UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

LEONARDO DE SOUZA AUGUSTO

Computationally-Efficient Neural Networks for Image Compression

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Mateus Grellert da Silva

Porto Alegre
February 2024

“You who are our future, Tell me this and tell me true.
Has your journey been good? Has it been worthwhile?”
— VENAT

ACKNOWLEDGMENTS

First of all, I would like to thank my father, mother and my three brothers, who always inspired and supported me throughout my entire life with the path that I have chosen.

To my lovely girlfriend, who went through thick and thin with me throughout all our graduation, always pushing me forward and supporting me. You inspire me and always make me be a better person.

To all of my friends that have been with me as long as I can remember, always making me laugh even in the toughest times, and always encouraging each other.

To all my professors who went above and beyond to teach with passion.

Finally, to my advisor, who actively helped me throughout all the work done in this thesis, giving me directions and valuable advice, and being a wonderful person.

I would not have reached this far without them, and I cherish every moment we had.

ABSTRACT

Neural-network image compression (NNIC) is an emerging field, with notable works achieving promising results in terms of image quality, but yet to achieve feasible computational times. NNIC solutions employ the use of Autoencoders (AEs) (compression networks made of an encoder and a decoder part), commonly built with convolutional layers. Recent publications show that NNIC networks are capable of obtaining equal or better rate-distortion performance and visual quality on image compression when comparing with traditional compression methods, while allowing more flexibility on the model design. However, these networks introduce new challenges regarding computational cost, since NNIC demands high computational power for compression and decompression and has not achieved great results in regards to processing time, even when specialized platforms like GPUs are used. This work proposes to optimize one of the reference models available in the literature to achieve better processing time while trying to maintain the compression quality. Differently from other solutions that aim at achieving higher compression or better image quality, our proposal will focus on reducing the computational cost of NNIC techniques, with special focus on the decoder side of these networks. Experimental results, gathered from compressing and decompressing images from the Kodak dataset, show that with small pruning-based changes on the decoder layers of the network, it is possible to achieve, on average, for the lowest compression ratio, 33.33% reduction in decompression time when using the best model for GPU, with an average PSNR loss of 0.01 dB. When using the best model for CPU, the decompression time reduction was of 55%, with a PSNR loss of 0.07 dB. The quality and performance metrics were gathered from compressing/decompressing the images of the Kodak dataset. Although, the CPU decompression time does not achieve ideal decompression times when compared to JPEG 2000, while the GPU reaches similar results as JPEG 2000. We expect that this work sparks interest in developing ways to make NNIC as accessible as possible with current technology and that it can help the development of current, and future, media formats.

Keywords: Deep Neural Networks. Autoencoders. Image Compression.

Redes Neurais para Compressão de Imagem Computacionalmente Eficientes

RESUMO

A compressão de imagens em redes neurais (NNIC) é uma área emergente, com trabalhos notáveis alcançando resultados promissores em termos de qualidade de imagem, mas ainda sem atingir tempos computacionais viáveis. As soluções NNIC empregam o uso de Autoencoders (AEs) (redes de compressão compostas por um codificador e uma parte decodificadora), comumente construídas com camadas convolucionais. Publicações recentes mostram que as redes NNIC são capazes de obter desempenho de taxa-distorção e qualidade visual iguais ou melhores na compressão de imagens quando comparadas com métodos de compressão tradicionais, ao mesmo tempo que permitem maior flexibilidade no design do modelo. Porém, essas redes introduzem novos desafios em relação ao custo computacional, uma vez que NNIC demanda alto poder computacional para compressão e descompressão e não tem alcançado grandes resultados em relação ao tempo de processamento, mesmo quando são utilizadas plataformas especializadas, como GPUs. Este trabalho propõe otimizar um dos modelos de referência disponíveis na literatura para obter melhor tempo de processamento e ao mesmo tempo tentar manter a qualidade da compressão. Diferentemente de outras soluções que visam obter maior compressão ou melhor qualidade de imagem, nossa proposta focará na redução do custo computacional das técnicas NNIC, com um foco especial no lado decodificador destas redes. Resultados experimentais, obtidos pela compressão e descompressão das imagens do conjunto de imagens Kodak, mostram que com pequenas alterações baseadas em poda nas camadas decodificadoras da rede, é possível atingir, em média, para a menor taxa de compressão, redução de 33.33% no tempo de descompressão, ao utilizar o melhor modelo para GPU, com uma perda média de PSNR de 0.01 dB. Ao utilizar o melhor modelo para CPU, a redução do tempo de descompressão foi de 55%, com perda de PSNR de 0.07 dB. Porém, o tempo de descompressão da CPU não atinge tempos de descompressão ideais quando comparado ao JPEG 2000, enquanto a GPU atinge resultados semelhantes aos do JPEG 2000. Esperamos que este trabalho desperte o interesse no desenvolvimento de formas de tornar o NNIC tão acessível quanto possível, com a tecnologia atual, e que possa ajudar no desenvolvimento de formatos de mídia atuais e futuros.

Palavras-chave: Redes Neurais Profundas, Autoencoders, Compressão de Imagem.

LIST OF ABBREVIATIONS AND ACRONYMS

NN	Neural Network
DNN	Deep Neural Network
CNN	Convolutional Neural Network
AE	Autoencoder
VAE	Variational Autoencoder
CAE	Convolutional Autoencoder
VR	Virtual Reality
HEVC	High Efficiency Video Coding
VVC	Versatile Video Coding
AV1	AOMedia Video 1
NNIC	Neural-Network Image Compression
NLT	Non-linear Transforms
CAVLC	Context-Adaptive Variable Length Compression
ReLU	Rectified Linear Unit
FPS	Frames Per Second
MSE	Mean Squared Error
PSNR	Peak Signal-to-Noise Ratio
SSIM	Structural Similarity Index Measure
MS-SSIM	Multiscale Structural Similarity Index Measure
LPIPS	Learned Perceptual Image Patch Similarity
BPG	Better Portable Graphics
JPEG	Joint Photographic Experts Group
LSTM	Long Short-term Memory Networks
GDN	Generalized Divisive Normalization

IGDN	Inverse Generalized Divisive Normalization
GAN	Generative Adversarial Networks
BPP	Bits Per Pixel
VRAM	Video Random Access Memory
CSV	Comma-Separated Values
PNG	Portable Network Graphics
BMP	Bitmap
DCT	Discrete Cosine Transform
FDCT	Forward Discrete Cosine Transform

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Today, most of the data that is transmitted through the Internet is from digital media, i.e., video streaming (YouTube, Twitch, etc.), images, audio, etc. To quantify this trend, the Sandvine Global Internet Phenomena Report says that, on the first half of 2022, almost 65.93% of Internet traffic were videos (this includes TV, video and streaming downloads) (SANDVINE, 2023). Almost all of this information is transmitted in a compressed form, since its raw (uncompressed) representation requires prohibitive amounts of data to be stored or transmitted through the Internet. To exemplify, a 3840×2160 (4K Ultra High Definition – UHD) uncompressed sequence with a frame rate of 60 frames per second (FPS), with 8 bit depth with 3 channels (24 bits for colored images), and 10 minutes long, it would occupy more than 800 Gigabytes, as seen on Equation 1.1. This clearly demonstrates the necessity of data compression, since it would be completely unfeasible to store or to transfer this much data through the Internet.

$$Video\ Size = \frac{600 * 60 * 3840 * 2160 * 24}{8 * 2\hat{}30} = 834.27\ Gb \tag{1.1}$$

Up to this date, we can highlight two ways of compressing data: using traditional encoders and decoders (jointly referred to as codecs) and using special deep neural networks (DNN) called autoencoders (AE) (BANK; KOENIGSTEIN; GIRYES, 2020). Traditional codecs are currently the most used form of compression, with most modern systems having hardware support for them. Popular codec examples are: High Efficiency Video Coding (HEVC) (SULLIVAN et al., 2012), Versatile Video Coding (VVC) (BROSS et al., 2021), and AOMedia Video 1 (AV1) (ROSENBERG, 2018). Neural-network image compression (NNIC) (O'NEILL, 2020), on the other hand, is a relatively new field that is rising on popularity. The main difference from the codecs is that, instead of using complex algorithms to encode and decode images and videos, it uses DNN and other machine learning techniques to achieve this task (YANG et al., 2023). Compression Neural Networks (NN) represent important alternatives to existing compression solutions, since the DNNs are highly flexible and can be easily adjusted for different input domains. With enough effort from research, they have the potential to completely change the field and even replace traditional codecs in the coming years.

Video and image compression is already a consolidated field, which has a wide adoption of both hardware and software solutions that were developed in the last decades. Traditional codecs like HEVC, VVC and AV1 employ what is called a block-based hy-

brid coding. Block-based coding is the idea of processing the data of each frame in small blocks to reduce complexity and also to allow a more fine-grain compression. Hybrid coding means that the codecs use both prediction and transforms to generate the compressed form of each block (BUDAGAVI et al., 2013).

One limitation of these codecs is clearly the time needed to finalize a project of a new codec. HEVC took 10 years after the launch of AVC to be finalized, and VVC took 7 years, launching in 2020 (BROSS et al., 2021). Even with VVC taking less time to release, it is still a considerable amount of time. However, This development time is required because it involves several steps like releasing a call for evidences of relevant new technology, performing experiments and adjustments, organizing expert meetings to decide what will effectively be included in the standard, patent/royalties agreements etc.

NNIC, on the other hand, is based on AE networks with non-linear transforms (NLT) capabilities. NLTs, although being harder to determine for desirable properties, have a lot of potential on closely adapting to the source, improving compression (BALLÉ et al., 2020). This also means that it is possible to use NLTs to focus on compression of specific sources, for example, nature images, space images and so on.

Recent research on NNIC have been tackling the challenge of compression using lossy compression, non linear transform coding and compressive AEs, for example (BALLÉ; LAPARRA; SIMONCELLI, 2016) (THEIS et al., 2017). The work proposed by (BALLÉ; LAPARRA; SIMONCELLI, 2016) employs an end-to-end AE network based on NLTs for an image compression model, to optimize the rate-distortion performance. The model was optimized to the Mean Square Error (MSE), but could provide even better results using a perception metric instead, according to the authors. This compression method obtained promising results of an improved rate-distortion over the JPEG and JPEG-2000 codecs for most images and bit rates and also provides more flexibility since the model can be optimized for other metrics. The paper by (THEIS et al., 2017) is also based on compressive AEs for lossy image compression, but this time targeting a computationally-efficient model. Similarly to Ballé, Theis' work aims to optimize the rate-distortion performance while also focusing on the quantization and entropy rate estimation.

