved higher visual quality for the same compression rates.

We compare the learned image compression models with a fixed codec (BPG) to ensure comparable results. We find that the impact of minor design decisions depends on network size, resulting in vastly different performances.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Recent advancements in deep learning and hardware accelerators led to the appearance of many novel solutions in image compression. While the usage of traditional image formats such as PNG (80.6%) and JPEG(76%) [91] dominates the web space [90], recent learning-based codecs match and surpass those formats in compression performance [12, 9, 13]. Although compression using neural networks dates back to 1989 [82], only recently end-to-end to end learned methods that convincingly outperform handcrafted standards [12, 81]. Data compression reduces bit rates by removing redundant information. In lossless compression, the codec output must be identical to the input. Conversely, lossy compression permits distortion to achieve significantly lower bitrates. Neural compression is the application of artificial neural networks (ANNs) to this task [102]. The core difference between handcrafted and learned methods is that the former relies on linear transformations based on expert knowledge. Contrarily, the latter is data-driven with non-linear transformations learned by neural networks to reduce dependencies from sources that are not jointly Gaussian [10].

1.2 Problem Statement

To adequately scope this thesis, we exclusively focus on lossy compression. By focusing on preserving structure in regions of interest, codecs can safely discard information for less critical areas (e.g., unfocused backgrounds).

Standardized benchmark datasets (e.g., Kodak [27] or CLIC [14]) exist to compare codec performance and demonstrate whether a novel method can improve rate-distortion performance.

Cheng et al. [25] work underlines the importance of our thesis. The work introduces a convolutional attention mechanism and a novel entropy model based on Gaussian Mixture Likelihoods (GMM). While the model demonstrates significantly better rate-distortion performance overall, the contribution of each component is unclear. Instead, improvements contributed by GMM and other design decisions, such as lightweight attention layers that emphasize local interactions [51], should be measured. Another example is the varying ways to handle gradient flow of *discrete quantization*. Some authors [75, 81] apply straight-through estimation, others [2] continuous relaxation or a combination [10]. The impact of such a seemingly minor design decision needs to be clarified. Therefore, a more streamlined method of precisely comparing and benchmarking fine-grained design decisions is needed.

To the best of our knowledge, we are the first to provide a comprehensive overview and taxonomy of lossy neural image compression that focuses on primary components and contributions instead of architectural changes only. We argue that the entropy model, the quantization, encoding, and the objective function primarily determine the novelty of the work.

Our study investigates the two limitations mentioned above in contemporary lossy neural image compression research. By addressing both limitations, we aim to ease the comparison of methods. Then, newer work may find promising directions and more readily determine the significance of method components. To this end, this thesis aims to facilitate the advancements in lossy neural image compression.

1.3 Research Questions

- RQ1: How do the finer-grained design decisions (e.g., model width and depth, architectural component, quantization methods) impact rate-distortion performance and perceptual quality? The primary focus is on runtime and improving rate-distortion performance using particular measures (e.g., MS-SSIM, PSNR). By answering RQ1, we may identify generalizable design decisions and facilitate discussions on developing a better understanding of their impact.

- RQ2: How can we classify neural compression methods by their core components? Neural compression is a popular but niche research area of deep learning (DL), and work in learned image compression (LIC) draws from advancements in DL for other vision tasks (e.g., [24]). By answering RQ2, we can aid novices in identifying parts of existing and future work that are native (and most relevant) to the compression task.

- RQ3: What are current publication trends and focus areas in neural data compression? Identifying trends can aid novices in providing contributions that the community is more likely to appreciate. Contrastingly, more seasoned researchers may identify worthwhile problems that the larger community needs to pay more attention to.

1.4 Aim of the work

The thesis aims to provide a comprehensive overview of lossy neural image compression work following a non-linear transform coding (NTC) approach [10]. We conceive a taxonomy that categorizes work exclusively according to components relevant to the compression objective with learned non-linear functions. Additionally, we implement a model registry and interface to benchmark various methods. Unlike existing libraries for neural data compression that implement existing methods, an essential requirement of our work is the configurability and exchangeability of individual components.

By addressing both limitations, we aim to ease the comparison of methods so that newer work may find promising directions and more readily determine the significance of method components.

We expect a streamlined integration of existing open-source neural data compression methods from third-party repositories and the ability to reproduce reported results in our environment. Lastly, we investigate the impact on rate-distortion performance by exchanging seemingly minor components of the existing compression model (e.g., GDN versus more common non-linearity functions). With this, we expect to draw meaningful insights that aid in the design of novel methods. Lastly, we expect our taxonomy to facilitate identifying worthwhile work and components to study.

The twofold aspect of our solution (taxonomy and registry) reflects the two limitations described in Section 1.2. With our comprehensive evaluation, taxonomy, and registry, we facilitate a fine-grained distinction of components relevant to the compression task. By combining the taxonomy with our benchmarking tool, newer work may find promising directions and more readily determine the significance of method components.

1.5 Structure

Chapter 2 describes the fundamental concepts and explains the domain knowledge required to understand the following chapters. Chapter 3 presents the related work, comparing existing benchmarking tools and literature reviews. Chapter 4 explains the methodology and procedures used to create an elaborate literature review, taxonomy, and benchmarking tool. It also includes the research questions posed in this thesis. Chapter 5 presents the developed taxonomy, which describes the classification of core components instrumental in learning image compression. Chapter 6 shows the implementation of the benchmarking tool Benchpresso, including software concepts, requirements, specific implementation details, UML diagrams, and caveats. Chapter 7 evaluates learned image compression models using the benchmarking tool and presents the results. The results are discussed and explained separately for each architecture. Chapter 8 concludes the thesis by answering the research questions and presenting an outlook to future work and possible continuation.

CHAPTER 2

Fundamentals

Section 2.1 defines image compression types. We explain the difference between traditional and learned image compression, including the rate-distortion theory. Section 2.2 introduces the commonly used evaluation metrics in literature, such as the mean-squared error, peak signal-to-noise ratio, and structural similarity index measure.

2.1 Image Compression

Compression reduces input data size by eliminating redundant information without distorting it. In image compression, the focus is on encoding the original image to be reconstructed from the compressed data. The goal is to reconstruct the image with the fewest differences from the original. Lossless image compression aims to reduce the data size while ensuring that the original and reconstructed images are identical. This limitation challenges the compression efficiency that can be achieved. Contrastingly, lossy image compression permits a bounded amount of error to lower bitrates further. The extent of detail and quality loss can vary and is often quantified using quality levels.

2.1.1 Lossy Image Compression Pipeline

Figure 2.1 shows a general pipeline in lossy image compression. The input image is represented as pixel intensities. These values are then quantized (rounded) because storing infinite real numbers is impossible. Quantization introduces error since the original real values cannot be exactly restored. Next, the quantized data is encoded into a bitstream and stored along with necessary metadata, such as the quality level. The resulting bitstream should be significantly smaller than the original image. The metadata is used to reverse the process during decoding. The bitstream is dequantized, and the inverse transform is applied. Depending on the set quality level, the reconstructed image may differ from the original to varying degrees.

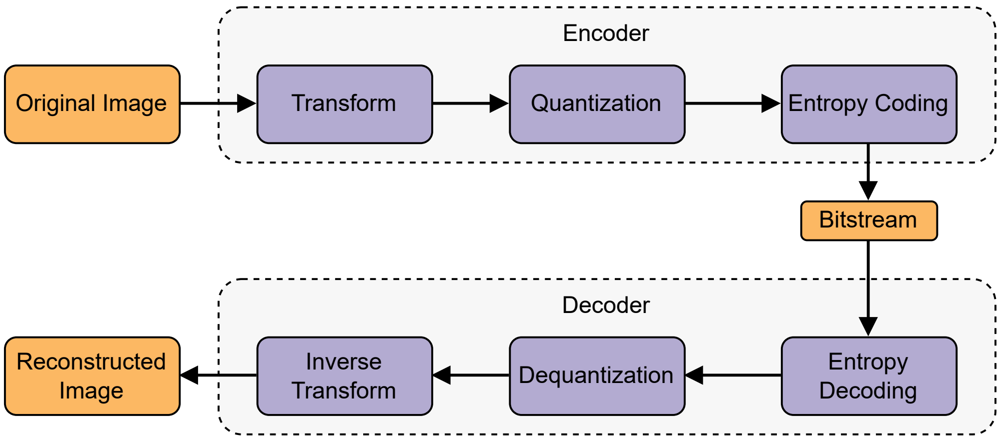

Figure 2.1: Lossy image compression pipeline.

2.1.2 Learned Image Compression

Learned image compression implements the general image compression pipeline that utilizes deep learning across several key steps. In LIC, the input image is transformed using a non-linear parametric transform. Traditional methods, such as the Discrete Cosine Transform (DCT) used in JPEG, are linear. Quantization was initially achieved through binarization [102], such as in the work by Toderici et al. [85, 86], which learns quantization by utilizing stochastic binarization. The image reconstruction process is the approximate inverse of the encoding. This architecture can be expanded with additional components that capture local and global similarities, focus attention on areas of high interest, and differentiate between relevant and redundant information. Neural networks with non-linear parametric transforms may achieve extremely low file sizes while maintaining high-quality reconstructed images. These results were not possible before with linear codecs such as JPEG.

2.1.3 Rate-Distortion Theory

The goal is to find a data representation that does not exceed a certain error threshold (distortion) while requiring minimal mutual information. Given the mutual information and the distortion, the r-d function theory estimates the minimum transmission bitrate needed to reconstruct the target representation without exceeding the error threshold [102]. In other words, Shannon's [79] rate-distortion theory estimates the achievable lower bound of bits needed to transmit the data from the encoder to the decoder. Assume an objective function L(c, λ), where c is a lossy codec and λ the rate-distortion lagrangian to relax the constrained optimization problem in Equation (2.1). As described by Yang et al.[102], given a fixed $\lambda > 0$, the function returns a new codec, where its R-D value lies on the R-D curve[1]. Thus, the λ hyperparameter controls visual quality and compression trade-offs. Commonly, models are trained for fixed and established λ values. Ballé et al.[12] defined the following λ values from lightest compression (best quality) to heaviest compression: [0.1800, 0.0932, 0.0483, 0.0250, 0.0130, 0.0067, 0.0035, 0.0018]. The λ values significantly impact the analysis and synthesis processes (Section 2.1.1). The analysis component utilizes non-linear transforms to convert the input into a latent representation. The synthesis component reconstructs the latent representation while minimizing distortion (Section 2.1.1). Compared to linear transforms, non-linear transforms are not limited to lattice-shaped quantization and can better adapt to the input[9, 10]. The Lagrangian multiplier controls both components. Higher λ values increase the loss for distortion, telling the model to focus more on quality during training. Lower λ values allow for higher distortion and lower rate, as seen in Equation (2.1).

$$L(\lambda, c) = \mathcal{R}(c) + \lambda\mathcal{D}(c) = \mathbb{E}[l(\mathbf{X})] + \lambda\mathbb{E}[\rho(\mathbf{X}, \hat{\mathbf{X}})]. \tag{2.1}$$

Rate measures the bits transmitted through the bitstream, directly determining the image's file size. Rate is commonly expressed using the bits-per-pixel (bpp) metric, which computes the average number of bits required to represent each pixel. This metric is calculated by dividing the total number of bits used by the total number of pixels of the reconstructed image.

Distortion quantifies the difference between the reconstructed and original images. The most common metric for measuring distortion is Mean Squared Error (MSE). However, Section 2.2 provides a detailed discussion of various metrics quantifying distortion (reconstruction error).

2.2 Metrics

Various metrics evaluate the quality or similarity of a given set of images. The most common and versatile metric is MSE. Over time, more specialized metrics were developed that focused on measuring perceptual quality.

2.2.1 Bjøntegaard Delta Rate

The Bjøntegaard Delta Rate [16] (BD-Rate) measures the compression performance of video codecs. This technique compares rate-distortion curves at various quality levels using PSNR and bpp value pairs. It also measures compression performance in the neural image compression field. The BD-Rate expresses the performance difference between the test and anchor codec. It measures the required bitrate to achieve the same quality level and outputs a percentage. A positive percentage means that the test codec requires more bits than the anchor codec to achieve the same quality level. Negative percentages imply that the test codec achieves the same quality level, requiring fewer bits than the anchor codec. BD-PSNR expresses the quality difference between both codecs at the

[1]The R-D curve is non-convex and typically only the local minimum is reached[102].

same bitrate. Positive BD-PSNR values suggest that the visual quality of the test codec is higher than the visual quality of the anchor codec at the same bitrate.

Barman et al. [15] conducted a study on the evolution and limitations of the BD-Rate metric, including a comparison of open-source implementations. Their research found that evaluating quality metrics other than PSNR may lead to unstable and insignificant results. The interpolation method varies between implementations, and the results may vary depending on the chosen method. Barman et al. found that the piece-wise interpolation mode (pchip) ensures the most stable results.

2.2.2 Reference-based vs No-Reference

Metrics used in image compression can be categorized as full reference-based, reduced reference-based, or no-reference [30, 18]. Full reference-based metrics compare the original and reconstructed images to assess quality, while reduced reference-based metrics evaluate the reconstructed images using only partial information from the original [30]. No-reference metrics operate solely on the reconstructed images without referencing the original [18]. Our work focuses exclusively on full reference-based metrics, such as Mean Squared Error (MSE), Multi-Scale Structural Similarity Index (MS-SSIM), Peak Signal-to-Noise Ratio (PSNR), and Learned Perceptual Image Patch Similarity (LPIPS).

2.2.3 Mean Squared Error

MSE is the most commonly used full-reference metric in learned image compression. Its simplicity and computational efficiency make it a highly performant metric [97]. Given the original image x and reconstructed image y, the MSE is calculated as the mean of the squared differences between the corresponding pixels of the two images, as seen in Equation (2.2). The goal is typically to minimize the MSE value, i.e., loss. However, MSE is not recommended for inference, as it poorly correlates with the human perception of visual quality [40, 98, 17, 103].

$$MSE(x, y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2 \tag{2.2}$$

2.2.4 Peak Signal-to-Noise Ratio

PSNR is derived from metric [98] and is defined in Equation (2.3). It does not provide additional information beyond MSE. However, it allows for comparing images with different intensity levels by accounting for dynamic range [98]. This work measures PSNR as:

$$PSNR = 10\log_{10}\frac{L^2}{MSE} \tag{2.3}$$

where L is the maximum signal value (e.g., $255 = 2^8$ for 8-bit images). The above PSNR definition scales logarithmically using a decibel (dB) scale. The dB growth can be interpreted as the reduction of the error signal. The logarithmic scaling of PSNR tends to flatten at higher dB-values. Reducing error at high dB-values becomes significantly more challenging. For instance, an improvement of 10 dB implies an error reduction by a factor of 10. Achieving a further 10 dB of improvement requires the same error reduction in addition to the previous one.

2.2.5 Multi-Scale Structural Similarity Index Measure

MS-SSIM is a metric first introduced by Wang et al. [96]. They argue that the human visual system (HVS) is sensitive to structural similarity in images [98], and it has been shown that the Structural Similarity Index Measure (SSIM) correlates better with perceived visual quality than MSE [96, 98]. However, SSIM still correlates poorly with visual quality [17]. Wang et al. initially developed the SSIM using a single-scale approach [96] but later improved upon it by incorporating multiple scales, each weighted according to its relative importance [97]. Given the original and reconstructed images, MS-SSIM begins at scale=1 and compares the structure and contrast of the images. The images are then iteratively downsampled by a factor of 2, affecting the detail and structure, before being compared again. This process is repeated until the set number of scales has been reached.

2.2.6 Learned Perceptual Image Patch Similarity

Traditional pixel-wise metrics, such as PSNR and MSE, do not accurately reflect human perception [40, 98, 17, 103]. Zhang et al. [103] introduced the Learned Perceptual Image Patch Similarity (LPIPS) metric, which utilizes a deep learning network to measure the similarity between two images. It surpasses the well-known metrics, such as PSNR, MSE, and MS-SSIM, in mimicking human perception and aligning with human judgment. The input images are expected to be normalized for the [-1, 1] range. The output is the similarity of both images, where lower is better and 0 means identical images. We set the network to $net = vgg$ for LPIPS evaluations in Chapter 7.

CHAPTER 3
Related Work

This section outlines the currently available approaches and related work which address similar topics.

3.1 Benchmarking Tools

CompressAI [20] is an open-source repository based on PyTorch that serves as a platform for end-to-end compression research. It provides pre-trained models and evaluation scripts and has partially ported the TensorFlow compression library to PyTorch. It is commonly used for its valuable implementations of standard functions. However, this platform does not meet our previously outlined requirements in Section 1.4 and chapters 4 and 6. Our objective is to develop a benchmarking tool that measures the performance of given models with well-known metrics. Our highly configurable tool offers a simple interface to replace components and quickly measure changes. Thus supporting the researcher in identifying components with the highest impact.

3.2 Literature Reviews

3.2.1 Learning-driven lossy image compression: A comprehensive survey

The survey by Jamil et al. [49] provides a general taxonomy of machine learning, which, in contrast to our taxonomy, does not focus on individual components of LIC. The publication strongly emphasizes architecture and measures the most common types, such as Autoencoders, Convolutional Neural Networks, and Principal Component Analysis. Conversely, our work argues that the components of an architecture, rather than the architecture itself, as classification on a component level permits a more fine-grained assessment of the significance of proposed changes.

3.2.2 Learning End-to-End Lossy Image Compression: A Benchmark

This work by Hu et al. [46] focuses on reviewing literature and benchmarking end-to-end learned image compression methods. It summarizes significant contributions up to 2021. In addition, it identifies issues with existing methods and addresses them by proposing a coarse-to-fine hyperprior framework.

3.2.3 Deep Architectures for Image Compression: A Critical Review

Mishra et al. [76] reviewed over 100 state-of-the-art (SOTA) publications. Contrary to our work, their review focuses on architectures in learned image compression, such as CNNs, GANs, RNNs, AEs, and VAEs. They conducted a comparative analysis of various models and traditional codecs such as JPEG and BPG. Recent advancements were summarized and visually presented on a timeline. In conclusion, they found that the joint rate-distortion optimization problem, which is present in traditional codecs, has not been fully solved with the utilization of deep neural networks.

3.2.4 Unveiling the Future of Human and Machine Coding: A Survey of End-to-End Learned Image Compression

Huang et al. [47] surveyed learned image compression codecs. They provide a high-level view of the JPEG compression flow and explain which components lend themselves to be optimized using neural networks. The survey summarizes previously conducted surveys and benchmarks, outlining their contributions. Huang et al. outline advancements in learned image compression, such as context models, transformers, attention modules, quantization, and generative adversarial networks. Their discussion points out the open challenges in learned image compression, such as computational complexity and subjective image assessment. In addition to reviewing MPEG Video Coding for Machines (VCM) and JPEG AI, the authors concluded by pointing out how much more flexible learned compression is compared to current traditional codecs.

3.2.5 Image compression with Neural Networks - A survey

Jiang et al.[50] published a survey regarding image compression with neural networks in 1999. They categorized the appliance of neural networks into three categories: 1) Direct Development, which focused on creating new codecs using neural networks, 2) Re-implementation of traditional codecs using neural networks, 3) Indirect application of neural networks to enhance existing pipelines. They used the standard metrics (described in Section 2.2), such as bits-per-pixel, Peak-Signal-to-Noise Ratio, and (normalized) Mean-Squared Error, to measure the compression performance and quality of reconstructed images. At the time, the goal was to match the performance of traditional codecs.

CHAPTER 4

Methodology

This section outlines the methods for achieving expected results and answering the research questions.

4.1 Literature Survey

The literature survey can be broken down into multiple steps:

4.1.1 Database Search.

The initial step is identifying critical publications based on selected keywords. The following terms can be derived from the aim of our thesis (Section 1.4): image compression, lossy compression, neural compression, and learned image compression. Our materials are primarily sourced from arXiv, IEEE Xplore, and Google Scholar databases. Publications with significant impact will be labeled foundational and will lay the groundwork for this thesis and subsequent steps of its literature survey. Additionally, arXiv provides tools for finding related publications called Influence Flower [80], Connected Papers [32], and CORE Recommender [57], which can be considered.

4.1.2 Backward and Forward Snowballing

Following the initial database search, we manually apply a twofold snowballing technique to find related publications. Backward snowballing involves collecting all materials cited by the publication currently under review; conversely, forward snowballing focuses on gathering materials that have cited the publication in question. We can gather a comprehensive amount of related papers by applying those techniques to all material from the initial step.

4.1.3 Screening and Selection

This step focuses on refining the gathered material by eliminating duplicates and irrelevant publications that do not contribute to the topic of lossy or learned image compression.

4.2 Taxonomy

The thesis introduces a comprehensive overview of work in learned image compression and non-linear transform coding. Our taxonomy categorizes works based on their primary components and contributions to the field of LIC, such as quantization and entropy models.

4.3 Benchmarking Tool

The findings and insights of our component-based taxonomy are further utilized in our benchmarking tool, Benchpresso. This tool verifies coarse-grained design decisions of notable works in neural image compression to establish a baseline. With an established baseline, we focus on the finer-grained design decisions and measure their performance. Our tests individually benchmark the significance of overlooked components and determine their impact on the model.

CHAPTER 5

Taxonomy

This chapter describes the classifications of our taxonomy, which are instrumental in the field of lossy LIC.

We differentiate the following categories in our taxonomy by identifying top-level components that pose cornerstones of the modern LIC paradigm:

- **Context Model:** computes probabilities and predicts the output image by sequentially decoding latents.
- **Architectural Archetypes:** break down and reconstruct images through analysis and synthesis transforms.
- **Side Information** provide additional context in encoding and causal modeling
- **Quantization:** reduces the image's continuous values into a discrete form, introducing error.
- **Variable Rate** specializes in variable quality settings without needing separate models for each quality setting.

Notably, we derive the interface of our benchmarking tool (Chapter 6) based on the top-level components. We systematically categorized and tagged the appropriate major components of publications derived in Section 4.1. Moreover, we introduced subcategories to accommodate varying properties, such as serial versus parallel processing in learned image compression or scalar versus vector quantization, which provides a finer granularity and cohesive structure.

5.1 Quantization

Quantization is a fundamental process in LIC. It approximates continuous values with a finite set of discrete symbols, reducing the amount of data and introducing error. We differentiate two major quantization types, scalar and vector quantization (Figure 5.1 and table 5.1).

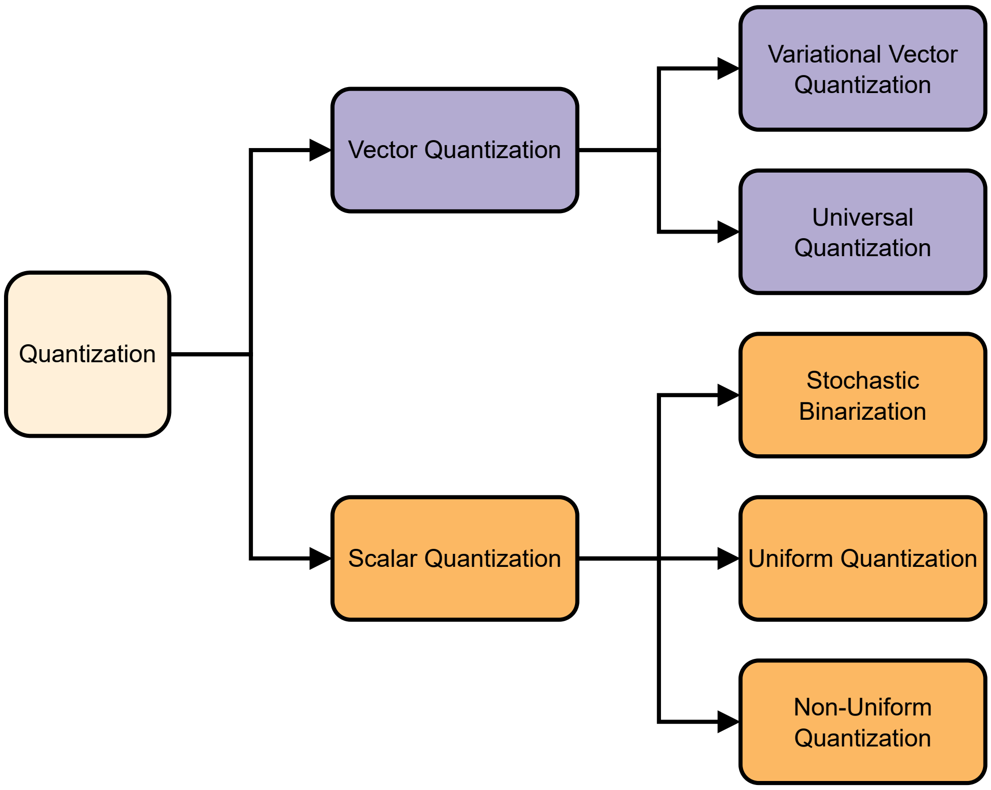

Figure 5.1: Quantization classification for learned lossy image compression.

Quantization Categories	**Related Papers**
Vector Quantization	[1, 3, 26, 107, 10, 105]
Scalar Quantization	[88, 33, 9, 13, 10, 74, 75, 43, 58, 77, 21, 85]

Table 5.1: Taxonomy of quantization methods in learned image compression.

5.1.1 Scalar Quantization

Scalar Quantization in LIC transforms each pixel of an image to its scalar representation. It is commonly employed due to its technical simplicity and computational efficiency.

