END-TO-END LEARNED IMAGE COMPRESSION WITH NORMALIZING FLOWS FOR LATENT SPACE ENHANCEMENT

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

FATİH YAVUZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2022
Approval of the thesis:

END-TO-END LEARNED IMAGE COMPRESSION WITH NORMALIZING FLOWS FOR LATENT SPACE ENHANCEMENT

submitted by FATİH YAVUZ in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University by,

Prof. Dr. Halil Kalıpçılara
Dean, Graduate School of Natural and Applied Sciences

Prof. Dr. İlkyUlusoy
Head of Department, Electrical and Electronics Engineering

Assoc. Prof. Dr. Fatih Kamışlı
Supervisor, Electrical and Electronics Engineering, METU

Examing Committee Members:

Prof. Dr. Gözde Akar
Electrical and Electronics Engineering, METU

Assoc. Prof. Dr. Fatih Kamışlı
Electrical and Electronics Engineering, METU

Prof. Dr. Şenan Ece Schmidt
Electrical and Electronics Engineering, METU

Assoc. Prof. Dr. Elif Vural
Electrical and Electronics Engineering, METU

Assist. Prof. Dr. Osman Serdar Gedik
Computer Engineering, Ankara Yıldırım Beyazıt University

Date:
I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Fatih Yavuz

Signature: 

iv
ABSTRACT

END-TO-END LEARNED IMAGE COMPRESSION WITH NORMALIZING FLOWS FOR LATENT SPACE ENHANCEMENT

Yavuz, Fatih
M.S., Department of Electrical and Electronics Engineering
Supervisor: Assoc. Prof. Dr. Fatih Kamışlı

September 2022, 99 pages

Learning based methods for image compression recently received considerable attention and demonstrated promising performance, surpassing many commonly used codecs. Architectures of learning based methodologies are typically comprised of a nonlinear analysis transform, which maps the input image to a latent representation, a synthesis transform that maps the quantized latent representation back to the image domain and a model for the probability distribution of the latent representation. Successful modelling of the probability distribution of the latent representation is critically important for their performance.

Inspired by the success of normalizing flows as generative models, this work proposes a framework that utilizes flow based neural networks to improve the modelling of the probability distribution of the latent representation and consequently, the performance of a commonly known learned image compression network that is used as a benchmark. Normalizing flows implement an invertible mapping from one distribution to another, allowing the latent representation to be mapped to another domain in which its probability distribution can better match an intended probability distribution. The proposed networks are trained in an end-to-end fashion and can outperform
the benchmark in rate-distortion performance.

Keywords: learned image compression, invertible neural networks, normalizing flows, transform coding
ÖZ

NORMALİZYE EDEN AKIŞLAR İLE GELİŞTİRİLEN SAKLI UZAY KULLANILARAK UÇTAN UCA ÖĞRENİLMİŞ GÖRÜNTÜ SIKIŞTIRMA

Yavuz, Fatih
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Fatih Kamışlı

Eylül 2022, 99 sayfa

Öğrenmeye dayalı görüntü sıkıştırma yöntemleri yakın geçmişte kayda değer miktarında dikkat çekmiş ve yaygın kullanılan pek çok görüntü sıkıştırma metodolojisinin așarak텀�ir. Öğrenmeye dayalı yöntemler; genel olarak girdi görselinin saklı bir şekilde temsilini sağlayan doğrusal olmayan bir analiz dönüşümünden, görselin nicelenmiş saklı temsilini tekrar görsel kümesine taşıyan bir sentez dönüşümünden ve saklı temsilin olasılık dağılımının modellenmesinden oluşmaktadır. Saklı temsilin olasılık dağılımının başarılı bir şekilde modellenmesi, öğrenmeye dayalı tekniklerin başarımında kritik önemine sahiptir.

Normalize eden akışların üretken modellemekede başarısından ilham alınan bir çalışma, normalize eden akışları kullanarak yaygın olarak bilinen, bu çalışma kapsamında referans alınan, öğrenmeye dayalı bir görüntü sıkıştırma sinir ağının saklı temsilinin olasılık dağılıının modellemesini ve bunun sonucu olarak da sıkıştırma başarımının geliştirilmesini amaçlayan bir çerçeve önermektedir. Normalize eden akışlar, bir dağılımdan başka bir dağılıma tersine çevrilebilir bir haritalama uygulamaktadır. Bu şekilde saklı temsilin olasılık dağılıının hedeflenen parametrir bir olasılık dağılımıyla
daha iyi uyuşması sağlanabilmektedir. Önerilen sinir ağıları uçtan uca eğitilmektedir ve referans sinir ağını oran-bozulma başarımında aşabilmektedir.

Anahtar Kelimeler: öğrenilmiş görüntü sıkıştırma, tersine çevrilebilir sinir ağıları, normalize eden akışlar, dönüşüm kodlama
To Emine, Metin, and Safa Yavuz – My Family
ACKNOWLEDGMENTS

Biggest thanks go to Fatih Kamışlı, who treated me with respect and understanding and without whose support this thesis would not have been possible. I would also like to thank TÜBİTAK SAGE and specifically Kemal Özcan and Talha Korkmaz from Division of Artificial Intelligence Applications, as well as Image Processing Division, who have been supportive of my endeavors.

This has been a long journey, details of which are too long and personal to cover here. All I wish to say is that I am once again, thankful to my family. None of this would become a reality without their love and support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>ÖZ</td>
<td>vii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>x</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xiv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xvi</td>
</tr>
<tr>
<td>LIST OF ABBREVIATIONS</td>
<td>xxi</td>
</tr>
<tr>
<td>CHAPTERS</td>
<td></td>
</tr>
<tr>
<td>1  INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Image Compression</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Anatomy of Common Lossy Image Compression Methodologies</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Contemporary Methods</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Deep Learning in Image Processing</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Learned Image Compression</td>
<td>4</td>
</tr>
<tr>
<td>1.6 Normalizing Flows</td>
<td>7</td>
</tr>
<tr>
<td>1.7 Motivation and Contribution</td>
<td>7</td>
</tr>
<tr>
<td>1.8 The Outline of the Thesis</td>
<td>8</td>
</tr>
<tr>
<td>2  PRELIMINARIES</td>
<td>9</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>2.1</td>
<td>Data Compression</td>
</tr>
<tr>
<td>2.2</td>
<td>Information and Entropy</td>
</tr>
<tr>
<td>2.3</td>
<td>Entropy Coding</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Huffman Coding</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Arithmetic Coding</td>
</tr>
<tr>
<td>2.4</td>
<td>JPEG</td>
</tr>
<tr>
<td>2.4.1</td>
<td>YCbCr Color Space</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Discrete Cosine Transform</td>
</tr>
<tr>
<td>2.5</td>
<td>BPG Image Format</td>
</tr>
<tr>
<td>2.6</td>
<td>Convolution Operation</td>
</tr>
<tr>
<td>2.7</td>
<td>Artificial Neural Networks</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Nonlinear Activation Functions</td>
</tr>
<tr>
<td>2.8</td>
<td>Training Neural Networks and Backpropagation</td>
</tr>
<tr>
<td>2.9</td>
<td>Stochastic Gradient Descent</td>
</tr>
<tr>
<td>2.10</td>
<td>Adam Optimizer</td>
</tr>
<tr>
<td>3</td>
<td>RELATED WORK</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview</td>
</tr>
<tr>
<td>3.2</td>
<td>Nonlinear Transform Coding Framework</td>
</tr>
<tr>
<td>3.3</td>
<td>End-to-end Trained Learned Image Compression</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Generalized Divisive Normalization</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Convolutional Layers</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Nonlinear Analysis and Synthesis Transforms</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Quantization</td>
</tr>
</tbody>
</table>
LIST OF TABLES

TABLES

Table 2.1 Huffman coding example: Probability distribution of the dictionary. 12

Table 4.1 Tabular description of \( \Theta(.) \) in EFNet. LeakyReLU is implemented as in PyTorch [1]. Note that the assumption here is that \( x_A \) & \( x_B \) have 0.5 \( C \) channels each. 53

Table 4.2 Tabular description of \( \theta(.) \). PReLU stands for the parametric ReLU function as implemented in PyTorch [1]. 56

Table 4.3 Tabular description of SNet. Notice that \( h_a \) and \( h_s \) are different from those in Scale Hyperprior Model [2]. 59

Table 4.4 Tabular description of Scale Hyperprior Model. Note that Scale Hyperprior Model includes all four transforms and Factorized Prior Model includes only \( g_a \) and \( g_s \) [2]. 60

Table 5.1 Lagrangian parameters, each governing a different image quality level. 66

Table 5.2 Number of parameters and runtime performance statistics of Scale Hyperprior Model, EFNet, and HCNet on Kodak dataset and number of parameters of SNet. 69

Table 5.3 Bitrate savings yielded by EFNet and HCNet over Scale Hyperprior Model [2] for a given PSNR range on Kodak test dataset. 70

Table 5.4 Bitrate savings yielded by EFNet and HCNet over Scale Hyperprior Model [2] for a given PSNR range on CLIC2020 Mobile test dataset. 70
Table 5.5  Bitrate savings yielded by EFNet and HCNet over Scale Hyperprior Model [2] for a given PSNR range on CLIC2020 Professional test dataset.
LIST OF FIGURES

FIGURES

Figure 1.1 Basic transform coding scheme ........................................... 2

Figure 2.1 Huffman tree example, redrawn accordingly from [3] ............... 12

Figure 2.2 Arithmetic coding example .................................................. 13

Figure 2.3 An illustration of YCbCr color space, on a picture of two cock-
atiels, Cesur and Sultan. The leftmost picture is the original RGB im-
age, the middle one is the Y or luminance channel and the rightmost
two are Cb and Cr chroma channels, from top to bottom .......................... 15

Figure 2.4 An illustration of HEVC intra prediction modes, taken from [4] . 17

Figure 2.5 A neural network with a single hidden layer. $x_i$ denotes the input
vector while $y_i$ denotes the output. $h_i$ represents the output of a learned
affine transform, of which, $b_i$ denotes bias terms, and $t_i$ represents ele-
mentwise nonlinear functions. $t_i(h_{1,i})$ denotes the hidden layer ........... 20

Figure 3.1 General nonlinear transform coding framework as proposed by
and redrawn from Ballé et al. [5][6] ...................................................... 26

Figure 3.2 Scale Hyperprior Model from a high level, redrawn from Ballé
et al. [2] ........................................................................................................ 34

Figure 3.3 Factorized Prior Model from a high level, as described by Ballé
et al. [2] ........................................................................................................ 35

Figure 4.1 Proposed framework, based on Scale Hyperprior Model [2] ....... 45
Figure 4.2 Operation of EFNet and HCNet

Figure 4.3 Before coupling, input is sliced into two along the channel dimension.

Figure 4.4 Channelwise Coupling Layer

Figure 4.5 Operation of SNet

Figure 4.6 Spatial Coupling Layer. Very similar to the channelwise coupling, with the only difference residing in the way the input partitions are generated.

Figure 4.7 Spatial Partitioning, separating input into two tensors along either of the two spatial dimensions. Here, separation is demonstrated for height dimension H, which works by taking even indices for one partition and odd indices for the other. For width dimension, the operation is very similar. The resultant partitions are concatenated along the channel dimensions.

Figure 4.8 Additive coupling layer design, used as building blocks of EFNet

Figure 4.9 Inverse of the additive coupling flow in EFNet

Figure 4.10 Building block of HCNet. Here, coupling over 4 groups is illustrated.

Figure 4.11 Overall structure of HCNet. Note that after final coupling no 1 × 1 convolution is applied. The coupling structure here is referred to as 4-2-1.

Figure 4.12 SNet in detail

Figure 4.13 Layer by layer depiction of Scale Hyperprior Model, redrawn from the work by Ballé et al. [2]. N=128 and M=192 was utilized for this work, which results in y having 192 channels.
Figure 4.14 Layer by layer depiction of Scale Hyperprior Model [2] with the addition of a flow based network. For SNet, the filter counts of $h_a$ and $h_s$ are altered.

Figure 5.1 Performance comparison between EFNet, HCNet, Scale Hyperprior Model [2], BPG, JPEG, and Factorized Prior Model [2] on Kodak images. EFNet, HCNet, and Scale Hyperprior models were trained, the remaining results were taken from CompressAI [7].

Figure 5.2 Performance comparison between EFNet, HCNet, Scale Hyperprior Model [2], BPG, JPEG, and Factorized Prior Model [2] on CLIC2020 Mobile test dataset. EFNet, HCNet, and Scale Hyperprior models were trained, the remaining results were taken from CompressAI [7].

Figure 5.3 Performance comparison between EFNet, HCNet, Scale Hyperprior Model [2], BPG, JPEG, and Factorized Prior Model [2] on CLIC2020 Professional test dataset. EFNet, HCNet, and Scale Hyperprior models were trained, the remaining results were taken from CompressAI [7].

Figure 5.4 Training instability of the first prototype of HCNet, $\lambda = 0.0035$. Notice the spikes of loss.

Figure 5.5 Improved stability of HCNet, $\lambda = 0.0035$. $s = 0.75 \sigma(\hat{s}) + 0.25$

Figure 5.6 Instability of HCNet during training for $\lambda = 0.0130$. $s = 0.75 \sigma(\hat{s}) + 0.25$

Figure 5.7 Training instability of SNet, $\lambda = 0.0035$. Notice the spikes of loss.

Figure 5.8 Training divergence of SNet, $\lambda = 0.0035$.

Figure 5.9 Training instability of the prototype of SNet with GDN, $\lambda = 0.0035$. Notice the divergence of loss.
| Figure | Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0$.0018. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots. | 77 |
| Figure | Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0$.0035. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots. | 78 |
| Figure | Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0$.0130. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots. | 79 |
| Figure | Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0$.0018. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots. | 80 |
| Figure | Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0$.0035. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots. | 81 |
| Figure | Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0$.0130. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots. | 82 |
Figure 5.16 Visual results for an image from Kodak test images . . . . . . 83

Figure 5.17 Visual results for an image from Kodak test images . . . . . . 84

Figure 5.18 Visual results for an image from Kodak test images . . . . . . 85
LIST OF ABBREVIATIONS

2D 2 Dimensional
3D 3 Dimensional
ANN Artificial Neural Network
BPG Better Portable Graphics
BPP Bits Per Pixel
CABAC Context Adaptive Binary Arithmetic Coding
CDF Cumulative Density Function
CNN Convolutional Neural Network
DCT Discrete Cosine Transform
DFT Discrete Fourier Transform
DST Discrete Sine Transform
EFNet Efficient Network
EBCOT Embedded Block Coding with Optimized Truncation
GAN Generative Adversarial Network
GDN Generalized Divisive Normalization
HEVC High Efficiency Video Coding
HCNet Hierarchical Channelwise Coupling Network
IDCT Inverse Discrete Cosine Transform
IGDN Inverse Generalized Divisive Normalization
JPEG Joint Photographic Experts Group
LIC Learned Image Compression
MSE Mean Squared Error
MS-SSIM Multiscale Structural Similarity
NF Normalizing Flow
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDF</td>
<td>Probability Density Function</td>
</tr>
<tr>
<td>PSNR</td>
<td>Peak Signal-to-Noise Ratio</td>
</tr>
<tr>
<td>SGD</td>
<td>Stochastic Gradient Descent</td>
</tr>
<tr>
<td>SNet</td>
<td>Spatial Coupling Network</td>
</tr>
<tr>
<td>VAE</td>
<td>Variational Autoencoder</td>
</tr>
<tr>
<td>VVC</td>
<td>Versatile Video Coding</td>
</tr>
</tbody>
</table>
In this day and age, it is commonplace to take photographs digitally, use online streaming services to watch TV series or enjoy the most recent works of music, and communicate with our friends and family on the Internet. This is to an extent where our reliance on storing and transmitting data may be considered indispensable. As such, the amount of data that is stored and transmitted has showed a strong trend of rising over the years. This reality is enabled by and calls for effective and robust methods of data compression, as uncompressed data would be impractical to store and transmit. Data compression can be defined as the science of representing data in compact form. This is achieved by exploiting structures in data [8].