One of the main problems that prevent a widespread adoption of NNIC solutions is related to the computational cost of the DNNs required to build them. We have seen a lot of progress towards making compression NNs as good as some classic codecs, like JPEG and JPEG 2000 for images (BALLÉ; LAPARRA; SIMONCELLI, 2016), in terms

of image quality, but we are still behind in terms of performance. As reported in (DICK et al., 2021), these models are much more complex than existing codecs, and, therefore, are much more costly on the performance. In (DICK et al., 2021), it is reported that even the fastest compression NN model is still 30× slower than JPEG2000 on decompression.

With that in mind, this work proposes a solution to lower the computational cost of image compression NNs. Our study builds upon the model proposed by (BALLÉ; LAPARRA; SIMONCELLI, 2016), since it is one of the most noteworthy solutions in the literature. It is also highly efficient in terms of compression, while still keeping a simpler network compared with more recent developments.

In this work, we will be focusing more on the decoder part of the model, since images and videos are encoded much less often than decoded. For instance, a video that is uploaded to a streaming platform like YouTube is typically encoded only once on the user device. On the other hand, any user that watches the same video will have to decode it every time it is played. The same applies to the images that are used as thumbnails for the videos. Therefore, solutions that reduce decoding complexity have a much greater impact on processing and energy performance.

Our motivation comes from the fact that lowering the computing requirements of these solutions will make them more accessible and enable their processing on consumer-grade hardware with limited processing and energy resources.

1.1 Document Organization

This document is organized as follows: Chapter 2 will present the theoretical foundation for this work, discussing how compression works, and how do NNs fit on this topic. Chapter 3 is going to show some of the related works that researched on compression NNs and that will serve as a base for our work. Chapter 4 will have our proposal and methodology for tackling the performance side of the proposed model for our compression NN. On Chapter 5 we will discuss the proposed models and the results obtained. Finally, on Chapter 6 we will conclude this work and discuss possible future studies on the topic.

2 BACKGROUND

The following sections explain the background of this work, starting on how compression works and some consolidated image compression methods that also serve as a basis for bench-marking some of the most prominent NNIC solutions. After that, NNs and deep learning are discussed, as well as how these models are seen as a promising alternatives compared with state-of-the-art image and video compression methods.

2.1 Data Compression

The compression of data relies on effectively removing the redundancy present on the bits that represent such data. Considering that we have a sequence of bits, zeros and ones, that repeat themselves multiple times, compression can be applied to it, by removing zeros and ones that repeat too many times concurrently and replacing it with some equivalent representation. This can effectively reduce the size of the sequence without losing its original representation. For example, the sequence of bits 110000, after compression, is equivalent to 2140 (two ones and four zeros). This example is based on run-length coding, that is called lossless compression, since, after decompressing, the data will be the exact same as it was before compression (YANG et al., 2023).

In a more formal way, compression uses the concept of information entropy as a way of measuring how much information value a symbol has (YANG et al., 2023). A symbol in this context can be understood as bit or a sequence of bits (YANG et al., 2023). If a sequence has symbols that occur very often, it has a low entropy, meaning it is easier to correctly predict the next symbol. By knowing the symbol distribution beforehand, we can devise en efficient way of compression this data using the probabilities of each symbol, such that the most common ones have the least amount of bits and can thus be compressed even more than symbols that do not repeat that often. This approach is employed in famous compression methods such as Huffman and Context-Adaptive Variable Length Compression (CAVLC) (RICHARDSON, 2004). The goal of these methods is to maximize compression for a given unknown distribution (new sequence being encoded) while keeping complexity relatively low.

To further increase compression, a technique called quantization can be employed. This technique improves compression by representing the same numbers in a reduced range of values. Current image and video codecs employ a quantization step in their

pipeline (although they also support lossless modes as well) (RICHARDSON, 2010). In AEs, the quantization was not so trivial to implement until not so long ago, since the quantization process is non-differentiable. Although, there are techniques that make quantization effective on AEs as seen in (THEIS et al., 2017). The main disadvantage is that the quantized data cannot be completely restored (RICHARDSON, 2010), configuring what is called a lossy compression. In lossy coding, the decoded data will not be the same as the original, so quality becomes a new parameter that must be considered.

2.2 Visual data representation

A video is a sequential sequence of images, also called frames, that are transmitted on a screen, with a target rate of frames per second, to create a visual experience. A frame is the same as an image, and is composed of pixels, which is the name given to a sample of a numerical value present on the matrix that represents the frame (BUBOLZ, 2021).

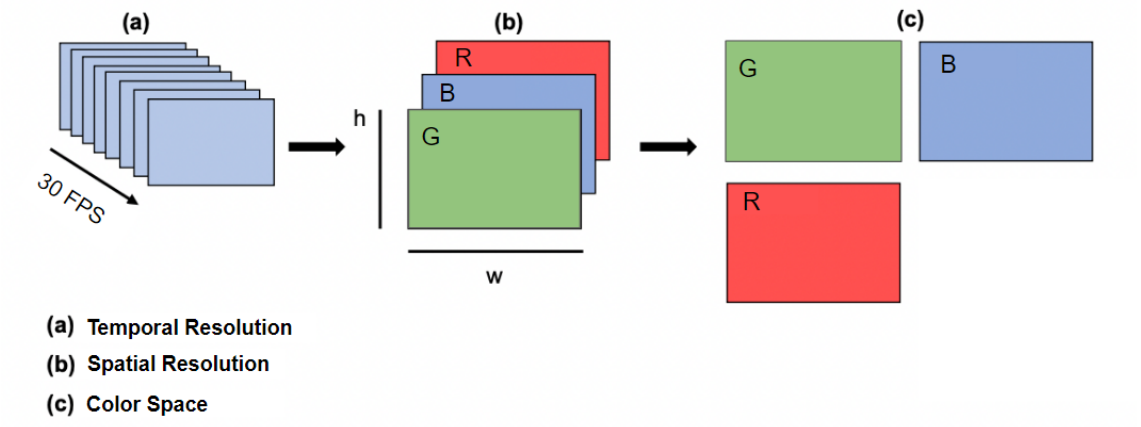
Within an image, or video, there are two most common color schemes, represented as 3 channels, the RGB (Red, Green and Blue) and the YCbCr (Luminance, blue chrominance and red chrominance). On RGB the colors are represented by a combination of the three colors, red, green and blue, while on YCbCr the luminance information is used to represent the colors (BUBOLZ, 2021).

Figure 2.1 shows on (a) the Temporal Resolution, where the video frames per second rate is defined. On (b), the Spatial Resolution, is where the height and width of the image is defined in pixels. On (c), the Color Space, is where the amount of pixels in each color space (RGB) are defined (BUBOLZ, 2021).

2.3 Conventional Image and Video Codecs

One of the most classic methods of compression is the JPEG standard, launched in 1992 and named after the group that created it, the Joint Photographic Expert Group, mainly used for compressing images (JPEG, 1992). JPEG can be considered one of the most successful standards because of its capacity to reduce images file size with relatively simple operations (linear transforms, quantization, and entropy). It is not very efficient in terms of image quality, but the efficiency on reducing the amount of data from the original can easily trade off the lack of image quality. JPEG, as it was first released, is considered

Figure 2.1: Basic concepts of visual data representation: resolution and color space.



Source: Adapted from (BUBOLZ, 2021)

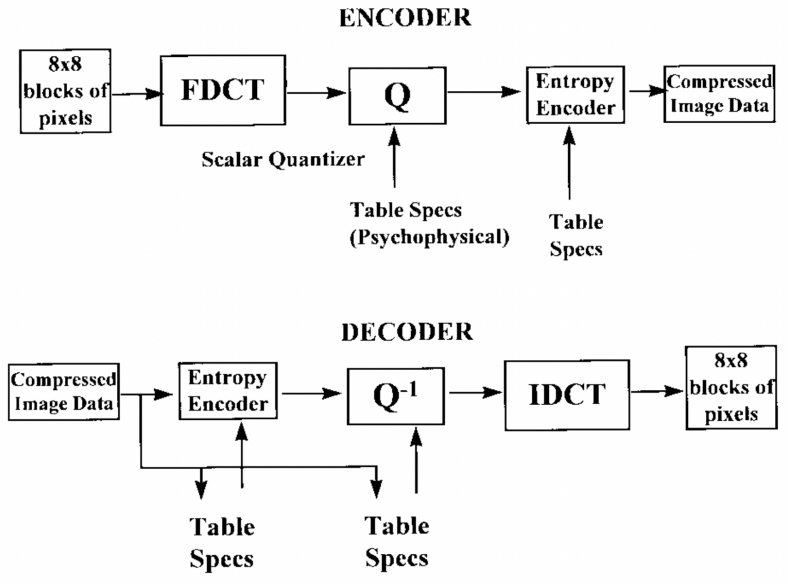
a lossy compression method. Following its release, JPEG Lossless was also introduced alongside the JPEG standard. The JPEG standard, although being widely used to this day, cannot reach lower bit rate compression at acceptable quality levels, as seen in (BALLÉ; LAPARRA; SIMONCELLI, 2016), in which it is easily surpassed in quality by the AE model proposed.

Figure 2.2 shows the block diagram of the JPEG codec. In this example, it compresses an image with dimension of 8 by 8 pixels, which is spectrally analyzed using a Forward Discrete Cosine Transform (FDCT) algorithm, and its resulting Discrete Cosine Transform (DCT) scalars are scalar quantized. Then, after quantization, the DCT quantized values are entropy coded, using run-length coding. The decompression is then done by the encoder, which does the inverse operations of the encoder (HASKELL et al., 1998).

The JPEG 2000 image compression standard, released in 2000, 8 years after the first release of JPEG, was developed as the successor to the JPEG providing options for both lossless and lossy compression on launch, contrary to standard JPEG, that was launched only with lossy compression. JPEG-2000 provides significantly lower distortion than traditional JPEG for the same bit-rate at the cost of more computational complexity. It has more options for compressing images, being able to focus on different goals for the compression and it works better than JPEG for high resolution images (MARCELLIN et al., 2000) (JPEG, 2000).