Uniform Quantization [9] is the most common form of quantization in lossy learned image compression. It has been utilized in several publications by Ballé et al. [9, 13, 10], Minnen et al. [74, 75], He et al. [43], Lee et al. [58], and Qian et al. [77]. In Uniform Quantization, continuous values are divided into equal-sized bins during training and rounded to the nearest integer. However, the rounding operation is non-differentiable, which poses a challenge for gradient-based optimization methods. To address this, Ballé et al. introduced additive uniform noise. They restore differentiability for the backpropagation process by simulating the quantization error with uniformly distributed additive noise.

Non-Uniform Quantization is a form of scalar quantization. The authors [21] address the inherent challenge of non-differentiability in quantization differently. Unlike uniform quantization [9], which applies the same quantization step across all values, non-uniform quantization updates itself based on the data distribution. Cai et al. [21] introduce an iterative optimization process that alternates between fixing the quantizer and the entropy model for optimization. The network is initially trained without the quantizer to minimize the MSE loss. The subsequent iteration fixes the network and aims to optimize the quantizer to reduce the quantization errors. According to the authors, this attempt showed overall improved compression efficiency.

Stochastic Binarization [85] introduces a probabilistic approach to quantization by transforming real-valued inputs into binary values, typically (-1,1). Unlike its deterministic counterparts, values are quantized based on the probability derived from the input, which may simulate noise. The challenge of non-differentiability in gradient-based optimization applies here as well. To address this issue, Toderici et al. [85] employ the straight-through gradient estimate [88, 33], which allows gradients to pass through unchanged. This approach mitigates the problem of gradients being zero or infinite almost everywhere, thereby allowing end-to-end optimization.

5.1.2 Vector Quantization

Vector Quantization in learned image compression involves segmenting an image into blocks of pixels as multi-dimensional vectors. Given a vector pattern, these vectors are approximated to the closest quantization bin in the latent space.

Soft-to-hard Vector Quantization [1] is an approach in quantization that anneals soft assignments to hard assignments during the training process. The key benefit of soft assignments lies in their differentiability, which facilitates end-to-end gradient-based optimization. The initial annealing rate σ estimates a probability distribution of the input value. Soft quantization assigns the value to multiple bins based on the estimated probability distribution. The authors systematically increase the σ-value, approaching infinity, to eventually focus the probability distribution on a single assignment, thus transitioning to a hard assignment. Agusstson et al. [1] warn about the annealing rate. Slow and rapid annealing rate may cause inflated weights or the vanishing gradients issue. Instead, they recommend parametrizing the σ-value or making it dependent on other values within the model.

Universal Quantization [3, 26] , also referred to as *dithered quantization*, is a form of quantization with a random offset [3], which aims to be universally applicable at training and test time. While Choi et al. [26] implemented universal quantization, Agustsson et al. [3] improved upon their work by eliminating the need for straight-through gradient estimates. Both successfully eliminated the training-test time mismatch and achieved end-to-end gradient optimization. Based on their previous work, soft-to-hard vector quantization in Section 5.1.2, Agustsson et al. combine soft rounding with additive uniform noise by Ballé et al. [9], to simulate quantization and use universal quantization [107] to communicate samples during training and test time and retain differentiability.

Variational Vector Quantization

Variational Entropy Constrained Vector Quantization is a vector quantization method based on non-linear transformations. Traditionally, vector quantization is computationally infeasible for multi-dimensional sources [10]. Transform coding simplified the problem by mapping the vectors into a latent space using linear transforms, which are constrained to simple transformations of that space. Ballé et al. utilized non-linear transform coding to adapt its quantization bins more closely to the source data. Stochastic gradient descent suffers from zero values almost everywhere and was avoided by employing dithered quantization and additive uniform noise [10].

Unified Multivariate Gaussian Mixture introduced by Zhu et al. [105] is a probabilistic vector quantization method with cascaded estimation. They identified redundancies in the inter- and intra-correlations in the latent representations. This novel vector quantization approach estimates the means and variances for the multivariate Gaussian mixture without utilizing a context model. Additionally, they found that using a single codebook was computationally too complex and introduced a multi-codebook approach instead. Their results and ablation study showed promising B-D rate improvements.

5.2 Architectural Archetypes

Analysis and synthesis transforms are the primary transforms in lossy image compression. Before the quantization step, the image must be deconstructed into a set of coefficients using an analysis transform. Conversely, the synthesis transform is used to reconstruct the image from the quantized coefficients.

Architectural Archetypes	**Related Papers**
Hierarchical Transformer	[71, 108, 106, 69, 68, 51, 56, 62, 55, 23, 7, 64, 34]
Non-linearity	[11, 12, 52, 78]
Convolutional	[35, 29, 92, 70, 106]

Table 5.2: Taxonomy of architectural archetypes in learned image compression.

A classic example of such transforms is the discrete cosine transform (DCT) and its inverse (IDCT), used in the JPEG image compression standard. It is a simple and efficient transform, limited to linear transformations. Such limitation influences the compression performance on complex images and introduces compression artifacts.

Convolutional Neural Networks apply filters to the input and create convolutions, which are further filtered [102]. They excel in feature extraction and image processing and are capable of approximating functions. CNNs outperformed [8, 59] handcrafted codecs, and their use reached 32% of all architectures for still image compression in 2022 [48]. Ballé et al. [12] helped CNN architectures achieve higher r-d performance by replacing the ReLU activation function with GDN and introducing the NTC [10] approach.

Further improvements include the introduction of deformable convolutions, which can shift attention to various details in the image. Wang et al. [92] described a promising new strategy for kernels, which introduces variable spatial ranges that are content-aware and deformable while adjusting their weights dynamically.

The constantly increasing depth of neural networks proved training to be challenging. The residual bottleneck block architecture mitigates the vanishing gradient problem. Skip connections bypass the depth and are directly combined with the output, enabling more profound and robust networks. We classify these approaches as *convolutional* (Table 5.2).

Other architectural archetypes include transformer-based models. Their general use involves classification and vision tasks, but they also found their way into learned image compression. We classify models predominantly consisting of self-attention or other transformer-like components as transformer-based.

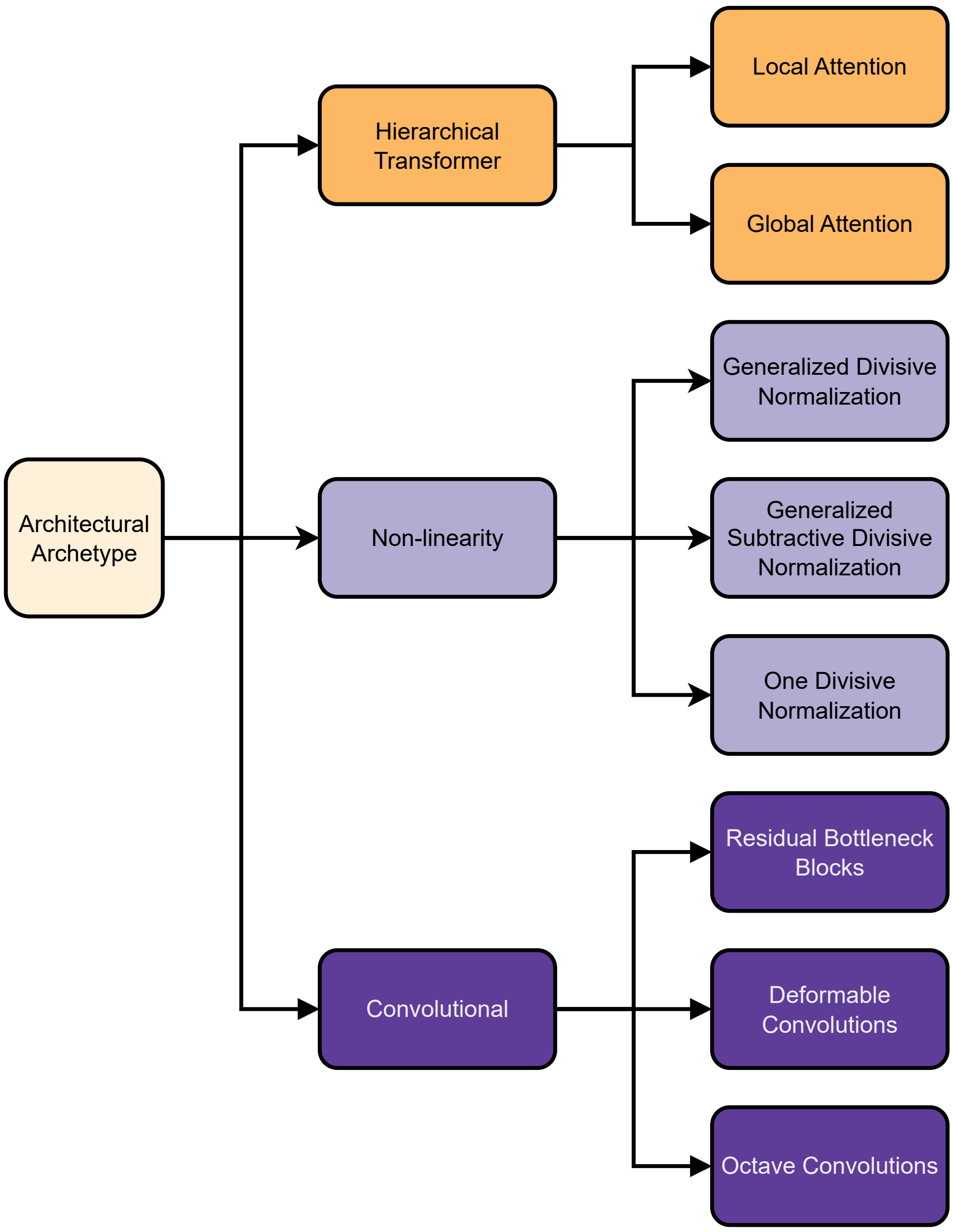

Figure 5.2: Architectural archetype classification for learned lossy image compression.

5.2.1 Hierarchical Transformer

Hierarchical transformers leverage a layered architecture that is capable of multi-scale processing. Initially used primarily in Natural Language Processing (NLP), they have since been adapted for high-level vision tasks and, more recently, low-level tasks such as LIC [71, 108, 106, 68]. Transformers start with high-resolution input and progressively reduce the resolution while increasing the number of extracted features. The feature extraction mechanism, or Attention, varies in implementation and technique and includes approaches such as Shifted Windows Transformer Attention, Residual Neighborhood Attention, or Convolutional Neural Networks.

Local Attention

Shifting-window-based Attention is an attention mechanism based on shifted windows [71, 108, 106, 69, 68, 51]. Swin splits the image into non-overlapping windows but can also do cross-window referencing. Zou et al. [108] developed a plug-and-play *window-based attention module* (WAM), which improved the R-D performance for learned image compression in Transformer- and CNN-based architectures. Their attention module was able to distribute bits unevenly, focusing more on regions of high contrast. Employing Swin-transformer-based Attention with non-linear transformations improved performance and shorter decoding times than CNN transforms [108, 106]. Liu et al. [68] showed in their ablation study that their *Transformer-CNN Mixture* (Section 5.2.1), which includes Swin Attention, resulted in significant performance gains over Transformer-only and CNN-only architectures.

Transformer-CNN Mixture is a combination of Transformer-based and CNN-based models that benefit from both strengths. CNNs are limited in modeling long-range dependencies [68, 56, 62]. However, Transformers excel in capturing global dependencies [56, 108]. We classify models as Transformer-CNN mixtures when they combine Transformer- and CNN-based characteristics.

Liu et al. [68] combined CNN's local modeling ability with a transformer's long-distance capturing ability. Their TCM Block splits the input tensor evenly and computes local and non-local features independently and in parallel. The outputs are concatenated into one tensor. This process is repeated in stages one and two. While the first stage employs Window-Based Multi-Head Self-Attention (W-MSA), the second stage uses Shifted W-MSA.

Khoshkhaht et al. [55] proposed a transformer-based non-linear transform, which utilizes two CNN-based hyperpriors and captures local and global information efficiently. Kim et al. [56] utilized a novel Transformer-based entropy model that combines a global and local hyperprior and outperforms existing entropy models regarding R-D performance. Both the entropy model and the local hyperprior consist of convolutional layers. Since the Informer utilized an autoregressive prior, the decoding process cannot be parallelized. Employing a channel-wise autoregressive [75] model would mitigate this limitation [55].

Residual Neighborhood Attention is classified as an archetype of residual block architecture, including attention mechanisms (Figure 5.2). Lu et al. [70] utilized Neighborhood Attention Transformer (NAT) [41] by Hassani et al. Before NAT, transformer-based attention mechanisms were considered unfeasible for low-level tasks such as compression[41]. Following the work [44], Lu et al. improved training and information aggregation by adding residual skip connections.

Global Attention

Khoshkhahtinat et al. [55] claims that SwinT's receptive field is insufficient to capture global information. Instead, they propose utilizing a hierarchical aggregated-windows transformer (AGWinT) to extract global dependencies. AGWinT extracts global query patches and matches them against local patches extracted by the local attention mechanism. Additionally, they employ a global context block based on an autoregressive model to exploit global context information context-dependently.

Kim et al. [56] introduced a novel entropy model based on the transformer architecture. The goal is to jointly model global and local dependencies using the Transformer's ability to capture long-range dependencies. They propose an Information Transformer (Informer), which uses a global hyperprior with a cross-attention mechanism to avoid the quadratic complexity of self-attention. Global dependencies are modeled in a context-dependent manner. However, the global hyperprior lacks spatial information provided by another local hyperprior.

5.2.2 Non-linearity

Generalized Divisive Normalization (GDN)

GDN is a non-linear transform used to gaussianize the local joint statistics. It was initially introduced by Ballé et al. [11] and works by normalizing pixel values, which account for neighboring pixels, to mimic the Gaussian distribution and make them statistically independent. Typical analysis and synthesis transforms have fixed parameters after training. In contrast, GDN is spatially adaptive and highly non-linear [12]. Its formula can be found in Equation (5.1).

$$z_i = \frac{x_i}{\left(\beta_i + \sum_j \gamma_{ij} \left|x_j\right|^{\alpha_{ij}}\right)^{\varepsilon_i}} \tag{5.1}$$

One Divisive Normalization (1DN)

GDN is a commonly used non-linear transform. Its significant performance comes at a price of high computational complexity. Johnston et al. [52] reduced the run-time of GDN by simplifying the formula in Equation (5.2). The authors fixed parameters to eliminate the need for square-root computation without performance degradation. They proposed One Divisive Normalization (1DN), a simplified version of GDN, resulting in

21.4% savings on normalization computations and over 50% of decoder run-time for state-of-the-art models at that time.

$$z_i = \frac{x_i}{\beta_i + \sum_j \gamma_{ij} |x_j|} \tag{5.2}$$

Generalized Subtractive Divisive Normalization

Qian et al. [78] argue that GDN suffers from a mean-shifting problem since its density should be zero-mean. They introduced Generalized Subtractive Divisive Normalization (GSDN), which aims to fix the issue by subtracting the mean from all values. This non-linearity component (Figure 5.2) improved their rate savings by about 2%.

5.2.3 Convolutional

Deformable Convolutions

Deformable Convolutions modify the standard convolution operation by allowing the shape of the convolution kernel to adapt dynamically to the input. This adaptation is typically achieved through an offset field that adjusts the kernel's receptive field [35, 29, 92]. There are subtypes of deformable convolutions, such as Deformable Residual Modules (DRM) and Lite Deformable Convolutions (LDCN). Deformable convolutions that incorporate residual block mechanisms are classified as DRMs [35, 29]. Meanwhile, simplified deformable convolutions that share kernel weights or apply softmax normalization are classified as LDCN [92].

Lite Deformable Convolution Deformable convolutions are similar in complexity to non-deformable convolutions [29]. Their complexity may be reduced without performance degradation. Wang et al. [92] introduced an LDCN that splits the input feature into groups and has the sharing modulated kernel weights within each group. They utilize softmax normalization to improve the training performance further.

Deformable Residual Module DRMs were first proposed by Dai et al. [29] and integrated into a LIC framework by Fu et al. [35]. The kernel's offset field is typically the size of the input feature map. Combining a classical convolution and the deformable module requires a skip connection inspired by the residual block mechanism. Modules in this group are highly compatible and can be used as a replacement for typical analysis and synthesis transforms. Due to their compatibility, we classify them as Deformable Convolutions and Residual Bottleneck Blocks (Figure 5.2).

Residual Blocks

Residual Blocks [44] allow layers to learn modifications to the identity mapping and improve the gradient flow. They utilize shortcut connections to skip through one or multiple layers and prevent degradation in deep neural networks. These shortcut connections are present in most modern architectures, from CNN to Transformers[89]. We classify works that use the original bottleneck blocks [44] with shortcut connections to this residual block subcategory.

Residual Neighborhood Attention Lu et al. [70] introduced a neighborhood attention mechanism for local attention that utilizes residual skip connections. This flexible mechanism is compatible with other window-based self-attention mechanisms [70] like SwinT [106]. Skip connections are necessary to facilitate better information aggregation and model training [70]. Section 5.2.1 describes further details on local attention in hierarchical transformers.

Octave Convolutions

Chen et al. [23] first introduced Octave Convolutions for computer vision tasks. They were motivated by areas of low detail (global structures) exhibiting low frequencies, while areas of high detail and lots of change exhibit high frequencies. They formulated an octave convolution (OctConv), which separated the feature maps into low and high frequencies [1] and at different resolutions, allowing them to be stored and processed efficiently.

Akbari et al. [7] observed that the work [23] used fixed interpolation methods in their octave convolutions, such as average pooling and nearest neighbor interpolation. They stated that these methods do not preserve spatial information and structure, which are not necessary for object detection. However, they developed a novel generalized octave framework to make OctConv suitable for LIC. Instead of fixed interpolation methods, they utilized learnable internal activation layers. Their proposed Generalized Octave Convolution (GoConv) effectively preserves the spatial structure and enhances the R-D performance [7, 64]. They also used the parameter estimator [74] to combine information for the context model and hyper-decoder.

Fu et al. [34] forgoes using an element-wise context-adaptive entropy model to avoid sequential decoding of elements. To compensate for the performance loss, they split image representations into LR and HR parts. Utilizing GoConv [7] in their hyper-encoder and decoder layers further eliminates spatial redundancies. They also introduce cross-resolution parameter estimation (CRPE), which combines LR parts from different layers and provides this information to decode elements. Avoiding the entropy model enables parallel decoding and accelerates decoding times by at least 73% while outperforming the VVC(4:2:0) codec.

5.3 Context Models

Context Models are an essential part of LIC. They utilize entropy-coded latents to improve R-D performance. They are typically sequential, but recent works in Table 5.3

[1] May be referred to as High Resolution (HR) and Low Resolution (LR) components.

show parallelizable variants. Figure 5.3 visualizes their categorization.

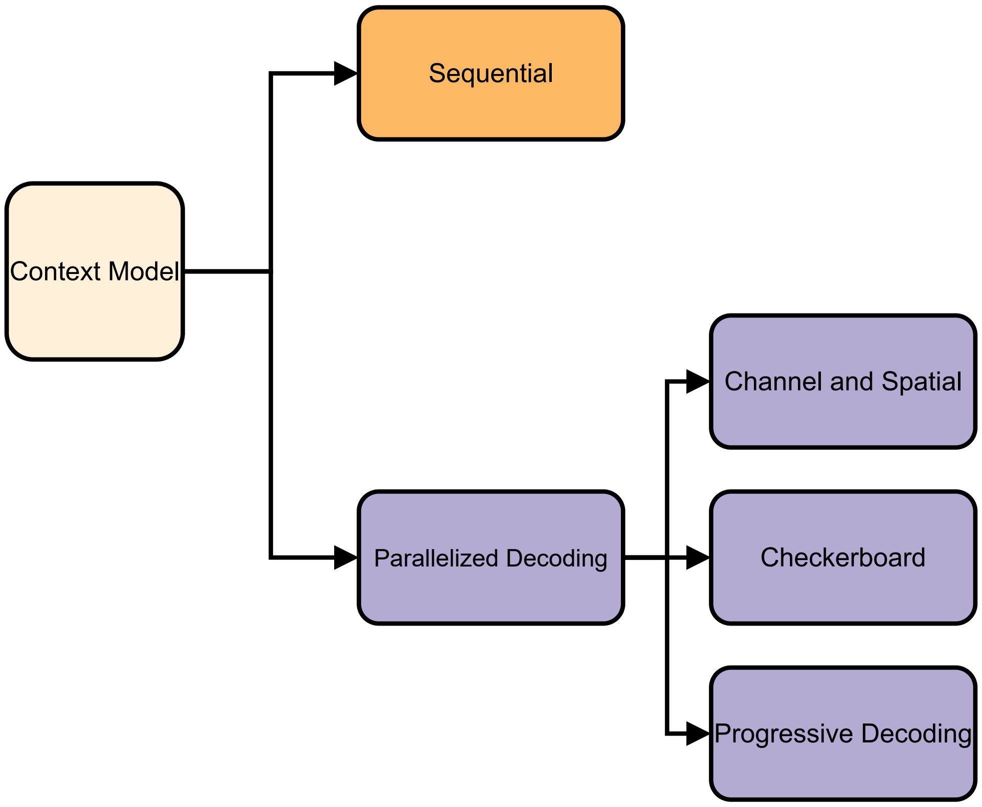

Figure 5.3: Context model classification for learned lossy image compression.

5.3.1 Sequential

This category encompasses partially and completely sequential models, i.e., models that do not focus on parallelization and utilize some form of context modeling.

Autoregressive priors have been successful in generative models [74]. Minnen et al. [74] examined their application in LIC. Their autoregressive model relied on the side information provided by a hyperprior. However, the decoding process depends on previously decoded pixels, so autoregressive models are inherently sequential [75, 102, 56]. Section 5.3.2 describes how Minnen et al. managed to parallelize the process.

Context-Adaptive models cannot be parallelized [56, 58, 34]. Lee et al. [58] introduced a context-adaptive model that splits the contexts into bit-free and bit-consuming ones. This distinction allowed them to allocate bits only where necessary, reducing spatial dependency. However, they utilize already decoded latents autoregressively to decode the bit-free context, which hinders parallelization.

5.3.2 Parallel Decoding

Checkerboard Context Model

He et al. [43] introduced the Checkerboard Context Model (CCM), a novel approach to parallel decoding. They proposed splitting the latent representation into anchors and non-anchors to facilitate decoding in two stages. Initially, the anchors are decoded in parallel. Subsequently, the non-anchor points are decoded in parallel using decoded anchors (context features) and hyperprior features. Many works [77, 43, 35, 92, 95, 42] utilize this two-stage, parallelizable checkerboard context model.

Fu et al. [35] noted a 0.2-0.3 dB decrease in R-D performance with the original CCM [43]. This issue was addressed by utilizing different probability distributions for anchors and non-anchors. They utilized the Gaussian-Laplacian Logistic Mixture Model (GLLMM) for anchor points and the Gaussian Mixture Model (GMM) the non-anchors to regain performance.

Jiang et al. [51] also observed a 4% performance degradation in the original CCM [43]. They enhanced the model by stacking checkerboards and utilizing shifted window-based checkerboard attention to address this issue. Stacked checkerboards also increase the non-linearity depth [51] while retaining the two-pass decode.

Wang et al. [92] divided their latent representation into slices. They applied the CCM to each slice for parallel decoding and achieved a BD-Rate of -7.68% over BPG(4:4:4) on the Kodak dataset.

He et al. [42] applied the CMM [43] pattern in combination with non-uniform channel grouping using an uneven grouping scheme to improve efficiency further. They allocated fewer channels in the early slices, resulting in fewer symbols that can be processed in parallel. However, they allocated more channels in the later slices to increase parallelization. This novel grouping scheme yields improved BD-Rate performance and lower latency. They argue that the later encoded channels contain less information and place higher emphasis on the early channels to reduce cross-group reference.

Similarly, Lu et al. [70] were inspired by the work [42] and adopted a similar grouping technique where the number of channels increases with each slice. However, instead of using the handcrafted slicing technique [42], they opted for cosine slicing, which performs better than linear slicing. In 3 of their four slices, they use the Generalized Checkerboard Pattern with variable granularity to group spatial neighbors.

Progressive Decoding

Progressive Decoding is the ability to split the complete decoding process into stages. We define it as the ability to preview an image without fully decoding it, which is highly practical for real-world applications. Progressive decoding is assumed to take only a fraction of the time compared to fully decoding an image. Prior works include those by Minnen et al. [75] and He et al. [42]

The Channel-wise Autoregressive Entropy Model by Minnen et al. [75] naturally supports progressive decoding, although this was not their primary goal. Their model first utilizes information from the hyperprior to generate an initial image. The missing information is inferred from the distribution of the hyperprior, and then the previously sliced latents are progressively added, increasing detail with each slice. Minnen et al. [75] highlight two crucial drawbacks. First, the R-D performance is much worse than standalone models tailored for low bitrates, making progressive decoding an infeasible solution for variable rate decoding. Second, their progressive decoding still requires the complete synthesis transform, which is computationally costly.

He et al. [42] addressed these issues by introducing the unevenly grouped channel-wise context model (described in Section 5.3.2). According to their claims, early channels contain most information, which lends itself to progressive decoding. To address the decoding speed issue, He et al. propose training a separate tiny network called a thumbnail synthesizer, which decodes images in microseconds thanks to its lightweight structure [42].

Channel and Spatial

We classify context models as "Channel and Spatial" that parallelized decoding by leveraging channel-wise or spatial-wise conditionals. This includes but is not limited to, slicing along the channel axis or using custom decoding orders that divide the latent space into independent and, therefore, parallelizable patches.