1.1 Image Compression

Image compression is the field of study that is concerned with finding compact representations for images. Images are unique in that there are typically strong constraints on pixel values, notably in images intended for human consumption. Lossy compression of images, which allows for some degradation and information loss while allowing for better compression is a more popular approach compared to lossless case [9]. This is because human vision is not particularly sensitive to mild distortions in images, which typically do not cause any loss of intelligibility or visual appeal while also offering large savings in required storage. This work follows the lossy compression approach.

It should also be noted that distortion is not permissible in all applications. For instance, scientific images and medical images [9] for instance do not allow for data
loss during compression. In these cases images are compressed without data loss, which is referred to as lossless compression. This type of compression requires more storage.

1.2 Anatomy of Common Lossy Image Compression Methodologies

As mentioned, humans are not particularly sensitive to mild distortions in images and are sensitive mostly to a small band of spatial frequencies \[10\]. Accordingly, natural images, such as common photographs, display a high degree of spatial correlation and most of their spectrum concentrates on a small band of spatial frequencies \[11\]. Following this phenomenon, a typical method of image compression is transform coding, which relies on using a transform to map the original image that offers a different representation of the image \[12\]. Main organization of such a compression methodology is given in Figure 1.1. The focus of this work is on this type of architecture.

The transform has commonly been chosen as a linear transform with efficient hardware implementation, which can be applied and inverted efficiently. The most common example is the Discrete Cosine Transform (DCT) \[12, 13\], which, for natural images, transforms the original image into a domain where most of the spectral components are concentrated in a few coefficients with the rest being much smaller. This effect is called energy compaction and is closely related to correlation reduction \[12\]. DCT is considered a very efficient energy compactor for natural images \[14\]. Other commonly used linear transforms with desirable properties include Discrete Sine Transform (DST) \[14\], Discrete Walsh-Hadamard Transform \[14\], Haar Transform \[12\] and Wavelet Transform \[15\]. Transforming the original image with an effective energy compacting transform allows the lossy compression method to prioritize the
most impactful coefficients in terms of human vision while discarding others. This prioritization ties directly to quantization. It is expedient to allocate a higher number of bits for the representation of coefficients that are more impactful \[16\]. After quantization, the resultant coefficients are subjected to some type of entropy coding, such as Huffman coding \[17\], and compressed losslessly before storage or transmission occurs. The process is inverted afterwards, where the entropy coded bit stream is first decoded, then dequantization is applied and finally the transform is inverted.

It follows that allowing for more bits and for more coefficients in the transformed domain would reduce distortion while demanding more storage. The number of bits used for storage is referred to as bitrate \[16\]. Hence, there is a tradeoff between the bitrate and distortion. When bitrate is fixed, lower distortion typically requires more clever and sophisticated implementations. This indicates that proper design has a critical impact on the performance of a given compression methodology.

1.3 Contemporary Methods

JPEG \[18\], JPEG2000 \[19\], and HEVC intra-frame compression modules \[20\] are among the most commonly used lossy image compression techniques today. In summary, JPEG relies on DCT for transforming, a fine-tuned quantization scheme that seeks to lessen the degradation from the perspective of human vision \[17\] and an entropy coding technique called Huffman coding. JPEG2000 uses wavelet transforms and an entropy coding scheme called Embedded Block Coding with Optimized Truncation (EBCOT) \[21\]. HEVC intra prediction relies on angular prediction, which uses previously processed blocks of the image followed by applying a form of DCT or DST to the residual between prediction and actual values. For entropy coding, a methodology named Context Adaptive Binary Arithmetic Coding (CABAC)\[22\]\[20\] is utilized.

As may be inferred from the above examples, designing of ever more effective image compression algorithms is a challenging task that requires the combined effort of many professionals to come to fruition. There are multiple considerations in such works that require significant domain expertise.
1.4 Deep Learning in Image Processing

With the advent of AlexNet and its significant success in image classification in 2012 [23], deep learning started to become increasingly widespread in image processing and computer vision applications. Deep learning refers to the use of Artificial Neural Networks (ANN), a machine learning technique, with multiple layers [24]. In a broad sense, ANNs are nonlinear and complicated functions with substantial amounts of adjustable or trainable parameters. Given a dataset, these parameters can be tuned with respect to a cost function. This tuning or optimization process is referred to as training. Stochastic gradient descent (SGD) or a related algorithm is used in this training process [25].

Unfortunately, training a neural network is a nonconvex optimization problem that requires many iterations, which can take hours, days or even longer. It is a compute-intensive problem. The upside is that deep learning solutions rely primarily on data and less on domain expertise.

As such, deep learning created an environment conducive to many developments and new state of the art results in many domains of image processing and computer vision, such as image classification [23, 26, 27, 28, 29, 30], semantic segmentation [31, 32], object detection [33, 34, 35], image super-resolution [36, 37], image inpainting [38, 39], and image generation [40, 41, 42, 43].

1.5 Learned Image Compression

Learned Image Compression (LIC) emerged as a result of applying deep learning to image compression. Work by Ballé et al. [5], a defining early example, defined the problem in tandem with variational inference [44] and suggested use of an end-to-end trained neural network that is conceptually similar to a variational autoencoder for image compression, achieving better results than JPEG [18] and JPEG2000 [19], which are non-data-driven codecs. The problem is stated with the following loss function:
\[ J = R + \lambda D \]  
\[ (1.1) \]

where \( R \) is the bitrate and \( D \) is a metric for distortion.

Where \( R \) denotes the rate term, the number of bits used to encode per pixel and \( D \) denotes distortion, such as in terms of mean squared error (MSE). \( \lambda \) is a Lagrangian parameter that is used to adjust the tradeoff between bit rate and distortion.

Rate is defined in terms of mean or expected information, or in other words, entropy. Information here is defined as follows if the quantized latent space to be encoded is denoted \( \hat{y} \):

\[ I = -\log_2 p_\hat{y}(\hat{y}) \]  
\[ (1.2) \]

Again, \( R \) is the entropy, given by:

\[ R = \mathbb{E}[I] = \mathbb{E}[-\log_2 p_\hat{y}(\hat{y})] \]  
\[ (1.3) \]

Distortion is typically defined in terms of MSE:

\[ D = ||x - \hat{x}||^2 \]  
\[ (1.4) \]

where \( x \) denotes the input image and \( \hat{x} \) denotes the reconstructed image. Combining our derivations, the final loss function ends up being:

\[ J = \mathbb{E}[-\log_2 p_\hat{y}(\hat{y})] + ||x - \hat{x}||^2 \]  
\[ (1.5) \]

Ballé et al. note that [5] although the work bears conceptual similarity to variational inference, changes in the definition of the problem may affect this similarity, such as a change in the employed distortion metric.

In its particular domain, LIC requires a treatment of several challenging issues in efforts to create useful compression models.

The first is quantization. Typically, after the input image is first transformed with a synthesis transform \( g_a \) and before coding, a quantization step must take place, such as rounding. Challenge is arisen by the fact that quantization is a nondifferentiable operation and poses a hindrance to end-to-end training of such architectures, as backpropagation would become problematic.
The quantization step is followed by entropy coding. This step requires probabilistic modelling to calculate likelihood values of the transformed components. Successful probabilistic modelling is conductive to powerful rate-distortion performance as will be discussed.

The third challenge, as may be inferred from (1.1), is posed by the tradeoff between rate and distortion. Depending on the application, different levels of quality and all else being equal, this will lead to a higher bitrate. A common way LIC deals with these different levels of tradeoff is by the employment of multiple variants of the same network, trained at different \( \lambda \) values.

Finally, the choice of nonlinear transforms themselves are not trivial. Layers and techniques used in the wider computer vision community such as batch normalization [45] may not be beneficial in LIC [5], and on similar note, a special nonlinearity called generalized divisive normalization (GDN) [46] is used instead in the field of LIC.

For quantization, Ballé et al. [5, 2] propose additive uniform noise to simulate the effects of quantization while maintaining differentiability. This technique is used widely [5, 2, 47, 48] and is also employed in this work. Alternative solutions, however, exist as in the work by Theis et al. [49], who replaced the derivative of quantization operation with the identity function during training.

Probabilistic modelling has been a primary focus [50]. [2] proposed two different models, each with different probabilistic modelling, namely the Factorized Prior Model and the Scale Hyperprior Model, the implementations of which are available in the CompressAI library [7]. Factorized Prior Model makes use of logits and the sigmoid function to map quantized values to likelihoods while Scale Hyperprior Model instead uses a Gaussian CDF scaled with some estimated scale for probabilistic modelling. In the work by Minnen et al. [47] the latter model was extended to include an estimation of means of the Gaussian random variables and further included an autoregressive component, both of which provided significant performance improvements.

Different requirements for different rate-distortion performances has been addressed by training a network multiple times for different quality levels by a large body of studies [5, 2, 47, 48, 50]. However, alternatives do exist, such as the one proposed by
the work of Toderici et al. \cite{51} that utilizes recurrent neural networks that need to be trained only once.

### 1.6 Normalizing Flows

Normalizing flows (NF), or simply flows, are a family of expressive generative models \cite{52}. Generative models are used in density estimation and prior construction \cite{52}, which are important considerations in image compression.

NFs are utilized in many applications \cite{52}, which include image generation, as exemplified by Flow++ \cite{53} and Glow \cite{54} studies, video generation, as exemplified by VideoFlow \cite{55}, and audio generation as in Waveglow \cite{56} and the study by Esling et al. \cite{57}. These networks are based on invertible transformations, which can be regarded as a divergence from many mainstream deep learning architectures. They are also relatively straightforward to train and allow for exact evaluation of densities by construction.

### 1.7 Motivation and Contribution

As mentioned, many works that focus exclusively on density modelling of LIC architectures provide significant compression performance gains. In the work by Ballé et al. \cite{2}, they remark a need for increasing the complexity of the synthesis and analysis transforms for higher quality requirements, which require better feature extraction. However, these transforms do provide a reasonable performance at lower quality levels. Indeed, considerable improvements have been seen without adding to the complexity of these transforms by instead enhancing the probabilistic modelling aspects of LIC architectures \cite{2, 47, 58}.

Recognizing the success of normalizing flows as generative models in density modelling, this work aims to improve probabilistic modelling of a learned image compression network proposed by Ballé et al. \cite{2} by combining the network with auxiliary networks based on coupling flows \cite{52}. While providing architectural alternatives and insights into benefits and limitations of flow based networks, significant improve-
ments are observed in rate-distortion performance.

1.8 The Outline of the Thesis

In Chapter 2, necessary background will be presented, most notably compression basics will be covered, along with deep learning foundations as neural networks are of central importance to this work.

In Chapter 3, a detailed discussion of the literature will take place, focusing on learned image compression frameworks and architectures, as well as generative models, namely generative adversarial networks (GAN), variational autoencoders (VAE) and normalizing flows (NF). Often, conceptual resemblance can be found between generative models and many image compression architectures of interest, making a discussion of them in order.

In Chapter 4, the exact details of the proposed framework and architectures are discussed. In Chapter 5, results are presented and evaluated. Finally, the thesis concludes with Chapter 6 with final remarks.
CHAPTER 2

PRELIMINARIES

2.1 Data Compression

Data compression, also referred to as source coding, is the art or science of representing information in compact form [8]. This is achieved by exploiting the structure present in a given type of data, such as frequently repeating patterns. Especially regarding many forms of data that are meant for human consumption, some information loss may be tolerable during compression as humans have limited perceptual abilities [8]. Small degradations in images or sound for instance rarely cause any loss of appeal or intelligibility. However, this is clearly not true for text. In addition, applications such as medical imaging commonly require perfect reconstruction. Consequently, depending on whether any information loss is tolerable or perfect reconstruction is required, there are two broad categories of compression: lossless compression and lossy compression, with the latter being the focus in this work. Regardless of its category, a compression methodology involves an encoder and a decoder.

Define the the $n$-dimensional input $\chi \in \mathbb{R}^n$. The encoder $\psi_e$ first transforms this input to another $k$-dimensional representation $\chi_r \in \mathbb{R}^k$. This is achieved using some transform $T : \mathbb{R}^n \rightarrow \mathbb{R}^k$. The resultant representation is then quantized with operator $Q : \mathbb{R}^k \rightarrow \mathcal{U}$ where $\mathcal{U}$ is some set. Some information loss occurs during quantization. After quantization, a bit stream is generated for transmission or storage via an entropy coder $E : \mathcal{U} \rightarrow \mathcal{S}$ where $\mathcal{S}$ is the set of bit streams. Thus, the encoder can be described mathematically as:

$$\psi_e = E \circ Q \circ T$$  \hspace{1cm} (2.1)

The decoder $\psi_d$ reverses this process as accurately as possible. $E$ is an invertible
operation, however, the quantization operation $Q$ typically involves non-invertible operations such as rounding and assumed not invertible. Depending on the design of the methodology, the transform $T$ may or may not be invertible. Define approximate inverse quantization operator $Q^{-1} : \mathcal{U} \to \mathbb{R}^k$ and (approximate) inverse transform $T^{-1} : \mathbb{R}^k \to \mathbb{R}^n$. Decoder first inverts entropy coding with $E^{-1} : \mathcal{S} \to \mathcal{U}$ and approximately inverts quantization with $Q^{-1}$ to obtain approximate transformed representation $\hat{\chi}_r$. Thereafter, $T^{-1}$ is applied to obtain the approximate reconstruction $\hat{x}$ of the input $x$. Thus, the decoder can be described mathematically as:

$$\psi_e = T^{-1} \circ Q^{-1} \circ E^{-1}$$

(2.2)

### 2.2 Information and Entropy

Ideas of quantitative measures of information, although they have existed before him, were put together and made into what is now referred to as information theory by Claude Elwood Shannon [59]. Conceptually, events with higher probability do not provide much information while events that are unlikely may provide considerable information. In layman’s terms, if something happens all the time, it is not news to people and is often ignored. However, if something happens very rarely, then it may often be considered newsworthy and attract attention.

For an event $A$ with probability $P(A)$, if information is to be stored in bits, the self-information associated with $A$ is defined as:

$$I(A) = -\log_2 P(A)$$

(2.3)

It follows that higher probability yields lower information. For a certain event, the relation (2.3) returns 0, implying there is no information. On the other hand, low probability events imply high information. As probability $P(A) \to 0$, the information $I(A)$ tends to $\infty$.

From this definition of information, one can define an average of information associated with some experiment $S$ with the sample space $A_i$ [60][59]. This is the definition of entropy:

$$H = \sum_i P(A_i) I(A_i) = \mathbb{E}[I(A)] \text{ for } A \in \mathcal{S}$$

(2.4)
We can extend this notion to random variables. For a discrete random variable $x$ with probability mass function $p_X(x)$, entropy can be defined as:

$$H = \mathbb{E}[I(x)] = \mathbb{E}[-\log_2 p_X(x)] = -\sum_x p_X(x) \log_2 p_X(x) \quad (2.5)$$

Importantly, entropy is also a measure of the average number of binary symbols needed to code the output of the source. As shown by Shannon, the best a lossless compressor can do is to encode data at an average bit rate equal to the entropy of the source [59].

### 2.3 Entropy Coding

Entropy coding refers to the use of coders that aim to use as few bits as possible to losslessly compress data. Modern data compression systems widely utilize sophisticated entropy coders [61]. Common examples include JPEG [18], JPEG2000 [19], HEVC [20] and VVC [62].
2.3.1 Huffman Coding

Huffman coding is an algorithm for optimal compression of symbols with known probabilities \cite{63}. It is a well known method to generate bit streams for a known alphabet with a known probability distribution. Simply put, Huffman coding aims to allocate fewer bits for frequently occurring symbols and longer sequences for less frequent symbols \cite{64}. As an illustration, assume that a source alphabet of 4 symbols has the distribution presented in Table 2.1\cite{3}.