After many years of the creation of JPEG, in 2014, the BPG (Better Portable Graphics) standard was created by the programmer Fabrice Bellard, proposing it as an successor for the original JPEG format. It was based on the High Efficiency Video Cod-

Figure 2.2: Block diagram of the JPEG Encoder and Decoder



Source: (HASKELL et al., 1998)

ing compression standard (explained later on) and it has a high compression ratio, with files being much smaller than JPEG, but with similar quality. It also supports lossless compression and compression for animations (BELLARD, 2018). The main difference between BPG and JPEG and JPEG-2000 is that BPG doesn't have a well consolidated hardware support for it, contrary to both JPEG and JPEG-2000 standards. However, the BPG format is supported by most web browsers with a JavaScript decoder (BELLARD, 2018).

Figure 2.3 shows a visual comparison of both codecs. Both images have the same file size. We can observe that BPG has better visual quality while the JPEG shows visible degradation in quality, noted by the blocks between different adjacent colors. Also, it is noticeable that on the red shirt the quality is better, that is due to that part of the image being mostly red, therefore having low entropy and being easier to compress.

In addition to image encoders, there are also specialized tools for video compression. The main advantage of video codecs over image ones is that temporal redundancy is also explored to improve compression even more. Among the several video codecs available, we can highlight three projects/standards as being the most widely used or discussed about by industrial and research experts: HEVC, VVC, and the AV1 codec.

The HEVC was created with the purpose of having increased video resolutions and increased parallel processing architectures, improving on top of the AVC codec (SULLIVAN et al., 2012).

The VVC is the successor to the HEVC codec. It has a 50% decrease in bit-rate

Figure 2.3: Image of a Football player encoded with JPEG (left) and BPG (right) image formats.

Source: (BELLARD, 2014)

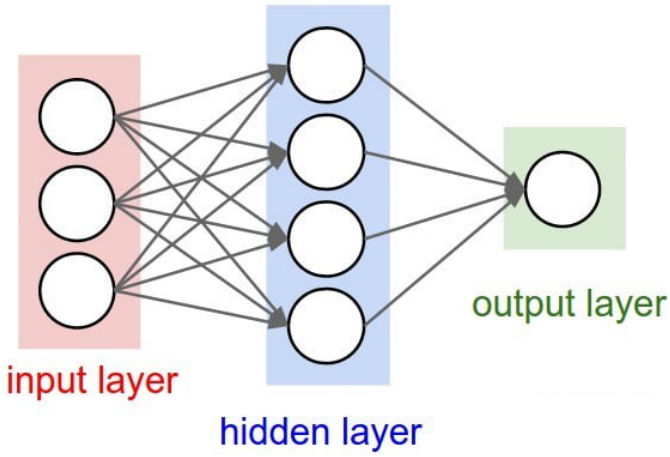
over HEVC, which in turn had 50% decrease in bit-rate over AVC (BROSS et al., 2021). VVC was designed to be capable of efficiently encode and decode videos with 4K and even higher resolutions, 360-degree videos and also for screen sharing and computer-generated content (BROSS et al., 2021).

AV1, contrary to the previous mentioned codecs, is open-source, created by the Alliance for Open Media (AOM). Their main goal was to create a royalty-free video format (GROIS; NGUYEN; MARPE, 2016) that was focused on streaming media (ROSENBERG, 2018). The AV1 has up to 50% increase in compression performance in comparison to AVC (LIU, 2018).

2.4 Neural Networks for Compression

The goal of DNNs for compression is to use NNs and machine learning techniques to replace the traditional compression tasks. In extreme cases, the entire codec is replaced by one or more DNNs, in a setup that is referred to as end-to-end compression networks. The following subsections explain some important concepts on this field.

Figure 2.4: One hidden layer between the input and output layers comprise a Neural Network.



Source: (JOHNSON, 2020)

2.4.1 Neural Networks

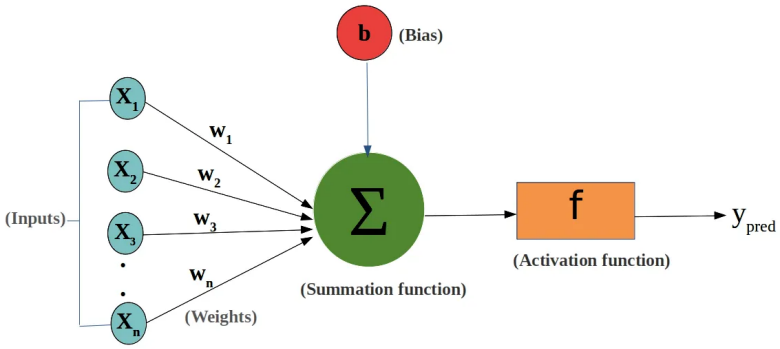
A NN is a directed graph of layers of nodes (also called neurons) that compute functions from the input layer to the output layer. Between the input and output layers, there are hidden layers (AGGARWAL, 2023). Figure 2.4 shows an example of a NN with one hidden layer.

The input layer is the set of features that are going to be computed by the NN to make predictions on the output (AGGARWAL, 2023). Each node corresponds to a feature that is going to be computed and stored on the next layer, the hidden layer. A hidden layer, and an output layer as well, has nodes that contains a variable that is the result of a computation.

The nodes are connected by edges that are parameterized with weights, and the resulting function of the NN is the accumulated result of the functions computed on each layer, considering the weights on its edges and the value on its nodes. This results in a NN that is capable of computing almost any function from the input to the output (AGGARWAL, 2023).

A NN learns by either unsupervised learning or supervised learning. On supervised learning the NN learns by creating a function based on the training data fed to the inputs and outputs and adjusting the weights to match the observed data. While on unsupervised learning there is no guidance on what the input should be, instead, it learns the data structure to identify natural patterns (GOOGLE, n.d.). The resulting value of a node is sometimes called an activation, and its value is calculated applying an activation

Figure 2.5: Representation of a node of a NN layer.



Source: (GANESH, 2020)

function to the sum of the multiplication of weights with features.

Figure 2.5 shows the representation of a NN layer node. The data is received on the input, then the weights are calculated for each value and then summed up to apply the activation function. There is also the bias value that serves to shift the activation function to the left or right, when added to the node value before the activation function.

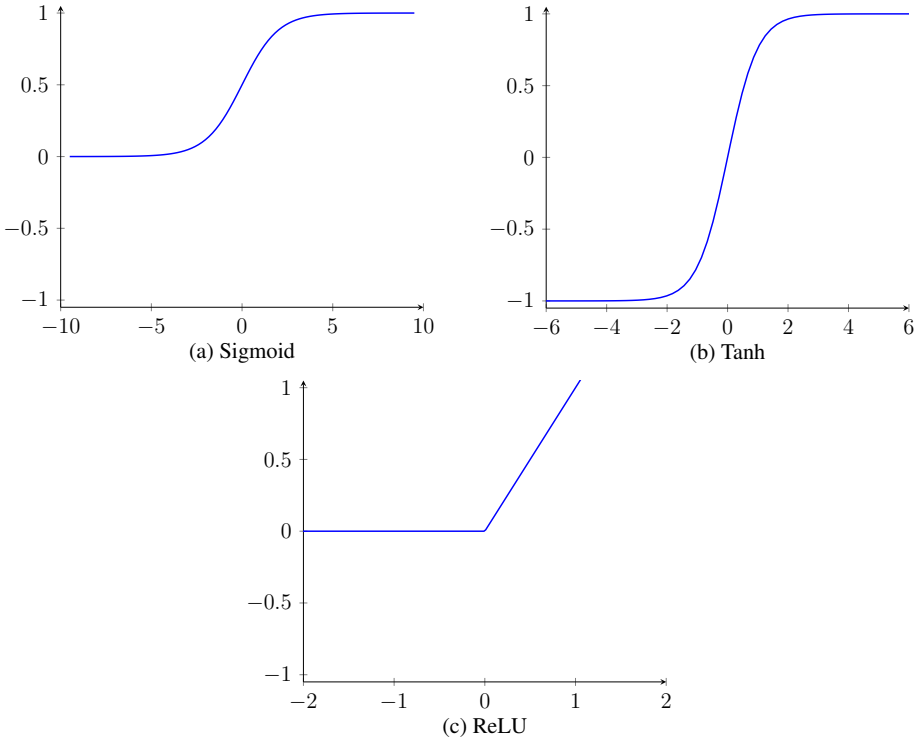
The activation function can be a linear function, like the identity function, or a nonlinear function, like sigmoid or hyperbolic tangent (tanh), for example (AGGARWAL, 2023). Figure 2.6 shows some commonly used activation functions on NNs. All of the activation functions showed are nonlinear.

The sigmoid and the tanh functions are often used on the hidden layers of NNs, with the difference between them being that when the desired output is either positive or negative, the tanh function is used. The Rectified Linear Unit (ReLU) function is similar to the sigmoid but is more simple to compute and has great results. The book from (AGGARWAL, 2023) is a great reference for further learning on activation functions and its types.

2.4.2 Deep Learning

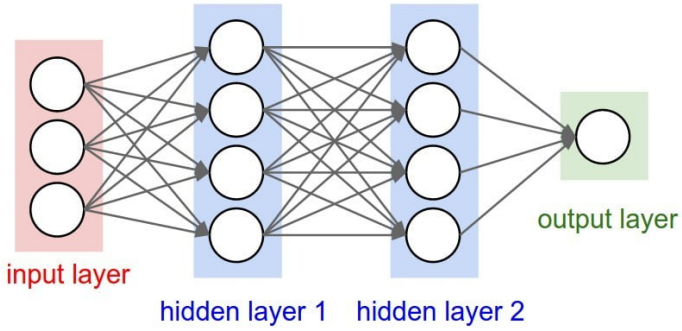
Deep learning is a type of learning that relies on NNs with multiple hidden layers, also known as DNNs. While other machine learning methods map relations from the input features directly to the output, the DNNs use the hidden layers to improve and add complexity in-between the input and output layers. Each successive layer can also be thought of as a feature extraction process, and the outputs of such layers are commonly

Figure 2.6: Commonly used Activation Functions



referred to as feature maps. Figure 2.7 shows the graph of a DNN with 2 hidden layers.

Figure 2.7: Two hidden layers, or more, between the input and output layers comprise a Deep Neural Network.



Source: (JOHNSON, 2020)

The objective of having multiple hidden layers is to make the model more powerful (more complex tasks with better performance). This happens due to the use of nonlinear activation functions, which helps modeling many complex tasks that are not linearly related with the inputs (AGGARWAL, 2023).

As you go deeper into the hidden layers you can have even more specific relations from the previous layers, due to the calculations using the activation functions and the weights on each edge, then finally you can decide the output based on it.