Channel-wise Autoregressive Context Model [74] Autoregressive models are inherently serial [74]. Minnen et al. [75] aimed to minimize serial processing while maintaining matching R-D performance in forward and backward adaptation. Forward adaptation makes use of side information, e.g., hyperpriors. Backward adaptation predicts already decoded patients, which are usually processed serially. They minimized serial processing by utilizing Latent Residual Predictions (LRP) and Channel-Conditioning (CC). The latent representation of an image is sliced along the channel axis instead of the spatial axis, and each slice can be processed in parallel. Their experiments improved encoding and decoding speeds compared to the checkerboard pattern approach [43] while maintaining better R-D performance than state-of-the-art codecs.

Fu et al. [36] utilize the channel-wise autoregressive model by Minnen et al. [75] employing a dual-branch encoder architecture. Two encoders process the input image: one handles high-resolution input to capture global dependencies and is fed as side information to the second encoder, which handles low-resolution input. Each encoder is processed by its channel-wise autoregressive context model (ChARM) separately and in parallel. Fu et al. [36] experimented with slicing the latent representation into 5 and 10 groups and found that more groups offer only slightly improved R-D performance.

Multistage Spatial Context Model [63] Lin et al. [63] mentioned that Lu et al.'s [70] extension to the checkerboard context model has similarities to their solution but is limited by the decoding order. They also consider the checkerboard context model by He et al. [43] to be a two-stage context model, where half of the codes (anchors) are decoded in stage 0 and the other half (non-anchors) in stage 1. Anchors do not benefit from the context model's information [63]. Lin et al. [63] introduced more stages to reduce the proportion of anchors to non-anchors. They provided two examples, dividing the latent space into 2x2 and 4x4 patches, decoded in 4 and 16 stages, respectively. All patches in each stage can be decoded in parallel and utilize the information from previously decoded stages. They found their solution [63] superior in R-D performance and 11.8% to 27.6% slower than the original checkerboard context model [43].

Context Model Categories		**Related Papers**
Sequential		[74, 75, 102, 56, 58, 34]
Parallelized Decoding	Channel and Spatial	[74, 75, 43, 36, 63, 70, 61]
	Checkerboard	[43, 77, 35, 92, 95, 42, 51, 70]
	Progressive Decoding	[75, 42]

Table 5.3: Taxonomy of context models in learned image compression.

Scanning Order and Dividing Technique Li et al. [61] utilize a spatial context-based convolutional context model to introduce a code-dividing technique that enables parallel decoding. First, they analyzed various 2D coding techniques, such as raster coding, which hindered parallelization or reduced entropy estimation accuracy. They chose the zigzag technique and extended it to 3D by interpreting the latent as a cube and stages as diagonal planes [61, 63]. These diagonal planes contain codes that can be independently decoded in parallel. Their Context-based Convolutional Network (CCN) utilized the 3D Zigzag Coding Order and Dividing Technique. Its degree of parallelization (DoP) depends on an image's width, height, and number of channels.

5.4 Hierarchical Priors

Hierarchical Priors utilize multiple layers of probability models. The primary prior models the latent of the input image. Then, additional layers of priors are built on top of the primary probability model, further processing the latent, which is fed back during the encoding and decoding process. The probability modeling assumes the prior distribution prior to computing the primary distribution.

Hierarchical Priors Categories		**Related Papers**
Causal Context Modeling		[13, 58, 74, 77, 71, 70, 41]
Global Context Modeling	Attention Based Models	[22, 62, 66, 65, 72, 77, 94, 93, 74, 71]
	Joint Priors	[74, 12, 13, 78, 36, 101, 104, 25, 56, 58, 45]

Table 5.4: Taxonomy of hierarchical priors in learned image compression.

5.4.1 Local Context Modeling

CNNs efficiently model the local context through their architectural design of convolutional filters and small receptive fields. Their main limitations are capturing long-range dependencies and global context modeling. This section classifies publications focusing on local context modeling and capturing local dependencies within the latent representations.

In [13], they introduce a variational autoencoder with a scale hyperprior. The distribution scale is estimated by their input-adaptive entropy model. Lee et al. [58] extend this work and introduce a context-adaptive entropy model that handles two contexts: bit-free and bit-consuming. The bit-consuming context provides global information and is described in more detail in Section 5.4.2. The bit-free context focuses on modeling the local context, which is bit-free, as it is already known to the encoder-decoder.

Minnen et al. [74] also based their solution on Ballé et al.'s [13] hyperprior architecture. They extended the scale hyperprior with an autoregressive component, which can efficiently exploit local dependencies by predicting the current latent based on previously decoded latent.

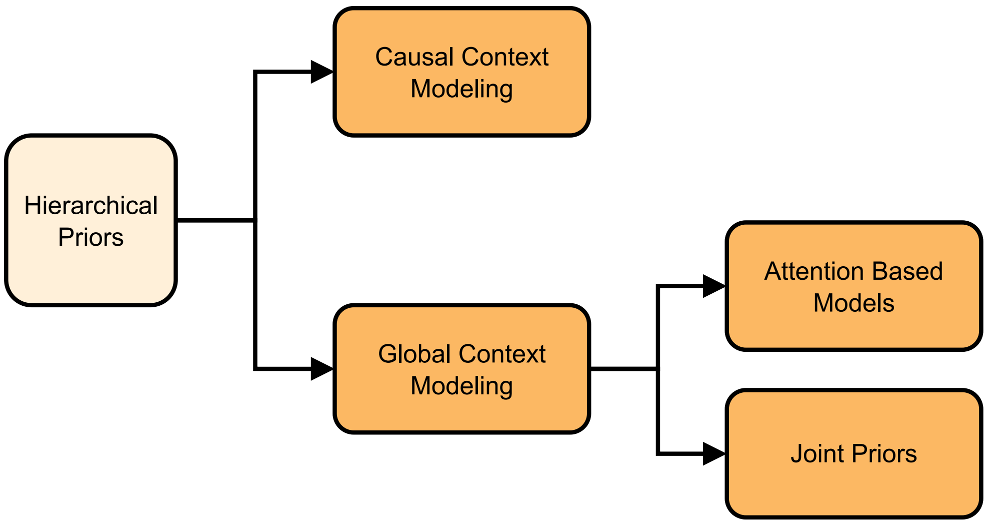

Figure 5.4: Hierarchical Priors classification for learned lossy image compression.

Qian et al. [77] utilized stacks of transformer layers as the autoregressive prior, with each layer including self-attention masks to learn the local context. Combined with a hyperprior, this predicted the Gaussian parameters, mean, and scale.

The causal attention module (CAM) by Lu et al. [71] is different from previous masked convolution-based mechanisms. In their Transformer-based Image Compression (TIC) architecture, they utilize Swin Transformer Blocks (STB) with convolutional layers jointly to extract both short—and long-range information. The combination of STB and convolutional layers is called a Neural Transformation Unit (NTU). CAM aggregates the information from NTUs using autoregressive neighbors with masked attention. These neighbors are then fused with the hyperpriors to predict the context.

Another notable work that utilizes self-attention for local context modeling is TinyLIC by Lu *et al* [70]. Their Integrated Convolution and Self-Attention (ICSA) units comprise Residual Neighborhood Attention Blocks (RNABs). The RNAB architecture can be implemented with window-based self-attention schemes, such as SwinT. However, Lu et al. [70] opted to use Neighborhood Attention Transformers (NAT) by Hassani et al. [41], as it provides a more flexible window with fewer irrelevant patches compared to SwinT. Both attention mechanisms are also mentioned in Section 5.2.1 about attention in hierarchical transformers.

5.4.2 Global Context Modeling

Attention Based Models

Non-local Attention Module (NLAM) [22, 62, 66] Chen et al. [22] argue that generating importance maps only at the bottleneck [65, 72] cannot handle complex regions effectively. Instead, they propose an attention module in the main encoder-decoder and the hyperprior branch. The Non-Local Attention Module (NLAM) is based on the Non-Local Network (NLN) by Liu et al. [65] and includes ResBlocks with ReLU activations. This NLAM architecture is identical to the NLAM architecture introduced 2 years prior by Liu et al. [66] Additionally, they produce multiscale attention masks, which are summed over all channels. [22] The key to their solution is applying the non-local attention module in both the primary and hyperprior branches to generate multiscale attention masks and exploit non-local correlations effectively.

Li et al. [62] also introduced a non-local attention block to exploit the global context. They propose a proxy similarity function to indirectly compute the similarity between the target code and other codes, which may be in different contexts. Additionally, they utilize a confidence score to omit dissimilar codes. The attention block combines this non-local information with the local representation to produce an output for entropy modeling.

Self-Attention Another approach that utilizes attention mechanisms in hierarchical priors is the Entroformer by Qian et al. [77] They introduce top-k self-attention for learned image compression, inspired by content-based sparse attention by Wang et al. [94, 93], which was used in vision tasks. The architecture is based on the work of Minnen et al. [74] and extended with a transformer-based entropy model. Additionally, the hyperprior consists of six transformer layers, each with a self-attention block. Qian et al. [77] tested multiple k-values for the top-k self-attention and found that the model performs best and converges quickly with k-values no larger than 64. The top-k scheme with large k-values may include too many irrelevant tokens [77], which worsens performance.

Swin Transformer-based Attention The Transformer-based Image Compression (TIC) model by Lu et al. [71] aims to utilize Swin Attention to capture both short-range and long-range dependencies. They construct Neural Transformation Units (NTUs), which consist of Swin Transformer Blocks (STB) combined with convolutional layers. These NTUs are present in both the encoder-decoder and the hyperprior branches, embedding long-range dependencies efficiently.

Joint Priors

Hierarchical Priors, combined with other priors jointly and utilized for global modeling, are classified as Joint Priors (Figure 5.4 and table 5.4).

Joint Autoregressive and Hierarchical Priors [74] After introducing one of the first end-to-end learned image compression models [12], Ballé et al. [13] extended their work by introducing a hierarchical prior to capture more information directly in the latent space. Their hyperprior became a well-known technique for capturing side information and improving R-D performance. Following this, a work by Minnen, Ballé, and Toderici [74] extended the previous scale-only hyperprior [13] to one that predicts both mean and scale. Additionally, they added an autoregressive prior, which jointly predicts the causal context.

Fu et al. [36] introduced a learned image compression model with two encoder branches, one for learning high-resolution latent and one for learning low-resolution latent. They utilize two Channel-Wise Autoregressive Models [74] (ChARM) for the output of the hyperprior branch. One ChARM is designed with larger filters to capture global information more efficiently, while the other uses smaller filters to capture local information. Furthermore, during the encoding and decoding of low-resolution latents, the information from the high-resolution latents is incorporated as conditional side information to enhance encoding and decoding efficiency and reduce redundancy.

Global Reference [78] Qian et al. [78] claim that the lack of global vision is a significant limitation and thus introduces a global reference model. Their solution extends the architecture by Minnen et al. [74] and adapts the entropy model to include a reference-based component. The global reference finds latent matching the current latent using already decoded latent. The search method is based on the works of Zheng et al. [104] and Yang et al. [101] However, Qian et al. [78] go beyond the similarity score by considering a confidence score to more accurately find relevant patents. The context model first estimates the entropy parameters and is then jointly updated by the global reference and hyperprior model.

Gaussian Mixture Models [25] Cheng et al. [25] observed some spatial redundancy in the latents in the work by Minnen et al. [74] They argue that a single Gaussian distribution is limited in shape and cannot fully represent the underlying distribution. To address this limitation, Cheng et al. [25] proposed a more flexible approach using a Gaussian mixture model (GMM), which achieved better spatial redundancy reduction.

Unified Multivariate Gaussian Mixture [105] Zhu et al. [105] argue that scalar values in univariate priors may be insufficient to capture inter- and intra-correlations. Inter-correlations are defined as visually and spatially similar regions, while intra-correlations are differences (covariances). The proposed method utilizes vectors instead of scalars to estimate means and covariances accurately without needing a context model. Their vectorized prior treats latents as vectors, improving compression speeds and reducing complexity. Their vector quantization method is described in Section 5.1.2.

Information Transformer [56] Kim et al. [56] found that the global reference model by Qian et al. [78] does not fully utilize global references, as it only finds one latent that matches the target latent. They improve upon this issue by introducing a global hyperprior, which extracts information from all the latent instead. Additionally, the solution by Qian et al. [78] has quadratic complexity, while the global hyperprior utilizes a cross-attention mechanism with fixed complexity regardless of image size. Kim et al. [56] introduce an Information Transformer (Informer), which jointly utilizes local and global hyperpriors and reduces computational complexity.

Bit-free and bit-consuming contexts [58] Lee et al. [58] introduced a context-adaptive entropy model that distinguishes between bit-free and bit-consuming contexts. The bit-free context does not require additional bit allocation and is usually included in the local context. Their work builds on the scale hyperprior by Ballé et al. [13], which uses the hyperprior as a bit-consuming context. The proposed framework adaptively decides whether the given information is bit-free (i.e., already known to both the encoder and decoder) or requires additional bit allocation.

Multi-layered Hyperpriors Hu et al. [45] proposed a coarse-to-fine framework that utilizes multiple hyperprior layers to eliminate spatial redundancy efficiently. They introduced a signal-preserving hypertransform designed to reduce spatial redundancy while preserving information. Hu et al. [45] used relatively small filters and 1x1 convolutions to achieve this, as large filters rely on local correlations. Additionally, they exploited the space-to-depth reshaping to place spatially adjacent elements in the exact location after reshaping. These novel solutions allowed them to stack multiple hyperlayers and better approximate the joint distribution.

5.5 Variable Rate

This category focuses on publications (Table 5.5) that aim to introduce variable rate models. Their characteristic is that they do not require separate models for each quality level. Instead, a single model can achieve variable target rates through scaling, modulation, adaptation, or multi-resolution.

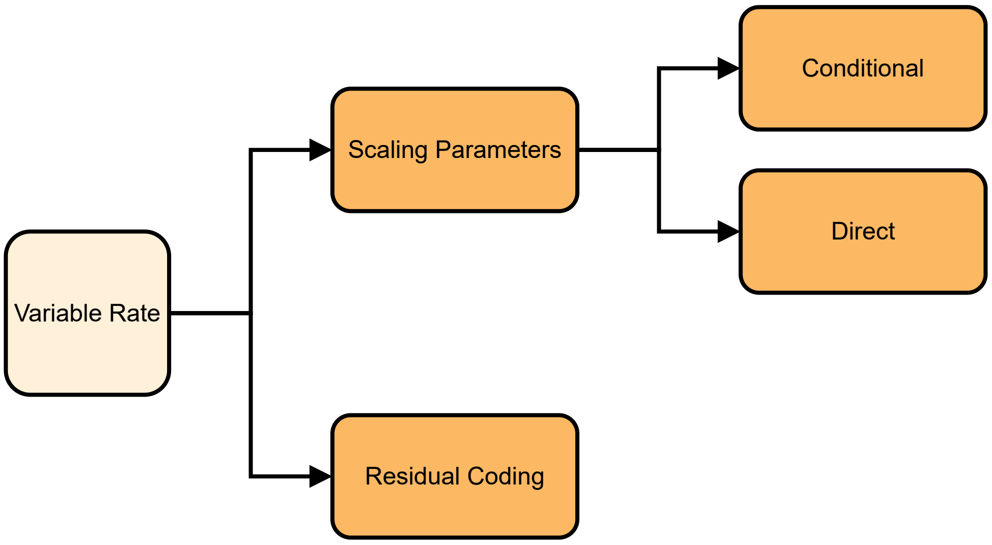

Figure 5.5: Variable Rate model classification for learned lossy image compression.

5.5.1 Scaling Parameters

Variable rate entropy models offer flexibility in setting the output quality. Unlike progressive decoding, which can stop the decoding process at any stage, variable rate models decode an image fully with a predetermined bit rate or quality target. This section focuses on variable rate models, which introduce variable rates through scaling parameters.

Variable Rate Categories		**Related Papers**
Scaling Parameters	Direct	[28, 87, 53]
	Conditional	[84, 26, 100, 83, 64]
Residual Coding		[6, 60, 5]

Table 5.5: Classification table for fig. 5.5

Conditional

Conditional Scaling Parameters can be seen as hyperparameters, which affect the entire network and the R-D optimization process. These parameters are typically present in analysis and synthesis transforms, the entropy model, and the quantization process. Choi et al. [26] proposed a variable rate framework that does not require training networks separately for each quality goal. Instead, they introduced two rate control parameters: the Lagrange multiplier λ and the quantization bin size. Both parameters can change the output quality without retraining or using multiple models. Theis et al. [84] introduced scale parameters λ to achieve variable rates. They initially trained the model using a fixed lagrangian multiplier (β), as described in Section 2.1.3. Then, they fixed the parameters and fine-tuned the model by introducing the scale parameters λ. The obtained scale parameters allow for fine-grained variable rate control with a single model. Choi et al. [26] trained the network using a predefined set of Lagrange multipliers λ and a bin size 1. They also adjusted the rate-distortion terms to accommodate the multiplier. Afterward, they fine-tuned the network using various bin sizes, enabling two controllable rate parameters. Yang et al. [100] published another novel. They introduced Modulated Autoencoders (MAE), which utilize modulation functions trained on a scaling factor λ. Unlike traditional methods that scale the bottleneck representation, these modulated networks adapt the feature maps at various layers within the model.

Direct

Direct Scaling Parameters typically scale the latent tensors directly before quantization and encoding. They also invert the scaling after entropy decoding and before the synthesis transform step. In contrast to the previous subcategory, direct scaling parameters are absent throughout the network as they scale the latent representation early and directly.

In contrast to the previous subcategory's work [26, 100], Cui et al. [28] introduce a variable rate model that utilizes channel-wise scaling of the latent representation through "gain units." Their exponential interpolation formula generates arbitrary scaling between learned discrete scaling points, providing finer-grained quality control. The gain control is applied directly to the latent representation before quantization and is reversed by applying the inverse gain before the synthesis transform step.

Tong et al. [87] introduce a quantization-error-aware variable rate framework to scale the latent representation directly. They formulate a univariate quantization regulator vector, which controls the quantization bin sizes and the quantization error during the quantization and entropy coding process. Tong et al. [87] claim that QVRF keeps the Gaussian entropy estimation nearly fixed. Experimental results suggest negligible performance degradation for discrete and continuous variable rates.

5.5.2 Residual Coding

This section focuses on achieving variable rates in a single model using residual coding. This approach controls the output quality by encoding the residual image losslessly [6, 60], typically with the BPG codec[2].

Octave Convolutions specialize in separating high and low-frequency data. More details on this architectural archetype can be found in Section 5.2.3. Their unique architecture lends itself well to variable rate models. Akbari et al. introduced a multi-resolution variable-rate model based on their previous work [5]. They replaced the analysis and synthesis transforms with generalized octave convolutions (GoConv) and transposed octave convolutions (GoTConv), allowing them to factorize high-frequency data and preserve important features across resolutions. Additionally, they modified the objective function to accommodate variable rates and introduced an enhancement layer called the residual image. The residual image is encoded losslessly into the bitstream using the BPG codec, further improving the performance by preserving finer details at higher bit rates.

Li et al. [60] proposed a variable rate model that utilizes vision transformer blocks to capture long-range dependencies. They extend the base framework previously used by Akbariet al. [6] To compensate for the time and computational cost of the context model, they divide the latent into segments, where each segment is processed in parallel with transformers. The residual image is encoded and decoded using the previously mentioned BPG codec.

Year	**Title**	**Taxonomy Categories**
2024	Variable-Rate Learned Image Compression with Multi-Objective Optimization and Quantization-Reconstruction Offsets [53]	Section 5.5
2024	Learned Image Compression with Gaussian-Laplacian-Logistic Mixture Model and Concatenated Residual Modules [36]	Section 5.4
2024	S2LIC: Learned Image Compression with the SwinV2 Block, Adaptive Channel-wise and Global-inter Attention Context [95]	Section 5.3
2023	QVRF: A Quantization-error-aware Variable Rate Framework for Learned Image Compression [87]	Section 5.5
2023	Neighborhood Attention Transformer [41]	Section 5.4
2023	An Introduction to Neural Data Compression [102]	Section 5.3
2023	Multistage Spatial Context Models for Learned Image Compression [63]	Section 5.3

Continued on next page

[2]Available at https://bellard.org/bpg/

Continued from previous page

Year	Title	Taxonomy Categories
2023	Fast and High-Performance Learned Image Compression With Improved Checkerboard Context Model, Deformable Residual Module, and Knowledge Distillation [35]	Section 5.3, Section 5.2
2023	Dynamic Kernel-Based Adaptive Spatial Aggregation for Learned Image Compression [92]	Section 5.3, Section 5.2
2023	MLIC: Multi-Reference Entropy Model for Learned Image Compression [51]	Section 5.3, Section 5.2
2023	Learned Image Compression with Mixed Transformer-CNN Architectures [68]	Section 5.2
2023	Multi-Context Dual Hyper-Prior Neural Image Compression [55]	Section 5.2
2022	Asymmetric Gained Deep Image Compression With Continuous Rate Adaptation [28]	Section 5.5
2022	Variable-Rate Deep Image Compression With Vision Transformers [60]	Section 5.5
2022	High-Efficiency Lossy Image Coding Through Adaptive Neighborhood Information Aggregation [70]	Section 5.4, Section 5.3, Section 5.2
2022	Entroformer: A Transformer-based Entropy Model for Learned Image Compression [77]	Section 5.4, Section 5.1, Section 5.3
2022	Ada-NETS: Face Clustering via Adaptive Neighbour Discovery in the Structure Space [94]	Section 5.4
2022	KVT: k-NN Attention for Boosting Vision Transformers [93]	Section 5.4
2022	Learning Accurate Entropy Model with Global Reference for Image Compression [78]	Section 5.4, Section 5.2
2022	Joint Global and Local Hierarchical Priors for Learned Image Compression [56]	Section 5.4, Section 5.3, Section 5.2
2022	Unified Multivariate Gaussian Mixture for Efficient Neural Image Compression [105]	Section 5.1
2022	Learned Image Compression with Generalized Octave Convolution and Cross-Resolution Parameter Estimation [34]	Section 5.3, Section 5.2
2022	ELIC: Efficient Learned Image Compression with Unevenly Grouped Space-Channel Contextual Adaptive Coding [42]	Section 5.3

Continued on next page

Continued from previous page

Year	Title	Taxonomy Categories
2022	The Devil Is in the Details: Window-based Attention for Image Compression [108]	Section 5.2
2022	Transformer-based Transform Coding [106]	Section 5.2
2021	Variable-Rate Deep Image Compression through Spatially-Adaptive Feature Transform [83]	Section 5.5
2021	End-to-End Learnt Image Compression via Non-Local Attention Optimization and Improved Context Modeling [22]	Section 5.4
2021	Transformer-based Image Compression [71]	Section 5.4, Section 5.2
2021	Taming Transformers for High-Resolution Image Synthesis [33]	Section 5.1
2021	Checkerboard Context Model for Efficient Learned Image Compression [43]	Section 5.1, Section 5.3
2021	Swin Transformer: Hierarchical Vision Transformer using Shifted Windows [69]	Section 5.2
2021	Learned Bi-Resolution Image Coding using Generalized Octave Convolutions [7]	Section 5.2
2020	Variable Rate Deep Image Compression With Modulated Autoencoder [100]	Section 5.5
2020	Learned Variable-Rate Multi-Frequency Image Compression using Modulated Generalized Octave Convolution [64]	Section 5.5, Section 5.2
2020	Learned Multi-Resolution Variable-Rate Image Compression with Octave-based Residual Blocks [6]	Section 5.5
2020	Learning Context-Based Non-local Entropy Modeling for Image Compression [62]	Section 5.4, Section 5.2
2020	Learning Texture Transformer Network for Image Super-Resolution [101]	Section 5.4
2020	Learned image compression with discretized gaussian mixture likelihoods and attention modules [25]	Section 5.4
2020	Coarse-to-Fine Hyper-Prior Modeling for Learned Image Compression [45]	Section 5.4
2020	Universally Quantized Neural Compression [3]	Section 5.1
2020	Nonlinear Transform Coding [10]	Section 5.1

Continued on next page

Continued from previous page

Year	Title	Taxonomy Categories
2020	Channel-wise Autoregressive Entropy Models for Learned Image Compression [75]	Section 5.1, Section 5.3
2020	Efficient and Effective Context-Based Convolutional Entropy Modeling for Image Compression [61]	Section 5.3
2019	Variable Rate Deep Image Compression With a Conditional Autoencoder [26]	Section 5.5, Section 5.1
2019	Learned Variable-Rate Image Compression with Residual Divisive Normalization [5]	Section 5.5
2019	Practical Stacked Non-local Attention Modules for Image Compression [66]	Section 5.4
2019	Conditional Probability Models for Deep Image Compression [72]	Section 5.4
2019	Context-adaptive Entropy Model for End-to-end Optimized Image Compression [58]	Section 5.4, Section 5.1, Section 5.3
2019	Drop an Octave: Reducing Spatial Redundancy in Convolutional Neural Networks with Octave Convolution [23]	Section 5.2
2019	Computationally Efficient Neural Image Compression [52]	Section 5.2
2018	Joint Autoregressive and Hierarchical Priors for Learned Image Compression [74]	Section 5.4, Section 5.1, Section 5.3
2018	Non-Local Recurrent Network for Image Restoration [65]	Section 5.4
2018	Variational image compression with a scale hyperprior [13]	Section 5.4, Section 5.1
2018	CrossNet: An End-to-end Reference-based Super Resolution Network using Cross-scale Warping [104]	Section 5.4
2018	Neural Discrete Representation Learning [88]	Section 5.1
2018	Deep Image Compression with Iterative Non-Uniform Quantization [21]	Section 5.1
2017	End-to-end Optimized Image Compression [12]	Section 5.4, Section 5.2
2017	Lossy Image Compression with Compressive Autoencoders [84]	Section 5.5

Continued on next page

Continued from previous page

Year	Title	Taxonomy Categories
2017	Soft-to-Hard Vector Quantization for End-to-End Learned Compression of Images and Neural Networks [1]	Section 5.1
2017	Deformable Convolutional Networks [29]	Section 5.2
2016	End-to-end optimization of nonlinear transform codes for perceptual quality [9]	Section 5.1
2016	Variable Rate Image Compression with Recurrent Neural Networks [85]	Section 5.1
2016	Density Modeling of Images using a Generalized Normalization Transformation [11]	Section 5.2
1985	On universal quantization [107]	Section 5.1

Table 5.6: List of selected publications, which are included in the taxonomies. This list is sorted by known release year and points to the section it has been categorized as. Section 5.2 is the taxonomy for architectural archetypes. Section 5.4 is the taxonomy for hierarchical priors. Section 5.1 is the taxonomy for quantization. Section 5.5 is the taxonomy for variable rate. Section 5.3 is the taxonomy for context models.