Table 2.1: Huffman coding example: Probability distribution of the dictionary

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
<th>Information</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>0.4</td>
<td>1.322</td>
<td>0</td>
</tr>
<tr>
<td>$a_2$</td>
<td>0.35</td>
<td>1.515</td>
<td>10</td>
</tr>
<tr>
<td>$a_3$</td>
<td>0.2</td>
<td>2.322</td>
<td>110</td>
</tr>
<tr>
<td>$a_4$</td>
<td>0.05</td>
<td>4.322</td>
<td>111</td>
</tr>
</tbody>
</table>

where information is given by $-\log_2(a_i)$ for symbol $a_i$. Huffman codes can be generated by a tree based approach as described in Figure 2.1.

![Huffman Tree Example](image)

Figure 2.1: Huffman tree example, redrawn accordingly from \cite{3}

Notice that the entropy, the expected information, of the alphabet is approximately 1.74, however, generated codes have an expected length of 1.85 per symbol. This inefficiency can be addressed by more sophisticated approaches, such as arithmetic coding.
2.3.2 Arithmetic Coding

Huffman coding typically provides suboptimal performance because by design, symbol probabilities are approximated by powers of \( \frac{1}{2} \). Arithmetic coding can be more efficient than Huffman coding in exchange for compute [61].

Instead of assigning each symbol a code directly, arithmetic coding relies on a cumulative density function (CDF) of the alphabet and recursively generates a bit stream symbol by symbol for a given source, almost like guessing game. Arithmetic coding is almost optimal as generated bit streams are always within two bits of the information content of the entire input string [65]. For an \( m \) symbol alphabet, the first step is to divide the unit interval \([0, 1)\) into subintervals with each subinterval of a size equal to the corresponding symbol’s probability. Thus, a CDF of symbols is created. For a given source, the interval of the first symbol on the CDF is identified. This interval is then partitioned exactly as the original CDF and then the interval of the combination of the first two symbols are identified. This process is repeated for the entire input sequence. A more rigorous treatment is provided in [65][66].

Assume an alphabet with symbols \( \{A, B, C\} \) with distribution \( \{0.4, 0.4, 0.2\} \). Based on this alphabet and distribution, arithmetic coding of the sequence \( ABC \) yields the interval \([0.288, 0.320]\). This process is illustrated in Figure 2.2. Closest contained interval in binary code is \([0.28125, 0.3215)\), corresponding to bit stream 01001.

![Arithmetic Coding Example](image)

**Figure 2.2:** Arithmetic coding example.
2.4 JPEG

The Joint Photographic Experts Group (JPEG) convened initially in 1987 to design an optimal image compression standard [67]. The widely known JPEG still image compression standard [18] emerged as a product of their work. In summary, JPEG first transforms the RGB image to the YCbCr color space. The obtained image is then partitioned into $8 \times 8$ pixel blocks called data units, on which the Discrete Cosine Transform (DCT) is applied. Thereafter, the resultant DCT coefficients are quantized and Huffman coding is utilized for the entropy coding of the quantized coefficients. The quantization eliminates less essential DCT components and is the main factor making JPEG lossy [68].

2.4.1 YCbCr Color Space

For JPEG, color spaces that represent chromatic information in two spaces and the achromatic information in one space, such as YUV and CIELUV, are recommended over RGB for better compression performance [18]. YCbCr is one such widely used color space, used in video codecs [69]. Transformations between YCbCr and RGB is defined in terms of invertible affine transformations, given by [69]:

\[
\begin{bmatrix}
  Y \\
  Cb \\
  Cr
\end{bmatrix} = \begin{bmatrix}
  0.299 & 0.587 & 0.114 \\
  -0.169 & -0.331 & 0.499 \\
  0.499 & -0.418 & -0.0813
\end{bmatrix} \begin{bmatrix}
  R \\
  G \\
  B
\end{bmatrix} + \begin{bmatrix}
  0 \\
  128 \\
  128
\end{bmatrix}
\] (2.6)

\[
\begin{bmatrix}
  R \\
  G \\
  B
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 1.402 \\
  1 & -0.344 & -0.714 \\
  1 & 1.772 & 0
\end{bmatrix} \begin{bmatrix}
  Y \\
  Cb - 128 \\
  Cr - 128
\end{bmatrix}
\] (2.7)

where for YCbCr, the Y channel represents the luminance, or intensity, of the image, while the Cb and Cr channels focus instead on the chrominance, or color, information. YUV and YCbCr color spaces differ only in terms of the DC component 128, however, the transformation matrix is the same [69]. JPEG is almost always stored in YCbCr. Unlike RGB, of which, all three channels are of similar importance, YCbCr concentrates the most informative portion of the image into one channel, i.e. the "Y" channel or the luminance component. This makes it possible to allocate more bits
to Y than other two channels, which is conductive for better compression [68]. An illustration of YCbCr color space is provided in Figure 2.3.

### 2.4.2 Discrete Cosine Transform

Discrete Cosine Transform (DCT) [13] is a linear transform that is closely related to the Discrete Fourier Transform (DFT). Unlike DFT, DCT is real valued and offers better energy compaction than DFT [12]. It is widely used in transform coding, being employed in JPEG [18], HEVC [20], and VVC [62].

Compared to DFT, DCT shines when operating on correlated sources, such as for Markov sources with a high correlation coefficient $\rho$ [14]:

$$\rho = \frac{\mathbb{E}[x_n x_{n+1}]}{\mathbb{E}[x_n^2]}$$  \hspace{1cm} (2.8)

In fact, it is well-known that DCT approaches the Karhunen-Loève Transform as $\rho$ approaches 1 in Equation (2.8), making DCT the optimal energy-compacting transform for the limiting case. As images can be modelled as Markovian, this helps explain the popularity and success of DCT in image compression [70]. In addition, DCT has fast algorithms [70], making it an efficient transform to compute. Mathematically, DCT coefficients $C_x(k_1, k_2)$ of a 2D $N_1 \times N_2$ sequence $x(n_1, n_2)$ can be defined as shown

---

**Figure 2.3:** An illustration of YCbCr color space, on a picture of two cockatiels, Cesur and Sultan. The leftmost picture is the original RGB image, the middle one is the Y or luminance channel and the rightmost two are Cb and Cr chroma channels, from top to bottom.
below in Equation (2.9) [71]:

\[
C_x(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} 4x(n_1, n_2) \cos \frac{\pi}{2N_1} k_1 (2n_1 + 1) \cos \frac{\pi}{2N_2} k_2 (2n_2 + 1)
\]  

(2.9)

Note that the DCT coefficients in Equation (2.9) are nonzero only in the support of \(0 \leq k_1 \leq N_1 - 1\) and \(0 \leq k_2 \leq N_2 - 1\). DCT can be inverted as follows [71]:

\[
x(n_1, n_2) = \frac{1}{N_1N_2} \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} w_1(k_1)w_2(k_2) \ldots
\]  

(2.10)

\[
C_x(k_1, k_2) \cos \frac{\pi}{2N_1} k_1 (2n_1 + 1) \cos \frac{\pi}{2N_2} k_2 (2n_2 + 1)
\]

where

\[
w_1(k_1) = \begin{cases} 
\frac{1}{2} & k_1 = 0 \\
1 & 1 \leq k_1 \leq N_1 - 1
\end{cases}
\]  

(2.11)

\[
w_2(k_2) = \begin{cases} 
\frac{1}{2} & k_2 = 0 \\
1 & 1 \leq k_2 \leq N_2 - 1
\end{cases}
\]  

(2.12)

2.5 BPG Image Format

BPG (Better Portable Graphics) was introduced by Fabrice Bellard [72] as a proposed replacement for JPEG. It utilizes a subset of HEVC [20], namely the intra-frame compression part. Therefore, a description of BPG requires a discussion of the intra prediction in HEVC.

Intra-frame coding in video coding standards is handled by block-based prediction, also referred to as intra prediction. This method relies on copying known adjacent pixel values along predefined directions, called modes, onto the block of interest. HEVC supports 33 such modes and block sizes \(4 \times 4, 8 \times 8, 16 \times 16\), and \(32 \times 32\). The prediction is then subtracted from the actual values of the block and a transform coding process that often relies on DCT is applied [73]. For entropy coding, CABAC [22, 20] algorithm is employed. An illustration of these prediction directions is provided in Figure 2.4.
Figure 2.4: An illustration of HEVC intra prediction modes, taken from [4]
2.6 Convolution Operation

Convolution operation for two discrete-time one-dimensional sequences is defined as follows:

\[ x[n] * y[n] = \sum_{k=-\infty}^{\infty} x[k] y[n-k] \] (2.13)

Convolution is a linear and commutative operation, modelling the behavior of a linear time invariant system [75]. As such, the following relationships hold true:

\[ x[n] * (a_1y_1[n] + a_2y_2[n]) = a_1(x[n] * y_1[n]) + a_2(x[n] * y_2[n]) \] (2.14)

where \( a_1, a_2 \in \mathbb{C} \)

\[ x[n] * y[n] = y[n] * x[n] = \sum_{k=-\infty}^{\infty} y[k] x[n-k] \] (2.15)

Relevant to digital images, this operation can be extended to higher dimensions while satisfying the above properties. For 2D sequences, convolution is defined as [76]:

\[ x[m,n] * * y[m,n] = \sum_{k=-\infty}^{\infty} \sum_{\ell=-\infty}^{\infty} x[k, \ell] y[m-k, n-\ell] \] (2.16)

2.7 Artificial Neural Networks

In simple terms, Artificial Neural Networks (ANNs), also referred to as neural networks, or "networks", comprise a machine learning technique that commonly utilizes nonlinear functions with many learnable (or trainable) parameters. They are referred to as ANNs because early learning algorithms were intended to be computational models of biological learning [24]. Deep learning (or ANNs) simply refers to the practice of the use of compositions of nonlinear and trainable functions tailored to a specific task. Deep learning gained commendable popularity after the work of Krizhevsky et al. [23] where their Convolutional Neural Network (CNN), a type of ANN that utilizes convolution operations, beat the previous state of the art in an image classification task.

For \( i \in \{1, \ldots, N\} \), let \( g_i(x_i; \theta_i) \) be a nonlinear function with \( \theta_i \) being a set of learnable parameters and \( x_i \) its input. Defining \( \theta = \{\theta_1, \theta_2, \ldots, \theta_N\} \), one can compose the
following:

\[
f(x; \theta) = g_N(x_N; \theta_N) \circ g_{N-1}(x_{N-1}; \theta_{N-1}) \circ \cdots \circ g_1(x_1; \theta_1)
\]  

(2.17)

where \( x = x_1 \). Note that each \( x_i \) may have a different dimensionality. In mathematical notation:

\[
g_i : \mathbb{R}^{M_i} \to \mathbb{R}^{M_{i+1}} \text{ where } i \in \{1, \ldots, N\}
\]  

(2.18)

If one sets \( M_i = M \) and \( M_N = K \), the network \( f(x; \theta) \) will have the following definition:

\[
f : \mathbb{R}^M \to \mathbb{R}^K
\]  

(2.19)

Each function \( g_i \) constitutes a layer of the neural network \( f \). Layers whose output is not the output of the neural network is called hidden layers. For \( f \), \( g_N \) is not a hidden layer as its output is directly observed.

A commonly used and relatively simple type of neural network is a fully connected neural network. These networks are comprised of affine transformations and nonlinear activation functions. Following the notation in (2.17), a fully connected network are comprised of \( g_i \)'s such that:

\[
g_i(x_i; \theta_i) = t_i(W_i x_i + b_i)
\]  

(2.20)

where \( W_i \) and \( b_i \) are learnable weights, constituting \( \theta_i \). An example network with a single hidden layer is provided in Figure 2.5.

It is worth noting that leaving nonlinearities out is usually not a reasonable idea because affine transformations are closed under composition and adding more layers do not increase the explicative power of a fully connected network that lacks nonlinear activation functions.

### 2.7.1 Nonlinear Activation Functions

Nonlinear functions, such as \( t_i \) as illustrated in Figure 2.5, are often called activation functions. In this work, several elementwise nonlinear functions are noteworthy, and they are provided below. PyTorch[1] deep learning library implementations and definitions are utilized.

\[
sigmoid(x) = \sigma(x) = \frac{1}{1 + e^{-x}}
\]  

(2.21)
Figure 2.5: A neural network with a single hidden layer. $x_i$ denotes the input vector while $y_i$ denotes the output. $h_i$ represents the output of a learned affine transform, of which, $b_i$ denotes bias terms, and $t_i$ represents elementwise nonlinear functions. $t_i(h_{1,i})$ denotes the hidden layer.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$  \hspace{1cm} \text{(2.22)}$$

$$\text{softplus}(x) = \log(1 + e^x)$$  \hspace{1cm} \text{(2.23)}$$

$$ReLU(x) = r(x) = \max(x, 0)$$  \hspace{1cm} \text{(2.24)}$$

$$\text{LeakyReLU}(x) = \max(x, 0) + \alpha \min(x, 0)$$  \hspace{1cm} \text{(2.25)}$$

where $\sigma(.)$ is also known as the logistic function, and $ReLU(.)$ is referred to as the rectified linear unit function and is mathematically the same as the unit ramp function.

### 2.8 Training Neural Networks and Backpropagation

Neural networks are commonly trained by minimizing the loss between the desired output of the network for a given input and the actual output of the network over the parameters of the network:

$$\min_\theta L(f(x_i, \theta), s_i)$$  \hspace{1cm} \text{(2.26)}$$

where for dataset $\mathcal{D}$, the pair $(x_i, s_i) \in \mathcal{D}$. In applications such as image classification, $s_i$ is often the label of the image. Notably in learning based image compression,
the choice \( s_i = x_i \) is very common. Depending on the application, the choice of \( L \) can vary. The above loss function (2.26) can also include regularization terms that often depend on \( \theta \):

\[
\min_{\theta} L(f(x_i, \theta), s_i) + \lambda \Omega(\theta)
\]  

(2.27)

where \( \lambda \) is a Lagrangian parameter that affects the impact of the regularization term \( \Omega \). Training of neural networks is generally a nonconvex optimization problem. As such, variants of gradient descent is used to iteratively minimize the loss (2.27). This requires the computation of first derivatives of the loss with respect to network parameters \( \theta \). To this end, backpropagation algorithms are utilized.

For a fully connected network with the structures described in (2.17) and (2.20), define the output of each layer \( g_i \) as \( h_i \) for a given \( x \). The forward pass is defined as [77]:

**Algorithm 1** Forward pass of a fully connected network

\[
\begin{align*}
h_0 &= x \\
\text{for } i = 1, \ldots, N \text{ do} & \quad a_i = W_i h_{i-1} + b_i \\
& \quad h_i = t_i(a_i)
\end{align*}
\]

\( \hat{y} = h_M \)

\( \ell = L(\hat{y}, s) + \lambda \Omega \)

where \( \hat{y} \) is the output of the network for input \( x \) and \( \ell \) is the loss calculated by some loss function \( L \) with respect to \( s \). Note that loss function may also include some regularization terms, here denoted by \( \lambda \Omega \). After the forward pass, the backward pass is defined as follows [77]:

21
Algorithm 2 Backward pass of a fully connected network

The first step is to calculate gradient of the loss $\ell$ with respect to the output $\hat{y}$:

$$d \leftarrow \nabla_{\hat{y}} \ell = \nabla_{\hat{y}} L(\hat{y}, s)$$

Now computations take place starting from the output layer:

for $i = N, \ldots, 1$ do

$$d \leftarrow \nabla_{a_i} \ell = d \odot t'(a_i)$$

Compute gradients with respect to all parameters in current layer:

$$\nabla_{b_i} \ell = d + \lambda \nabla_{b_i} \Omega$$

$$\nabla_{W_i} \ell = d h_{i-1}^T + \lambda \nabla_{W_i} \Omega$$

Propagate gradients for the next layer:

$$d \leftarrow \nabla_{h_{i-1}} \ell = W_i^T d$$

end for

As such, gradients of loss with respect to network parameters can be iteratively and efficiently computed thanks to the layered structure of neural networks.