DNNs are widely employed due to their efficiency in solving numerous complex

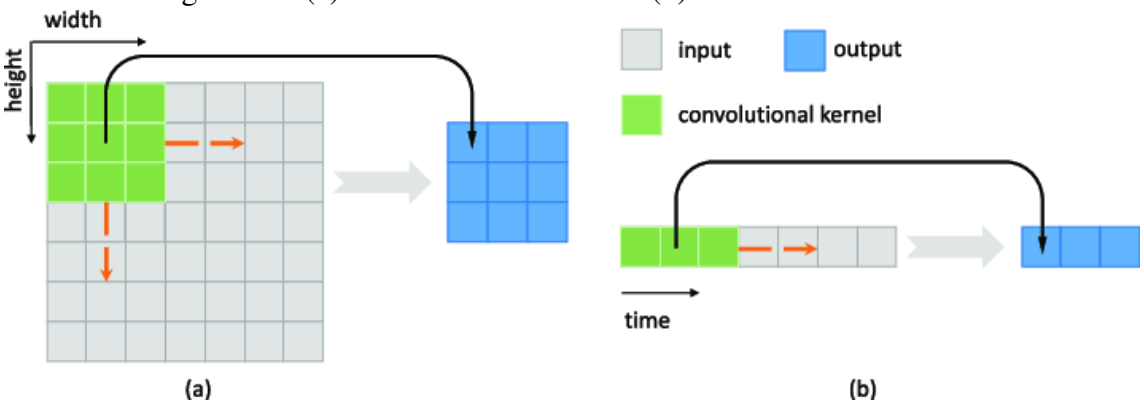
tasks, such as object detection, language translation, and even image/video compression (AGGARWAL, 2023).

One of the most efficient operation used to extract features from input or from intermediate layers is called convolution, so DNNs are commonly composed of one or more layers that employ this operation. A NN mostly composed by layers with convolutional operations is called a Convolutional Neural Network (CNN) (AGGARWAL, 2023).

2.4.3 Convolutional Neural Networks

An important type of DNN for this work is the CNN, which are composed of one or more layers that employ single-dimensional or multi-dimensional convolution. This operation, illustrated in Figure 2.8, works by using a kernel that filters the original input by applying a mathematical operation at each sample point while moving the kernel along the input until the operation has been applied to all elements of the input.

Figure 2.8: (a) 2D kernel convolution. (b) 1D kernel convolution.



Source: (WANG et al., 2019)

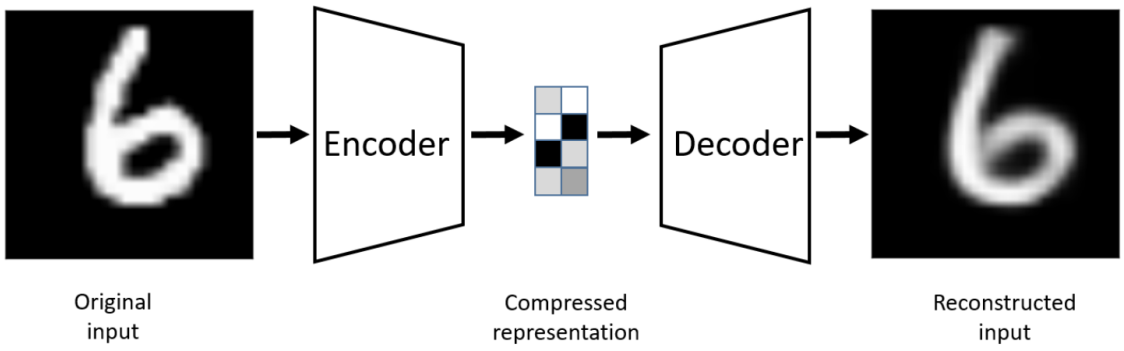
When 2D convolution is employed, CNNs become capable of handling matrices as inputs, capturing the spatial relations between the input features (AGGARWAL, 2023). This is specially interesting for images, since they are represented as 2D matrices (one for each color channel). In addition, the spatial relation that is captured by 2D convolutions is useful because it allows these networks to detect patterns in objects or particular patterns in the image. For the purpose of image compression, this property makes CNN-based AEs capable of detecting spatial relations and redundancies, making them more efficient for this task when compared to 1D CNN-based AEs.

2.4.4 Autoencoders

AEs are a type of NN that contains an encoding and a decoding part, making them appealing for compression. The goal of an AE is to encode the input data and compress it into a meaningful form, commonly referred to as latent representation or latent-space data. The latent representation is then sent to the decoding part, in which data as reconstructed back to its original form (BANK; KOENIGSTEIN; GIRYES, 2020).

Figure 2.9 shows a simplified AE diagram, with an image of the number 6 as the input. The encoder is the layers that compress the information to the latent space, and the decoder layers do the opposite, decompressing the information on the latent space to a reconstructed representation. The combination of these layers compose the NN of the AE. In this example the image is encoded to a compressed representation (latent space) and then decoded to a recreation of the original input. It is clear that the image on the output is the number 6, although not being exactly as the original, in terms of quality.

Figure 2.9: Simplified AE diagram.

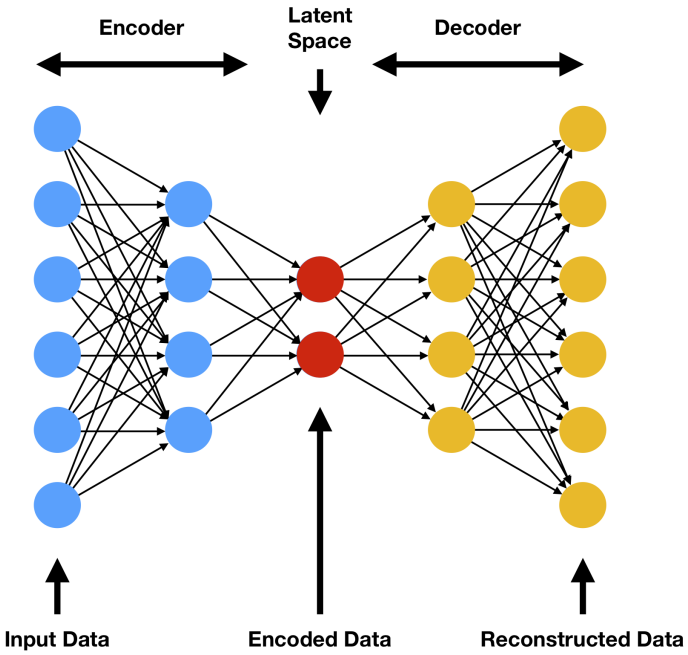


Source: (BANK; KOENIGSTEIN; GIRYES, 2020)

Figure 2.10 shows the NN inside an AE. The input image is first encoded on the encoder layers and the middle hidden layer, called latent space, is the result of the encoding, then the decoder receives the latent space data (encoded data) and reconstructs the image. It is important to note that the latent space is a compressed form of the original data, thus having a smaller layer than the original input layer. This representation is the one packed as a bitstream for transmission or storage. We also note the presence of decoder layers that are built as a reflection of the encoder ones. Although this is not mandatory, it is definitely the most common approach employed by similar works.

AEs are rising in popularity due to several promising characteristics. They are flexible in terms of complexity, meaning that it is possible to have an encoder and decoder with different computational costs. Flexibility also means that AE models can evolve

Figure 2.10: Layers of an AE.



Source: (FLORES, 2019)

together with new technology and adapt to new media formats, for example 360-degree videos or Virtual Reality (VR) content (THEIS et al., 2017), while, as mentioned before, a new codec standard could take years of development until its release.

AEs can be trained end-to-end, like the proposed by (BALLÉ; LAPARRA; SIMONCELLI, 2016), or layer by layer (BANK; KOENIGSTEIN; GIRYES, 2020). There are multiple types of AEs, and the most relevant to this work are the Convolutional Autoencoders (CAE) and Variational Autoencoders (VAE).

As mentioned on Section 2.4.3, any NN that uses convolutional operations are considered CNNs, and this also applies to AEs, being called CAEs. Since AEs are primarily focused on images, they heavily benefit from having convolutional layers in their composition and are one of the main points of compression AEs.

2.4.5 Variational Autoencoder

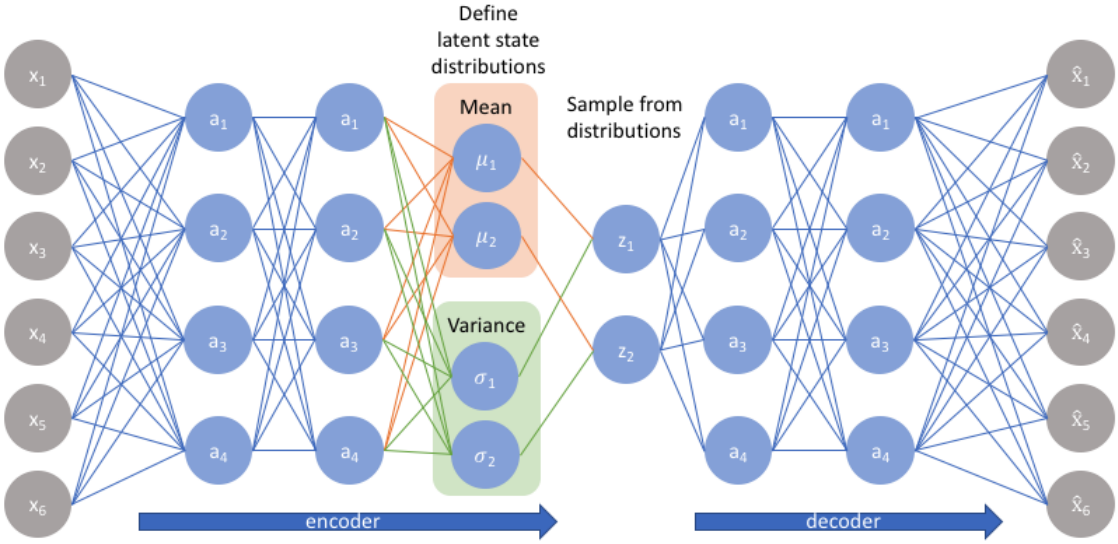
As an improvement to regular AEs, the VAE is a generative model that uses probabilistic distribution to describe data generation (BANK; KOENIGSTEIN; GIRYES, 2020). Figure 2.11 shows the diagram of a VAE, similar to the diagram of an AE on Figure 2.10.