CHAPTER 6

Benchmarking Tool - Benchpresso

This chapter focuses on the details of implementing the Benchpresso benchmarking tool. Section 6.3 outlines the features and usage of the Benchpresso command line interface, including training models with various datasets, testing different datasets for inference, and dynamic configuration. Section 6.4 provides a base model implementation to train and test models designed to be used with the Benchpresso tool. Section 6.5 provides a base implementation for data modules to facilitate loading various data sets when using Benchpresso. Section 6.6 discusses customization options, such as saving and loading checkpoints, setting the log-directory and streamlining development.

6.1 Requirements

The framework's goal is to standardize the benchmarking process by offering an end-to-end process of fitting, validating, and testing a model. It interprets configuration files and executes any or all steps. Modifying configuration files and restarting the end-to-end process should be possible. The framework ensures that the results are reproducible and evaluated directly or in the future. This section outlines the requirements that the Benchpresso framework must meet, the problems it aims to solve, and the processes involved in configuring it.

- The framework must be configurable using configuration files.
- The framework must load objects of given class paths.
- The framework must execute single or all stages, such as fit, test, and validation.
- The framework must persist results in standard formats.
- The framework must perform seeding to ensure reproducible results.

- The configuration files must be open, i.e., not specifying a field assumes the default value.
- The configuration files must accept class paths to core components.
- The configuration files must be reusable.
- The configuration files must be compatible with all stages, such as fit, test, and validation.

6.2 System Design

This section offers a comprehensive description of the system behind the framework. It is highly configurable and extensible. Figure 6.1 presents the model diagram in more detail. Core components are abstract and can be extended or replaced. They are inspired by the typical variational autoencoder (VAE) architecture. Therefore, implementing any VAE model and modifying its core components is possible. The exact way of creating new variations is described in Section 6.4.

Loading data for the fit, validation, and test stages is an essential step. Benchpresso applies its rules to datasets as well. Figure 6.2 shows the data module diagram, which shows that datasets for various stages are easily replaceable. The implementation may consist entirely of custom transforms, setups, preprocessing, and datasets.

6.3 Command Line Interface

The Command Line Interface (CLI) is designed to load various models that extend the `LightningModule` and dataloaders that extend the `LightningDataModule`. Our `BaseModel` and `BaseDataModule` classes implement these and can be further extended. Section 6.3.1 describes the implementation in further detail.

6.3.1 Command Line Interface

```python
# These imports let you use the classes through the CLI
from data_modules import *  # noqa: F401
from models import *  # noqa: F401


def cli_main():
    cli = LightningCLI()


if __name__ == "__main__":
    cli_main()
```

Additionally, the CLI requires a specific execution stage, model, and data. The available stages are `fit` for training, `validate` for validation, and `test` for inference. Instead of defining the models and data loaders using the `--model` and `--data` flags, it is possible to use the `--config` flag to pass a configuration file. Using configuration files ensures that each run (referred to as experiments) is reproducible and highly customizable. We recommend creating separate configuration files for each dataset, stage, and model. The CLI also supports overriding by passing the arguments in the `jsonargparse`[1] syntax. Section 6.3.2 provides and explains the usage.

6.3.2 CLI Command Usage

```
# Run training with FactorizedPrior model and Flicker2W dataset
$ python benchpresso-cli.py fit \
  --model models.google_models.FactorizedPrior \
  --data data_modules.flicker_2w_data_module.Flicker2WDataModule

# Run training with FactorizedPrior model, Flicker2W dataset,
# and override batch_size
$ python benchpresso-cli.py fit \
  --model models.google_models.FactorizedPrior \
  --data data_modules.flicker_2w_data_module.Flicker2WDataModule
  --data.batch_size 32

# Run inference with FactorizedPrior model, Kodak dataset,
# batch_size=1, and ms_ssim loss metric
$ python benchpresso-cli.py test \
  --model models.google_models.FactorizedPrior \
  --data data_modules.kodak_data_module.KodakDataModule
  --data.batch_size 1
  --model.rate_distortion_function
  '{class_path: loss.rate_distortion_loss_compressai.RateDistortionLoss,
  init_args: {loss_metric: ms_ssim}}'

# Run inference with the specified configuration file
$ python benchpresso-cli.py test \
  --config configs/factorized_prior_test_q8_mse.yaml

# Run training with the specified configuration file
$ python benchpresso-cli.py fit \
  --config configs/factorized_prior_fit_q8_mse.yaml
```

[1]https://jsonargparse.readthedocs.io/en/v4.32.0/

6.4 Models - loading and customizing

Our benchmarking tool enables modular and customizable models. We have defined a base class for models that consists of necessary arguments, such as:

- **encoder & decoder** non-optional input arguments used to encode and decode data
- **rate_distortion_function** is the objective function
- **entropy_bottleneck** is the entropy bottleneck with the default value provided by the CompressAI framework.
- ****kwargs** An optional argument that can be used to pass a dictionary of additional parameters.

We expect the users to implement the abstract methods themselves. Additionally, we leverage the PyTorch Lightning framework by LightningAI, which reduces boilerplate code, such as the training loop, as seen in Section 6.4.1.

6.4.1 Base Model

```
class BaseModel(L.LightningModule, ABC):
    def __init__(self,
                 encoder: nn.Module = None,
                 decoder: nn.Module = None,
                 rate_distortion_function: nn.Module = None,
                 entropy_bottleneck = None,
                 **kwargs: dict[str, Any]):
        super(BaseModel, self).__init__()
        # implementation details

    @abstractmethod
    def forward(self, x):
        pass

    @abstractmethod
    def training_step(self, batch, batch_idx):
        pass

    @abstractmethod
    def validation_step(self, batch, batch_idx):
        pass

    @abstractmethod
    def test_step(self, batch, batch_idx):
        pass

    def on_train_epoch_end(self) -> None:
        pass

    def configure_callbacks(self) -> Union[Sequence[Callback], Callback]:
        pass

    def configure_optimizers(self):
        pass
```

The BaseModel class should be extended and customized to your needs. We recommend adding additional functions, such as saving images during inference or computing additional metrics. Section 6.4.2 provides a simple implementation of the training and test loop. The datasets are modular and are automatically loaded depending on the current stage. A correctly implemented data module should load the training and validation datasets for the training stage, as shown in Section 6.5.1.

6.4.2 VAE Model Example

```
class VariationalAutoEncoder(BaseModel):

    # ...

    def forward(self, x: torch.Tensor) -> Dict[str, Any]:
        y = self._encoder(x)
        y_hat, y_likelihoods = self._entropy_bottleneck(y)
        x_hat = self._decoder(y_hat)
        return {
            "x_hat": x_hat,
            "likelihoods": {
                "y": y_likelihoods,
            },
        }

    def training_step(self, batch, batch_idx):
        return self.compute_one_epoch(batch, batch_idx)

    def validation_step(self, batch, batch_idx):
        return self.compute_one_epoch(batch, batch_idx)

    def test_step(self, batch, batch_idx):
        return self.compute_one_epoch(batch, batch_idx)

    def compute_one_epoch(self, batch, batch_idx):
        x, _ = batch
        x_hat = self.forward(x)
        loss = self._rate_distortion_function(x_hat, x)
        return loss

```

By keeping the extensions of the base class consistent and modular, it is possible to integrate various models with different datasets. Section 6.4.3 provides the configuration file for the factorized model by Ballé et al. [13] In this configuration, we set the convolutional layers for the encoder and decoder according to their publication and set the *class_path* accordingly. Additionally, we set the activation function to Generalized Divisive Normalization (GDN). As described in the work of Ballé et al. [13], for $\lambda = 0.0018$, we set N to 128 and M to 192. We chose mean-squared-error as our loss metric.

6.4.3 Factorized Prior Model Configuration File

```yaml
model:
  class_path: models.google_models.FactorizedPrior
  init_args:
    encoder:
          class_path: google_encoder_decoder.FactorizedPriorEncoder
          init_args:
            activation:
                class_path: activations.GeneralizedDivisiveNormalization
                init_args:
                    channels: *N
                    inverse: False
                    beta_min: 1e-6
                    gamma_init: 0.1
            channels: &N 128 # saves value 128 in variable N
            expansion_channels: &M 192 # saves value 192 in variable M
    decoder:
      class_path: google_encoder_decoder.FactorizedPriorDecoder
      init_args:
        activation:
            class_path: activations.GeneralizedDivisiveNormalization
            init_args:
                    channels: *N
                    inverse: True # set to true utilizes IGDN
                    beta_min: 1e-6
                    gamma_init: 0.1
            channels: *N
            expansion_channels: *M
        rate_distortion_function:
          class_path: loss.rate_distortion_loss_compressai.RateDistortionLoss
          init_args:
            lagrangian_multiplier: 0.0018
            loss_metric: mse
            return_type: all
        entropy_bottleneck: null # default entropy bottleneck is used
        expansion_channels: *M
```

6.5 Data Modules - handling data loaders and datasets

The `Data Module` handles datasets for training, validation, and testing. The base class in Section 6.5.1 provides basic functionality for datasets and is intended to be extended by a concrete class implementation. The following parameters can be customized:

- **data_dir** specifies the path to the dataset directory. When using the `ImageFolder` class by `torchvision`, ensure that the data structure complies with its requirements and provide folders for training, validation, and testing.
- **batch_size** can be customized depending on your hardware. Loading 16-32 images per batch should be sufficient for training. All images in a batch must be of the same size. Set $batch\_size = 1$ for inference to handle various resolutions. The default value is 1.
- **num_workers** can accelerate the process depending on your hardware. The default value is 1.
- **transform** is used to load a dataset. If none is provided, the base class defines a default transform, which crops each image to a $256 * 256px$ patch and transforms it into a tensor.
- ****kwargs** is optional and provide a dictionary of additional parameters.

6.5.1 The Base Class for Data Modules

```
class BaseDataModule(L.LightningDataModule):
    def __init__(self,
                 data_dir: str = "./data",
                 batch_size: int = 1,
                 num_workers: int = 1,
                 transform=None,
                 **kwargs: dict[str, Any]):
        super().__init__()
        # assign or initialize variables

        default_transform = transforms.Compose([
            transforms.CenterCrop(256),
            transforms.ToTensor()
        ])
        self.transform = transform or default_transform

    def prepare_data(self):
        pass

    def setup(self, stage):
        train_path = os.path.join(self.data_dir, "train")
        # initialize eval_path and test_path

        if stage == 'fit' or stage is None:
            self.train_dataset = ImageFolder(root=train_path,
                                             transform=self.transform,
                                             allow_empty=True)
            self.val_dataset = ImageFolder(root=eval_path,
                                           transform=self.transform,
                                           allow_empty=True)

        if stage == 'validate' or stage is None:
            # analog to the training stage

        if stage == 'test' or stage is None:
            # analog to the training stage

    def train_dataloader(self):
        return DataLoader(self.train_dataset,
                          batch_size=self.batch_size,
                          shuffle=False,
                                           num_workers=self.num_workers)

    # val_ and test_dataloaders analog to train_dataloader(self)
```

6.5.2 Loading a Dataset using the CLI

The Kodak Dataset is commonly used in testing. Creating a concrete implementation for a dataset such as Kodak, can be realized in two ways. The fastest option is to extend our base class and override the arguments accordingly, e.g., *data_dir* = "*./data/kodak*." Section 6.5.2 provides an example implementation.

Kodak Data Module

```python
class KodakDataModule(BaseDataModule):
    def __init__(self,
                 data_dir: str = "./data/kodak",
                 **kwargs: dict[str, Any]):
        super(KodakDataModule, self).__init__(data_dir=data_dir,
                                              **kwargs)
```

6.5.3 Loading a Dataset using a configuration file

The alternative way is to define the data module in the configuration file. We recommend extending the base class with the concrete dataset first. This enables customization of the concrete class and introduces additional parameters without changing the base class. To this end, we propose a configuration file for inference using the Kodak dataset (Section 6.5.4). We set the transform to NoCrop to compress the full-resolution image. Each batch is a tensor and requires consistent resolution. A simple solution is to set $batch\_size = 1$ and $num\_workers = 2$. Alternatively, divide the dataset into $class\_n, n \geq 0$ folders, each with images of identical resolution.

6.5.4 Kodak DataModule Configuration File

```yaml
data:
  class_path: data_modules.kodak_data_module.KodakDataModule
  init_args:
    transform:
      class_path: transform.NoCrop
    data_dir: ./data/kodak
    batch_size: 1
    num_workers: 2
```

We have implemented a `transform` class that consists of common transforms such as `CenterCrop`, `RandomCrop` (Section 6.5.6), and `NoCrop`. Both our `RandomCrop` and `CenterCrop` transforms include the *patch_size* argument. When loading a training dataset, using`RandomCrop` with a *patch_size* of 256 and *batch_size* of 16 is typical. These settings are similar to the Kodak configuration file in Section 6.5.5.

6.5.5 Flickr2W DataModule Configuration File

```yaml
data:
  class_path: data_modules.flicker_2w_data_module.Flicker2WDataModule
  init_args:
    transform:
      class_path: transform.RandomCrop
      init_args:
        patch_size: 256
    data_dir: ./data/flicker_2W_images
    batch_size: 16
    num_workers: 2
```

6.5.6 RandomCrop Transform Implementation

```python
class RandomCrop:
    def __init__(self, patch_size: int = 256):
        self.transform = transforms.Compose([
            transforms.RandomCrop(patch_size),
            transforms.ToTensor()
        ])

    def __call__(self, img):
        return self.transform(img)
```

6.6 Configuration Files

We use configuration files to achieve high customizability while maintaining consistency. Our naming convention follows the syntax
`<model_name>_<stage>_<quality>_<loss_metric>.yaml`; for example, `FactorizedPrior_fit_q1_mse.yaml` for training the factorized prior model [13] or `Cheng2020Attention_test_q8_ms_ssim.yaml` for testing the model by Cheng et al. [25] When using the CLI, executing commands saves the configuration in the logs directory to document and reproduce results in the future. Checkpointing can occur after a specified number of batches, as shown in Section 6.6.1. For development purposes, we

recommend configuring $fast\_dev\_run : True$. Furthermore, loading checkpoints may be accomplished by setting the path in the configuration file. For more details, refer to the PyTorch Lightning documentation[2].

6.6.1 Miscellaneous Configuration

```yaml
# ... trimmed configuration

trainer:
  logger: null
  callbacks:
    - class_path: lightning.pytorch.callbacks.ModelCheckpoint
      init_args:
        # save checkpoint every 1000 steps
        every_n_train_steps: 1000
        # keep only the latest checkpoint
        save_top_k: 1
  # runs only one epoch and one batch
  fast_dev_run: true
  enable_checkpointing: true
  enable_progress_bar: true
  enable_model_summary: true
  default_root_dir: logs
# loads a checkpoint
ckpt_path: '/app/logs/factorized_prior_relu_q8_mse.ckpt'
```

- –stage is either train or test.
- –data specifies the data module configures the dataset, batch size, transforms, and directories.
- –rates_config
- –trainer loads a trainer config. We use different trainers for local development, training and testing to ensure consistent settings across all runs. The development trainer limits the number of steps and epochs, disables checkpointing, limits batch sizes, etc.

6.7 Measuring and benchmarking

Benchpresso measures each model's encoding and decoding latency across multiple resolutions. Additionally, it measures the throughput across multiple resolutions and

[2]https://lightning.ai/docs/pytorch/stable/

batch sizes to evaluate the architecture's capability of working on images in parallel. The coding latencies are measured in milliseconds, while the throughput is measured in images per second.

Geifman [39] shows the correct way to measure latency in CUDA-supported workflows. Traditional methods of measuring the elapsed time between tasks may result in inaccurate measurements. The author describes the correct way of synchronizing the GPU with the CPU, warming up the GPU with dummy data, and recording execution times according to the GPU.

6.7.1 GPU Warm-up

Benchpresso warms up the GPU by generating random images in a loop and utilizing CUDA functions. At the end of the loop, the torch framework synchronizes all kernels by waiting for the CUDA cores to finish.

6.7.2 Measuring time

Benchpresso measures execution times using *cuda.Event*[3] markers, which are designed to measure them accurately. Create a starter and ending event. Record the time on the starter event and execute your operation. After the operation ends, record the ending event. Wait for the cuda cores to synchronize, then measure the elapsed time between both events. These steps ensure precise measurements by excluding external factors.

Benchpresso reuses this function for latency measurements and throughput evaluation. Measuring throughput also tracks the number of batches to evaluate the images per second value accurately.

[3]Documentation: `https://pytorch.org/docs/stable/generated/torch.cuda.Event.html`

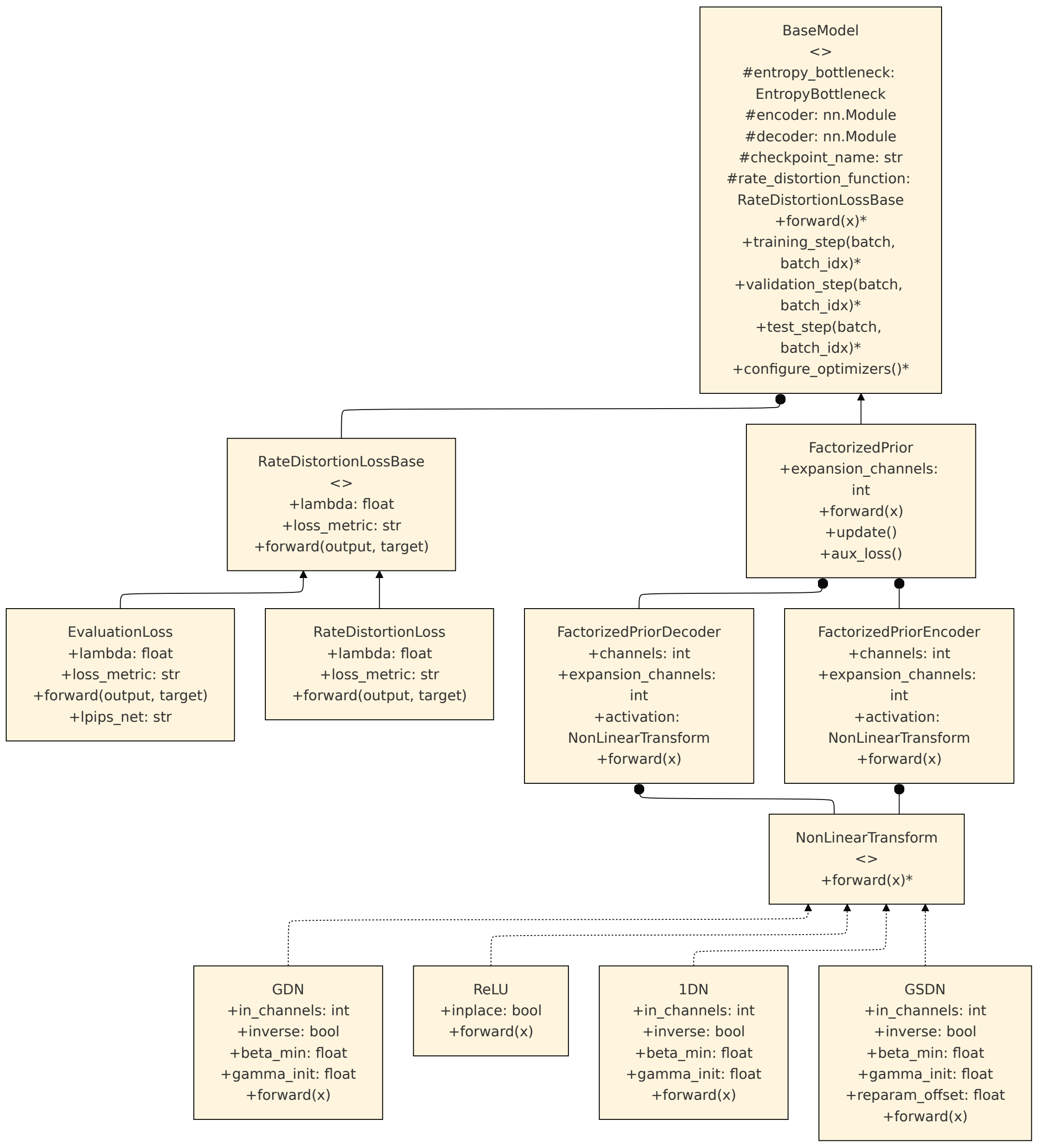

Figure 6.1: This diagram shows the basic structure of Benchpresso. The Base Model defines input variables and is necessary to implement functions. The default implementation is provided for consistency. The exemplary model FactorizedPrior requires two optimizers (main and aux) and an update function for the cumulative distribution function required by the entropy bottleneck. The non-linear transform interface requires only the forward function. Furthermore, our base class for the rate-distortion may be extended, and training and evaluation may differ.

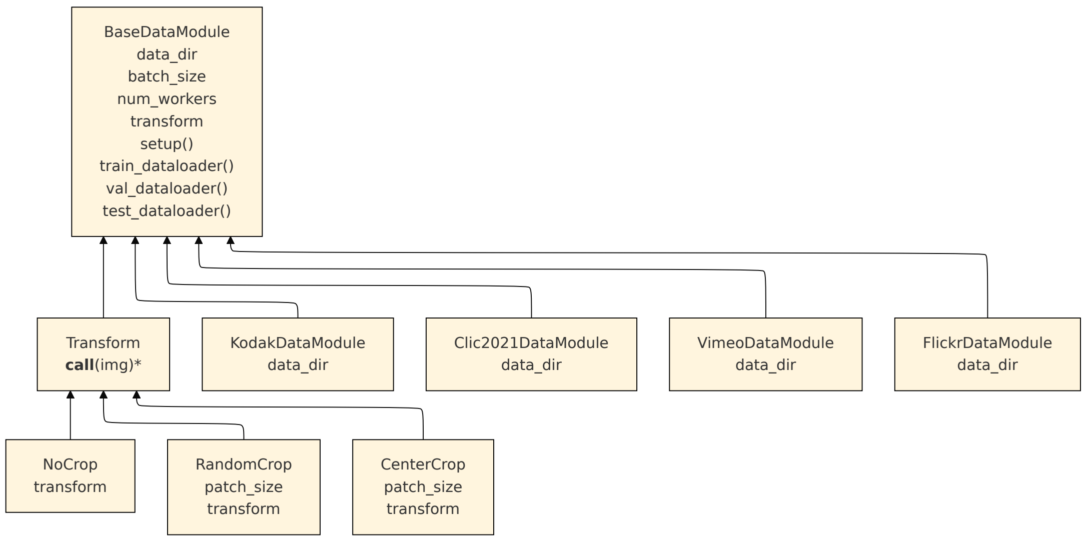

Figure 6.2: This diagram shows the basic structure of data modules in Benchpresso. The Base defines input variables necessary to implement functions. The default implementation is provided for consistency. We implement the data module for training and testing datasets. The transform interface offers an abstraction to implement common image transforms such as center crop, random crop, and no crop (for full-resolution images).

CHAPTER 7

Evaluation

This chapter evaluates variants of learned lossy compression models using our benchmarking tool. We choose proven models as our baseline and systematically modify their core elements. Benchpresso trains and evaluates the baseline models and their variants using the same procedures and datasets. We draw comparisons with handcrafted codecs to reduce variance and ensure comparable results across all tested models.

We utilize the BD-Rate and BD-PSNR to compare trained models with the anchor codec. Unless otherwise specified, we employ BPG (4:4:4) as the anchor codec. Results are interpolated using the Piecewise Cubic Hermite Interpolation (PCHIP), as recommended by Barman et al. [15] and Boyce et al. in their work on the common test conditions, following the 10th JVET meeting [19].

7.1 Setup

This section describes the environment and application setup, as future changes in software and hardware may influence the currently presented results. The environment section provides detailed information on hardware, the operating system, and global dependencies. The Application section includes detailed information on the libraries and programs, such as docker.

7.1.1 Hardware and Software Environment

We conduct our experiments on a machine running Ubuntu 22.4 LTS, equipped with an NVIDIA GeForce RTX 3090 GPU with 24GB VRAM. We utilize the official PyTorch Docker image[1] with CUDA 12.4 Runtime, Python 3.11.9, and Docker v27.1.1. For the anchor codec, which serves as a reference for comparison, we employed BPG[2] 0.9.5, compressing images with 4:4:4 sampling and quality settings ranging from 0 to 51.

7.1.2 Datasets and Training Parameters

We train the models on two datasets, Flickr2K and Vimeo90K. Flickr2K comprises 20'715 images sampled from the Flickr dataset and curated by Liu et al. [67]. Vimeo90K [99] Triplet dataset comprises 73'171 3-frame images and is loaded with the dataloader by CompressAI [3]. For testing and validation, we employ the standard set of 24 raw Kodak images [27]. For architectures based on [13], we use $\lambda \in$ (0.0250, 0.0130, 0.0067, 0.0035, 0.0018) with $N = 128, M = 192$ set to five lower rates.