2.9 Stochastic Gradient Descent

Stochastic gradient descent (SGD) and its variants are likely the most widely used optimization algorithms in machine learning [25]. If the samples $(x_i, s_i) \in D$ are independent and identically distributed, then a small subset $m$, called batch, of training samples are sufficient to estimate true gradient $d$ of $D$ by batch gradient $\hat{d}$. Mathematically:

$$\mathbb{E}[\hat{d}] = d \quad (2.28)$$

One can define SGD as follows [25]:
Algorithm 3 SGD Algorithm

Require: Learning rate $\eta$

Require: Initial parameter set $\theta$

Require: Batch size $p$

while Stopping criterion not met do

Set $m$ as a random sample from $\mathcal{D}$. Each iteration will take a non-overlapping sample subset of $p$ samples until $\mathcal{D}$ is exhausted and sampling will begin again.

$m = \{x_1, \ldots, x_p\}$

Set gradient estimate $\hat{d} = 0$

for $i = 1, \ldots, p$ do

Compute batch gradient estimate $\hat{d} \leftarrow \hat{d} + \nabla_{\theta}(L(f(x_i, \theta), s_i) + \lambda\Omega)/p$

end for

Update gradient: $\theta \leftarrow \theta - \eta\hat{d}$

end while

While SGD is widely used, learning can be slow with it if gradients are small [25]. If the gradients have been consistent for recent batches, then the optimizer might afford to be more confident in taking steps in that direction. This is achieved by adding a velocity component, also referred to as momentum. With this addition, SGD with momentum is described below.

Algorithm 4 SGD with Momentum

Require: Momentum parameter $\alpha$, Initial velocity $v$, Learning rate $\eta$

Require: Initial parameter set $\theta$, Batch size $p$

while Stopping criterion not met do

$m = \{x_1, \ldots, x_p\}$

Set gradient estimate $\hat{d} = 0$

for $i = 1, \ldots, p$ do

Compute batch gradient estimate $\hat{d} \leftarrow \hat{d} + \nabla_{\theta}(L(f(x_i, \theta), s_i) + \lambda\Omega)$

end for

Compute velocity update: $v \leftarrow \alpha - \eta\hat{d}$

Update gradient: $\theta \leftarrow \theta + v$

end while
2.10 Adam Optimizer

Adam [78] optimizer is a widely used stochastic optimization technique, described below [25]:

**Algorithm 5** Adam Optimizer

**Require:** Step size $\alpha$, Decay rates $\rho_1$ and $\rho_2$, constant $\epsilon$

**Require:** Initial parameter set $\theta$, Batch size $p$

Set first and second moment variables $s = 0$, $r = 0$

Set timestep $t = 0$

**while** Stopping criterion not met **do**

$m = \{x_1, \ldots, x_p\}$

Set gradient estimate $\hat{d} = 0$

for $i = 1, \ldots, p$ do

Compute batch gradient estimate $\hat{d} \leftarrow \hat{d} + \nabla_\theta (L(f(x_i, \theta), s_i) + \lambda \Omega)$

end for

$t \leftarrow t + 1$

Get biased first moment: $s \leftarrow \rho_1 s + (1 - \rho_1) \hat{d}$

Get biased second moment: $r \leftarrow \rho_2 r + (1 - \rho_2) \hat{d}^2$

Bias corrected first moment: $\hat{s} \leftarrow \frac{s}{1 - \rho_1}$

Get biased second moment: $\hat{r} \leftarrow \frac{r}{1 - \rho_2}$

Gradient update: $\Delta \theta = -\alpha \frac{\hat{s}}{\sqrt{\hat{r}} + \epsilon} \hat{d}$

Apply gradient update: $\theta \leftarrow \theta + \Delta \theta$

**end while**

Adam keeps decaying averages of momentum directions and magnitudes. When gradient directions are relatively consistent across updates, the network takes larger and larger steps when training with the Adam optimizer. It is worth noting that many LIC studies use Adam for training [5, 48, 58, 50].
CHAPTER 3

RELATED WORK

3.1 Overview

This chapter aims to cover the roots of the work in this thesis. These include an overview of a relevant end-to-end learned image compression framework, a detailed discussion of individual components of common end-to-end learned image compression architectures, and generative models. As will be discussed, learning based image compression is conceptually related to variational autoencoders, a generative model. Normalizing flows are also studied under the context of generative models. In this light, a discussion of generative models is in order.

Research in learning based image compression has seen considerable growth over the recent years with many studies focusing on providing development to various aspects of transform coding. These include a nonlinear analysis transform, a nonlinear synthesis transform, and various entropy models (called priors) to handle the probabilistic modelling of the latent space.

3.2 Nonlinear Transform Coding Framework

Traditional image compression methods are not data-driven and grounded in theoretical derivation. JPEG [18] and JPEG2000 [19] are prominent examples. Briefly, for instance in JPEG, as discussed in preliminaries, compression is performed by first transforming the input image by applying DCT [13] then quantizing and entropy coding the resultant DCT coefficients. However, DCT is a linear transform, and so are other transforms such as wavelet transforms and DST. On the upside these transforms
are easy to invert and often have efficient algorithms, with ODST-3 being a notable exception as it lacks efficient algorithms [79] despite being employed in a widely used video codec, HEVC or H.265 [20]. Furthermore, they have strong theoretical grounds, such as by modelling images as Markov processes as discussed by Kamışlı in detail [70]. However, their disadvantage is that their linearity might actually be too strong a limitation and may hinder performance. Although DCT and DST are optimal for certain Markovian processes [70], this is likely too strong a simplification for natural images, which may be assumed to have more complicated behavior.

An early nonlinear transform coding architecture was developed by Ballé et al. [5] to address the mentioned limitations. Conceptually, they provided a description of nonlinear transform coding procedure in three different spaces as seen in Figure 3.1, where $D$ is a metric of distortion and $R$ is the bitrate. The perceptual space transform is generated using a fixed transform $g_p$. MS-SSIM [80] is a popular metric for perceptual space, providing a measure of perceptual quality.

$x$ and $\hat{x}$ represent the original image and the reconstructed image after lossy compression, respectively. $y$ is obtained by applying a nonlinear parametric function (e.g. neural network) $g_a(x; \theta)$ to $x$. $q$ is a discrete tensor, obtained by quantizing $y$. $q$ is then entropy coded, and its entropy is the lower bound for $R$. After decoding, $\hat{y}$ is obtained, which is then passed through another network $g_s(\hat{y}, \phi)$ to generate $\hat{x}$.

For the end-to-end training of the architectures in the works by Ballé et al. [5][2], the
following cost function is utilized:

\[
\min_{\theta, \phi} J = R + \lambda D \tag{3.1}
\]

Where \( \lambda \) is a Lagrangian parameter, controlling the tradeoff between rate and distortion. As mentioned, the rate \( R \) is the entropy of the quantized representation. For a chosen distortion metric \( d(. , ) \), input images \( x \) from the dataset \( D \) and reconstructed images \( \hat{x} \), we then have \[6\]:

\[
\min_{\theta, \phi} J = H(P_q) + \lambda E[d(z, \hat{z})] \tag{3.2}
\]

where \( z \) and \( \hat{z} \) represent the perceptual representation of the input and the reconstructed image, respectively and \( H(P_q) \) represents the discrete entropy of \( q \) in Figure 3.1. Note that both terms above are stochastic and neither of them have a known probability distribution. The right term, the distortion, is estimated through minibatch averages from \( D \) and the left term, bitrate, is estimated using minibatch averaging of the self-information of the probabilities of \( q \), which is based on a neural-network based model.

As mentioned in the introduction, there are certain important considerations and challenges to training a successful learning based nonlinear transform coding architecture for image compression. These include the design of the nonlinear transforms \( g_a \) and \( g_s \) in Figure 3.1, design of differentiable quantization approximations as quantization commonly involves rounding operations that hinders end-to-end training, and probabilistic modelling of the latent variables \( y \) in Figure 3.1 for bitrate calculations.

### 3.3 End-to-end Trained Learned Image Compression

Learned image compression architectures address the mentioned considerations in different ways. This section aims to provide an executive summary based on the solutions proposed in the works by Ballé et al. \[5, 2\], which will be adopted as the baseline models in this work.

One of the primary concerns of learned image compression architectures is the design of nonlinear analysis and synthesis transforms. As will be discussed, these transforms
are commonly comprised of convolutional layers and generalized divisive normalization (GDN) layers. Contribution of GDN layers is to model nonlinear dependencies while convolutional layers do the heavy lifting.

Another important consideration is that quantization, as mentioned, is a nondifferentiable operation. To see this, one can define $\hat{x}$ in Figure 3.1, the reconstructed image, as a function of $x$, the input image, through the following relationship:

$$\hat{x} = g_s(Q^{-1}(Q(g_a(x; \theta))), \phi)$$ (3.3)

where $Q^{-1}$ is the dequantization operation, which here is simply casting discrete variables to continuous domain. However, quantization operation, $Q$, involves rounding operations that prevents backpropagation of derivatives for end-to-end training.

Finally, bitrate estimation is another important consideration. Computing the bitrate requires a probability estimate for the latent representation. While the construction of such models is definitely not trivial, successful probabilistic modelling is critically important in a learning based image compression architecture.

### 3.3.1 Generalized Divisive Normalization

Generalized divisive normalization (GDN) was proposed by Ballé et al. [46] as a parametric nonlinear transform to Gaussianize natural images. The operation is differentiable and efficiently invertible, hence well-suited for end-to-end training. After its inception, various studies [5, 2, 47, 48, 50] utilized this operation.

GDN generalizes and motivates from divisive normalization, a common form thereof is [46]:

$$y_i = \frac{x_i^\alpha}{z_i^{\beta} + \sum_j x_j^\alpha}$$ (3.4)

where $\Psi = \{\alpha, \beta, \gamma\}$ are the parameters of the divisive normalization layer and $x_i$ and $y_i$ are components of inputs and outputs of the layer, respectively. Motivated by divisive normalization, Ballé et al. [46] define GDN as follows:

$$z = Hx$$ (3.5)

$$y_i = \frac{z_i}{(\beta_i + \sum_j \gamma_{ij}z_j |^\alpha_{ij})^{\epsilon_i}}$$ (3.6)
where $H$, $\alpha$, and $\gamma$ are matrices and $\beta$, and $\epsilon$ are vectors, constituting the parameter set $\Psi$ of the GDN layer. Typically, $\alpha = 2$ and $\epsilon = 1/2$ are set as in Ballé et al. [5] and $Hx$ is implemented as a convolution. In this case, for the input-output pair $(x, y)$, the (approximate) inverse GDN (or IGDN) operation is given by:

$$y_i = x_i(m, n)(\hat{\beta}_i + \sum_j \hat{\gamma}_{ij}(x_j(m, n))^2)^{1/2}$$ (3.7)

A simplified variant of GDN was derived by Johnston et al. [81] as:

$$y_i = \frac{x_i}{\beta_i + \sum_j \gamma_{ij}|x_j|}$$ (3.8)

which can be approximately inverted for an input-output pair $(x, y)$ by:

$$y_i = x_i(\hat{\beta}_i + \sum_j \hat{\gamma}_{ij}|x_j|)$$ (3.9)

### 3.3.2 Convolutional Layers

Affine convolution operation, or an (affine) convolutional layer, is defined as follows:[5]

$$z_i(m, n) = \sum_j (h_{ij} * x_j)(m, n) + c_i$$ (3.10)

where $*$ denotes 2D convolution operation, $z$ and $x$ are the input and output tensors of this operation respectively, and $h$ and $c$ are trainable filters (or kernels) and bias terms respectively. The index $i$ represents the channel dimension. This means, to compute each spatial channel of $z$, every channel of $x$ is convolved with a separate, learnable kernel and a fixed bias term is added to the result.

Commonly, the convolutional layers are upsampling or downsampling layers, often by 2. The rate of downsampling and upsampling is referred to as stride. Downsampling operation can be described as:

$$y_i(m, n) = z_i(m, n)(sm, sn), \ s \in \mathbb{Z}^+$$ (3.11)

where $s$ denotes the stride, $y_i$ denotes the downsampled output, and $z_i$ denotes the result of the raw convolution operation in equation 3.10. The final output of the downsampling convolutional layer with input $x$ is $y$. 29
When a convolutional layer is upsampling, commonly a "bed of nails" type of approach is followed. Before convolution, between each column and row, columns and rows of zeros, respectively, equal in number to stride is added. Mathematically, the upsampling convolutional layer with input $x_i$ and output $y_i$ can be defined as [5]:

$$z_i(m, n) = \begin{cases} x_i(m/s, n/s) & \text{if } m/s \text{ and } n/s \in \mathbb{Z} \\ 0 & \text{otherwise} \end{cases}$$

(3.12)

$$y_i(m, n) = \sum_j (h_{ij} \ast z_j)(m, n) + c_i$$

(3.13)

### 3.3.3 Nonlinear Analysis and Synthesis Transforms

Of relevance to this work, Ballé et al. introduce two compression architectures in their work [2], differing in the way the entropy coding process and the latent representation is handled. Those are namely the Factorized Prior Model, which uses a learned factorized probability density, and the Scale Hyperprior Model, which uses a Gaussian probability model that relies on a side network to estimate the scale for each latent variable. It is worth noting that Factorized Prior Model is similar to the architecture in another paper by Ballé et al. [5].

For both models the analysis and synthesis transforms are the same respectively. Recall that the analysis transform is the $g_a$ function in Figure [3.1]. Define downsampling convolutional layer with stride $s$ and a trainable parameter set $\eta_i$ as $\text{conv}_s(x; \eta_i)$ and a GDN layer with a trainable parameter set $\omega_i$ as $\text{gdn}(x; \omega_i)$. Define the following composition:

$$c_s(x; \theta_i) = \text{gdn}(x; \omega_i) \circ \text{conv}_s(x; \eta_i)$$

(3.14)

where $\theta_i = \{\omega_i, \eta_i\}$. With some abuse of notation, this leads to the following definition of $g_a$:

$$g_a(x; \theta) = (\text{conv}_{s_2}(\theta_4) \circ c_2(\theta_3) \circ c_2(\theta_2) \circ c_2(\theta_1))(x)$$

(3.15)

where $\theta$ is the set of all learnable parameters in the composition on the right side. Similar to the convolutional layer, with parameter set $\eta_i$ one can define the upsampling convolution with stride $s$ as $\text{deconv}_s(x; \eta_i)$. Please note that unlike what the
name "deconv" implies, this layer does not perform deconvolution operation, however, conceptually it is treated as a sort of approximate inverse to the convolutional layer and hence erroneously called as such.

Define IGDN operation with a trainable parameter set \( \omega_i \) as \( igdn(x; \omega_i) \). Using the developed notation for IGDN and upsampling convolution, define the following composition:

\[
dc_s(x; \phi_i) = igdn(x; \omega_i) \circ deconv_s(x; \eta_i)
\]  

(3.16)

where \( \phi_i = \{\omega_i, \eta_i\} \) is the set of learnable parameters. Once again with some abuse of notation, the nonlinear synthesis transform can be defined as:

\[
g_s(x; \phi) = (deconv_2(\phi_4) \circ dc_2(\phi_3) \circ dc_2(\phi_2) \circ dc_2(\phi_1))(x)
\]  

(3.17)

The derived nonlinear synthesis and analysis transforms were utilized in multiple works \[2, 47, 58\]. However, some works utilized different transforms. The work of Cheng et al. \[48\] used residual connections and subpixel convolutions. The same work makes use of an attention mechanism, a type of operation popularized by the work of Vaswani et al. \[82\]. Recently, transformer networks, a type of ANN that makes extensive use of attention, were popularized in computer vision applications by Dosovitsky et al. \[30\], who achieved commendable results in image classification with their Vision Transformer (ViT). Transformer architectures were also adopted in learned image compression. The work by Zhu et al. \[50\] utilized a type of transformer called Swin Transformer \[83\] to improve the design of analysis and synthesis transforms and argued that Swin Transformers might outperform CNN type architectures in learned image compression.