Contrary to a standard AE, the output of the encoder, that would go into the latent

space, is a probability distribution. In this case, the latent space from a standard AE is replaced by two vectors, as seen on Figure 2.11, one representing the mean distribution and the other representing the standard deviation (or variance) distribution. From a sample of these distributions as an input of the decoder, the original input is then reconstructed (JORDAN, 2018).

However, the addition of probability distribution creates a new problem during backpropagation since it cannot be done when using the random sampling process. The solution to this problem is called the reparameterization trick. This will not be discussed in this work, but these references provide great insights on this topic (BANK; KOENIG-STEIN; GIRYES, 2020; FLORES, 2019; JORDAN, 2018).

Figure 2.11: Layers of a VAE.



Source: (JORDAN, 2018)

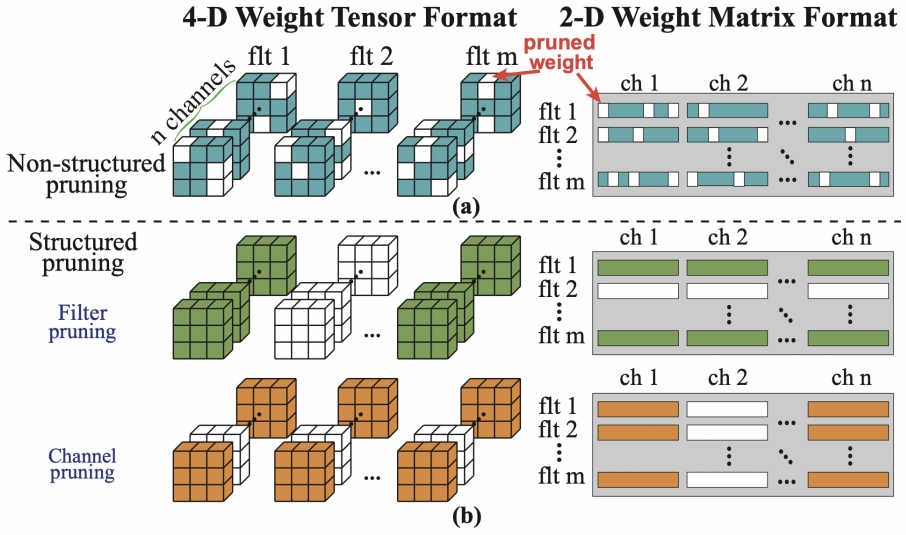
2.5 Techniques for complexity reduction of DNNs

Since our primary objective is to optimize the inference time on the decoder part of the AE, compression techniques for compressing NNs are crucial. The goal of these techniques is to simplify NNs structures, while trying to keep a balance between computational cost and performance. Most of these techniques are thoroughly explained on a paper by James O'Neill (O'NEILL, 2020).

One simple technique is NN pruning (illustrated in Figure 2.12), which consists of removing nodes or weights from the layers, that are not as beneficial to the model performance. This technique is a great starting point for lowering the AE complexity

since it is relatively straightforward. One can select the removed instances, for example, by setting a specific threshold to determine what is pruned and what is not. Some authors also advocate the use of structured pruning, which is the idea of removing regular sets of nodes/connections with the purpose of optimizing the computing parallelism.

Figure 2.12: Non-structured (a) vs structured (b) pruning.

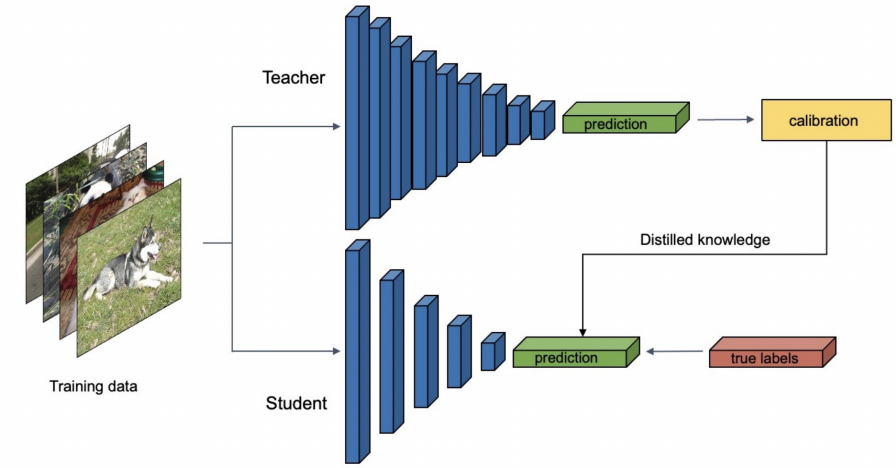


Source: (CAI et al., 2021)

Knowledge distillation (illustrated in Figure 2.13 involves learning a smaller NN (called student network) based on a larger one (called teacher network), while minimizing the entropy, distance or divergence between the probabilistic estimates (O'NEILL, 2020). Since DNN models for image compression have a substantial size, this technique could prove to be very useful on lowering their complexity.

Quantization is a technique to represent the values on a reduced number of bits, which, when training on a GPU, are typically 32-bit floating point numbers, 16-bit floating point and 16-bit integers. With quantization, these numbers are represented on a reduced type representation, going down to even to 1 bit representations in some cases (O'NEILL, 2020).

Figure 2.13: Knowledge distillation training scheme.



Source: (YANG; SONG, 2021)

3 RELATED WORKS

This chapter will review state-of-the-art research that serve as inspiration for this work. The papers (BALLÉ; LAPARRA; SIMONCELLI, 2016), (TODERICI et al., 2016) and (THEIS et al., 2017) focus on presenting new NN compression methods based on AEs, except for (DICK et al., 2021) paper, that focuses more on analyzing the complexity and quality of these types of compression.

3.1 End-to-end Optimized Image Compression

(BALLÉ; LAPARRA; SIMONCELLI, 2016) propose an end-to-end optimized compression method, consisting of a nonlinear analysis, an uniform quantizer and a nonlinear synthesis, optimizing the model for MSE. Unlike most CNNs, the authors use joint nonlinearity to introduce some form of local gain control.

The analysis and synthesis transforms, which are the encoder and decoder part of their AE, were built using Generalized Divisive Normalization joint nonlinearity (GDN), thus forming a cascade of linear convolutions and nonlinearities, followed by a uniform scalar quantization. Then the image is reconstructed using a approximate parametric inverse nonlinear transform. The GDN is a continuous, differentiable and invertible transform, inspired by neuroscience, that is used as the activation function of the convolution layer, focusing on creating a form of local gain control (BALLÉ; LAPARRA; SIMONCELLI, 2016), (BALLé; LAPARRA; SIMONCELLI, 2016). It implements a type of locally-informed normalization using the equation 3.1:

$$[h!]y_i = \frac{x_i}{(\beta_i + sum_j(\gamma_{j,i} * |x_j|^{\alpha}))^{\epsilon}} \tag{3.1}$$

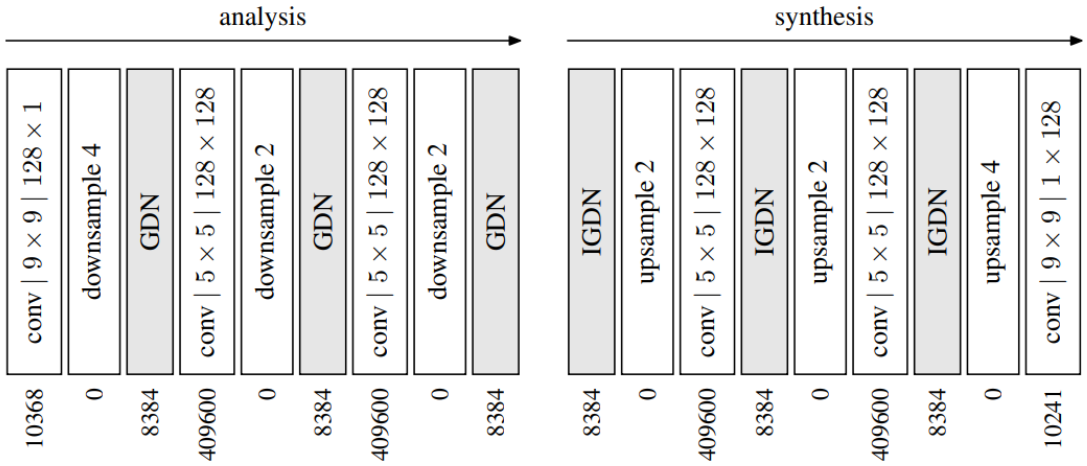
In Equation 3.1, i and j iterate over channel dimensions, and the α, β, γ, and ϵ parameters are trainable in order to customize the normalization based on the input data.

To fix the problem of non-differentiability on the quantization step during backpropagation, the authors switched the quantization for additive uniform noise. The analysis transform consists of three stages of normalization, subsampling and GDN, and the synthesis transform mirrors the analysis, consisting of three stages of convolution followed by a upsampling and an approximate inverse for GDN (IGDN).

Figure 3.1 shows the layer structures of the greyscale model proposed in his work,

with the number of parameters in each layer. The analysis arrow represents the encoder layers, while the synthesis arrow represents the decoder layers. The amount of filters here are lower than the ones on the model used for colored images, which use 192 filters, and 3 channels, while this uses 128 filters and 1 channel. The GDN/IGDN and Upsampling/Downsampling are jointly implemented on the convolutional layers (BALLÉ; LAPARRA; SIMONCELLI, 2016).

Figure 3.1: Grayscale model layers scheme. The number of parameters of each layer is given at the bottom.



Source: (BALLÉ; LAPARRA; SIMONCELLI, 2016)

The main objective of their work is to optimize the weighted sum of rate and distortion, given by $R + \lambda D$, with λ being a scalar controlling the trade-off between them. Optimization was done using a subset of images from ImageNet database and the model trained for each value of λ. For evaluation the images were compressed using uniform quantization, since the additive noise was only for training (backpropagation), using the Kodak image dataset (KODAK, 1999), and compared against JPEG and JPEG 2000. They evaluate the distortion by using the Multiscale Structural Similarity Index Measure (MS-SSIM) (WANG; SIMONCELLI; BOVIK, 2003) metric, which is a more advanced form of the Structural Similarity Index Measure (SSIM) (WANG et al., 2004). Their method shows improvements over all images tested and all bit rates. And when evaluating with Peak Signal-to-Noise Ratio (PSNR), the method exhibits better rate-distortion performance on most images, specially on lower bit rates, when compared to JPEG and JPEG 2000.