We optimize all models using the Adam optimizer with an initial learning rate of 10^{-4} (decay to 10^{-6}). We use a batch-size = 16 for Ballé et al. architectures. We conducted training for a minimum of 1'000'000 steps with early stopping (patience=20). All models are optimized for MSE and forgo fine-tuning. We lower the default precision to *bfloat16* (medium precision) to achieve faster convergence.

7.1.3 Application

The experiments run in a dockerized environment with volumes mapped to log output, datasets, checkpoints, and compression output. We utilize the Weights&Biases Logger[4] (v0.18.5) for experiment tracking. Essential libraries include torch (v2.4.1), torchvision (v0.18), CompressAI (v1.2.6), and lightning[5] (v2.4.0).

We utilize YAML configuration files for training and testing stages, each assigned a descriptive name. For instance, `google_fp_gdn` in the training folder signifies the

[1]PyTorch Images can be found here: `https://hub.docker.com/r/pytorch/pytorch/tags`
[2]Available at `https://bellard.org/bpg/`
[3]Source code available at: `https://github.com/InterDigitalInc/CompressAI/blob/master/compressai/datasets/vimeo90k.py`
[4]Available at `https://wandb.ai/`
[5]Available at `https://lightning.ai/docs/pytorch/stable/`

training process of the factorized prior model by Ballé et al. using GDN as the activation function, while `train/google_fp_relu` signifies the same training process with ReLU as the activation function.

7.2 Ablation Studies

We perform ablation studies on two well-researched architectures by Ballé et al. The baseline of the factorized prior (fp) architecture features a simple encoder and decoder, which utilizes the generalized divisive normalization (Section 5.2.2) activation function. Similarly, we refer to the second architecture as mean-scale hyperprior (msh). In addition to the previous architecture, it features a component to extract and utilize side information. Our flexible framework allows us to construct diverse configurations based on these two architectures.

Section 7.3 describes the modifications for the fp model, and Section 7.4 for the msh model.

7.3 Factorized Prior Model

7.3.1 Activation Functions

The factorized prior model by Ballé et al. utilizes GDN [13] as the activation function. CompressAI[6] replaced this activation function with ReLU, while Johnston et al. [52] introduced a simplified version of GDN described in Section 5.2.2. Qian et al. [78] introduced a subtractive version of GDN, described in Section 5.2.2. We utilize the framework Benchpresso to train and evaluate the factorized prior model with the above activation function variants: FP with GDN, FP with 1DN, FP with ReLU, and FP with GSDN. To modify the baseline architecture, we adjust the classpath of the activation function by pointing at different implementations to profit from the framework's modularity. We measure the performance gains of various activation functions and compare computationally expensive functions with cheaper alternatives. Those architecture variants are visualized in Figure 7.10.

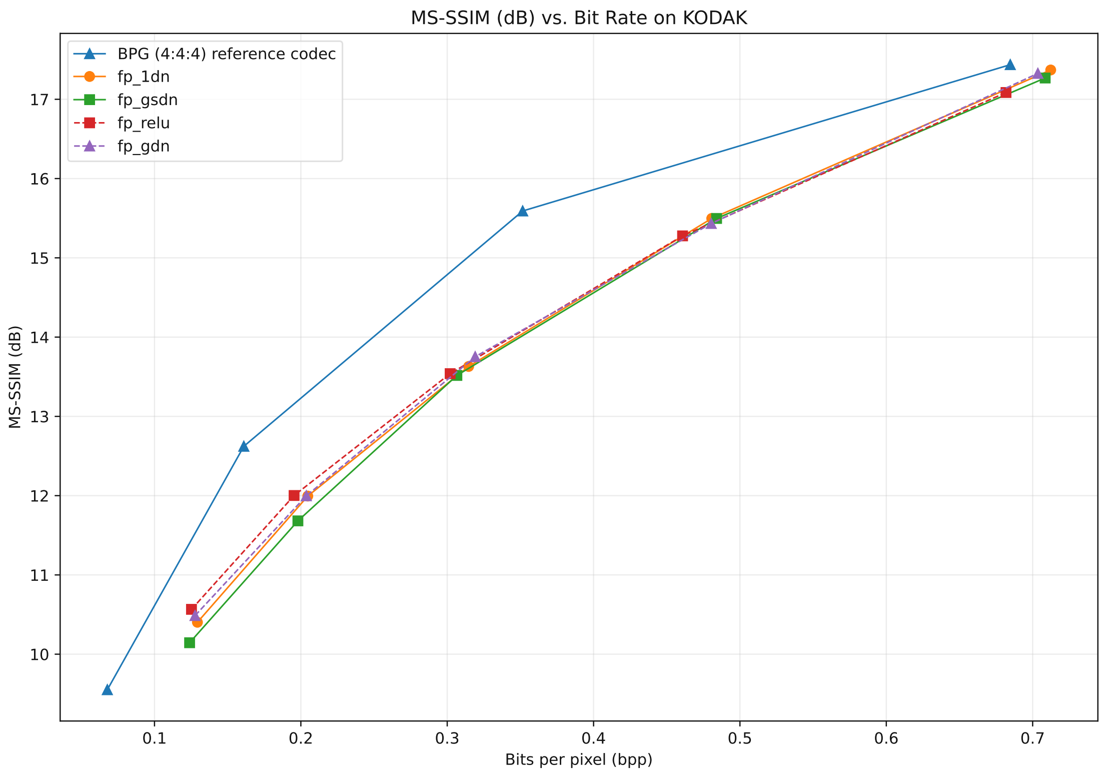

Figure 7.1: Performance evaluation in MS-SSIM of Factorized Prior Variants on the Kodak dataset.

[6]https://github.com/InterDigitalInc/CompressAI/blob/master/compressai/models/google.py#L167

7.3.2 R-D Performance

We evaluate the models on the Kodak [27] and CLIC [14] datasets and report Bits-per-pixel, MS-SSIM, LPIPS, and PSNR. Figure 7.2 and Figure 7.1 show the rate-distortion curves, with the distortion measures in PSNR and MS-SSIM evaluated on the Kodak dataset.

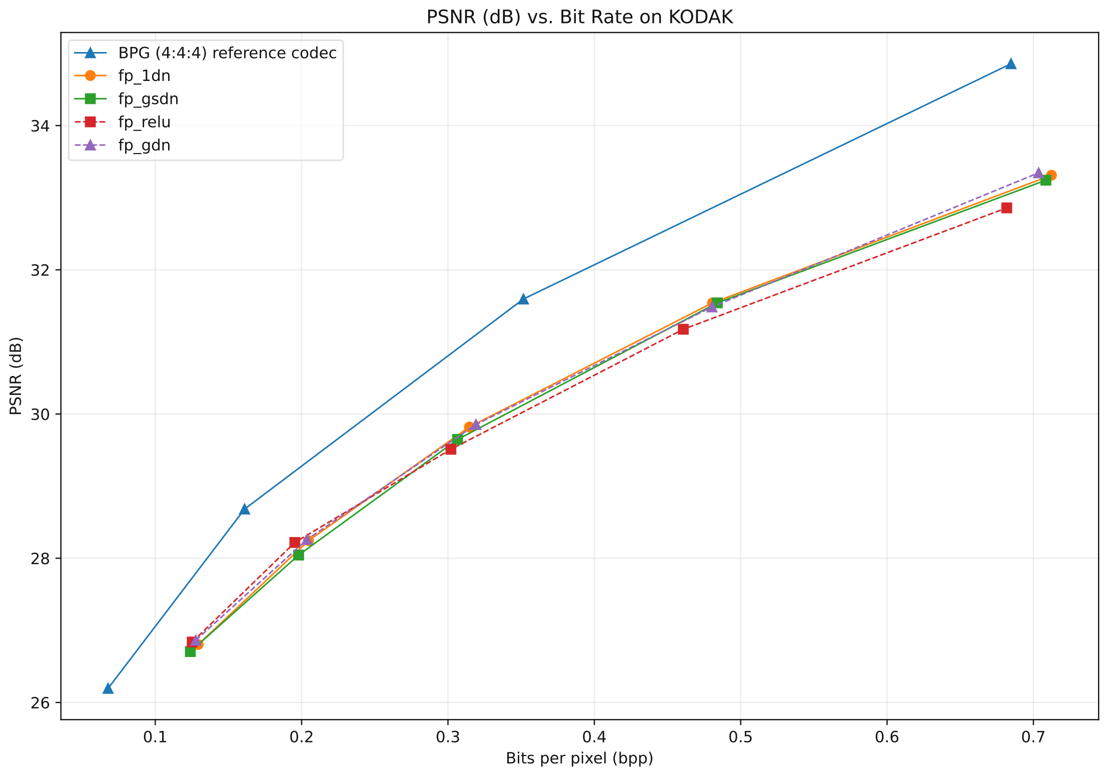

Figure 7.2: Performance evaluation in PSNR of Factorized Prior Variants on the Kodak dataset.

Figure 7.3 and Figure 7.4 illustrate rate-distortion curves evaluated on the CLIC dataset. These figures also provide a comparison to other non-linear components within the same architecture. We choose a similar range of bitrates for the anchor codec (BPG) to ensure stable and comparable results

Model	Bjøntegaard Delta Rate↓	Bjøntegaard Delta PSNR↑
fp_relu	44.37%	-1.40 dB
fp_gdn	41.67%	-1.35 dB
fp_1dn	42.08%	-1.36 dB
fp_gsdn	44.21%	-1.41 dB

Table 7.1: Bjøntegaard Delta Rate and Bjøntegaard Delta PSNR of factorized prior variants with modified activation functions. The anchor codec is BPG(4:4:4). All codecs are evaluated on the Kodak dataset.

Table 7.2 shows the performance of the factorized prior model with GDN as the non-linear component. This baseline achieves a BD-Rate of 41.67% and BD-PSNR of -1.35 dB, as shown in Table 7.1. Table 7.3 shows the performance of the fp_gsdn model, which utilizes the GSDN function, instead of GDN in its non-linear component.

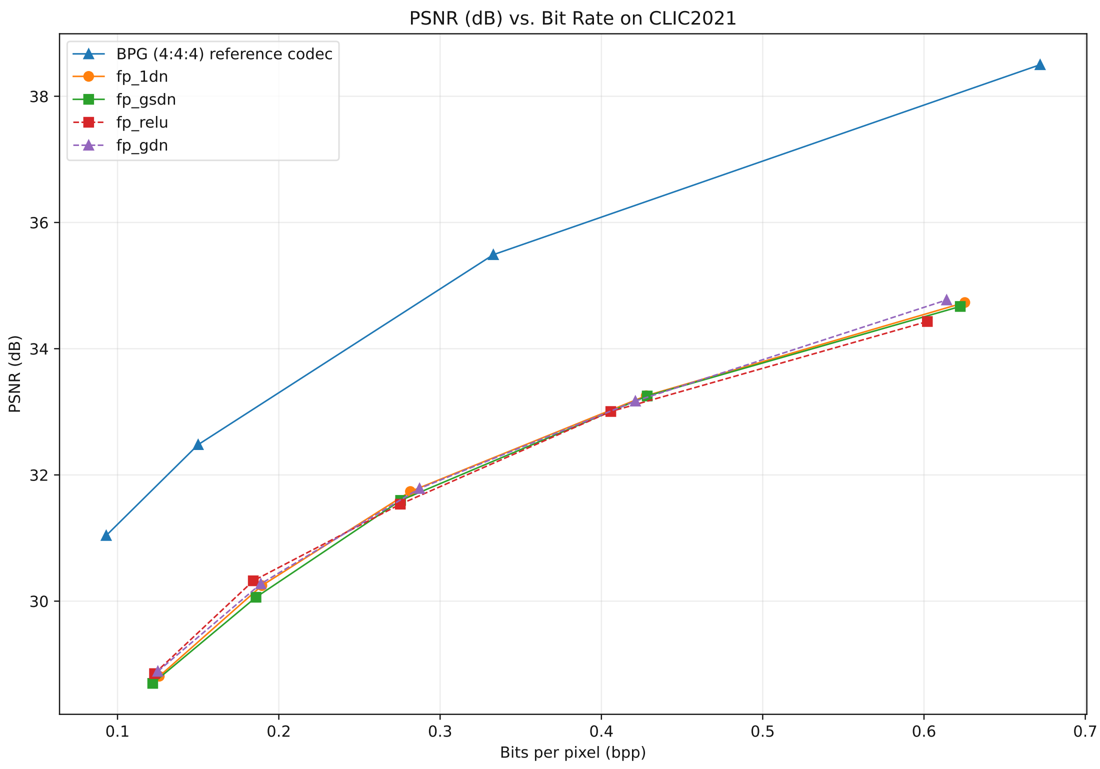

Figure 7.3: Performance evaluation in PSNR of Factorized Prior Variants on the CLIC dataset.

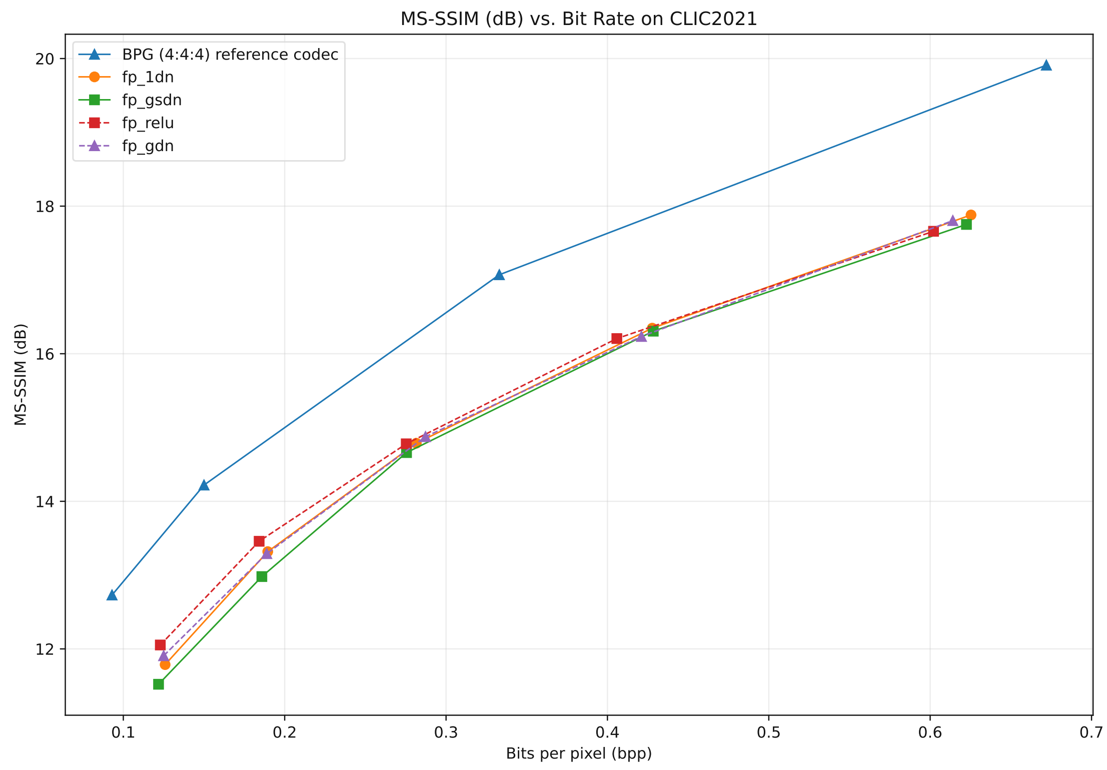

Figure 7.4: Performance evaluation in MS-SSIM of Factorized Prior Variants on the CLIC dataset.

We show that the GSDN and ReLU variants perform 3% worse than the baseline compared to our anchor codec. ReLU is computationally cheap to compute and has a lower capacity of non-linear transforms.

Kodak				**CLIC2021**			
bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓	bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓
0.132	9.657	26.881	0.458	0.117	9.544	28.770	0.448
0.208	10.572	28.186	0.407	0.176	9.866	30.095	0.414
0.325	11.524	29.673	0.348	0.267	10.824	31.410	0.373
0.503	13.051	31.284	0.292	0.402	11.453	32.728	0.335
0.703	17.326	33.345	0.235	0.626	11.989	34.083	0.292

Table 7.2: Metrics for the factorized prior model with GDN as the activation function.

Table 7.4 shows the performance of the fp__relu model, which uses a computationally inexpensive non-linear transform, ReLU. While the computation is less complex and

Kodak				CLIC2021			
bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓	bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓
0.123	10.144	26.705	0.468	0.121	11.520	28.697	0.469
0.198	11.681	28.044	0.413	0.186	12.980	30.061	0.429
0.306	13.517	29.648	0.357	0.275	14.660	31.596	0.387
0.484	15.496	31.541	0.297	0.428	16.305	33.251	0.346
0.708	17.270	33.243	0.248	0.622	17.752	34.669	0.311

Table 7.3: Metrics for the factorized prior model with GSDN as the activation function.

achieves a higher throughput per second on our hardware, the performance degradation is significant.

Kodak				CLIC2021			
bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓	bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓
0.125	10.565	26.838	0.468	0.123	12.053	28.851	0.468
0.195	12.001	28.218	0.412	0.184	13.458	30.322	0.425
0.302	13.538	29.511	0.363	0.275	14.779	31.536	0.391
0.460	15.276	31.175	0.308	0.406	16.205	33.004	0.354
0.681	17.085	32.858	0.251	0.602	17.659	34.430	0.310

Table 7.4: Metrics for the factorized prior model with ReLU as the activation function.

Table 7.5 shows the performance of the fp_1dn variant with the 1DN transform, which achieves a BD-Rate of 44%.

Kodak				CLIC2021			
bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓	bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓
0.129	10.402	26.803	0.455	0.126	11.787	28.810	0.454
0.205	11.994	28.243	0.405	0.189	13.318	30.247	0.419
0.314	13.630	29.820	0.352	0.281	14.787	31.738	0.384
0.480	15.497	31.541	0.294	0.428	16.346	33.255	0.346
0.712	17.369	33.310	0.242	0.625	17.880	34.732	0.305

Table 7.5: Metrics for the factorized prior model with 1DN as the activation function.

7.3.3 Latency and Throughput

We evaluate latency and throughput by measuring the encoding times, decoding times, and number of images encoded and decoded in specific time intervals. First, the GPU is warmed up, and the clocks are synchronized. We generate random pictures using the "randn" function by torch in batches of 2-16. The initial resolution is 256px and increases by 128px until 1024px. We repeat our runs 50 times across all resolutions and network configurations. Small network configurations utilize fewer channels (N=128, M=192). Larger network configurations utilize more channels (N=192, M=320). The latency is expressed in milliseconds (ms), while the throughput is in images per second.

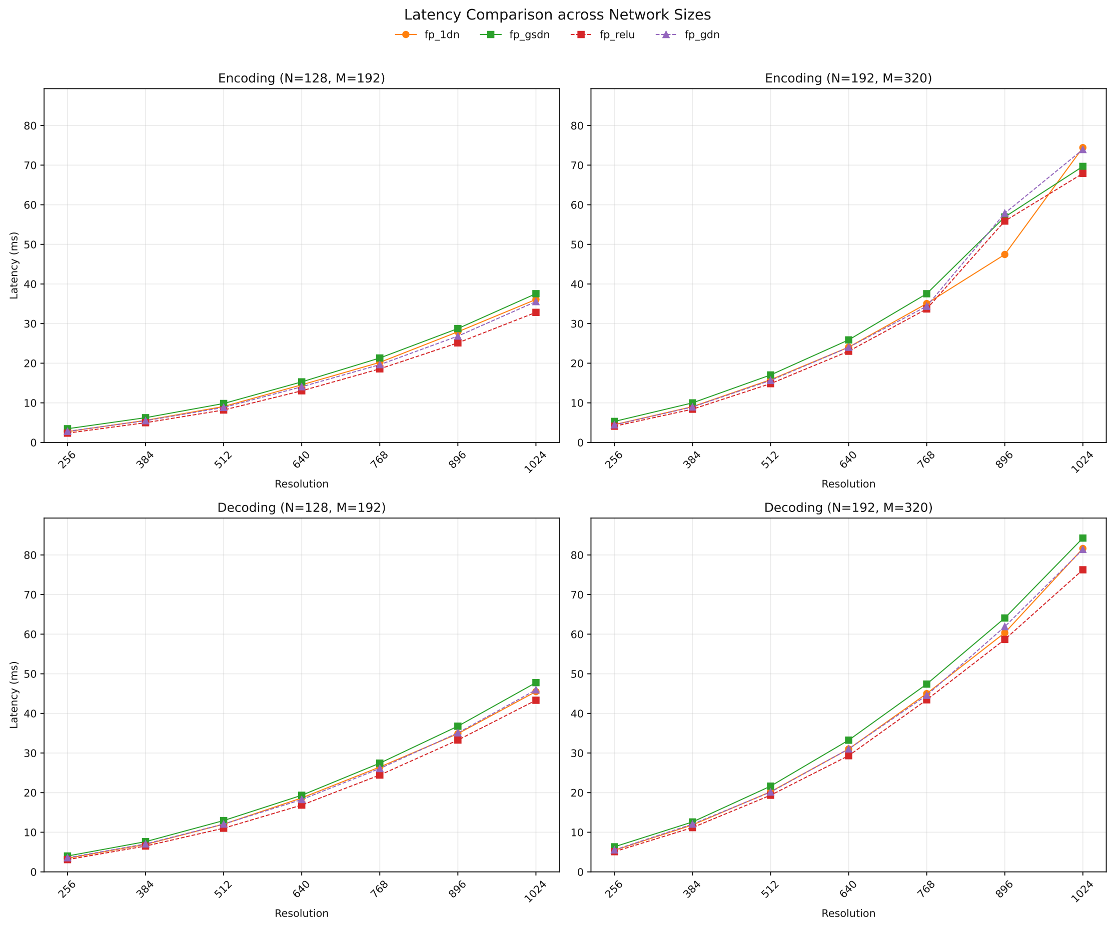

Figure 7.5: Encoding and decoding latency performance across different variants of the factorized prior architecture. The networks are tested in two capacities: N=128, M=192 and N=192, M=320, and across image resolutions ranging from 256x256 to 1024x1024 pixels. All measurements are in milliseconds.

We compare the encoding and decoding latency of each factorized prior variant in Figure 7.5. The left side shows results for the small network size of N=128 and N=192. The fp__relu variant achieves the lowest encoding and decoding times across all resolutions, while the fp__gsdn (Section 5.2.2) variant performs the slowest. ReLU is computationally inexpensive, so we confirm that utilizing the fp__relu activation function in the non-linearity block achieves the fastest encoding and decoding times. GSDN is an extension of GDN, with an addition of a subtractive-divisive normalization causing worse encoding and decoding performance than the fp__gdn variant. The right side of the figure shows the encoding and decoding latencies evaluated on a more significant number of channels (N=192, M=320). The increase in network size causes higher latencies across all variants, almost doubling the measured encoding and decoding times. On average, the fp__relu variant is the fastest, while the fp__gsdn variant is the slowest. However, the fp__1dn variant shows significantly better results for the 896x896 resolution while achieving similar results to fp__relu in lower resolutions.

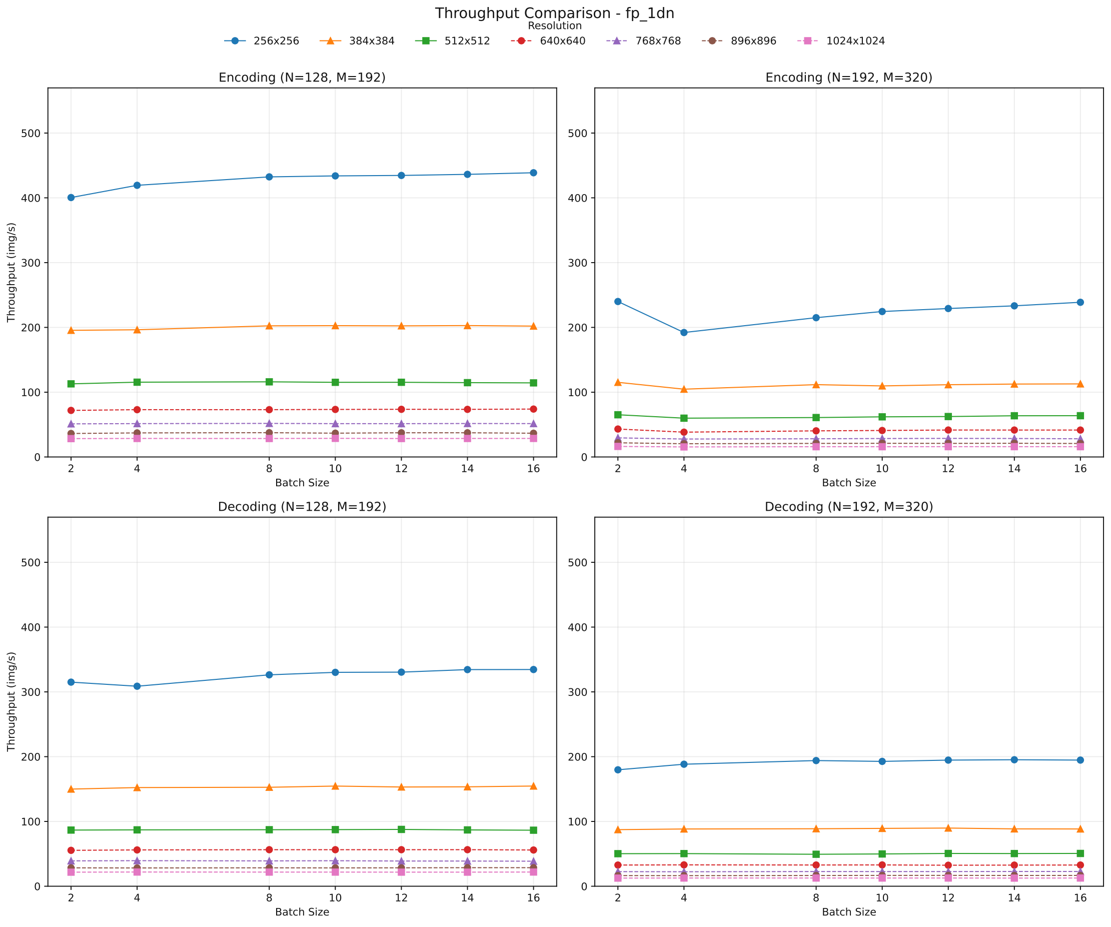

Figure 7.6: The encoding and decoding throughput of the factorized prior 1DN variant.