### 3.3.4 Quantization

The relation described in (3.3) utilizes quantization operation, which involves nondifferentiable operations. This hinders end-to-end training.

One possible relaxation of the nondifferentiable quantization operation is to add uniform noise to \( y \) to simulate the quantization operation during training, as utilized by
This can be described as:

\[ y = g_a(x; \theta) \]  

\[ q = Q_t(y) = y + \tilde{n} \text{ with } p_\tilde{n}(n) = U(-\frac{1}{2}, \frac{1}{2}) \]

where \( Q_t \) represents the quantization operation used during training and \( \tilde{n} \) is a random variable with uniform distribution.

When the training is completed, an actual quantization operation is substituted. This approach has been employed widely \[47, 58, 48, 50\] and used in this work in the light of its evident success.

Alternatively, Theis et al. \[49\] propose keeping the forward pass the same, i.e. retaining the rounding operation while redefining the derivative in backpropagation as a constant 1. To this end, they define a stochastic rounding operation as follows:

\[ \{y\} \approx \lfloor y \rfloor + \epsilon, \ \epsilon \in \{0, 1\}, P(\epsilon = 1) = y - \lfloor y \rfloor \]

with \( y \) being the input of the rounding operation. In simple terms, they define the input variable \( y \) in two parts: A quantized, integer part and a stochastic decimal part. The derivative is defined as follows:

\[ \frac{d}{dy}\{y\} := \frac{d}{dy}\mathbb{E}[\{y\}] = \frac{d}{dy}y = 1 \]

### 3.3.5 Probabilistic Modelling

Importantly, during training, the baseline networks, namely the Scale Hyperprior and Factorized Prior models, do not contain an actual entropy coder and a bitrate estimate is necessary for training. Recall that bitrate \( R \) is defined as:

\[ R = H(P_q) = -\mathbb{E}[\log_2 P_q] \]

In practice, expectation operation is replaced by an averaging operation during training. This leaves a need for a probabilistic model of \( P_q \). Successful probabilistic modelling is critically important to the success of a learning based image compression network and to this end many approaches have been proposed over the years.
Factorized Prior Model defines a non-parametric univariate density function. Recall that for a valid univariate CDF $c(x)$ and its associated probability density function (PDF) $p(x)$ the following hold \([2]\):

$$c(-\infty) = 0, c(-\infty) = 1, p(x) = \frac{\partial c}{\partial x} \geq 0$$  \hfill (3.23)

Define the following compositions:

$$c = f_K \circ f_{K-1} \ldots f_1$$  \hfill (3.24)

$$p = f'_K \cdot f'_{K-1} \cdot \ldots \cdot f'_1$$  \hfill (3.25)

For $k = \{1, \ldots, K - 1\}$, define $f_k$ as:

$$f_k(x) = g_k(H^{(k)}x + b^{(k)})$$  \hfill (3.26)

where $g_k$ is defined as follows:

$$g_k(x) = x + a^{(k)} \odot \tanh(x)$$  \hfill (3.27)

The final stage guarantees $c$ will map to $[0, 1]$ with the use of the sigmoid function:

$$f_K(x) = \sigma(H^{(K)}x + b^{(K)})$$  \hfill (3.28)

Note that for density $p$ to be valid, it must also be nonnegative. This is achieved by reparameterization:

$$H^{(k)} = \text{softplus}(\hat{H}^{(k)})$$  \hfill (3.29)

$$a^{(k)} = \tanh(\hat{a}^{(k)})$$  \hfill (3.30)

Using the developed learnable densities, $P_\psi$ is then evaluated as:

$$a^{(k)} = \tanh(\hat{a}^{(k)})$$  \hfill (3.31)

Factorized Prior Model defines the probability distribution of the latent space $\tilde{y}$ as a composition of independent components $\tilde{y}_i$. Explicitly:

$$p_{\tilde{y} | \psi}(\tilde{y} | \psi) = \prod_i (p_{y_i | \psi^i}(\psi^i) * \mathcal{U}(-\frac{1}{2}, \frac{1}{2}))(\tilde{y}_i)$$  \hfill (3.32)

Where $\mathcal{U}$ represents the uniform distribution, $p_{y_i | \psi^i}$ terms are constructed based on the $p$ and $c$ formulation above. The convolution part with the uniform distribution is added to better match the marginal distribution.
On the other hand, the Scale Hyperprior Model adopts a different approach. This model improves probabilistic modelling by introducing an auxiliary variable $\tilde{z}$ through a "hyperprior" network, which has the same prior distribution as the one for the Factorized Prior model while the latent space $\tilde{y}$ is now modelled by a zero-mean Gaussian prior. This is illustrated in Figure 3.2. Mathematically, hyperprior distribution is handled by the same model as in Equation (3.32) given by:

$$p_{\tilde{z}|\psi}(\tilde{z}|\psi) = \prod_i (p_{z_i|\psi_i}(\psi_i) * U(-\frac{1}{2}, \frac{1}{2}))(\tilde{z}_i)$$  \hspace{1cm} (3.33)

where non-quantized $z$ is obtained using the hyper-analysis transform $h_a$ as:

$$z = h_a(y, \phi_h)$$  \hspace{1cm} (3.34)

The utility of $\tilde{z}$ is to provide the scale components, denoted as $\tilde{\sigma}$, to a parametric Gaussian conditional that models the latent space $\tilde{y}$. More explicitly:

$$p_{\tilde{y}|\tilde{z}}(\tilde{y}|\tilde{z}, \tilde{\sigma}) = \prod_i (N(0, \tilde{\sigma}_i) * U(-\frac{1}{2}, \frac{1}{2}))(\tilde{y}_i)$$  \hspace{1cm} (3.35)

where the scale terms $\tilde{\sigma}$ are obtained using the hyper-synthesis transform as follows:

$$\tilde{\sigma} = h_s(\tilde{z}, \theta_h)$$  \hspace{1cm} (3.36)

The scale hyperprior simply focuses on improving the probabilistic modelling of the Factorized Prior Model while leaving the analysis and synthesis transforms, as well
as the quantization practice intact. This alone yields significant compression performance improvements as will be discussed. This is indeed not a trivial observation, reflecting the clear impact that successful probabilistic modelling has on compression performance. The work by Minnen et al. [47] builds on this model by introducing two models. One of these networks are built upon the Scale Hyperprior Model by having the hyperprior estimate not only a scale but also a mean component for the Gaussian component. They further extend this model in the same paper by introducing an autoregressive model to the prior model. Both of these approaches yield performance improvements. A follow up work by Minnen and Singh [58] extends this model by adding channel-conditioning and latent residual prediction, which improves the compression further still.

### 3.3.6 End-to-End Training

This chapter focuses on the end-to-end training process of the Factorized Prior Model and Scale Hyperprior Model. A general loss function for the framework described in Figure 3.1 is given by:

$$L = R + \lambda D$$  \hspace{1cm} (3.37)

where $R$ refers to bitrate per pixel, which is given by the total entropy of the latent space, divided by the number of pixels. $D$ will be discussed shortly. For Factorized Prior Model, the probability distribution of the latent space is given by the following equation, rewritten here for convenience:

$$p_{\tilde{y}|\psi}(\tilde{y}|\psi) = \prod_i (p_{y_i|\psi}(\psi^i) \ast U(-\frac{1}{2}, \frac{1}{2}))(\tilde{y}_i)$$  \hspace{1cm} (3.38)
It is useful here to expand uniform random distribution in Equation (3.38) as follows:

$$U\left( -\frac{1}{2}, \frac{1}{2} \right) = u(x + \frac{1}{2}) - u(x - \frac{1}{2})$$  \hspace{1cm} (3.39)

where $u()$ denotes the unit step function. Recall that convolution with the unit step function is equivalent to taking an integral. It follows that [2]:

$$p_{y_i|\psi^i}(\tilde{y}_i) * U\left( -\frac{1}{2}, \frac{1}{2} \right) = \int_{-0.5}^{0.5} p_{y_i|\psi^i}(\tilde{y}_i)d\tilde{y}_i = c(\tilde{y}_i + 0.5) - c(\tilde{y}_i - 0.5)$$  \hspace{1cm} (3.40)

where $c$ is the univariate non-parametric learned cumulative density function, described in Equation (3.24). $c$ is differentiable by design, therefore, rate can be computed differentiably as well, meaning the probabilistic model can be trained through gradient descent or a related method.

In this work, the distortion metric is chosen as the mean squared error, averaged over each pixel. During training, quantization is replaced by additive uniform noise. Consequently, the transition from $x$ to $\tilde{x}$, as shown in Figure 3.2 during training is differentiable. All in all, the loss function for Factorized Prior Model can be written as:

$$J = R + \lambda D = \mathbb{E}[-\log_2 p_{\tilde{y}}(\tilde{y})] + \mathbb{E}\{|\|x - \tilde{x}\|^2\}$$  \hspace{1cm} (3.41)

Scale Hyperprior Model differs in the computation of the rate component, as the rate now needs to be computed for both $\tilde{y}$ and $\tilde{z}$. For $\tilde{z}$, the rate formula remains the same. The formula for $\tilde{y}$ can be derived similarly. In this case a Gaussian distribution $\mathcal{N}(0, \tilde{\sigma})$ is convolved with a uniform distribution, described in Equation (3.35). Simply by substituting the Gaussian for $p_{y_i|\psi^i}$ in the Equation (3.40), a straightforward result for probability computation is obtained:

$$p_{\tilde{y}_i|\tilde{z}}(\tilde{y}_i|\tilde{z}, \tilde{\sigma}_i) = \int_{-0.5}^{0.5} \mathcal{N}(\tilde{y}_i; 0, \tilde{\sigma}_i)d\tilde{y}_i = \Phi\left(\frac{\tilde{y}_i + 0.5}{\tilde{\sigma}_i}\right) - \Phi\left(\frac{\tilde{y}_i - 0.5}{\tilde{\sigma}_i}\right)$$  \hspace{1cm} (3.42)

where $\Phi$ denotes the cumulative density function of zero-mean unit-variance univariate Gaussian distribution. Combining rate formulas, the final loss function of the Scale Hyperprior Model is given by:

$$J = \mathbb{E}[-\log_2 p_{\tilde{y}|\tilde{z}}(\tilde{y})] + \mathbb{E}[-\log_2 p_{\tilde{z}}(\tilde{z})] + \mathbb{E}\{|\|x - \tilde{x}\|^2\}$$  \hspace{1cm} (3.43)
3.4 Generative Models

Modelling of probability distributions of samples taken from some distribution is a major objective of machine learning [52]. This process is also referred to as unsupervised learning or generative modelling. Some applications of generative models include prior construction and probability density estimation, which are of critical importance for learned image compression.

There are many types of generative models. These include generative adversarial networks (GAN) [40], variational autoencoders (VAE) [44, 84], and normalizing flows (NF) [52, 85].

3.4.1 Generative Adversarial Networks

After its inception by Goodfellow et al. [40], GANs have seen extensive use in fields including computer vision, natural language processing, and time-series synthesis [86].

Commonly, a GAN is constituted by two networks, namely a discriminator $D$ and a generator $G$. The goal of the discriminator is to distinguish real data from fake data. In contrast, the aim of the generator $G$ is to generate convincing synthetic data to deceive the discriminator [86]. Therefore, there is an adversarial relationship between the two networks. Mathematically, the optimization of such a construct can be described as [86]:

$$\min_G \max_D \mathbb{E}_x \log[D(x)] + \mathbb{E}_z \log[1 - D(G(z))]$$  \hspace{1cm} (3.44)

where $x$ is data that is sampled from a real dataset $\mathcal{D}$ and $z$ is drawn from a distribution $P_z$, usually some kind of noise. The generator $G$, using the noise sample $z$, generates a sample $G(z)$ that attempts to mimic the distribution of $\mathcal{D}$. The goal is to make this generated sample convincing for the discriminator. The discriminator $D$ is trained to "believe" samples from $\mathcal{D}$ and try not to be fooled by the generator. A successfully trained GAN can distinguish real and fake images well and generate convincing images.
GANs can yield commendable results. An example of a successful GAN is the Style-GAN by Karras et al. [87], whose work can generate high quality natural images. However, evaluating the observed distribution $P_D$ is usually not possible.

### 3.4.2 Variational Autoencoders

Variational autoencoders were proposed by Kingma et al. [44] as probabilistic latent variable models. Assume $x \in \mathcal{D}$ has an unknown distribution $p_x(x)$. Define a latent space with random variable $z$ with the distribution $p_z(\theta)$ a desired distribution e.g. Gaussian, where $\theta$ constitutes the set of parameters of the generative network. Define the decoder $p_\theta(x|z)$. This leads to the following straightforward relationship, which is called a deep latent variable model [84]:

$$
p_\theta(x) = \int p_\theta(x, z) dz \quad \text{where} \quad p_\theta(x, z) = p_\theta(z)p_\theta(z|x) \quad (3.45)
$$

A common goal is to maximize the likelihood of $p_\theta(x)$, however, in this context so far, this is problematic. $p_\theta(x)$ is assumed not to have an analytic solution or an efficient estimator. This is directly related to the posterior $p_\theta(z|x)$ being intractable.

VAEs address this problem by introducing the inference model $q_\phi(z|x)$, which is a separately parameterized encoder. Using sufficiently flexible neural networks, the hope is to approximate the posterior as accurately as possible. Returning to the likelihood maximization, the problem can be stated as [84]:

$$
\log p_\theta(x) = \mathbb{E}_{\log q_\phi(z|x)}[\log p_\theta(x)] \quad (3.46)
$$

Note that $\log$ refers to the natural logarithm. The above equality holds because the marginal $p_\theta(x)$ does not depend on $z$, therefore, the expectation has no impact. Expanding the relationship with Bayes’ rule yields the following [84]:

$$
\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x)] = \mathbb{E}_{q_\phi(z|x)} \log \left[ \frac{p_\theta(x, z)}{p_\theta(z|x)} \right] \quad (3.47)
$$

$$
\mathbb{E}_{q_\phi(z|x)} \log \left[ \frac{p_\theta(x, z)}{p_\theta(z|x)} \right] = \mathbb{E}_{q_\phi(z|x)} \log \left[ \frac{p_\theta(x, z)}{q_\phi(z|x)} \frac{q_\phi(z|x)}{p_\theta(z|x)} \right] \quad (3.48)
$$

which finally yields:

$$
\log p_\theta(x) = \mathbb{E}_{q_\phi(z|x)} \log \frac{p_\theta(x, z)}{q_\phi(z|x)} + \mathbb{E}_{q_\phi(z|x)} \log \frac{q_\phi(z|x)}{p_\theta(z|x)} \quad (3.49)
$$
The left term on the right hand side of the above equation is an important metric for VAEs. Define \[ L_{\theta,\phi}(x) = \mathbb{E}_{q_\phi(z|x)} \log \frac{p_\theta(x,z)}{q_\phi(z|x)} \] (3.50)

\[ D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \mathbb{E}_{q_\phi(z|x)} \log \frac{q_\phi(z|x)}{p_\theta(z|x)} \] (3.51)

With the relationships (3.50) and (3.51), the equation (3.49) can be rewritten as \[ \log p_\theta(x) = L_{\theta,\phi}(x) + D_{KL}(q_\phi(z|x)||p_\theta(z|x)) \] (3.52)

where \( L_{\theta,\phi}(x) \) is the mentioned metric called Evidence Lower Bound (ELBO). To see why it is useful, notice that the quantity \( D_{KL}(q_\phi(z|x)||p_\theta(z|x)) \), which is the KL divergence between the posterior distribution \( q_\phi(z|x) \) and the (intractable) posterior distribution \( p_\theta(z|x) \), is always nonnegative. Therefore \[ L_{\theta,\phi}(x) \leq \log p_\theta(x) \] (3.53)

This implies that ELBO can be maximized to indirectly maximize the likelihood \( p_\theta(x) \). For a fixed \( x \), maximizing the likelihood also ensures the approximate posterior will approach the true posterior by minimizing the KL Divergence in the equation (3.51).