3.2 Variable Rate Image Compression With Recurrent Neural Networks

In (THEIS et al., 2017), the authors propose a solution for image compressing utilizing a NN framework that focuses on compressing low scale images. The reason being, is that traditional codecs, like the previously mentioned JPEG and JPEG 2000, are highly efficient for compressing large scale images that benefits from having high spatial redundancy and by using techniques to capitalize on that. Although, when compressing low scale images, the same properties do not apply, since they are images that likely contain difficult to compress information. This problem, allied to the fact that there is a high traffic of low scale images through search engines, video thumbnails, photos and so on, can be worked upon to vastly improve the user experience while navigating the Internet.

To achieve this task, they used LSTM with extensions to introduce spatial information to the network and non-LSTM networks using a feed-forward architecture, feeding the residuals (output) from one AE to another. Four networks were used, two non-LSTMs and two LSTMs, all following the same stages: an encoder network, followed by a quantizer and then a decoder network. The simplest one is a non-LSTM composed of stacked fully-connected layers with constant amount of 512 units in each layer, the next is similar to the previous, but utilizing LSTM on the layers for both encoder and decoder parts. The last two networks are improvements on the previous LSTM and non-LSTM networks, by utilizing convolution and deconvolution on its layers. The training used learning rates of $\{0.1, 0.3, 0.5, 0.8, 1\}$ and the Adam training algorithm.

The SSIM perceptual measure is used to evaluate the models compression performance. Utilizing scores ranging from 0 (worst) to 1 (perfect reconstruction) the best result from the traditional codecs was from the JPEG with a score of 0.80 on the best case. The worst case performance was from the convolutional and deconvolutional non-LSTM network with a score of 0.45, while the best case performance was from the convolutional and deconvolutional LSTM network with a score of 0.87, an 8.75% increase in performance from the JPEG.

3.3 Lossy Image Compression With Compressive Autoencoders

The work proposed by (THEIS et al., 2017) demonstrates an AE that optimizes the rate-distortion trade-off by making minimal changes to the rate-distortion trade-off, due

to its non-differentiable nature. This AE is composed of three components: an encoder, a decoder and a probabilistic model which is used for entropy coding.

For this end, the authors used the smooth approximation function $r(y) = y$ (identity) only in the derivative of the rounding function during backpropagation. This approach fixes the non-differentiability of the rounding-based quantization and also makes it easier to implement, since the gradients are passed without modifications from the decoder to the encoder. Since the probabilistic model is also non-differentiable, they used a continuous, differentiable approximation to solve this. For the encoder and decoder, they used common CNNs, while the decoder mirrors the encoder, it uses sub-pixel convolutions to do the upsampling. In order to have variable bit rates, they used a method called scale parameters, which is better explained on their paper.

All models were trained using the Adam algorithm applied to batches of 32 images with a resolution of 128 by 128 pixels. The training has two steps, first the model is trained for a fixed rate-distortion trade-off with the learning rate starting from 10^{-4} until 10^{-5}. Then, they introduce the scale parameters for other values of the trade-off while keeping all other parameters fixed, with a learning rate starting from 10^{-3} and continuously decreased by a factor of $1000^{0.8}/(1000 + t)^{0.8}$ with t being the number of updates.

For evaluation, they used multiple perception metrics: PSNR, SSIM and MS-SSIM. The comparison was made between JPEG, JPEG 2000 and the implementation of (TODERICI et al., 2017) compression method. For PSNR, their method performs similar to JPEG 2000, although slightly worse at low and medium bit rates and slightly better at higher bit rates. For SSIM, their method outperforms all of the others. For MS-SSIM, their methods performs similar to the other methods except for very low bit rates. Also, they found the results to be very image dependant.

3.4 Quality and Complexity Assessment of Learning-Based Image Compression Solutions

Contrary to the previously mentioned works, the work of (DICK et al., 2021) presents an analysis on learning-based compression methods, comparing eight models present on the Compression package (TENSORFLOW, 2022) with JPEG 2000 and BPG in regards to quality and processing time. While BPG wasn't evaluated on the previously mentioned works of this section, it still plays a major role in compression, since it is one of the biggest differences from traditional codecs.

The models from TensorFlow are all reproductions of models from some of the authors mentioned on this section, with the only difference being the use of integer arithmetic operations related to conditional priors that provide consistent performance over different hardwares. The models are based on the CNN architectures of nonlinear transform coders, VAs and Generative Adversarial Networks (GAN), and all of them have slight changes for better evaluation that is explained in more details in (DICK et al., 2021). The training was made using the MSE distortion metric, and the evaluation was done using the Kodak data set using the PSNR, MS-SSIM and Learned Perceptual Image Patch Similarity (LPIPS) (ZHANG et al., 2018) perception metrics.

Their work makes a thorough comparison between the models and codecs visual quality, but the most interesting part is the complexity comparison between the models and codecs, that compares the processing time for compressing and decompressing on CPU and GPU. Some interesting results were obtained for the models. For example, the VAs model and the GAN, while obtaining higher quality results, also showed significant increase in complexity as is expected, with the GAN model having the slowest decompression time overall, reaching at impressive 4.53 seconds to decompress on CPU, while JPEG 2000 took only 0.01 seconds and the slowest learning-based model besides the GAN model reached 1.01 seconds. It is also observed that there is a low increment in complexity for nonlinear transform models, but at the same time having better quality results than JPEG 2000. There is also a direct relation to the model size and processing time. On average, the learning-based compression methods take 49.1% more time to decompress on a CPU and 16.12% on a GPU, when compared to traditional transform-based compression. This brings great attention to the purpose of this work, that intends to improve on the decompression time. GPUs also were shown to have savings in processing time, since there has been a growth in machine learning specs in these components, but are not a viable solution for resource-constrained devices.

3.5 Chapter Summary

Table 3.1 shows a comparison of the related references presented in this chapter, as well as the goals set for this work based on this analysis. Note that Bits Per Pixel (BPP) range and PSNR ranges are not the exact value, since they were obtained from graphics on the respective works.

We will focus on using (BALLÉ; LAPARRA; SIMONCELLI, 2016) work as ref-

erence, while also having a BPP range a little bigger, and evaluation the quality with LPIPS together with PSNR and MS-SSIM. The range values of BPP and PSNR defined on the table are approximations taken by looking at the graphs of the experimental results on each work, since most of those works did not define a range.

Table 3.1: Summary of related references and a comparison with our proposal.

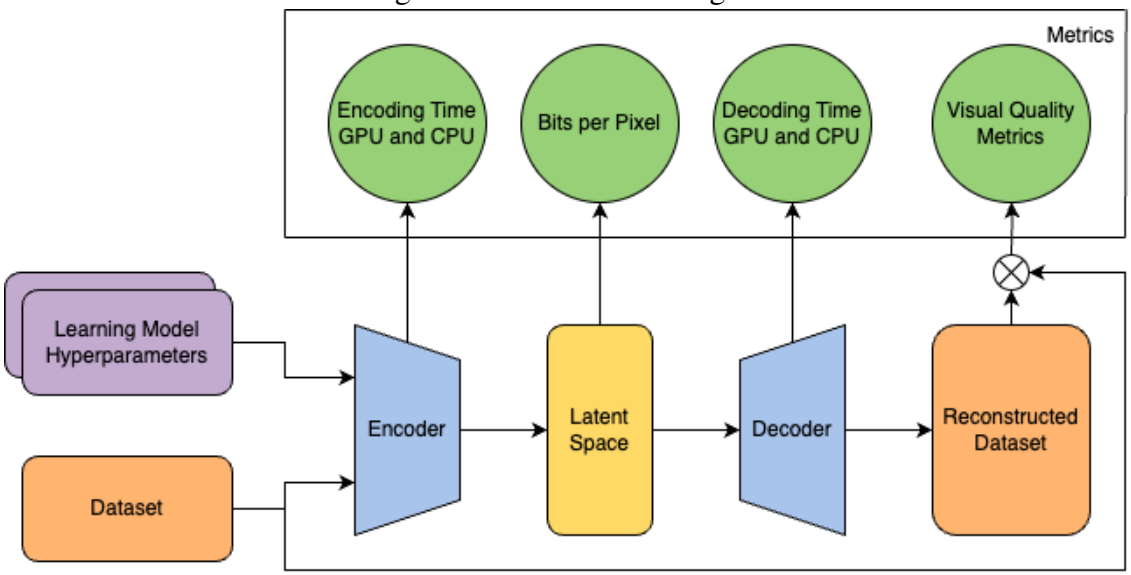
Work	*Image-scale focus*	*Evaluation metric*	*Architecture*	*Quality analysis*	*Complexity analysis*	*BPP range*	*PSNR range (dB)*
(BALLÉ; LAPARRA; SIMONCELLI, 2016)	General	PSNR and MS-SSIM	CNN	Yes	No	0.025 - 0.45	20 - 39
(TODERICI et al., 2016)	Thumbnail	SSIM	LSTM and non-LSTM	Yes	No	0.62 - 1.37	N.A.
(THEIS et al., 2017)	General	PSNR, SSIM and MS-SSIM	CNN	Yes	No	0.4 - 2.4	29 - 41
Proposed work	General	PSNR, MS-SSIM and LPIPS	Low-Complexity CNN (Inspired by (BALLÉ; LAPARRA; SIMONCELLI, 2016))	Yes	Yes	0.15 - 1.2	20 - 39

Source: The Author

4 PROPOSAL AND METHODOLOGY

This chapter discusses the proposal and methodology for achieving our main objective, which is lowering the computational cost of image compression NNs while maintaining image quality, in this case, using the model proposed by (BALLÉ; LAPARRA; SIMONCELLI, 2016) as the base model. Figure 4.1 shows the methodology employed in this work.

Figure 4.1: Evaluation Diagram



Source: The Author

As mentioned in Section 3.4, (DICK et al., 2021) work showed that there is a lot of room for research on improving the decompression time of learning-based models.

The proposal for this work is to evaluate the feasibility of a simplified learning-based model, in terms of complexity, that has a low decompression time while still keeping a decent amount of visual quality. The chosen coding platform was Google Colab as it is a great place to code machine learning models with the TensorFlow API and its tools.

As mentioned in Section 1, our goal is to utilize the model proposed by (BALLÉ; LAPARRA; SIMONCELLI, 2016) and build a simplified version (or versions) of it. The model will be mainly optimized on the decoder end as to increase the possibility of using the compression method on general-use or low-end computers. This is possible since learning-based compression methods have the added flexibility of being able to implement encoders and decoders separately. Our focus is on the decoder end, because the encoder can be computed in another machine that has high computational power that can bear with the higher complexity, while the decoder end will be the end-user, a person using a

smartphone or a personal computer for example.