Figure 7.6 shows the throughput performance of the 1DN variant in both network configurations. The most significant difference can be observed at the 256px resolution in the small-sized network configuration. With the increasing number of images per batch, the small network encoded and decoded more images per second. Higher-resolution images cause a constant throughput regardless of batch size or network configuration.

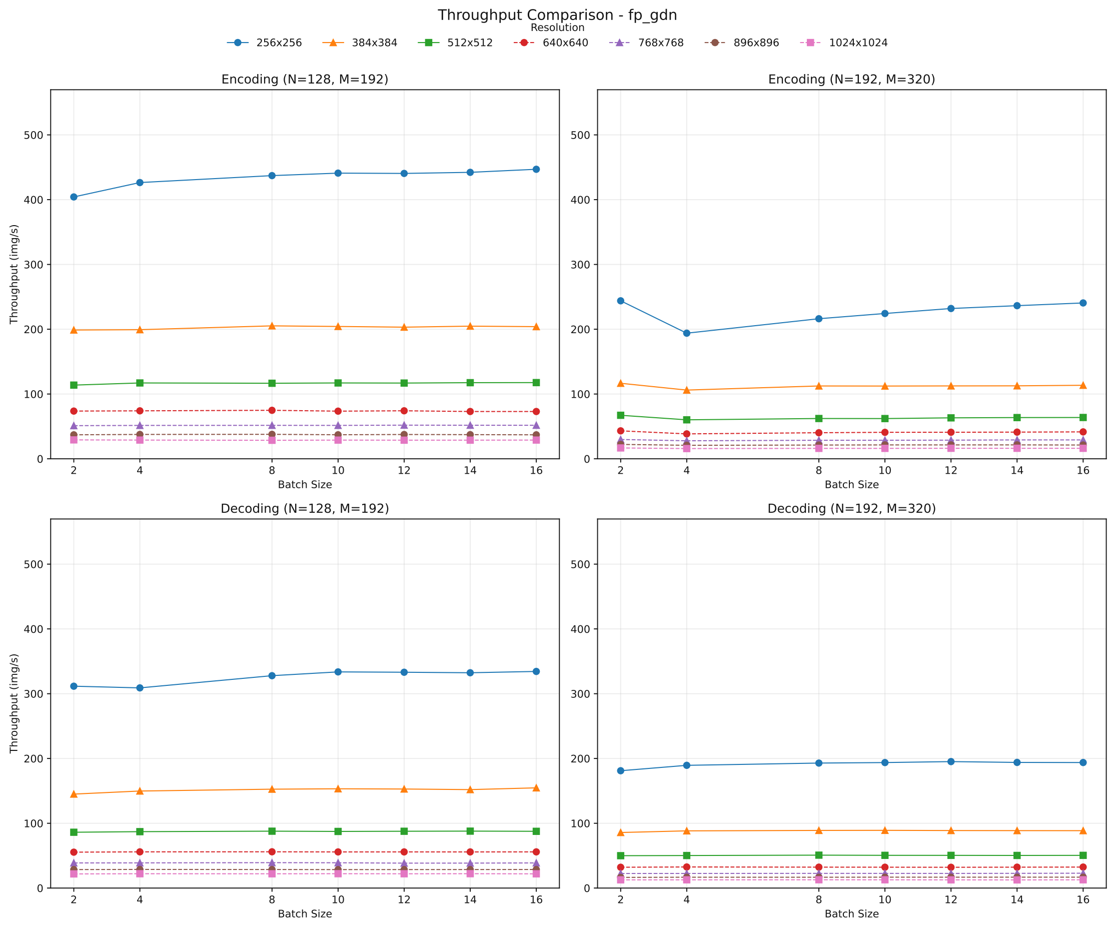

Figure 7.7: The encoding and decoding throughput of the factorized prior GDN variant.

Figure 7.7 shows the throughput performance of the GDN variant. It follows the same trend as the 1DN variant yet performs slightly better.

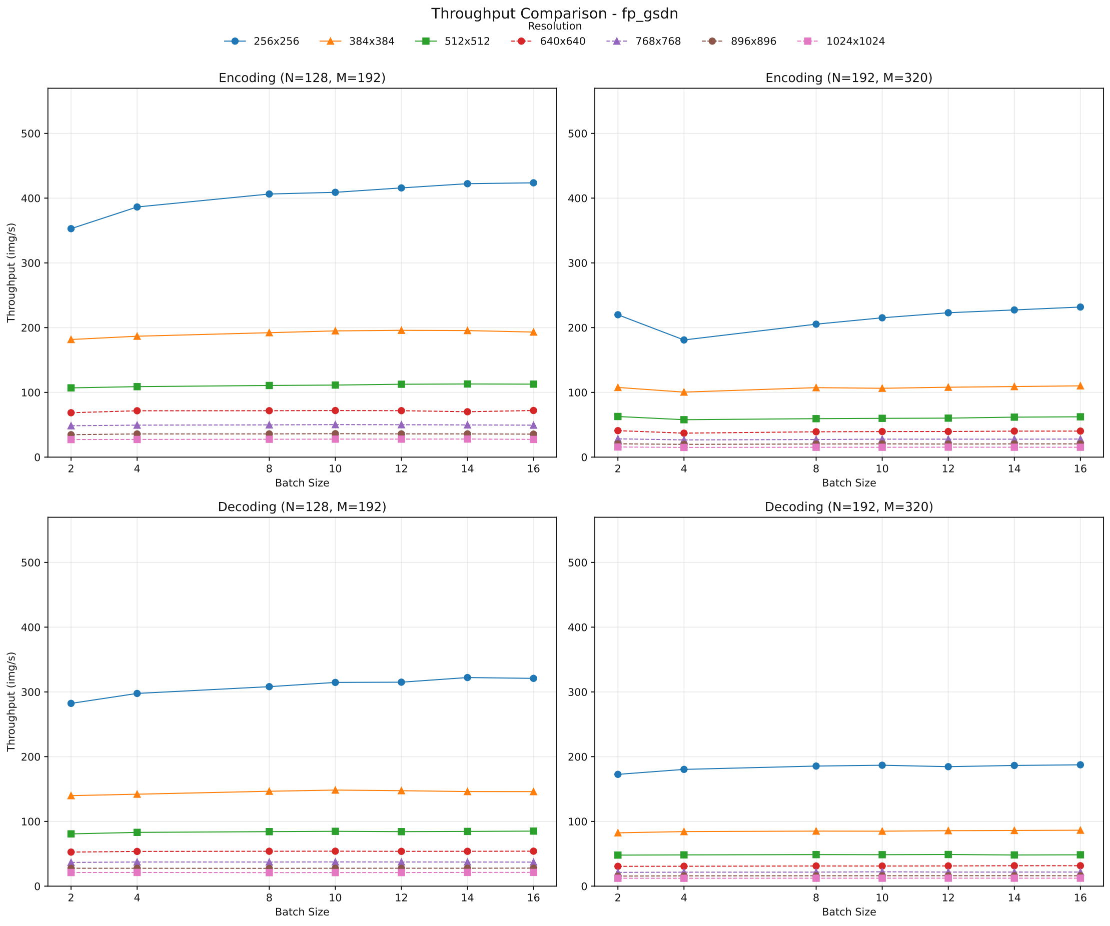

Figure 7.8: The encoding and decoding throughput of the factorized prior GSDN variant.

Figure 7.8 shows the throughput performance of the GSDN variant. The general performance gains and losses are relatively similar to previous variants. This variant performs the worst due to additional computational complexity.

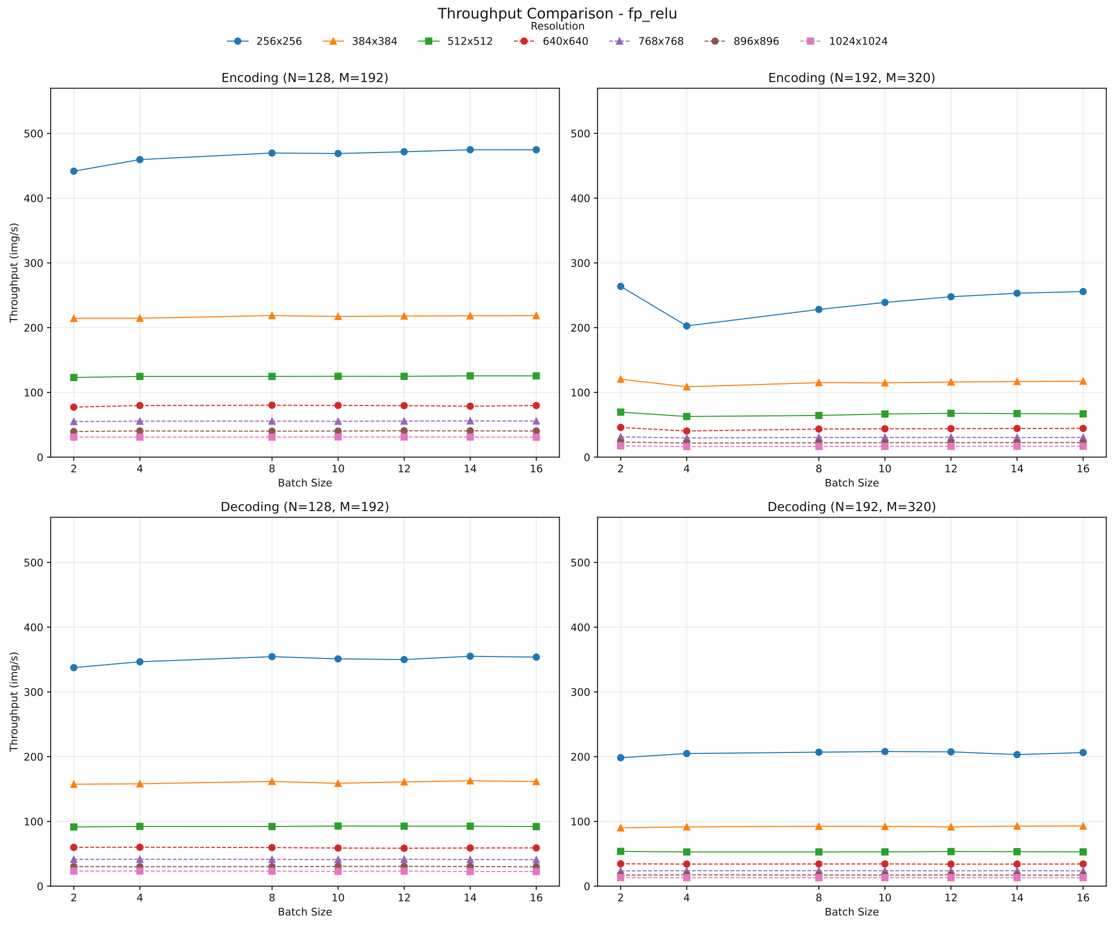

Figure 7.9: The encoding and decoding throughput of the factorized prior ReLU variant.

Figure 7.9 shows the throughput performance of the ReLU variant. It is the fastest-performing variant, almost achieving 500 img/s at 256px resolution and batch sizes 8-16. As expected, the computationally inexpensive ReLU function performs the best in latency and throughput tests.

7.3.4 Discussion

The GSDN variant reports worse performance and slower coding times than the baseline GDN performance and shows deficiencies when used in a small network, such as the factorized prior architecture. The throughput ranges from 20 images per second at 1024px to 420 images per second at 256px. The BD-Rate is 3% higher than the baseline, meaning 3% higher costs in bitrate to output the same quality of images. Figure 7.5 shows that the GSDN variant consistently achieves, on average, the highest encoding and decoding times. The authors tested GSDN on a more extensive network with a hyperprior supported by global reference and local context. Modifying the original GDN function proves computationally costly, not achieving the desired improvement in simple networks such as the factorized prior architecture. The ReLU variant achieves results similar to those of GSDN in perceptual quality, performing 3% worse than the baseline in the BD-Rate metric. However, the low computational cost of the ReLU function allows for the fastest encoding and decoding times among all variants. It is the best-performing variant in encoding and decoding latencies and throughput. The 1DN variant is a minor modification of the baseline. Figure 7.5 shows almost identical performance in encoding and decoding latencies. Analogously, the throughput performance behaves equally to its baseline. Table 7.1 shows an insignificant performance degradation in BD-Rate by 0.5%.

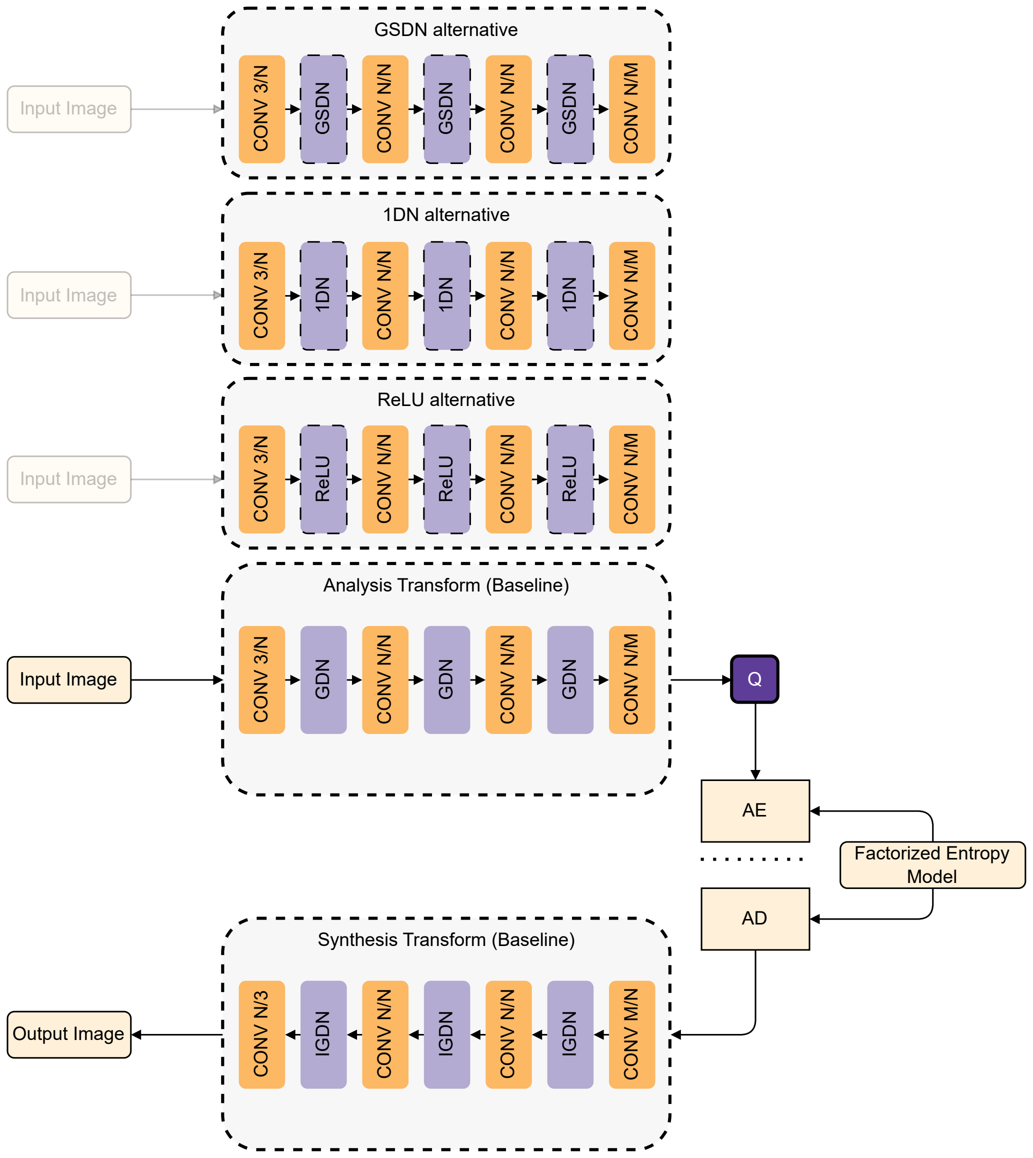

Figure 7.10: Factorized Prior baseline architecture consisting of convolutional layers and non-linear activations (GDN). The input image gets encoded into a latent representation, quantized (Q), and arithmetically encoded (AE) into a bitstream. The stream can be decoded with an arithmetic decoder (AE). Both AE and AD share a factorized entropy model. The baseline synthesis transform comprises deconvolutions and inverse GDN (IGDN)[11] elements to synthesize an output image. We visualize the modified architectures with the analysis transform element. We replace the GDN activations with ReLU, 1DN, and GSDN activations in the synthesis and analysis transforms while keeping the rest of the architecture unchanged.

7.4 Mean-Scale Hyperprior Model

Our baseline is the architecture of the mean-scale hyperprior model by Ballé et al. The baseline configuration utilizes GDN in its non-linear component. Other configurations utilize GSDN, 1DN, or ReLU instead.

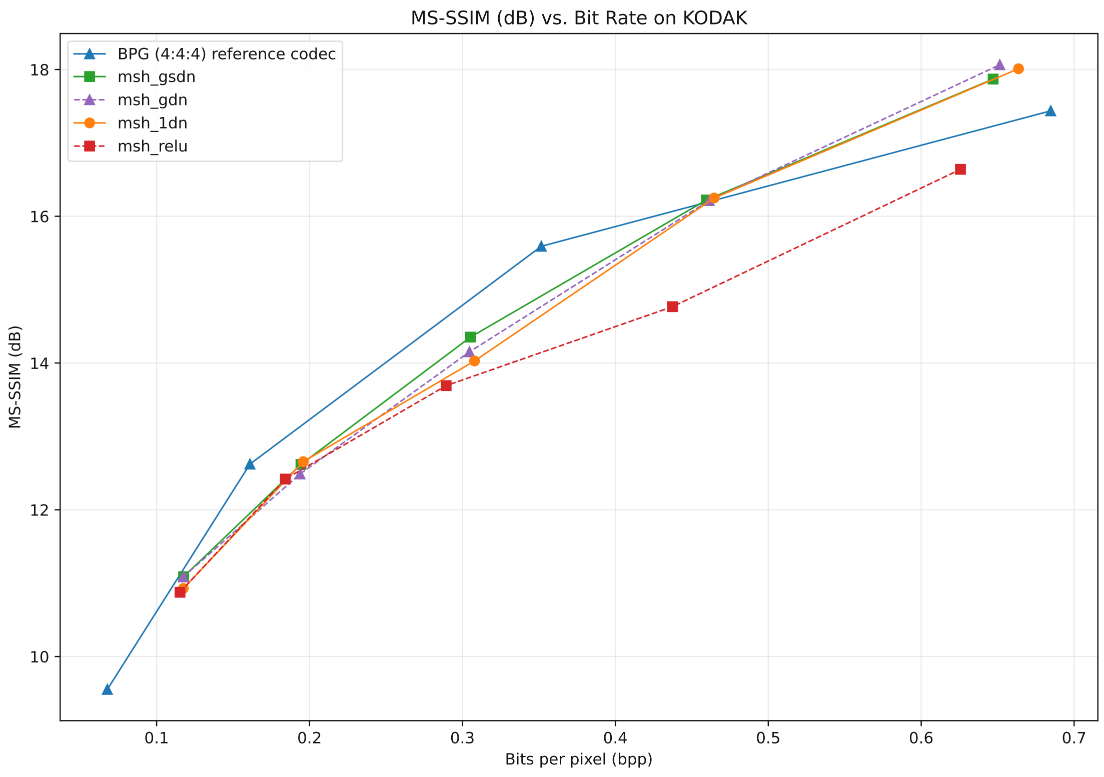

Figure 7.11: Performance evaluation in MS-SSIM of Mean-Scale Hyperprior Variants on the Kodak dataset.

7.4.1 R-D Performance

We evaluate each configuration on the Kodak and CLIC datasets. We measure Bits-per-pixel, MS-SSIM, LPIPS, and PSNR metrics. Figures 7.11 and 7.12 demonstrate the r-d performance curves of all configurations compared to the BPG reference codec and evaluated on the Kodak dataset. Similarly, Figures 7.13 and 7.14 show the evaluation performed on the CLIC dataset.

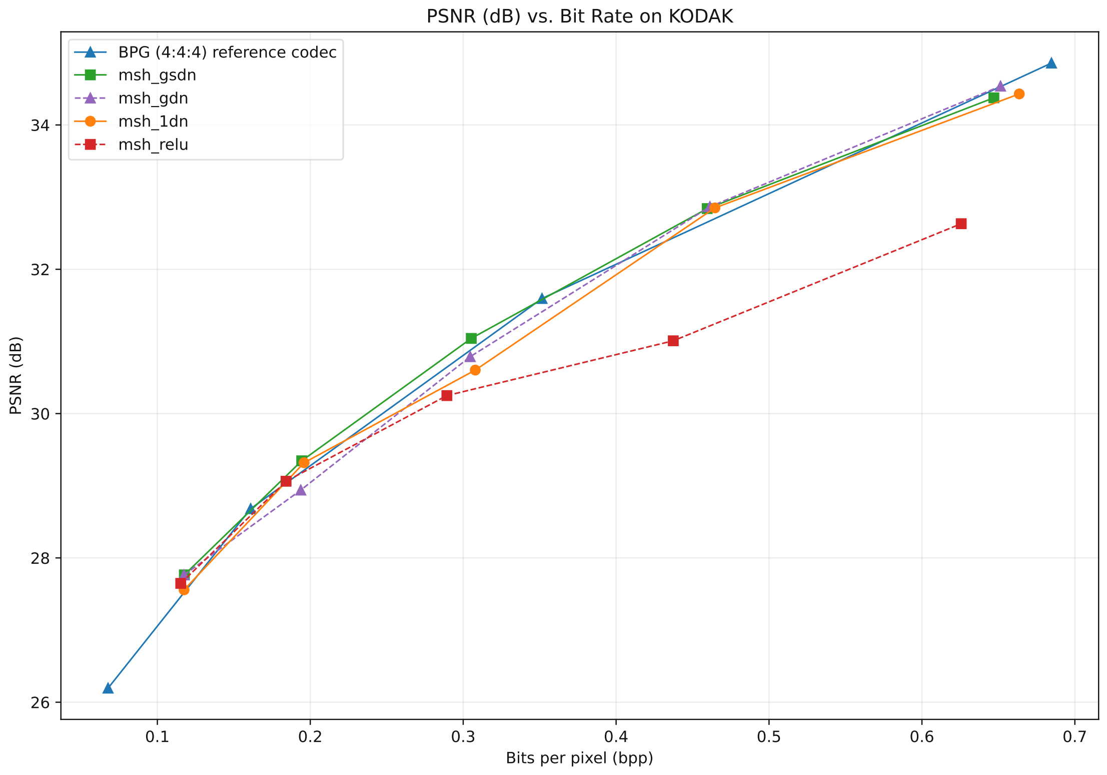

Figure 7.12: Performance evaluation in PSNR of Mean-Scale Hyperprior Variants on the Kodak dataset.

Model	Bjøntegaard Delta Rate↓	Bjøntegaard Delta PSNR↑
msh_relu	20.02%	-0.68 dB
msh_gdn	3.89%	-0.15 dB
msh_1dn	3.72%	-0.14 dB
msh_gsdn	-0.55%	0.01 dB

Table 7.6: Bjøntegaard Delta Rate and Bjøntegaard Delta PSNR of mean-scale hyperprior variants with modified activation functions. The anchor codec is BPG(4:4:4). All codecs are evaluated on the Kodak dataset.

The CLIC dataset consists of images that are easier to compress than the Kodak dataset. Tables 7.7 to 7.10 show greater compression performance when evaluated on CLIC. The reference codec is evaluated using the same data to ensure fair comparison.