The networks of interest in this work bear conceptual similarity to VAEs [2]. In Bayesian Variational Inference, the goal is to estimate the true posterior \( p_{y|x}(y|x) \), which is assumed intractable [84], by an approximate posterior \( q(y|x) \). The KL divergence between \( q \) and \( p \) looks like [5, 2]:

\[
\mathbb{E}_{x \sim p_x} [D_{KL}(q||p_{y|x})] = \mathbb{E}_{x \sim p_x} [\mathbb{E}_{y \sim q} [\log q(y|x)] - \mathbb{E}_{y \sim q} [\log p_{y|x}(y|x)]] \\
= \mathbb{E}_{x \sim p_x} [\mathbb{E}_{y \sim q} [\log q(y|x)] - \mathbb{E}_{y \sim q} [\log p_x(y|x)y] - \mathbb{E}_{y \sim q} [\log p_y(y)] + C
\] (3.54)

In the second line, the first term will evaluate to zero across all \( x \) as it has a uniform distribution due to quantization. The second term corresponds to distortion and the third term corresponds to the bitrate.

### 3.4.3 Normalizing Flows

A normalizing flow, or flow, is a transformation of a simple distribution to a more complex one that relies on a sequence of differentiable and invertible mappings. Normalizing flows are expressive generative models with tractable distributions that are
efficient to evaluate and sample from, unlike VAEs and GANs. Intuitively, the idea is to start with a known starting distribution (also called the base distribution) and progressively generate more and more complicated distributions, using a series of nonlinear and invertible functions also called bijections [52].

Let \( x \) be a D-dimensional random vector. Assume it is related to a random vector \( u \) with the following relationship

\[
x = T(u) \quad \text{with} \quad u \sim p_u(u)
\]

where the distribution \( p_u(u) \) is assumed known and tractable. This distribution is referred to as the base distribution of the flow based model. The defining property of normalizing flows is that not only the transformation \( T(.) \) must be differentiable but also that \( T^{-1}(.) \) has to exist and be differentiable [85]. This imposes an additional requirement that the vector \( u \) must also be D-dimensional. These conditions culminate in \( x \) having a well-defined probability density function, given simply by the following change of variables formula [85, 52]:

\[
p_x(x) = p_u(u) |\det J_T(u)|^{-1} \text{ where } u = T^{-1}(x)
\]

In this case the above is naturally a bijective relationship. We can equivalently express the same notion as follows:

\[
p_x(x) = p_u(T^{-1}(x)) |\det J_{T^{-1}}(x)|
\]

The Jacobian determinant component can be interpreted as a volume correction term in this context where the Jacobian matrix \( J_T(u) \) is a \( D \times D \) matrix of first partial derivatives of \( T \) with respect to \( u \) [85]:

\[
J_T(u) = \begin{bmatrix}
\frac{\partial T_1}{\partial u_1} & \frac{\partial T_1}{\partial u_2} & \cdots & \frac{\partial T_1}{\partial u_D} \\
\frac{\partial T_2}{\partial u_1} & \frac{\partial T_2}{\partial u_2} & \cdots & \frac{\partial T_2}{\partial u_D} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial T_D}{\partial u_1} & \frac{\partial T_D}{\partial u_2} & \cdots & \frac{\partial T_D}{\partial u_D}
\end{bmatrix}
\]

In the context of generative models, base distribution \( p_u(u) \) is operated on by (or "flows through") the bijection \( T \), yielding a more complex distribution. This is called the generative direction. Notice that it is easy to sample a random point from the
base distribution and apply the transform $T(.)$ to generate its corresponding point in the more complex distribution $[52]$. For example, base distribution $p_u(u)$ can be a simple Gaussian and sampling a noise vector, the flow can generate a natural image corresponding to the particular sample.

On the other hand, the inverse direction $T^{-1}(.)$ is called the normalizing direction. In this direction, the complex and irregular form of the original distribution is progressively converted to a simpler, more regularized and "normalized" form. This perspective inspired the term "normalizing flows" as the transformation $T^{-1}(.)$ can be thought of as normalizing the original data distribution $[52]$.

Training of a normalizing flow is relatively straightforward. Assume that the base distribution $p_u(u)$ is parameterized by the set of parameters $\phi$ and the flow is parameterized by a set of parameters $\theta$. Define the set of parameters as $\Theta = \{\theta, \phi\}$. A common approach is to maximize the log-likelihood of the data distribution. For a dataset $D = \{x^{(i)}\}_{i=1}^{M}$ the log-likelihood under the model is given as follows:

$$
\log(p(D | \Theta)) = \sum_{i=1}^{M} \log(p_x(x^{(i)} | \Theta)) \quad (3.59)
$$

Using relation (3.57) we can express the log-likelihood as:

$$
\log(p(D | \Theta)) = \sum_{i=1}^{M} \log(p_u(T^{-1}(x^{(i)} | \theta) | \phi) + \log |det J_{T^{-1}}(x^{(i)} | \theta)| \quad (3.60)
$$

The above equation (3.60) can be used as a cost function to optimize the parameters $\Theta$ of a flow based neural network. $[52, 85]$ Notice that the likelihood is calculated for an entire dataset. The summation in the cost function can also be regarded as expectation. In practice we typically sample the dataset and apply mini-batch based gradient descent, using batches as unbiased estimators of the dataset.

Currently, one of the most widely used flow architectures are coupling flows $[52]$. Coupling method was introduced by Dinh et al. $[88]$ and is conductive to expressive transformations for flows.

Let us partition the input $x \in \mathbb{R}^D$ into two disjoint subspaces $(x^A, x^B) \in \mathbb{R}^d \times \mathbb{R}^{D-d}$. Define the bijection $h(., \theta)$ with a set of parameters $\theta$. Then one can define a function
$g : \mathbb{R}^D \to \mathbb{R}^D$ as follows \cite{52}:

$$
y^A = h(x^A; \Theta(x^B)) \\
y^B = x^B
$$

where the set of parameters $\theta$ for $h(.; \theta)$, called the coupling function, are defined by an arbitrary function $\Theta(x^B)$, not necessarily invertible. The output is obtained in full length simply by concatenating $y^A$ and $y^B$ again. It is very straightforward to invert this transform $g$:

$$
x^A = h^{-1}(y^A; \Theta(x^B)) \\
x^B = y^B
$$

To elaborate, notice that $y^A$ and $y^B$ are easily obtained given $y$ by partitioning. $x^B$ is immediately equal to $y^B$, and this allows for the recalculation of $\Theta(x^B)$. The coupling transform $h$ is often chosen as an easily invertible function, such as simple addition, in which case $h^{-1}$ ends up being subtraction. Therefore, $x^A$ is obtained in a simple manner, meaning the transform $g$ is very simple to invert. Furthermore, most coupling functions $h$ operate on $x^A$ in an elementwise fashion, making the overall Jacobian of $g$ triangular \cite{52}. This is important as the Jacobian determinant can be computed very efficiently for such flows.

This type of flow is highly expressive, and this is closely related to the possibility of making the function $\Theta(x^B)$ arbitrarily complex \cite{52}. The work by Kingma and Dhariwal \cite{54} in particular made use of a shallow ResNet architecture and obtained very impressive results, generating sizable, realistic looking images.
CHAPTER 4

PROPOSED METHOD

An important observation is that the analysis and synthesis transforms $g_a$ and $g_s$ in Factorized Prior Model and Scale Hyperprior Model, while straightforward in design, are expressive transforms. In particular, $g_a$ can be viewed as a feature extractor, relying on convolutional layers to obtain useful features that summarizes the input image. $g_s$ plays a similar role in the reverse direction by generating the image from summarized features, being an approximate inverse to $g_a$. A summary of these transforms are provided in Table 4.4 and a detailed layer-by-layer schematic is provided in Figure 4.13.

This expressivity of these transforms are primarily controlled by number of channels in the convolutional layers. An important realization is that increasing the number of channels may not yield much benefit as noted by Ballé et al. [5], especially when training for lower bitrates. While improving these transforms is still possible, as exemplified by Zhu et al. [50], they do not appear to be the main objective for improvement. In fact, this is evident simply by comparing Scale Hyperprior Model and Factorized Prior Model. They only differ in terms of their probabilistic modelling with the former having a more sophisticated prior model, meaning both networks have similar feature extraction capabilities. However, Scale Hyperprior Model significantly outperforms the latter. This trend continues in the follow up work by Minnen et al. [47], who implement more sophisticated prior models, this time in comparison to the one in Scale Hyperprior Model, to improve compression performance, all while retaining $g_a$ and $g_s$.

The mechanism behind these improvements is the reduction of mismatch between the actual joint distribution of all latent variables and the prior joint distribution that the
designer assumes. In fact, entropy, or bitrate, is minimized if the chosen prior model is identical to the marginal distribution of the latent representation \cite{5}. The compression performance improvements provided by sophisticated prior models reflect that the latent representation has indeed a complicated distribution. This is where normalizing flows can be helpful.

Normalizing flows, or simply flows, are intuitive and expressive generative models. The basic problem statement is to map a dataset $D$ with unknown distribution to a latent representation with a chosen distribution, effectively learning the complicated distribution of the dataset with the flow. If the flow is expressive enough, this mapping is achievable, as exemplified by the many studies mentioned in introduction. Furthermore, this comes at no expense of a loss of information, as these flow based transforms are fully invertible.

In this light, we hypothesize that a flow based transform may improve the performance of the prior model of the network chosen as the benchmark, Scale Hyperprior Model, resulting in bitrate savings. As mentioned, it is assumed here that the transforms $g_a$ and $g_s$ are expressive, therefore they are left unchanged. Instead, the latent representation of Scale Hyperprior Model, which is comprised of $y$ and $z$, is extended by introducing a third variable $u$, which corresponds to the space of extracted features at the output of $g_a$. The overall framework is described at a high level in Figure \ref{fig:4.1} and in layer level detail in Figure \ref{fig:4.14}.

$y$ is generated simply by pushing $u$ through the flow based transform. At this point, the operation is similar to the one in Scale Hyperprior Model. $z$ is generated by passing $y$ through the transform $h_a$. $z$ is then quantized, then transformed by $h_s$ to get scale components for the prior model of $y$. Using these scale components, $y$ is also quantized. Bit streams for both sets of quantized variables are generated during inference, marking the end of the compression process.

In decompression, the same scale components have to be computed again by decompressing quantized $z$ variables. The quantized $y$ variables are recovered by using these scales. We return to the $u$ space by inverting the flow based transform. The recovered $u$ is used to synthesize the reconstructed image.
4.1 Overall Framework and Operation

The overall proposed framework is described in Figure 4.1. The framework is architecturally based on Scale Hyperprior Model. The analysis and synthesis transforms $g_a$ and $g_s$, and the hyper-analysis and hyper-synthesis transforms $h_a$ and $h_s$ are all kept the same, except for a case where number of filters in the convolutional layers of $h_a$ and $h_s$ needed to be altered.

The first step is to apply the nonlinear analysis transform $g_a$ to the input $x$ to obtain the latent representation $u$:

$$u = g_a(x, \phi_g)$$

(4.1)

We denote the Flow Network in Figure 4.1 as $FN(\cdot; \theta_{NF})$, through which $y$ is obtained as:

$$y = FN(u; \theta_{NF})$$

(4.2)

$$z = h_a(y, \phi_h)$$

(4.3)

The utility of $\hat{z}$ is to provide the scale components, denoted as $\hat{\sigma}$, to a parametric
A Gaussian conditional that models the latent space $\tilde{y}$. More explicitly:

$$p_{\tilde{y}|\tilde{z}}(\tilde{y}|\tilde{z}, \tilde{\sigma}) = \prod_i (N(0, \tilde{\sigma}_i) \ast U(-\frac{1}{2}, \frac{1}{2}))(\tilde{y}_i) \quad (4.4)$$

where the scale terms $\tilde{\sigma}$ are obtained using the hyper-synthesis transform $h_s$ as follows:

$$\tilde{\sigma} = h_s(\tilde{z}, \theta_h) \quad (4.5)$$

The entropy model works the same way as in Scale Hyperprior Model. The hyper-latent $z$ is given by:

$$z = h_a(y, \phi_h) \quad (4.6)$$

which is used to compute the scale components as:

$$\tilde{\sigma} = h_s(\tilde{z}, \theta_h) \quad (4.7)$$

Just as it is the case in Scale Hyperprior Model, the final distribution of the latent representation $y$ is defined as:

$$p_{\tilde{y}|\tilde{z}}(\tilde{y}|\tilde{z}, \tilde{\sigma}) = \prod_i (N(0, \tilde{\sigma}_i) \ast U(-\frac{1}{2}, \frac{1}{2}))(\tilde{y}_i) \quad (4.8)$$

The latent representation $y$ is quantized during inference and during training additive uniform noise is utilized to simulate quantization effects. No quantization is performed on latent representation $u$, however since $u$ is obtained from the quantized $\hat{y}$ or noisy $\tilde{y}$ during decompression, we use another symbol, $\hat{u}$, for the reconstructed $u$ in either scenario. $\hat{u}$ is given by:

$$\hat{u} = FN^{-1}(\hat{y}; \theta_{NF}) \quad (4.9)$$

One can replace $\hat{y}$ above by $\hat{y}$ to describe the operation during training. Finally, the synthesis transform $g_s$ is utilized to reconstruct $x$:

$$\hat{x} = g_s(\hat{u}; \theta_g) \quad (4.10)$$

### 4.2 Architectural Choices for the Flow Network

Architectural choices for FN is nontrivial. Given a set of available layers, design of an optimal neural network is a combinatorial task. In the scope of this work, we restrict our attention to three particular designs.
The first will be referred to as the Efficient Network, or EFNet. This network relies on simple additive coupling blocks, comprised of separable convolutions and orthogonal transforms for channel mixing. To increase expressivity, many coupling blocks, namely 10, are utilized in a serially connected manner, which is expected to increase the runtime.

The second design will be referred to as the Hierarchical Channelwise Coupling Network, or HCNet in short. This design does not aim to restrict parameter count and relies on affine coupling layers, which introduce scale components to the coupling transform, 1 × 1 invertible convolutions, which is a more general transform for channel mixing compared to orthogonal transforms, as well as regular convolutional layers. The resultant model is more parameter intensive than EFNet. With the addition of an increased number of parameters and an additional inductive bias, i.e. the affine coupling, we expect this network to outperform the EFNet.

The final design is called Spatial Coupling Network, or SNet in short. The previous two architectures create partitions along the channel dimension for coupling. Here, we test an alternative. Inspired by wavelet transforms, we create partitions along spatial dimensions. However, concatenation of spatial partitions take place along the channel dimension as will be illustrated, which requires a much larger filter count at the hyperprior layers, resulting in a parameter intensive network. As such, additional improvements over HCNet is expected.

EFNet and HCNet can be summarized as illustrated in Figure 4.2. Channel dimension is fixed by setting M=192. Input and output tensors have the exact same shape, or more specifically have the same number of voxels, which is a necessary condition for invertibility.

These networks rely on channelwise coupling. The first step is to partition the input tensor along the channel dimension. In these two implementations, the input tensor is simply sliced in half as demonstrated in Figure 4.3. Note that this operation is clearly invertible by simple concatenation.

The slices are then coupled using an affine or additive coupling layer as shown Figure 4.4. Note that there is an abuse of notation here: \(x_A\) and \(x_B\) are not necessarily fed
Flow Network

Input Tensor: $u$

$K$ Channels

$H$ $W$

Output Tensor: $y$

$K$ Channels

$H$ $W$

Figure 4.2: Operation of EFNet and HCNet.

Input Tensor: $x$

$K$ Channels

$H$ $W$

Slice Along Channel Dimension

Create Two Chunks

$C$

$K/2$ Channels

$C$

$K/2$ Channels

Figure 4.3: Before coupling, input is sliced into two along the channel dimension.
in the order they are created in Figure 4.3. In EFNet, the two partitions are fed after swapping their order, as in Figure 4.8.