4.1 Dataset

To train the models, it was initially planned to use one of the following data sets: CIFAR100 (KRIZHEVSKY, 2009) and COCO or its reduced version, COCOmini (LIN et al., 2015)(SAMET; HICSONMEZ; AKBAS, 2020). The CIFAR-100 (KRIZHEVSKY, 2009) is a small image dataset composed of millions of small 32×32 images. The original COCO dataset (LIN et al., 2015) is a large-scale object detection, segmentation, and captioning dataset, and has 330 thousand images, while the COCOmini dataset (SAMET; HICSONMEZ; AKBAS, 2020) is a subset of images from the original COCO dataset, having 25 thousand images that are randomly sampled from COCO, while also preserving the overall ratio of objects in the images.

We ended up choosing the COCOmini, with images with resolution larger or equal to 256 by 256 pixels, as our training dataset. It has a good amount of images, and will guarantee better reproducibility during training, since it would be difficult to train the entire dataset if it was bigger. Also, since the model has an input of 128x128, the CIFAR-100 dataset would not be adequate for training due to having images smaller than the input.

As for evaluation, we used the (KODAK, 1999) dataset that is an uncompressed set of images commonly used to evaluate image compression methods and is also used on (BALLÉ; LAPARRA; SIMONCELLI, 2016) work. Since the model is fully convolutional, we can train the model with lower resolution images (128×128) and, after training, test the model with the Kodak dataset with the 768×512 images from the Kodak dataset.

Table 4.1 shows the amount of images per dataset which were used for training, validation and evaluation of the models. Only the filtered version of the COCOmini dataset (256×x256 images and larger only) was utilized for the final results.

4.2 Baseline Model

Figure 4.2 shows the convolution kernel size, channels size layers structure of the baseline model, from (BALLÉ; LAPARRA; SIMONCELLI, 2016), used in this work. In this work, the original model presented in Chapter 3 was slightly modified to accept

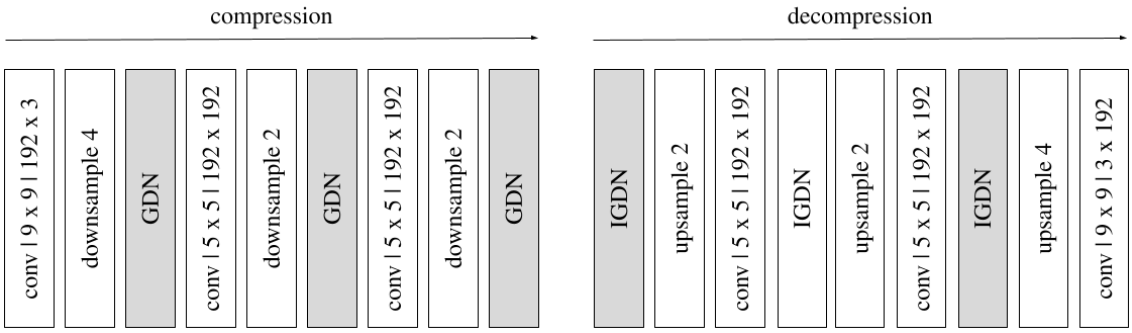
Table 4.1: Datasets utilized in this work

Dataset	Training Images	Validation Images	Evaluation Images
COCOmini	25000	5000	-
COCOmini (256x256 and larger only)	24831	4958	-
Kodak	-	-	24

Source: The Author

RGB-color images (3 input channels).

Figure 4.2: Parameters used in the compression and decompression layer.



Source: Based on the diagram from (BALLÉ; LAPARRA; SIMONCELLI, 2016)

In Figure 4.2, *conv* stands for convolution layers, and the subsequent values represent the kernel size (9×9 in the first layer), the number of output filters (192×3). The downsample *N* layers represent an $N : 1$ downsampling operation (the same goes for upsampling). The GDN/IDGN and the Upsampling/Downsampling operations are jointly implemented in the Convolutional layer, which is called SignalConv2D in the Tensorflow Compression library (TENSORFLOW, 2022).

4.3 Optimizations

To optimize the base model, the techniques mentioned on Section 2.5 were considered, but the only one that was successfully employed in this work was pruning.

The only technique that was not tested, was knowledge distillation, because that it would be quite complex to do it using the base model, since it is a custom model, with custom layers and loss function, that does not work well with the rest of the TensorFlow API without some adjustments, which would take more work. This is also the reason why the weight pruning and quantization techniques did not work so well with the base model.

The quantization and weight pruning techniques were tested using the TensorFlow Lite API, which has support for these techniques, but again, since the base model is very customized, the optimizations did not work due to incompatibilities between the implementation of the model and the API.

As mentioned in (BALLÉ; LAPARRA; SIMONCELLI, 2016), training with a perceptual metric instead of MSE could provide better results in terms of quality, leaving space for more optimizations on the decoder. With this in mind we tested utilizing the LPIPS[1] metric instead of MSE, but due to its implementation being made for older versions of TensorFlow 1 (we used TensorFlow 2), it was incompatible with our code. Even with a non-official conversion of the LPIPS metric for TensorFlow 2[2], it still didn't work with the base model, so we scrapped that idea and continued using MSE.

For optimizing the base model through pruning, we tried changing only the model with $\lambda = 0.001$, with the lowest BPP, and then reflecting the best changes to the other models, since it takes roughly 5 hours to train each model for 500 epochs. The following list shows the changes to the decoder side from the base model, together with the labels used to represent their respective optimized models on the figures:

- 2L: Removing the intermediate convolutional layer.
- K3x3: Lowering the kernel size from the first two convolutional layers from 5 by 5 to 3 by 3.
- 2L, K3x3: Removing one convolutional layer and decreasing the kernel size from the first convolutional layer from 5x5 to 3x3.
- 2L, K9x9: Removing one convolutional layer and increasing the kernel size from the first convolutional layer from 5x5 to 9x9.

Table 4.2 details the amount of parameters of each proposed network, as well as the one proposed by (BALLÉ; LAPARRA; SIMONCELLI, 2016). It is noticeable that the *2L, K9x9* model is actually more complex than all the other models, but even so, the decompression times are still lower when using GPU, but slower when using CPU, as will be seen on Chapter 5. That is due to the GPU taking advantage of the bigger kernel size, while the CPU can not.

[1] Available at https://github.com/alexlee-gk/lpips-tensorflow
[2] Available at https://github.com/moono/lpips-tf2.x

Table 4.2: Number of parameters of each model, including the model size in Mb.

Model Name	*Encoder Parameters*	*Decoder Parameters*	*Total Parameters*	*Model Size (Mb)*
Ballé, 2016 (3L, K5x5)	2338176	2338179	4676355	17.84
2L	2338176	1195011	3533187	13.48
K3x3	2338176	1011075	3349251	12.78
2L, K3x3	2338176	531459	2869635	10.95
2L, K9x9	2338176	3406851	5745027	21.92

Source: The Author

4.4 Evaluation

For the evaluation, we used the work of (DICK et al., 2021) as an inspiration for comparing the processing time between the learning-based models, and also compare the quality of the compression using the quality metrics MSE, PSNR, MS-SSIM and LPIPS. Together with the amount of BPP on the compressed image to measure the compression rate. With BPP, we can calculate the approximate amount of bits on an image, without considering any file overhead. Figure 4.1 shows a diagram with the evaluation flow very similar to the one used on their work.

The evaluation flow will start by feeding the encoder network with the models parameters and the (KODAK, 1999) image dataset to be compressed, then it will follow the compression steps, but at each step we will save the respective metrics needed for the evaluation, them being: Encoding and Decoding time for the Encoder and Decoder, respectively, Bits per Pixel (BPP) on the latent space and lastly, the reconstructed dataset will be compared to the original dataset using the visual quality metrics PSNR, MS-SSIM and LPIPS to measure the effectiveness of the model(s).

4.5 Metrics

The chosen metrics for visual quality were PSNR, MS-SSIM and LPIPS and also we will be measuring the processing time of the model running on GPU and CPU to evaluate the computational cost, as well as the number of parameters.

The visual quality metrics PSNR and MS-SSIM were chosen due to being present on both of the main works of inspiration (BALLÉ; LAPARRA; SIMONCELLI, 2016) and (DICK et al., 2021), so we have a good basis of comparison. And LPIPS is a good

metric since it is specifically targeted to learning-based methods and its scores have a good correlation with human perception (ZHANG et al., 2018).

5 RESULTS AND DISCUSSION

The following sections will present the specifications of the computer used to train the model, and also all of the main parameters set to train and evaluate the model and its optimizations, while discussing the reason for choosing these parameters and optimizations. After presenting the implementation, the performance and quality results obtained from both the base model and the optimized models will be evaluated against each other, as well as comparing them to Kakadu[1] implementation of the JPEG 2000 compression algorithm with similar BPP for the images.

5.1 Environment and Hardware Setup

As mentioned before, the AE model from (BALLÉ; LAPARRA; SIMONCELLI, 2016) was used as a base model on which to improve performance time. All of the code used was run on a Google Colab notebook, while using a local environment for code execution. The code for (BALLÉ; LAPARRA; SIMONCELLI, 2016) model was imported from the TensorFlow Compression GitHub page (BALLé, 2022), with some minor adjustments to fit our base code. A local environment was used instead of regular Google Colab hosted execution environment, since it provided faster training and execution times due to the dedicated GPU.

The hardware setup of the local environment is the following:

- CPU: Intel I5 11400F with 6 cores and 12 threads, and max core clock of 4.40GHz.
- GPU: NVIDIA GeForce RTX 4070 with 12 Gb of Video RAM (VRAM).
- RAM: 32 Gb.

To be able to take advantage of the CUDA cores capabilities from the GPU, the Jupyter Notebook used as local environment on Google Colab was running on WSL2 with Ubuntu 22.04.3 LTS distro, on Windows 10, with Python 3.10 and TensorFlow 2.14.0 library, the CUDA SDK 12.3.0 and the NVIDIA driver GeForce Game Ready 546.65.

[1] Available at https://kakadusoftware.com/

5.2 Setup of the Training Process

For training the AE model, it was used the COCOminitrain (SAMET; HICSONMEZ; AKBAS, 2020) dataset containing 25000 images for training and 5k images for validation. To optimize the dataset training speed, only crops of the original images were utilized, and images that were smaller than 256 by 256 were discarded to maintain a reasonable level of image quality. Table 5.1 shows the chosen model parameters for training, and the next paragraphs explains them.