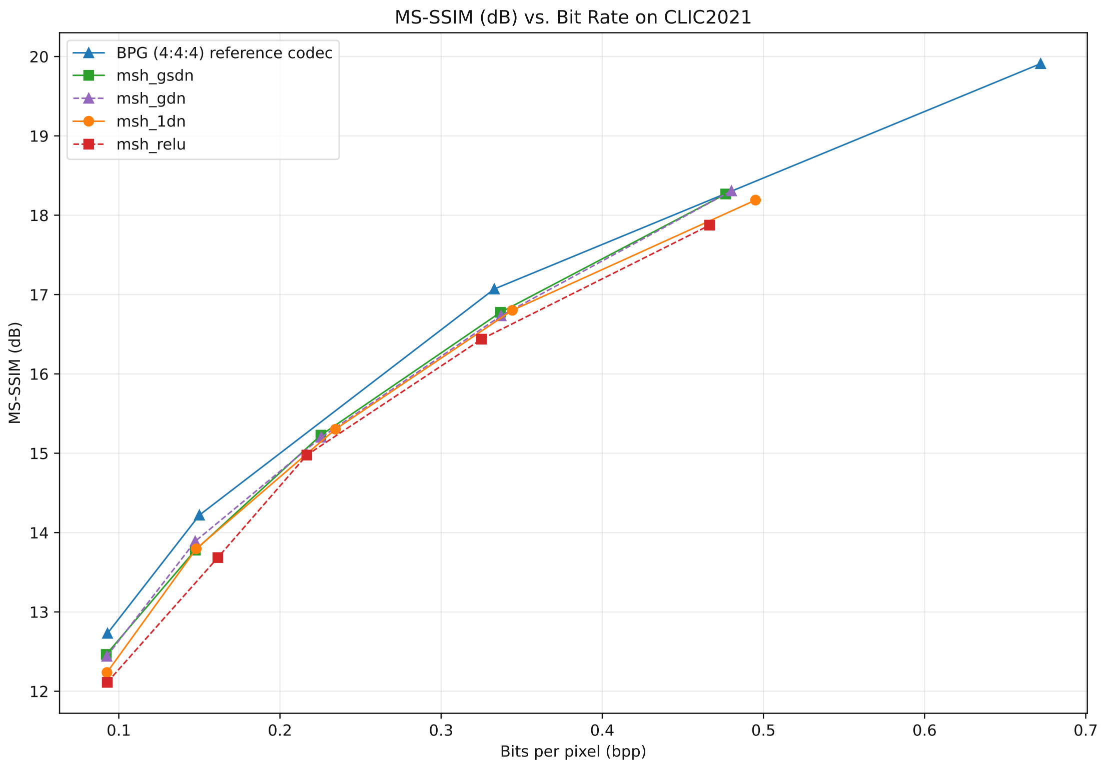

Figure 7.13: Performance evaluation in MS-SSIM of Mean-Scale Hyperprior Variants on the CLIC dataset.

When evaluated on the Kodak dataset, the r-d performance of all mean-scale hyperprior configurations closely follows the performance of the reference codec (Figure 2). The BD metrics shown in Table 5 reveal variations in performance between variants compared to the previous architecture. The GSDN variant achieves a slight improvement over the

Kodak				CLIC2021			
bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓	bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓
0.118	11.088	27.765	0.445	0.092	12.464	29.774	0.447
0.194	12.618	29.346	0.392	0.147	13.779	31.210	0.420
0.305	14.353	31.042	0.333	0.225	15.228	32.694	0.380
0.460	16.222	32.841	0.276	0.337	16.776	34.232	0.339
0.647	17.870	34.376	0.227	0.477	18.268	35.688	0.296

Table 7.7: Metrics for the mean-scale hyperprior model with GSDN as the activation function

anchor codec and a significant improvement over the baseline in terms of BD-Rate and BD-PSNR performing the best across all configurations. In contrast, the GDN and 1DN

Kodak				CLIC2021			
bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓	bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓
0.115	10.876	27.647	0.458	0.093	12.112	29.405	0.466
0.184	12.419	29.062	0.403	0.161	13.685	31.047	0.424
0.289	13.692	30.249	0.354	0.217	14.977	32.393	0.387
0.437	14.768	31.009	0.316	0.325	16.438	33.789	0.345
0.626	16.640	32.631	0.261	0.467	17.875	35.199	0.304

Table 7.8: Metrics for the mean-scale hyperprior model with ReLU as the activation function

variants show nearly identical performance results, as seen in Tables 2 and 3. The ReLU variant shows a significant drop off in performance at 20% BD-Rate. When evaluated on the CLIC dataset, all variants show a better performance pattern in metrics, such as MS-SSIM, PSNR, and bits-per-pixel. At the same time, LPIPS scores are superior on the Kodak dataset.

Kodak				CLIC2021			
bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓	bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓
0.117	10.927	27.557	0.451	0.093	12.237	29.576	0.456
0.196	12.656	29.319	0.389	0.148	13.795	31.163	0.413
0.308	14.029	30.603	0.341	0.234	15.302	32.657	0.379
0.465	16.249	32.851	0.272	0.344	16.801	34.230	0.334
0.664	18.011	34.430	0.232	0.495	18.190	35.559	0.301

Table 7.9: Metrics for the mean-scale hyperprior model with 1DN as the activation function

The CLIC dataset has a higher performance gap between the anchor codec and the variants. This suggests that more competitive results are achieved on traditional photographic content (Kodak) instead of diverse images (CLIC).

Kodak				CLIC2021			
bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓	bpp↓	MS-SSIM↑	PSNR↑	LPIPS↓
0.117	11.085	27.764	0.446	0.093	12.441	29.774	0.454
0.194	12.488	28.940	0.393	0.147	13.890	31.233	0.417
0.305	14.152	30.789	0.337	0.226	15.201	32.681	0.377
0.461	16.214	32.867	0.276	0.337	16.732	34.231	0.339
0.651	18.065	34.538	0.222	0.480	18.306	35.674	0.294

Table 7.10: Metrics for the mean-scale hyperprior model with GDN as the activation function

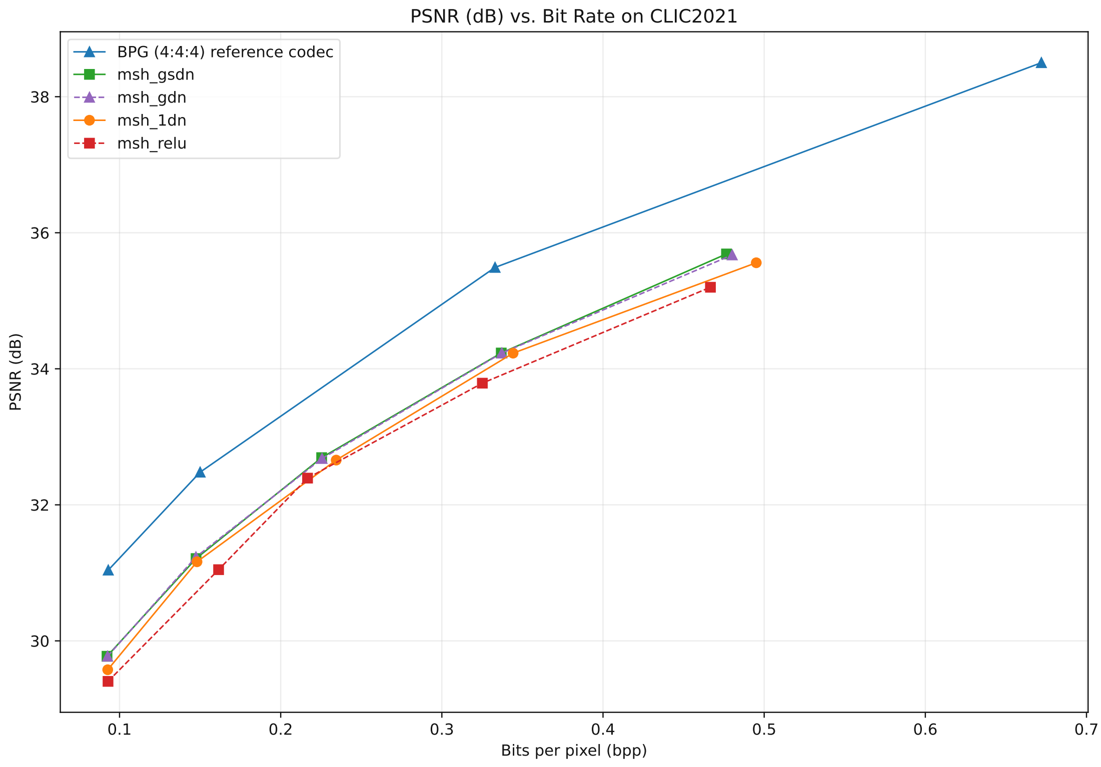

Figure 7.14: Performance evaluation in PSNR of Mean-Scale Hyperprior Variants on the CLIC dataset.

7.4.2 Latency

We measure the latency in encoding and decoding times across 7 square resolutions and two network sizes. The initial resolution starts at 256px and increases by 128px until 1024px. The small network size consists of N=128, M=192 channels, while the bigger network consists of N=192, M=320 channels. We repeat those measurements 50 times and compute the standard deviation.

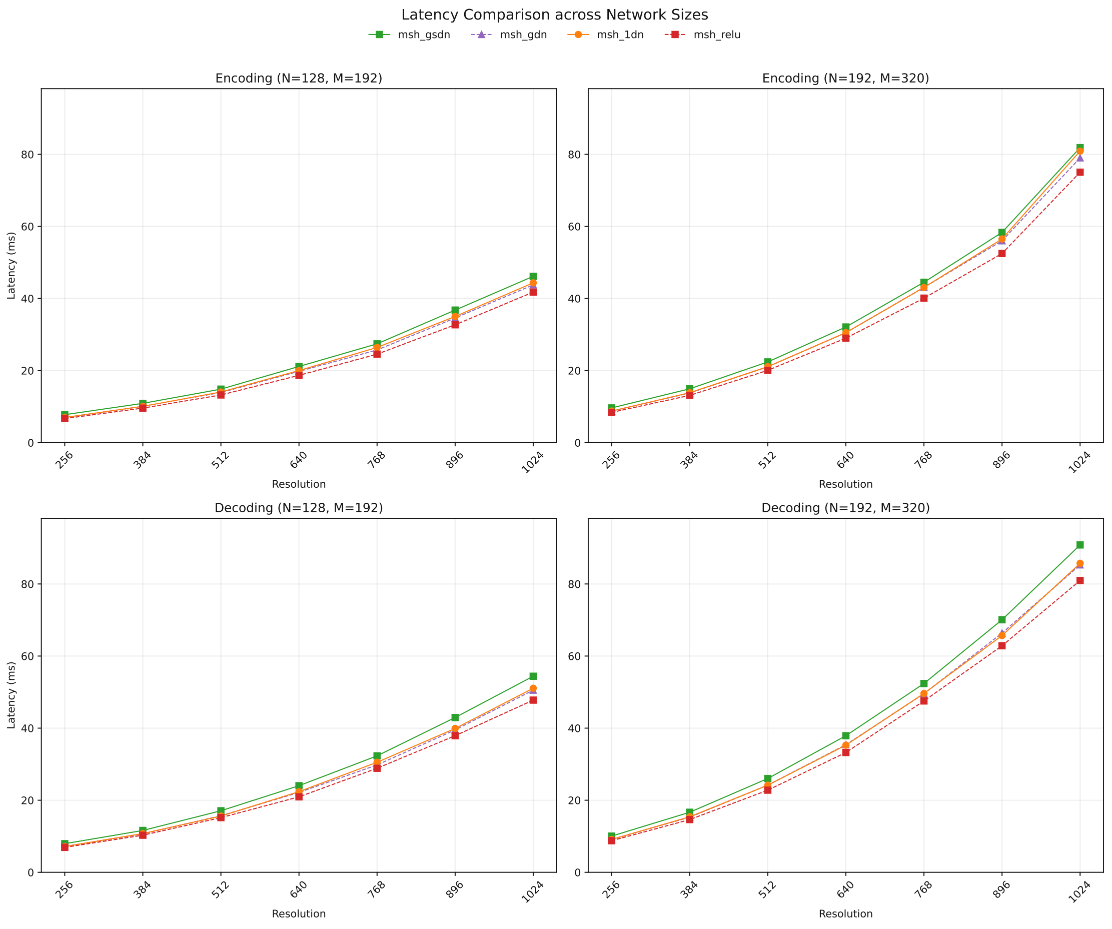

Figure 7.15: Encoding and decoding latency performance across different variants of the mean-scale hyperprior architecture.

Figure 7.15 shows the latencies divided vertically by encoding and decoding and divided horizontally by network size. The y-axis is equal in all subplots to improve readability. The latencies were measured in milliseconds. The GSDN variant performs the slowest, with latencies varying from 7ms at 256px to 46ms at 1024px. The ReLU variant performs the fastest, with latencies varying from 6ms at 256px to 41ms at 1024px. The encoding latency gap between the slowest and fastest variant at low resolutions is 1ms, increasing to 5ms at 1024px. For decoding latencies, it is 1ms at 256px and 7ms at 1024px.

7.4.3 Throughput

We measure throughput across all resolutions, network sizes, and batch sizes. The resolution and network configurations are identical as in the latency measurements. The batch size starts at two and increases to 4, 8, 10, 12, 14, and 16, respectively. We repeat those measurements 50 times as well.

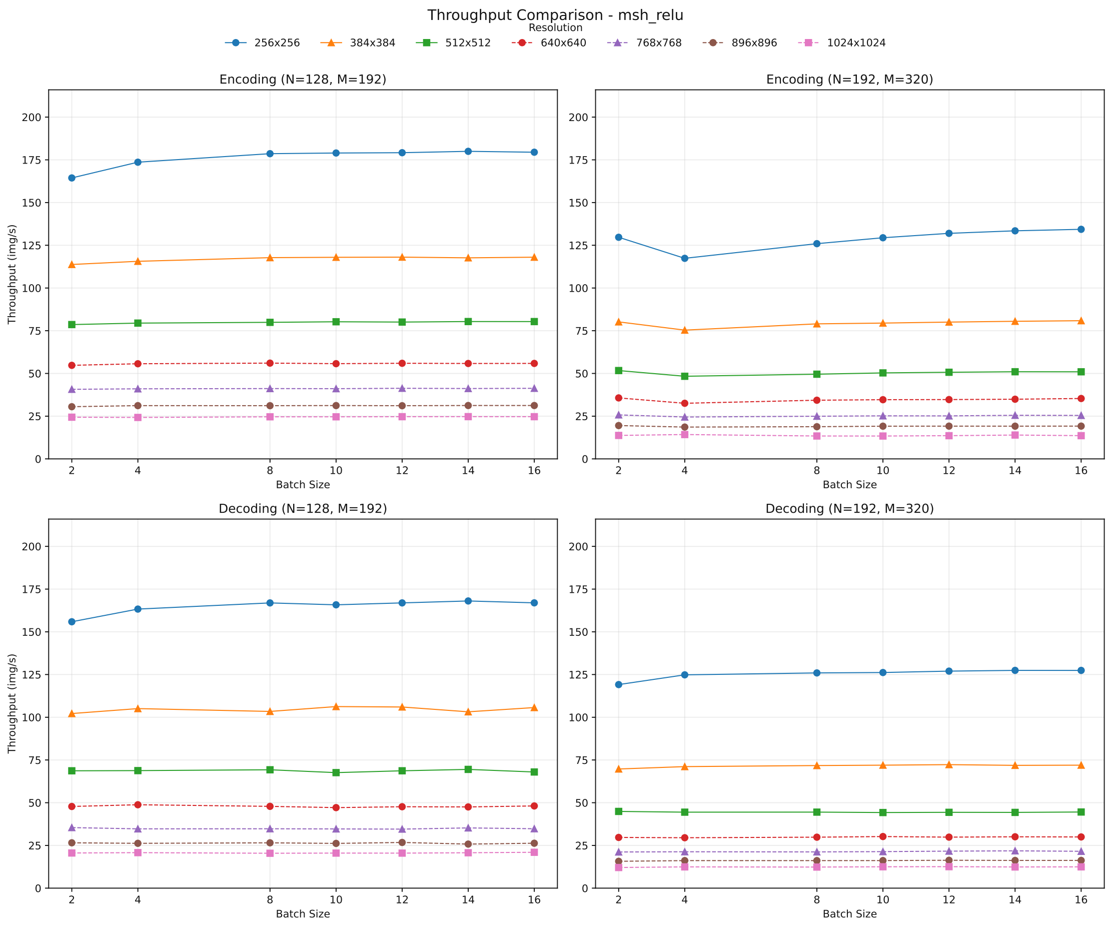

Figure 7.16: The encoding and decoding throughput of the mean-scale hyperprior ReLU variant.

Figure 7.16 shows the throughput performance for the ReLU variant. For the resolution of 256px and a small number of channels, each increase in the batch size also increases the encoding throughput. With the increase in resolution, the throughput gains diminish entirely. This variant peaks at 179 images per second. Generally, increasing the number of channels penalizes the throughput growth gained from bigger batch sizes.

Figure 7.17 shows the throughput performance for the GDN variant. The baseline performs slightly worse than the ReLU variant but generally follows the same trend. In networks with more channels, the throughput dips at a batch size of 4 images, with the highest drop at the lowest resolution.

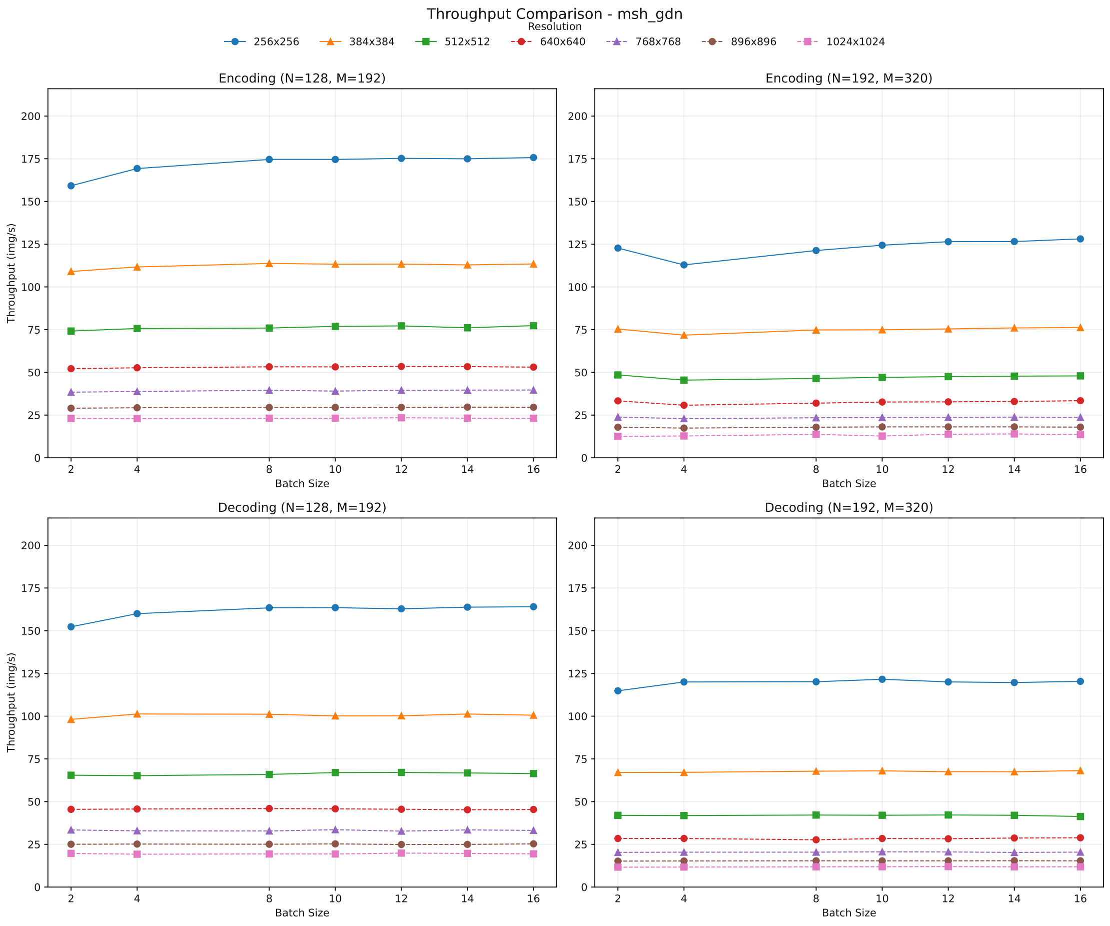

Figure 7.17: The encoding and decoding throughput of the mean-scale hyperprior GDN variant.

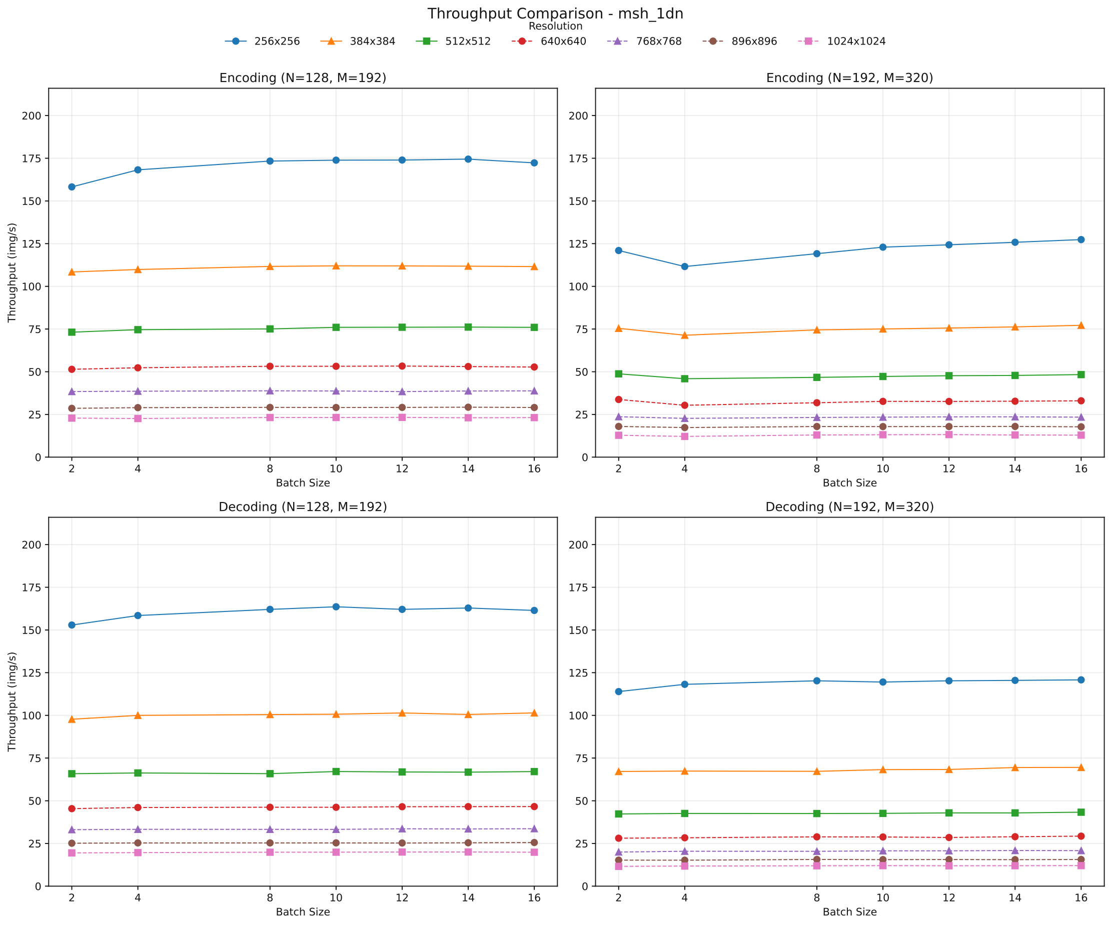

Figure 7.18: The encoding and decoding throughput of the mean-scale hyperprior 1DN variant.

Figure 7.18 shows the throughput performance for the 1DN variant. It matches almost exactly the performance of the GDN variant. Table 7.6 and Figure 7.15 confirm that.

Figure 7.19 shows the throughput of the GSDN variant. It performs the worst out of the evaluated variants with at most 170 images per second. However, the difference between the fastest and GSDN variants is less than 10 images per second.

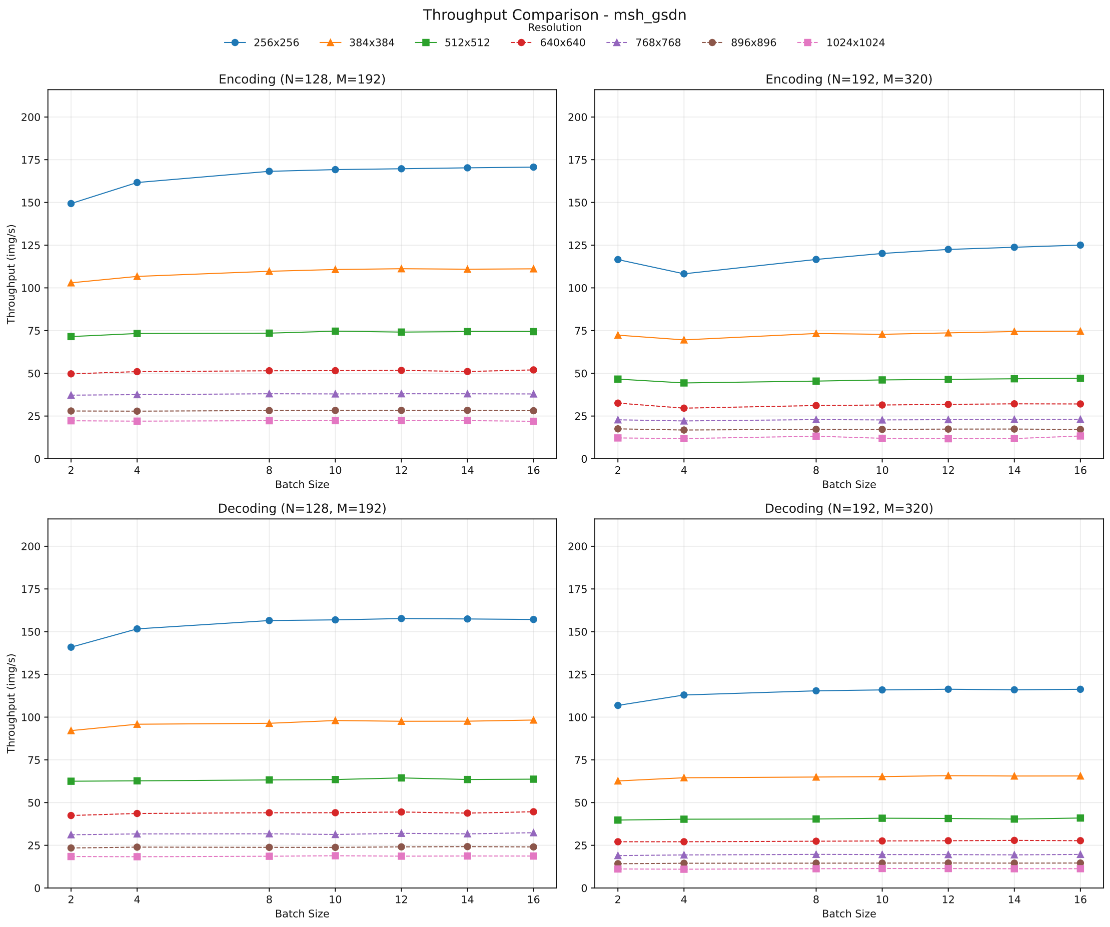

Figure 7.19: The encoding and decoding throughput of the mean-scale hyperprior GSDN variant.