In EFNet, the coupling is additive and the function \( h(\cdot) \) in Figure 4.4 is simply a summation operation, which is easily reversed as illustrated in Figure 4.9. However in HCNet, it is instead replaced by an affine function that includes a scale component, which will be discussed. The orthogonal transform \( Q \) in Figure 4.4 is simply set to identity in HCNet. However, in EFNet, it is an orthogonal transform along the channel dimension, i.e. treating each \( C \times 1 \times 1 \) chunk of its input tensor with dimensions \( C \times H \times W \) as a vector. As such, EFNet relies on simple shuffling and parameter-efficient orthogonal transforms that operate only on a smaller chunk of the input for channel mixing. To generalize this mixing paradigm, HCNet relies on invertible \( 1 \times 1 \) convolutions as demonstrated by Kingma and Dhariwal in their Glow paper [54]. These operate similarly to the orthogonal transforms that were mentioned in that they also utilize matrix multiplication along the channel dimension. They are also initialized as orthogonal matrices, but instead, they can learn arbitrary invertible matrices. Although these layers are expected to be more efficient channel mixers, their inverses are not computed efficiently.

SNet, on the other hand, can be summarized as illustrated in Figure 4.5. Notice that dimensions of the output is different from those of the input. This is, however, not
a barrier to invertibility as the total number of voxels in both tensors are exactly the same. The input tensor is reversibly reorganized inside the flow to create the shape of the output tensor. Due to the shape of the output tensor, namely its higher number of channels, causes the hyperprior model to be more heavily parameterized in order to match its inputs shape. The resultant hyperprior model is described in Table 4.3.

As an alternative to channelwise partitioning, we test spatial partitioning for SNet as shown in Figure 4.7. The resultant partitions are then subjected to affine coupling as in HCNet. This is illustrated in Figure 4.6. The input tensor is first sliced as in Figure 4.3, just like in HCNet. However, note that the input was previously constituted by spatial partitions that were concatenated along the channel dimension. Thus, this time the coupling occurs along the spatial dimension.
Create Two Chunks

Apply Coupling

Channelwise Concatenation of $y_A$ and $y_B$

Figure 4.6: Spatial Coupling Layer. Very similar to the channelwise coupling, with the only difference residing in the way the input partitions are generated.

4.3 EFNet

The basic building block of Efficient Network (EFNet) is a coupling flow denoted by $E F N e t_i$, as shown in Figure 4.8. This flow is simply comprised of channel partitioning as in Figure 4.3, swapping the partitions, and applying additive coupling as in Figure 4.4 by setting $h(.)$ to simple addition.

For this implementation, the latent space representations $u$ and $y$ both have a channel dimension of 128. $x_A$ and $x_B$ are obtained by partitioning the input $x$ in channel dimension into two tensors with a channel dimension of 64 each. This shuffling operation is performed to add to the expressivity of the flow. It has a Jacobian determinant of 1, so it does not affect efficient Jacobian determinant computation.

The function $\Theta(\cdot; \theta_i)$ is designed in a non-invertible fashion, and is given by:

$$\Theta(\cdot; \theta_i) = \text{LeakyReLU} \circ \text{conv}_1(\eta_i, 2) \circ \text{LeakyReLU} \circ \text{sepconv}_1(\eta_i, 1) \quad (4.11)$$

where $\text{sepconv}$ is a depthwise separable convolution operation, which acts on each input channel individually. In this particular implementation, the kernel size for the
Figure 4.7: Spatial Partitioning, separating input into two tensors along either of the two spatial dimensions. Here, separation is demonstrated for height dimension H, which works by taking even indices for one partition and odd indices for the other. For width dimension, the operation is very similar. The resultant partitions are concatenated along the channel dimensions.
A separable convolution layer is 7 and for the regular convolutional layer, 1. This means that the $\text{conv}$ term yields linear combinations of the channels of its input. Neither of the convolutions perform any downsampling or upsampling and the number of channels are preserved at the output. LeakyReLU \cite{1} is given below again for convenience. The negative slope is set statically to 0.01.

$$\text{LeakyReLU}(x) = \max(0, x) + \text{negative\_slope} \times \min(0, x) \quad (4.12)$$

In tabular form the transform $\Theta$ is in Table 4.1.

<table>
<thead>
<tr>
<th>$\Theta(.)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sepconv, 1, 0.5C, 7x7</td>
</tr>
<tr>
<td>LeakyReLU</td>
</tr>
<tr>
<td>conv, 1, 0.5C, 1x1</td>
</tr>
<tr>
<td>LeakyReLU</td>
</tr>
</tbody>
</table>

It follows that $y_B$ is given by:

$$y_B = x_A + \Theta(x_B; \theta_i) \quad (4.13)$$

The transform $Q$ in Figure 4.8 is constituted by a learnable orthogonal matrix of size $64 \times 64$ and a bias term $b$. The orthogonality is ensured by using the following parametrization \cite{1}:

$$Q = exp(A) \quad (4.14)$$
where $A$ is a skew-symmetric matrix. It follows that $y_A$ is given by:

$$y_A = Qx_B + b$$  \hspace{1cm} (4.15)$$

where the matrix multiplication operation takes place by treating $x_A$ as a set of vectors along the channel dimension. The overall Jacobian is given in the form of a blocked matrix by:

$$J = P \frac{\partial [y_A; y_B]^T}{\partial [x_B; x_A]^T} = P \begin{bmatrix} Q_{64\times64} & 0_{64\times64} \\ D(\Theta) & I_{64\times64} \end{bmatrix}$$  \hspace{1cm} (4.16)$$

where $P$ is the permutation matrix that shuffles $x_A$ and $x_B$. $D(\Theta)$ represents the derivative of $y_B$ with respect to the additive component $\Theta(x_B)$ in Equation 4.13. Notice that the determinant of the Jacobian is exactly 1. This allows for the log-determinant terms to be dropped from the loss function during training.

Inversion of the flow EFNet is straightforward. It is based on reversing the operation of each block as shown in Figure 4.9. The first step is to get $x_B$ once again from $y_A$. It is indeed convenient that the inverse of an orthogonal matrix is simply its transpose.

$$x_B = Q^T(y_A - b)$$  \hspace{1cm} (4.17)$$

Thereafter, it is once again straightforward to obtain $x_A$:

$$x_A = y_B - \Theta(x_B, \theta_i)$$  \hspace{1cm} (4.18)$$

The overall flow EFNet is obtained in this work by serially connecting 10 additive coupling flows in Figure 4.8 as follows:

$$E F N e t = E F N e t_1 \circ E F N e t_2 \circ \cdots \circ E F N e t_{10}$$  \hspace{1cm} (4.19)$$

Figure 4.9: Inverse of the additive coupling flow in EFNet
4.4 HCNet

The Hierarchical Channelwise Coupling Network (HCNet) is based on affine coupling layers and invertible $1 \times 1$ convolutions as described before. The hierarchical nature will be described shortly. This network relies on partitioning the input into groups first and then applying coupling within each group in parallel. The operation for one such block for 4 groups is demonstrated in Figure 4.10.

In a nutshell the hierarchical nature stems from the reliance on coupling within groups. HCNet studied here is comprised of a succession of 3 such blocks: Namely, coupling within 4 groups in parallel, followed by coupling within 2 groups, and coupling within 1 group i.e. the entire tensor. This is illustrated in Figure 4.11.

Each group in each stage of the network has its own channelwise partitioning followed by an affine coupling flow. Referring to Figure 4.4 we make the following definitions:

$$Q = I \quad (Q \text{ is replaced with identity})$$

$$p = \theta(x_A) \quad \text{where } x_A \in \mathbb{R}^{C'/2 \times H \times W}, \ p \in \mathbb{R}^{C \times H \times W}$$

$$\hat{s} \triangleq \text{channels of } p \text{ with even index}, \ \hat{s} \in \mathbb{R}^{C'/2 \times H \times W}$$
Figure 4.11: Overall structure of HCNet. Note that after final coupling no $1 \times 1$ convolution is applied. The coupling structure here is referred to as 4-2-1.

\[
t \triangleq \text{channels of } p \text{ with odd index}, \quad t \in \mathbb{R}^{C/2 \times H \times W} \tag{4.23}
\]

where $t$ is the translation component of the affine transform and $\hat{s}$ is the precursor of the scale component. The nonlinear transform $\theta(.)$ is defined in tabular form sequentially as in Table 4.2. Note that PReLU is defined as

\[
PReLU(x) = \max(0, x) + \min(0, x) \tag{4.24}
\]

where $a$ is learnable, as implemented in PyTorch [1].

Table 4.2: Tabular description of $\theta(.)$. PReLU stands for the parametric ReLU function as implemented in PyTorch [1].

<table>
<thead>
<tr>
<th>$\theta(.)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv,1,C,5x5</td>
</tr>
<tr>
<td>PReLU</td>
</tr>
<tr>
<td>conv,1,C,5x5</td>
</tr>
<tr>
<td>PReLU</td>
</tr>
</tbody>
</table>

Note that for invertibility, one needs to ensure that scale component is indeed nonzero and is also not too large or too small. Therefore, one may apply the following transform to $\hat{s}$:

\[
s \triangleq \sigma(\hat{s}) \text{ where } s \in (0, 1) \tag{4.25}
\]
where \( \sigma \) is the sigmoid function. We can then define the affine coupling function \( h(.) \) as:

\[
y_B = h(x_B, \theta(x_A)) = s(x_A) \times x_B + t(x_A)
\]  
(4.26)

For the initial partitions \( x_A \) and \( x_B \), noting that \( s \) and \( t \) were obtained using \( x_A \) one may define:

\[
y_A = x_A
\]  
(4.27)

\[
y_B = s(x_A) \times x_B + t(x_A)
\]  
(4.28)

As long as the scale component \( s \) is nonzero, the coupling can easily be inverted:

\[
x_A = y_A
\]  
(4.29)

\[
x_B = \frac{y_B - t(x_A)}{s(y_A)}
\]  
(4.30)

Theoretically the sigmoid function always returns nonzero values, however, in practice very small values of \( s \) may affect the stability of the network. This will be addressed again in the next chapter, where we describe some precautions taken to prevent \( s \) from becoming too small.

The Jacobian is easy to compute for the coupling layer. For each coupling group we have:

\[
J = \frac{\partial[y_A, y_B]^T}{\partial[x_A, x_B]^T} = \begin{bmatrix} I & 0 \\ D(s(x_A)) & \text{diag}(s(x_A)) \end{bmatrix}
\]  
(4.31)

The overall Jacobian for each grouped coupling will be comprised of a blockwise diagonal matrix for each group. Therefore, the Jacobian log-determinant of the overall coupling will simply be the sum of the log-determinants of each group. HCNet is described in detail below in Algorithm 6.
Algorithm 6 HCNet 4-2-1 structure.

Input: $X$

for i=1,2,3 do

Partition $X$ channelwise into $2^{3-i}$ groups in sequence, as in Figure 4.10

for each group do

Current group: $x$

Partition group into two partitions $x_A$ and $x_B$ as shown in Figure 4.3

Apply the coupling flow illustrated in Figure 4.4 once

Output: $y$

end for

Obtain $X$ by concatenating all outputs $y$.

if $i \neq 3$ then

$X := \text{Invertible1x1}(X)$

end if

end for

Output: $X$

Finally, it is worth noting that grouped coupling layers connected in parallel can efficiently be computed using the grouped convolution of PyTorch [1].

4.5 SNet

The spatial coupling in SNet is very similar to the channelwise coupling in HCNet without the hierarchical implementation. However, unlike HCNet, SNet lacks $1 \times 1$ convolutions and instead utilizes spatial partitioning, as described in Figure 4.7

Overall architecture is illustrated in Figure 4.12

The first step is to apply spatial partitioning to the input tensor as demonstrated in Figure 4.7. This results in two partitions obtained by sampling even and odd indices along the height(H) dimension of the tensor respectively.

Two layers of coupling is applied to this tensor. Each coupling layer is the same as those described in HCNet with appropriate channel dimensions, as shown in Figure

58
Table 4.3: Tabular description of SNet. Notice that $h_a$ and $h_s$ are different from those in Scale Hyperprior Model [2].

<table>
<thead>
<tr>
<th>$g_a$</th>
<th>$g_s$</th>
<th>$h_a$</th>
<th>$h_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv,2,128,5x5</td>
<td>deconv,2,128,5x5</td>
<td>abs</td>
<td>deconv,2,384,5x5</td>
</tr>
<tr>
<td>GDN</td>
<td>IGDN</td>
<td>conv,1,576,3x3</td>
<td>ReLU</td>
</tr>
<tr>
<td>conv,2,128,5x5</td>
<td>deconv,2,128,5x5</td>
<td>ReLU</td>
<td>deconv,2,576,5x5</td>
</tr>
<tr>
<td>GDN</td>
<td>IGDN</td>
<td>conv,2,384,5x5</td>
<td>ReLU</td>
</tr>
<tr>
<td>conv,2,128,5x5</td>
<td>deconv,2,128,5x5</td>
<td>ReLU</td>
<td>deconv,1,768,3x3</td>
</tr>
<tr>
<td>GDN</td>
<td>IGDN</td>
<td>conv,2,128,5x5</td>
<td>ReLU</td>
</tr>
<tr>
<td>conv,2,192,5x5</td>
<td>deconv,2,3,5x5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.6 Denote the spatial partitions as $x_A$ and $x_B$. In the first coupling layer, $x_A$ is kept the same, but $x_B$ is subjected to affine coupling as described. Then, the transformed $x_B$ is kept the same and now $x_A$ is subjected to affine coupling using the transformed $x_B$ to generate the required $s$ and $t$ parameters. Refer to the operation so far as height coupling. The resultant $x_A$ and $x_B$ are then concatenated along channel dimension and the resultant tensor is now spatially partitioned along the width(W) dimension. Each of the two partitions are coupled separately in the same way as described for height coupling. Finally, the resultant tensors are all concatenated. This results in a very "tall" tensor. This requires a change in the number of filters in $h_a$ and $h_s$, as tabulated in Table 4.3.
Table 4.4: Tabular description of Scale Hyperprior Model. Note that Scale Hyperprior Model includes all four transforms and Factorized Prior Model includes only $g_a$ and $g_s$.

<table>
<thead>
<tr>
<th>$g_a$</th>
<th>$g_s$</th>
<th>$h_a$</th>
<th>$h_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv,2,128,5x5</td>
<td>deconv,2,128,5x5</td>
<td>abs</td>
<td>deconv,2,128,5x5</td>
</tr>
<tr>
<td>GDN</td>
<td>IGDN</td>
<td>conv,1,128,3x3</td>
<td>ReLU</td>
</tr>
<tr>
<td>conv,2,128,5x5</td>
<td>deconv,2,128,5x5</td>
<td>ReLU</td>
<td>deconv,2,128,5x5</td>
</tr>
<tr>
<td>GDN</td>
<td>IGDN</td>
<td>conv,2,128,5x5</td>
<td>ReLU</td>
</tr>
<tr>
<td>conv,2,128,5x5</td>
<td>deconv,2,128,5x5</td>
<td>ReLU</td>
<td>deconv,1,192,3x3</td>
</tr>
<tr>
<td>GDN</td>
<td>IGDN</td>
<td>conv,2,128,5x5</td>
<td>ReLU</td>
</tr>
<tr>
<td>conv,2,192,5x5</td>
<td>deconv,2,3,5x5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 4.12: SNet in detail.
Figure 4.13: Layer by layer depiction of Scale Hyperprior Model, redrawn from the work by Ballé et al. \cite{2}. N=128 and M=192 was utilized for this work, which results in \( y \) having 192 channels.
Figure 4.14: Layer by layer depiction of Scale Hyperprior Model \cite{2} with the addition of a flow based network. For SNet, the filter counts of $h_a$ and $h_s$ are altered.
CHAPTER 5

EXPERIMENTAL RESULTS

In the previous section, based on the framework proposed in Figures 4.14 and 4.1, three different architectures were proposed. Namely:

- the Efficient Network (EFNet), which is comprised of lightweight additive channelwise coupling layers.
- the Hierarchical Channelwise Coupling Network (HCNet), which extends the ideas in the EFNet and utilizes a more parameter intensive network.
- the Spatial Coupling Network (SNet), which utilizes the spatial partitioning paradigm shown in Figure 4.7 unlike the previous two models. The main objective for this network is to observe the performance of an alternative partitioning paradigm.