7.4.4 Discussion

The GSDN variant reports the slowest encoding and decoding times and the lowest throughput. This trend can be observed in both factorized prior and mean-scale hyperprior architectures and small and large network configurations. However, the GSDN variant achieves the best r-d performance in the mean-scale hyperprior architecture. Unlike the factorized prior architecture, msh uses side information with its hyperprior component. Tables 7.1 and 7.6 show that deeper networks benefit the more computationally expensive GSDN component.

The ReLU variant reports the fastest encoding and decoding times and the highest throughput, which is valid for both architectures and network sizes. However, the quality gap in r-d curves became larger compared to the factorized prior architecture. It implies that the simple ReLU component has a more significant impact on deeper networks. Figures 7.11 and 7.12 show that its non-linear modeling capabilities are far weaker than those of the GDN, 1DN, and GSDN variants.

The 1DN variant performs identically as the baseline variant (GDN) on average. The throughput performance in Figures 7.15, 7.17 and 7.18 shows no significant gaps in performance. The simplification in 1DN of the GDN variant has no impact on encoding and decoding speeds. Table 3 shows an almost identical BD-Rate and BD-PSNR, implying identical r-d performance. The exact results shown in Tables 7.9 and 7.10 support the argument, as no significant differences in performance are found.

CHAPTER 8

Conclusion

This chapter concludes the thesis by answering the research questions posed in the beginning, summarizing the evaluation results and giving an outlook on the future work.

8.1 Research Questions

RQ1: How do the finer-grained design decisions impact rate-distortion performance and perceptual quality? The evaluation in Chapter 7 shows measurements for two architectures in multiple configurations. The configurations vary in network depth and core components. We tested the non-linearity component in four variants. The results show that increasing the network size considerably reduces the throughput and increases the encoding and decoding latencies. However, the rate-distortion performance is significantly higher for all variants. Interestingly, the GSDN variant was ranked lower than other variants in a shallow network while performing the best in the deeper network. The model depth influences architectural components to different degrees. For instance, the ReLU variant performed within 1% compared to other shallow network variants. However, the ReLU variant performed 16% worse in a deep network than other variants. Simple variants do not benefit in metrics for perceptual quality as much as more complex variants from increasing the depth of the network.

RQ2: How can we classify neural compression methods by their core components? We find relevant papers through the snowballing, screening, and deduplication methodology described in Chapter 4. The focus is on publications within the lossy learned image compression field. Publications dedicated to lossless learned image compression or object detection were discarded. The inherent property of snowballing is linking publications together. For instance, many works are related and based on the works by Ballé et al. We classify publications into common groups, which may consist of areas of opposite stakes, such as serial and parallel processes. In our taxonomy, publications are cited multiple times in different groups. The taxonomy identifies core elements of the lossy learned image compression field: Quantization, Architectural Archetypes, Side Information, Variable Rate, and Context Models. These future-proof groups are the core components of state-of-the-art frameworks that describe the publications found in Tables 5.1 to 5.5. The systematic analysis of connected publications reveals one possible permutation, which we express as multiple isolated taxonomies. The resulting categorization of core components supports the development of the Benchpresso framework, which offers interfaces to perform and benchmark fine-grained decisions.

RQ3: What are current publication trends and focus areas in neural data compression? The taxonomy in Chapter 5 categorizes the field of lossy learned data compression into core components. Each had a period of growth. Early progress can be attributed to quantization mechanisms and hierarchical priors: seven publications on quantization, three on architectural archetypes, five on hierarchical priors, one on context models, and one on variable rate from 2016 to 2018. In the following years, 2019 and 2020, the number of publications for hierarchical priors, architectural archetypes, variable rate, and context models increased. Table 5.6 shows that more current publications focus mainly on increasing parallelism, introducing more side information components, reducing local and global redundancies, and improving efficiency. In summary, the first publication focused on establishing the first end-to-end optimized learned image compression architectures and later on shifted the focus to increasing the visual quality. Afterward, the next goal was to improve encoding and decoding latency and efficiency while avoiding quality degradation.

8.2 Future Work

This section provides an outlook on possible continuations of this thesis. It highlights open research left to be done and further questions to be answered separately.

8.2.1 Pypi Package and Open-Source

Given the requirements and system design, the next logical step is to open-source the software. In the future, the Benchpresso project needs to be refactored and brought to a state where it is viable to be open-sourced. In addition, we plan to turn the project into a Pypi package to integrate its features into existing projects easily. Special annotations would register objects as models and datasets, which can be trained and benchmarked.

8.2.2 Combining novel with existing solutions

This thesis focused heavily on established solutions and based all its variants mainly on two architectures. In the future, more exotic architectures should be tested. Given the challenge of mismatched channels and implementation differences, not all combinations and variants are viable. It would be interesting to measure the impact of attention modules on small networks, such as the factorized prior compared to their original deep networks. With the given flexibility of benchpresso, it is possible to mix various architectures by different authors easily.

8.2.3 Extended Literature Review

An follow-up of our work may consider work on *perceptual compression*. Perceptual compression aims to maintain perceptual quality scores, which is challenging especially at lower bitrates [73]. Typically, methods rely on generative models to model the source distribution [4, 54, 73]. Additionally, a follow-up may focus on *task-oriented compression*, where the objective is not limited to image reconstruction [81, 31, 38, 37].

Overview of Generative AI Tools Used

I have used ChatGPT in Version 3.5 and 4.0 as an aid. ChatGPT helped me do research. I have generated summaries in the form of bullet point lists for my notebook, which has helped me find relevant papers more easily.

I have used Grammarly to correct my text. Grammarly offers features similar to MS Word. Grammarly gives suggestions on how to fix grammar and punctuation. I have not used its text generation feature. This AI Tool is limited to the English language.

I have used GitHub Copilot in its student version. This AI Tool helped me write code. I did not use its prompt feature and, at most, accepted automated suggestions.

Übersicht verwendeter Hilfsmittel

Ich habe ChatGPT in Versionen 3.5 und 4.0 als Hilfsmittel verwendet. ChatGPT hat mir bei der Recherche geholfen. Ich habe Zusammenfassungen in Form von nummerierten Listen für mein Notizbuch generiert, um einfacher relevante Arbeiten zu finden.

Ich habe Grammarly verwendet um meinen Text zu korrigieren. Grammarly bietet ähnliche Funktionen wie MS Word an. Man bekommt Vorschläge sowohl bei grammatikalischen als auch bei Interpunktionsfehlern. Ich habe keinen Text mit Grammarly generiert. Das KI-Tool ist auf die englische Sprache begrenzt.

Ich habe Github Copilot in Student-Version verwendet. Dieses KI-Tool hilft Code zu schreiben. Ich habe das Prompt Feature nicht verwendet. Ich habe nur die automatisierten Vorschläge akzeptiert.

List of Figures

List of Tables

Bibliography

[1] Eirikur Agustsson, Fabian Mentzer, Michael Tschannen, Lukas Cavigelli, Radu Timofte, Luca Benini, and Luc Van Gool. Soft-to-hard vector quantization for end-to-end learned compression of images and neural networks. *CoRR*, abs/1704.00648, 2017.

[2] Eirikur Agustsson, Fabian Mentzer, Michael Tschannen, Lukas Cavigelli, Radu Timofte, Luca Benini, and Luc Van Gool. Soft-to-hard vector quantization for end-to-end learning compressible representations, 2017.

[3] Eirikur Agustsson and Lucas Theis. Universally quantized neural compression, 2020.

[4] Eirikur Agustsson, Michael Tschannen, Fabian Mentzer, Radu Timofte, and Luc Van Gool. Generative adversarial networks for extreme learned image compression, 2019.

[5] Mohammad Akbari, Jie Liang, Jingning Han, and Chengjie Tu. Learned variable-rate image compression with residual divisive normalization, 2019.

[6] Mohammad Akbari, Jie Liang, Jingning Han, and Chengjie Tu. Learned multi-resolution variable-rate image compression with octave-based residual blocks, 2020.

[7] Mohammad Akbari, Jie Liang, Jingning Han, and Chengjie Tu. Learned bi-resolution image coding using generalized octave convolutions. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(8):6592–6599, May 2021.

[8] Sharon Ayzik and Shai Avidan. Deep image compression using decoder side information, 2020.

[9] Johannes Ballé, Valero Laparra, and Eero P. Simoncelli. End-to-end optimization of nonlinear transform codes for perceptual quality. *CoRR*, abs/1607.05006, 2016.

[10] Johannes Ballé, Philip A. Chou, David Minnen, Saurabh Singh, Nick Johnston, Eirikur Agustsson, Sung Jin Hwang, and George Toderici. Nonlinear transform coding, 2020.

[11] Johannes Ballé, Valero Laparra, and Eero P. Simoncelli. Density modeling of images using a generalized normalization transformation, 2016.

[12] Johannes Ballé, Valero Laparra, and Eero P. Simoncelli. End-to-end optimized image compression, 2017.

[13] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior, 2018.

[14] Johannes Ballé, George Toderici, Luca Versari, Nick Johnston, Lucas Theis, Andrey Norkin, Krishna Rapaka, Erfan Noury, Ross Cutler, Radu Timofte, Fabien Racapé, Yan Ye, Ali Bilgin, Andrew Segall, Balu Adsumilli, and Ramzi Khsib. Challenge on learned image compression. `https://compression.cc/tasks/#image`. Last accessed 7th, August, 2024.

[15] Nabajeet Barman, Maria G. Martini, and Yuriy Reznik. Bjøntegaard delta (bd): A tutorial overview of the metric, evolution, challenges, and recommendations, 2024.

[16] Gisle Bjøntegaard. Calculation of average psnr differences between rd-curves, 2001.

[17] Yochai Blau and Tomer Michaeli. The perception-distortion tradeoff. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, June 2018.

[18] Sebastian Bosse, Dominique Maniry, Klaus-Robert Müller, Thomas Wiegand, and Wojciech Samek. Deep neural networks for no-reference and full-reference image quality assessment. *IEEE Transactions on Image Processing*, 27(1):206–219, 2018.

[19] Jill Boyce, Karsten Suehring, Xiang Li, and Vadim Seregin. Jvet-j1010: Jvet common test conditions and software reference configurations, 07 2018.

[20] Jean Bégaint, Fabien Racapé, Simon Feltman, and Akshay Pushparaja. Compressai: a pytorch library and evaluation platform for end-to-end compression research, 2020.

[21] Jianrui Cai and Lei Zhang. Deep image compression with iterative non-uniform quantization. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 451–455, 2018.

[22] Tong Chen, Haojie Liu, Zhan Ma, Qiu Shen, Xun Cao, and Yao Wang. End-to-end learnt image compression via non-local attention optimization and improved context modeling. *IEEE Transactions on Image Processing*, 30:3179–3191, 2021.

[23] Yunpeng Chen, Haoqi Fan, Bing Xu, Zhicheng Yan, Yannis Kalantidis, Marcus Rohrbach, Shuicheng Yan, and Jiashi Feng. Drop an octave: Reducing spatial redundancy in convolutional neural networks with octave convolution, 2019.

[24] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Deep residual learning for image compression, 2019.

[25] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Learned image compression with discretized gaussian mixture likelihoods and attention modules. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 7939–7948, 2020.

[26] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Variable rate deep image compression with a conditional autoencoder, 2019.

[27] Eastman Kodak Company. Kodak lossless true color image suite. `https://r0k.us/graphics/kodak/`. Last accessed 20th, October, 2024.

[28] Ze Cui, Jing Wang, Shangyin Gao, Bo Bai, Tiansheng Guo, and Yihui Feng. Asymmetric gained deep image compression with continuous rate adaptation, 2022.

[29] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 764–773, 2017.

[30] Keyan Ding, Kede Ma, Shiqi Wang, and Eero P. Simoncelli. Comparison of full-reference image quality models for optimization of image processing systems. *International Journal of Computer Vision*, 129(4):1258–1281, Apr 2021.

[31] Yann Dubois, Benjamin Bloem-Reddy, Karen Ullrich, and Chris J. Maddison. Lossy compression for lossless prediction, 2022.

[32] Alex Tarnavsky Eitan, Eddie Smolyansky, Itay Knaan Harpaz, and Sahar Perets. Connectedpapers. `https://www.connectedpapers.com/about`. Last accessed 14th, January, 2024.

[33] Patrick Esser, Robin Rombach, and Björn Ommer. Taming transformers for high-resolution image synthesis, 2021.

[34] Haisheng Fu and Feng Liang. Learned image compression with generalized octave convolution and cross-resolution parameter estimation, 2022.

[35] Haisheng Fu, Feng Liang, Jie Liang, Yongqiang Wang, Guohe Zhang, and Jingning Han. Fast and high-performance learned image compression with improved checkerboard context model, deformable residual module, and knowledge distillation, 2023.

[36] Haisheng Fu, Feng Liang, Jianping Lin, Bing Li, Mohammad Akbari, Jie Liang, Guohe Zhang, Dong Liu, Chengjie Tu, and Jingning Han. Learned image compression with gaussian-laplacian-logistic mixture model and concatenated residual modules, 2024.

[37] Alireza Furutanpey, Philipp Raith, and Schahram Dustdar. Frankensplit: Efficient neural feature compression with shallow variational bottleneck injection for mobile edge computing. *IEEE Transactions on Mobile Computing*, 23(12):10770–10786, 2024.

[38] Alireza Furutanpey, Qiyang Zhang, Philipp Raith, Tobias Pfandzelter, Shangguang Wang, and Schahram Dustdar. Fool: Addressing the downlink bottleneck in satellite computing with neural feature compression, 2024.

[39] Amnon Geifman. The correct way to measure inference time of deep neural networks. *link: https://towardsdatascience. com/the-correct-wayto-measure-inference-time-of-deep-neural-networks-304a54e5187f*, 2020.

[40] Bernd Girod. *What's wrong with mean-squared error?*, page 207–220. MIT Press, Cambridge, MA, USA, 1993.

[41] Ali Hassani, Steven Walton, Jiachen Li, Shen Li, and Humphrey Shi. Neighborhood attention transformer, 2023.

[42] Dailan He, Ziming Yang, Weikun Peng, Rui Ma, Hongwei Qin, and Yan Wang. Elic: Efficient learned image compression with unevenly grouped space-channel contextual adaptive coding, 2022.

[43] Dailan He, Yaoyan Zheng, Baocheng Sun, Yan Wang, and Hongwei Qin. Checkerboard context model for efficient learned image compression, 2021.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[45] Yueyu Hu, Wenhan Yang, and Jiaying Liu. Coarse-to-fine hyper-prior modeling for learned image compression. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(07):11013–11020, Apr. 2020.

[46] Yueyu Hu, Wenhan Yang, Zhan Ma, and Jiaying Liu. Learning end-to-end lossy image compression: A benchmark, 2021.

[47] Chen-Hsiu Huang and Ja-Ling Wu. Unveiling the future of human and machine coding: A survey of end-to-end learned image compression. *Entropy*, 26(5), 2024.

[48] Sonain Jamil, Md. Jalil Piran, and MuhibUrRahman. Learning-driven lossy image compression; a comprehensive survey, 2022.

[49] Sonain Jamil, Md. Jalil Piran, MuhibUr Rahman, and Oh-Jin Kwon. Learning-driven lossy image compression: A comprehensive survey. *Engineering Applications of Artificial Intelligence*, 123:106361, 2023.

[50] J. Jiang. Image compression with neural networks – a survey. *Signal Processing: Image Communication*, 14(9):737–760, 1999.

[51] Wei Jiang, Jiayu Yang, Yongqi Zhai, Peirong Ning, Feng Gao, and Ronggang Wang. Mlic: Multi-reference entropy model for learned image compression. In *Proceedings of the 31st ACM International Conference on Multimedia*, MM '23. ACM, October 2023.

[52] Nick Johnston, Elad Eban, Ariel Gordon, and Johannes Ballé. Computationally efficient neural image compression, 2019.

[53] Fatih Kamisli, Fabien Racape, and Hyomin Choi. Variable-rate learned image compression with multi-objective optimization and quantization-reconstruction offsets, 2024.

[54] Byeongkeun Kang, Subarna Tripathi, and Truong Q. Nguyen. Toward joint image generation and compression using generative adversarial networks, 2019.

[55] Atefeh Khoshkhahtinat, Ali Zafari, Piyush M. Mehta, Mohammad Akyash, Hossein Kashiani, and Nasser M. Nasrabadi. Multi-context dual hyper-prior neural image compression, 2023.

[56] Jun-Hyuk Kim, Byeongho Heo, and Jong-Seok Lee. Joint global and local hierarchical priors for learned image compression, 2022.

[57] Petr Knoth, Vako Batiashvili, Valery Budko, Matteo Cancellieri, Ekaterine Chkhaidze, Artem Minakov, Suchetha N.Kunnath, Viktoriia Pavlenko, David Pride, Friedrich Summann, Halyna Torchylo, Beso Tsiklauri, Andrew Vasiliev, Anton Zhuk, and Michael Upshall. Core recommender. `https://core.ac.uk/services/recommender`. Last accessed 14th, January, 2024.

[58] Jooyoung Lee, Seunghyun Cho, and Seung-Kwon Beack. Context-adaptive entropy model for end-to-end optimized image compression, 2019.

[59] Jooyoung Lee, Seunghyun Cho, and Munchurl Kim. An end-to-end joint learning scheme of image compression and quality enhancement with improved entropy minimization, 2020.

[60] Binglin Li, Jie Liang, and Jingning Han. Variable-rate deep image compression with vision transformers. *IEEE Access*, 10:50323–50334, 2022.

[61] Mu Li, Kede Ma, Jane You, David Zhang, and Wangmeng Zuo. Efficient and effective context-based convolutional entropy modeling for image compression. *IEEE Transactions on Image Processing*, 29:5900–5911, 2020.

[62] Mu Li, Kai Zhang, Wangmeng Zuo, Radu Timofte, and David Zhang. Learning context-based non-local entropy modeling for image compression, 2020.

[63] Fangzheng Lin, Heming Sun, Jinming Liu, and Jiro Katto. Multistage spatial context models for learned image compression, 2023.

[64] Jianping Lin, Mohammad Akbari, Haisheng Fu, Qian Zhang, Shang Wang, Jie Liang, Dong Liu, Feng Liang, Guohe Zhang, and Chengjie Tu. Learned variable-rate multi-frequency image compression using modulated generalized octave convolution, 2020.

[65] Ding Liu, Bihan Wen, Yuchen Fan, and Chen Change Loy. Non-local recurrent network for image restoration, 06 2018.

[66] Haojie Liu, Tong Chen, Qiu Shen, and Zhan Ma. Practical stacked non-local attention modules for image compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.

[67] Jiaheng Liu, Guo Lu, Zhihao Hu, and Dong Xu. A unified end-to-end framework for efficient deep image compression. *arXiv preprint arXiv:2002.03370*, 2020.

[68] Jinming Liu, Heming Sun, and Jiro Katto. Learned image compression with mixed transformer-cnn architectures, 2023.

[69] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows, 2021.

[70] Ming Lu, Fangdong Chen, Shiliang Pu, and Zhan Ma. High-efficiency lossy image coding through adaptive neighborhood information aggregation, 2022.

[71] Ming Lu, Peiyao Guo, Huiqing Shi, Chuntong Cao, and Zhan Ma. Transformer-based image compression, 2021.

[72] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Conditional probability models for deep image compression, 2019.

[73] Fabian Mentzer, George Toderici, Michael Tschannen, and Eirikur Agustsson. High-fidelity generative image compression, 2020.

[74] David Minnen, Johannes Ballé, and George Toderici. Joint autoregressive and hierarchical priors for learned image compression. *CoRR*, abs/1809.02736, 2018.

[75] David Minnen and Saurabh Singh. Channel-wise autoregressive entropy models for learned image compression, 2020.

[76] Dipti Mishra, Satish Kumar Singh, and Rajat Kumar Singh. Deep architectures for image compression: A critical review. *Signal Processing*, 191:108346, 2022.

[77] Yichen Qian, Ming Lin, Xiuyu Sun, Zhiyu Tan, and Rong Jin. Entroformer: A transformer-based entropy model for learned image compression, 2022.

[78] Yichen Qian, Zhiyu Tan, Xiuyu Sun, Ming Lin, Dongyang Li, Zhenhong Sun, Hao Li, and Rong Jin. Learning accurate entropy model with global reference for image compression, 2022.

[79] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[80] Minjeong Shin, Jiahao Zhang, Jakub Nabaglo, Alexander Soen, Benjamin Readshaw, Lexing Xie, Mitchell Whitelaw, Stephen Blackburn, and Kee Siong Ng. Influenceflower. `https://influencemap.cmlab.dev/`. Last accessed 14th, January, 2024.

[81] Saurabh Singh, Sami Abu-El-Haija, Nick Johnston, Johannes Ballé, Abhinav Shrivastava, and George Toderici. End-to-end learning of compressible features, 2020.

[82] Sonehara, Kawato, Miyake, and Nakane. Image data compression using a neural network model. In *International 1989 Joint Conference on Neural Networks*, pages 35–41 vol.2, 1989.

[83] Myungseo Song, Jinyoung Choi, and Bohyung Han. Variable-rate deep image compression through spatially-adaptive feature transform, 2021.

[84] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy image compression with compressive autoencoders, 2017.

[85] George Toderici, Sean M. O'Malley, Sung Jin Hwang, Damien Vincent, David Minnen, Shumeet Baluja, Michele Covell, and Rahul Sukthankar. Variable rate image compression with recurrent neural networks, 2016.

[86] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks, 2017.

[87] Kedeng Tong, Yaojun Wu, Yue Li, Kai Zhang, Li Zhang, and Xin Jin. Qvrf: A quantization-error-aware variable rate framework for learned image compression, 2023.

[88] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning, 2018.

[89] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[90] W3Techs. W3tech: Usage statistics of image file formats for websites. `https://w3techs.com/technologies/overview/image_format`. Last accessed 4th, October, 2024.

[91] G.K. Wallace. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992.

[92] Huairui Wang, Nianxiang Fu, Zhenzhong Chen, and Shan Liu. Dynamic kernel-based adaptive spatial aggregation for learned image compression, 2023.

[93] Pichao Wang, Xue Wang, Fan Wang, Ming Lin, Shuning Chang, Hao Li, and Rong Jin. Kvt: k-nn attention for boosting vision transformers, 2022.

[94] Yaohua Wang, Yaobin Zhang, Fangyi Zhang, Ming Lin, YuQi Zhang, Senzhang Wang, and Xiuyu Sun. Ada-nets: Face clustering via adaptive neighbour discovery in the structure space, 2022.

[95] Yongqiang Wang, Feng Liang, Jie Liang, and Haisheng Fu. S2lic: Learned image compression with the swinv2 block, adaptive channel-wise and global-inter attention context, 2024.

[96] Z. Wang, E.P. Simoncelli, and A.C. Bovik. Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, volume 2, pages 1398–1402 Vol.2, 2003.

[97] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[98] Zhou Wang and Alan C. Bovik. Mean squared error: Love it or leave it? a new look at signal fidelity measures. *IEEE Signal Processing Magazine*, 26(1):98–117, 2009.

[99] Tianfan Xue, Baian Chen, Jiajun Wu, Donglai Wei, and William T Freeman. Video enhancement with task-oriented flow. *International Journal of Computer Vision (IJCV)*, 127(8):1106–1125, 2019.

[100] Fei Yang, Luis Herranz, Joost van de Weijer, Jose A. Iglesias Guitian, Antonio M. Lopez, and Mikhail G. Mozerov. Variable rate deep image compression with modulated autoencoder. *IEEE Signal Processing Letters*, 27:331–335, 2020.

[101] Fuzhi Yang, Huan Yang, Jianlong Fu, Hongtao Lu, and Baining Guo. Learning texture transformer network for image super-resolution, 2020.

[102] Yibo Yang, Stephan Mandt, and Lucas Theis. An introduction to neural data compression, 2023.

[103] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric, 2018.

[104] Haitian Zheng, Mengqi Ji, Haoqian Wang, Yebin Liu, and Lu Fang. Crossnet: An end-to-end reference-based super resolution network using cross-scale warping, 2018.

[105] Xiaosu Zhu, Jingkuan Song, Lianli Gao, Feng Zheng, and Heng Tao Shen. Unified multivariate gaussian mixture for efficient neural image compression, 2022.

[106] Yinhao Zhu, Yang Yang, and Taco Cohen. Transformer-based transform coding. In *International Conference on Learning Representations*, 2022.

[107] J. Ziv. On universal quantization. *IEEE Transactions on Information Theory*, 31(3):344–347, 1985.

[108] Renjie Zou, Chunfeng Song, and Zhaoxiang Zhang. The devil is in the details: Window-based attention for image compression, 2022.