5.1 Methodology

For training, the following loss function is utilized:

\[
J = H(P_{y,z}) + \lambda \mathbb{E}\{||x - \hat{x}||^2\} \tag{5.1}
\]

where \(H(P_{y,z})\), the entropy, corresponds to the bitrate calculated over the coded latent space variables \(y\) and \(z\), \(\lambda\) is the Lagrangian parameter governing the tradeoff between rate and distortion, \(x\) is the input and \(\hat{x}\) is the reconstructed image. Training was performed with the five \(\lambda\) values shown in Table 5.1 with increasing \(\lambda\) corresponding to higher reconstruction accuracy in exchange for more storage.
Table 5.1: Lagrangian parameters, each governing a different image quality level

<table>
<thead>
<tr>
<th>Quality</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>0.0018</td>
<td>0.0035</td>
<td>0.0067</td>
<td>0.0130</td>
<td>0.0250</td>
</tr>
</tbody>
</table>

Scale Hyperprior Model \[2\] was used as the benchmark. EFNet, HCNet and the benchmark model were trained five times each for each quality level, yielding 15 networks. Batches of 8 random crops of size \(256 \times 256\) from CLIC2020 dataset (1633 images, obtained by combining Professional and Mobile training sets) were used and optimization was performed by the Adam optimizer \[78\] at an initial learning rate of \(10^{-4}\) for approximately 1.8 million steps, lowering the learning rate to \(10^{-5}\) for the last 0.2 million steps for fine tuning for a total training step count of approximately 2 million.

Training and all implementations of the networks were made using PyTorch \[1\] and CompressAI \[7\] libraries. Flow based networks were built benefiting from the library by Vincent Stimper\[1\]. The results are compared against JPEG, BPG, and Factorized Prior Model, the results for which are taken from CompressAI \[7\]. SNet was observed to be unsuccessful and training was terminated early. This will be addressed.

For rate-distortion curves, instead of MSE, PSNR is adopted. PSNR is a logarithmic scale of lack of distortion, related to MSE by:

\[
PSNR = 10 \log_{10} \frac{1}{MSE}
\]  

A higher PSNR corresponds to lower distortion. PSNR is given as a function of bits per pixel (bpp) in the plots, which corresponds to the storage required per pixel in the processed image, in bits.

Using the datapoints for rate-distortion curves, one may calculate a measure of bitrate saving between two given curves. To this end, consider two functions \(f(k)\) and \(g(k)\), where \(k\) is a given PSNR value. Each of these functions return a bpp value for a given PSNR. These functions here are approximated by linearly interpolating available data points.

\[1\] github.com/VincentStimper/normalizing-flows
At a particular PSNR value \( k \), we define percentage bitrate saving \( s(k) \) for \( g(k) \) over the baseline \( f(k) \) as:

\[
    s(k) = 100 \frac{f(k) - g(k)}{f(k)}
\]

(5.3)

We are interested in the average bitrate saving \( s_{m,n} \) over a period \([m, n] \), where \( m \) and \( n \) are PSNR values. Here we define \( s_{m,n} \) as:

\[
    s_{m,n} = \frac{1}{|m - n|} \int_{m}^{n} s(k) \, dk
\]

(5.4)

### 5.2 Results and Findings

The performance of the benchmark model, EFNet, and HCNet are evaluated on the well-known Kodak test images dataset, CLIC2020 Mobile test dataset, and CLIC2020 Professional test dataset, illustrated in Figure 5.1, Figure 5.2, and Figure 5.3 respectively. Curves for BPG, JPEG and Factorized Prior Model, data for which are taken from CompressAI [7], are also included.

Using datapoints for the benchmark model, EFNet, and HCNet in Figure 5.1, Figure 5.2 and Figure 5.3 bitrate savings are estimated based on the formulation in Equations (5.3) and (5.4). The results are provided in Table 5.3, Table 5.4, and Table 5.5.

HCNet, summarized in Figure 4.11 and Table 4.2 underwent some changes during development. Its final form has the coupling hierarchy 4-2-1, as described in Figure 4.11. Initially it had the same coupling layout, however the scaling component of the affine coupling, defined in Equation (4.25), had the following implementation:

\[
    s \triangleq \sigma(\hat{s}) + 0.01
\]

(5.5)

This appeared to contribute to the networks instability, as \( s \) was allowed to be very small, which could lead to very large values during inversion. A related discussion is provided in the work by Behrmann et al. [89]. Validation loss for the first implementation is given in Figure 5.4, where spikes of loss where the network loses training stability are very apparent. However, training eventually converged for this implementation.
Changing the hierarchy of the network from 4-2-1 to 1-2-4 also appeared to exacerbate this phenomenon, yielding NaN (not a number) values, causing training to collapse. The exact cause of this instability, which likely stemmed from a diminished numerical invertibility of a theoretically invertible network, was not precisely identified. A backup shortly before the occurrence of NaN was investigated. $1 \times 1$ invertible convolutions retained singular values reasonably small and away from zero, approximately in the range $[0, 4]$. Similarly, the remaining layers did not have any apparent anomalies either.

Benefiting from the findings of Behrmann et al. [89], the training instability problem was primarily alleviated by making the following change to the scaling component:

$$s \triangleq 0.75 \sigma(\hat{s}) + 0.25$$  \hspace{1cm} (5.6)

With this modification, the 4-2-1 hierarchy demonstrated significantly improved performance, as shown in Figure 5.5. Unexpectedly, however, for higher $\lambda$ values, namely $\lambda = 0.0130$ and $\lambda = 0.0250$, HCNet appeared to suffer from training instability despite the modification. The instability is illustrated in Figure 5.6. Furthermore, for the higher $\lambda$ values, the network also suffered from reduced performance, visible in Figure 5.1 and especially evident in Figure 5.2 and Figure 5.3. In this case as a precaution, for a handful of times when NaN values occurred during training, the epoch would be repeated from a previous backup.

SNet is not included in rate-distortion curves because a stable enough model with a notable performance did not emerge. SNet implementations, either frequently returned NaN values or simply showed divergent behavior, illustrated in Figure 5.7 where the training was terminated early. In the most successful experiment, coupling layers after spatial partitioning along width dimension, which is illustrated in Figure 4.12, were removed. This meant that partitioning and coupling along height dimension took place, while only the partitioning step took place along width dimension. Training initially appeared promising, however diverged slowly after a point, illustrated in Figure 5.8. In an attempt to improve training stability, GDN layers were introduced as a replacement to PReLU, however the resultant network did not converge, as illustrated in Figure 5.9.

Figures 5.16, 5.17, and 5.18 demonstrate performances of the compression meth-
ods of interest. Figures starting from Figure 5.10 up to and including Figure 5.15 demonstrate the latent representations created by EFNet, HCNet, and Scale Hyper-prior Model respectively for two different Kodak test images at three different quality levels each. To generate these plots, three channels with the highest information were picked and averaged. Information was derived from likelihood values for the quantized latent representation $y$ with the following expression:

$$I = -\log_2(P(\hat{y})) \quad (5.7)$$

Average runtimes over Kodak test images and number of parameters are provided in Table 5.2. The runs were made on a NVIDIA RTX3060 Laptop GPU for CUDA runs and on an Intel(R) Core(TM) i7-10875-H CPU for the others.

Table 5.2: Number of parameters and runtime performance statistics of Scale Hyper-prior Model, EFNet, and HCNet on Kodak dataset and number of parameters of SNet.

<table>
<thead>
<tr>
<th>Architecture</th>
<th># Params</th>
<th>Enc(GPU)</th>
<th>Dec(GPU)</th>
<th>Enc(CPU)</th>
<th>Dec(CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale Hyperprior</td>
<td>5.08M</td>
<td>0.098s</td>
<td>0.026s</td>
<td>0.540s</td>
<td>0.700s</td>
</tr>
<tr>
<td>EFNet</td>
<td>5.31M</td>
<td>0.141s</td>
<td>0.045s</td>
<td>0.597s</td>
<td>0.757s</td>
</tr>
<tr>
<td>HCNet</td>
<td>7.57M</td>
<td>0.137s</td>
<td>0.035s</td>
<td>0.628s</td>
<td>0.783s</td>
</tr>
<tr>
<td>SNet</td>
<td>57.66M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5.3: Bitrate savings yielded by EFNet and HCNet over Scale Hyperprior Model \[2\] for a given PSNR range on Kodak test dataset.

<table>
<thead>
<tr>
<th>PSNR Range</th>
<th>EFNet</th>
<th>HCNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.5dB-31.5dB</td>
<td>1.89%</td>
<td>5.35%</td>
</tr>
<tr>
<td>31.5dB-35dB</td>
<td>1.34%</td>
<td>-1.64%</td>
</tr>
<tr>
<td>Total Saving</td>
<td>1.63%</td>
<td>2.09%</td>
</tr>
</tbody>
</table>

Table 5.4: Bitrate savings yielded by EFNet and HCNet over Scale Hyperprior Model \[2\] for a given PSNR range on CLIC2020 Mobile test dataset.

<table>
<thead>
<tr>
<th>PSNR Range</th>
<th>EFNet</th>
<th>HCNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.75dB-32.5dB</td>
<td>-7.01%</td>
<td>0.20%</td>
</tr>
<tr>
<td>32.5dB-36dB</td>
<td>-1.08%</td>
<td>-12.62%</td>
</tr>
<tr>
<td>Total Saving</td>
<td>-4.15%</td>
<td>-5.99%</td>
</tr>
</tbody>
</table>

Table 5.5: Bitrate savings yielded by EFNet and HCNet over Scale Hyperprior Model \[2\] for a given PSNR range on CLIC2020 Professional test dataset.

<table>
<thead>
<tr>
<th>PSNR Range</th>
<th>EFNet</th>
<th>HCNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.75dB-33.5dB</td>
<td>-6.2%</td>
<td>1.63%</td>
</tr>
<tr>
<td>33.5dB-36.75dB</td>
<td>-2.01%</td>
<td>-7.79%</td>
</tr>
<tr>
<td>Total Saving</td>
<td>-4.25%</td>
<td>-1.13%</td>
</tr>
</tbody>
</table>
Figure 5.1: Performance comparison between EFNet, HCNet, Scale Hyperprior Model [2], BPG, JPEG, and Factorized Prior Model [2] on Kodak images. EFNet, HCNet, and Scale Hyperprior models were trained, the remaining results were taken from CompressAI [7].
Figure 5.2: Performance comparison between EFNet, HCNet, Scale Hyperprior Model [2], BPG, JPEG, and Factorized Prior Model [2] on CLIC2020 Mobile test dataset. EFNet, HCNet, and Scale Hyperprior models were trained, the remaining results were taken from CompressAI [7].
Figure 5.3: Performance comparison between EFNet, HCNet, Scale Hyperprior Model [2], BPG, JPEG, and Factorized Prior Model [2] on CLIC2020 Professional test dataset. EFNet, HCNet, and Scale Hyperprior models were trained, the remaining results were taken from CompressAI [7].
Figure 5.4: Training instability of the first prototype of HCNet, $\lambda = 0.0035$. Notice the spikes of loss.

Figure 5.5: Improved stability of HCNet, $\lambda = 0.0035$. $s = 0.75 \sigma(\hat{s}) + 0.25$
Figure 5.6: Instability of HCNet during training for $\lambda = 0.0130$, $s = 0.75 \sigma(\hat{s}) + 0.25$

Figure 5.7: Training instability of SNet, $\lambda = 0.0035$. Notice the spikes of loss.
Figure 5.8: Training divergence of SNet, $\lambda = 0.0035$.

Figure 5.9: Training instability of the prototype of SNet with GDN, $\lambda = 0.0035$. Notice the divergence of loss.
Figure 5.10: Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0.0018$. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots.
Figure 5.11: Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0.0035$. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots.
Figure 5.12: Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0.0130$. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots.
Figure 5.13: Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0.0018$. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots.
Figure 5.14: Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0.0035$. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots.
Figure 5.15: Compression performance and latent space visualization on a Kodak test image for the benchmark, EFNet, and HCNet, respectively, with $\lambda = 0.0130$. Darker pixels represent locations with less information content, and brighter pixels represent locations with higher information content on the latent space plots.
Figure 5.16: Visual results for an image from Kodak test images.
Figure 5.17: Visual results for an image from Kodak test images.
Figure 5.18: Visual results for an image from Kodak test images.
In terms of bitrate saving, both EFNet and HCNet outperform Scale Hyperprior Model [2] in the widely used Kodak dataset as illustrated in Figure 5.1 and Table 5.3. This is also observable in the latent representations, shown in figures from Figure 5.10 to Figure 5.12. EFNet and HCNet appear more concentrated in the distribution of information content in the latent space, evident by their darker illustrations. As bitrate increases, EFNet and HCNet, similar to Scale Hyperprior Model, have their latent spaces more information-dense. This is evident as their latent representations in the illustrations become lighter.

EFNet, consisting of additive coupling layers, appears to provide overall worse performance than Scale Hyperprior Model on the other two datasets, approaching or slightly exceeding the performance of Scale Hyperprior Model at higher bitrates, as illustrated in Figure 5.2 and Figure 5.3 as well as on Table 5.4 and Table 5.5. On the other hand, at lower PSNR values, HCNet often notably surpasses the other two networks.

This might reflect that simple additive coupling may not provide the best performance or generalization. In this particular case, and unsurprisingly, affine coupling layers comprised of reasonably sized neural networks appear more promising.

Interestingly, even though HCNet suffers from reduced PSNR at higher bitrates as observed in rate-distortion curves, in latent representation plots this is not observable. HCNet appears to outperform the other two networks. Referring to the training instability as mentioned in the previous section, stabilizing HCNet further may completely prevent this reduction in reconstruction accuracy as it is highly likely that the network relinquishes stability for certain images or textures with the way HCNet is currently implemented. Even though normalizing flows appear to improve the compression performance, a primary concern is their stability. A theoretically invertible network may not be invertible in practice. Although certain precautions were taken in this work as mentioned, there likely remain many improvements that can be made in this regard.

EFNet did not appear to generalize well at lower bitrates to the two datasets other
than Kodak test images. Indeed, the flow in EFNet can learn the identity function, and in this regard the finding is somewhat unexpected. The reason for this reduction is unclear. It can be expected that changes in training dataset, as well as allowing for longer training than was allowed in this work may alleviate or solve this problem.

SNet, which relied on a spatial partitioning scheme described in Figure 4.7, was not found to be a feasible flow based network. The concept, as implemented in this work, leads to considerably large networks that often suffer from training instabilities. The network was simply investigated as an alternative coupling strategy.

Finally, in terms of runtime performance, EFNet and HCNet generally did worse than the Scale Hyperprior Model. This is expected. The sequential nature of the implemented flows introduces additional delays.
CHAPTER 6

CONCLUSION AND OUTLOOK

This thesis work studied the performance of a framework comprised of flow based networks and a known learned image compression architecture. All networks were trained in an end-to-end fashion, showed evidence of validating the initial hypothesis that an addition of a flow would enhance the latent representation, improving the performance of the prior model and reducing bitrate. The main reference of comparison was the work by Ballé et al. [5]. Proposed networks show notable improvement over the benchmark.

Although training was performed on a particular dataset for a given number of steps, making alterations in training strategy may prove beneficial. This may require investigation.

A common concern emerged as the stability of flows. While this work made some observations in this regard, more work needs to be done to understand and implement effective measures to keep normalizing flows stable and indeed invertible.

The focus of this work was three different designs of coupling flows. Alternative flow types and alternative coupling mechanisms, as well as flows with different architectural choices may improve performance further while ensuring stability. This may be an important direction for future work.

Finally, I would like to state that it has been a pleasure.
REFERENCES