Table 5.1 summarizes all the parameters mentioned above.

Table 5.1: Parameters used for model training

Parameter	*Value*
Input size	128x128
Steps per epoch	1000
Batch size	8
Train images/epoch	8000
Epochs	100
Lambda	0.001, 0.004, 0.01, 0.03
Filter size	192
Color mode	RGB
Learning rate	10^{-4}

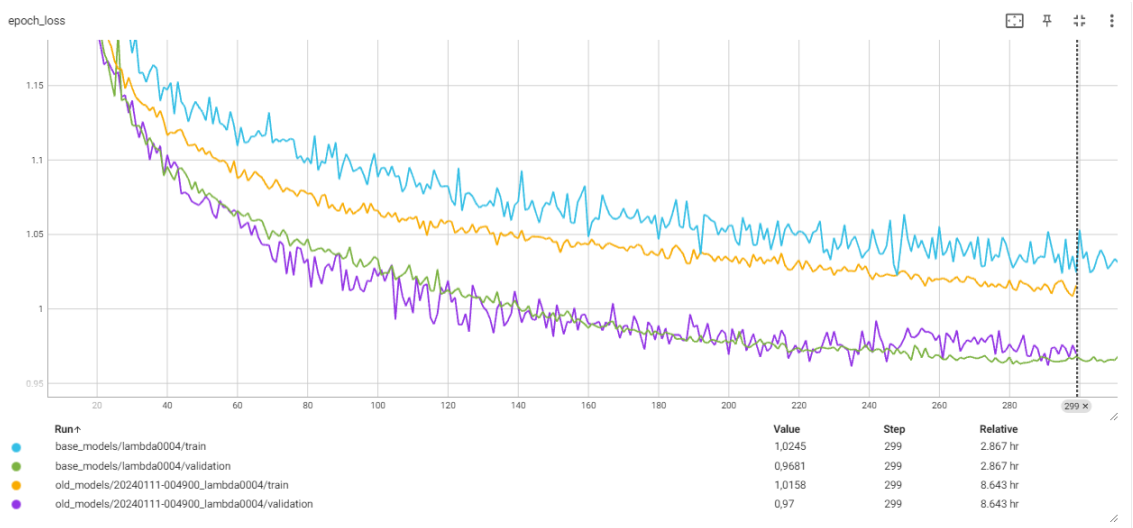
The input size is the dimension of the images used in the model during training. During training, only images with this exact dimension can be used. For inference, any image dimension can be used, since there is no defined size for the input, and the model is fully convolutional. We maintained the initial training input size of 128 by 128, same as Ballé's (TENSORFLOW, 2022).

The COCOmini dataset was used through a TensorFlow API dataset object, to load images during training. Previously we were utilizing all of the images in the training, during an epoch, but later, tried the approach of using only 1000 steps per epoch, which meant that for each step of the epoch, we loaded eight samples from the batch. This means that instead of loading all the images in a single epoch, around 25000, we used only 8000 images per epoch (1000x8 = 8000). This choice of steps per epoch made the training almost three times faster. It also lowered the "smoothness" of the loss function during training, but the contrary happened for the validation, having even better loss on

validation as exemplified in Figures 5.1 and 5.2

The graph on Figure 5.1 shows the difference between using all images or one third of the images from the training dataset in one epoch. The blue and green lines are from the model using the partial dataset per epoch, and the yellow and purple are using the entire dataset per epoch. Using only a portion of the dataset per epoch is clearly better for the validation loss and even for the training time, being almost three times lower, which allows training the model for more epochs in less time.

Figure 5.1: Loss during validation and training for each epoch, using $\lambda = 0.004$. The graph shows the difference in loss on training when using the entire dataset per epoch, versus using only a third of the dataset per epoch.



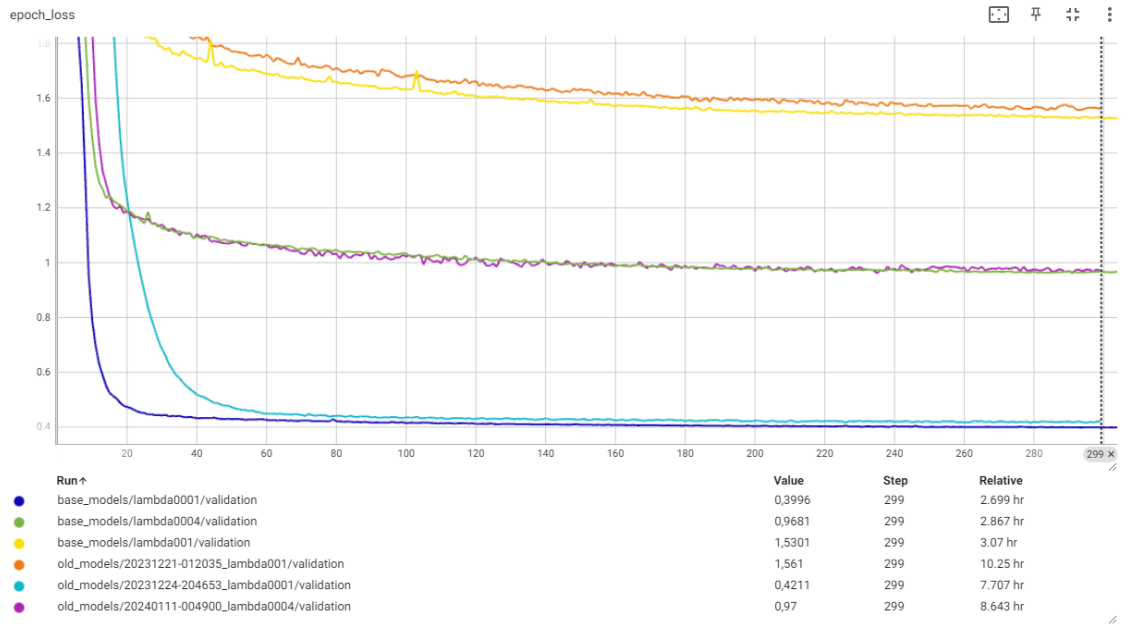
Source: The Author

The graph on Figure 5.2 shows the difference between using all images or one third of the images from the training dataset in one epoch. The dark blue, yellow and green lines are from the model using one third of the images per epoch, and the orange, cyan and purple are using the entire dataset per epoch. Using only a portion of the dataset per epoch is clearly better for the validation loss and even for the training time, being roughly three times lower, which allows training the model for more epochs in less time.

Initially the models would be trained for 1000 epochs, as is defined in the base code (TENSORFLOW, 2022), but the models were trained for 500 epochs, since the loss function for 500 epochs and 1000 epochs were similar, as is seen on Figure 5.3, and took half the time to train the models, facilitating the experiments.

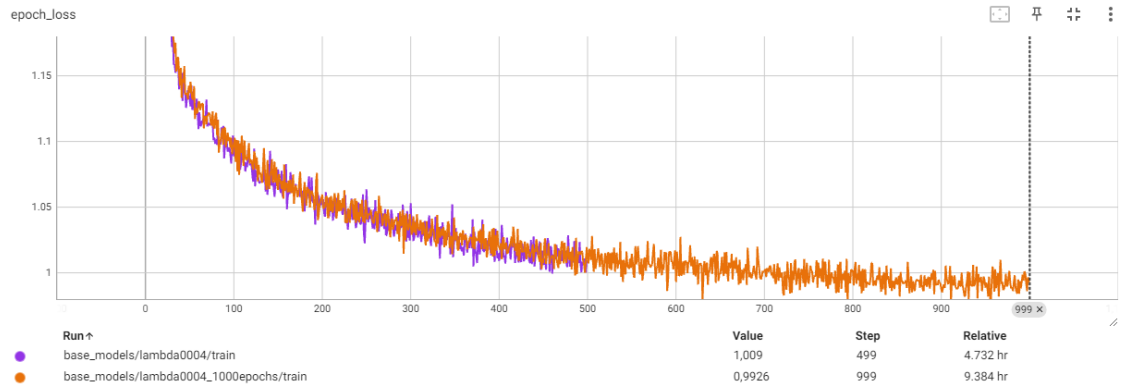
The batch size is the number of samples (images) that are processed before updating the model's parameters. The batch size for training is eight, same as Ballé's. This value was already good enough and wouldn't bring much improvements to the training

Figure 5.2: Loss during validation with default, and different steps per epoch, for different λ.



Source: The Author

Figure 5.3: The difference in loss between the model trained for 1000 epochs (in orange) and the model trained for 500 epochs (in purple) is approximately 0.0164, while the training time for 1000 epochs is almost the double than for 500 epochs.



Source: The Author

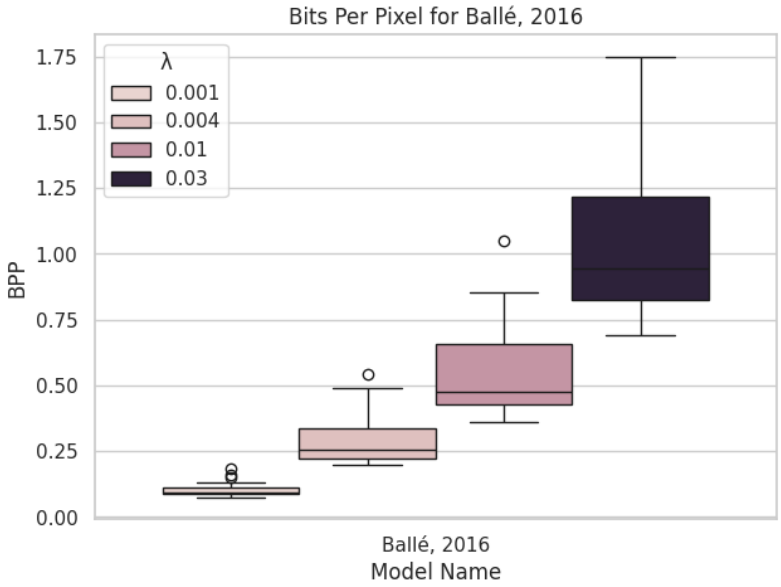
with other values.

The loss function used to train our models, presented in Eq. 5.1 is the rate-distortion trade-off function presented earlier, with the Rate being the amount of Bits per Pixel (BPP) and the Distortion being Mean Squared Error (MSE).

$$Loss = BPP + \lambda * MSE \tag{5.1}$$

In (5.1), the λ parameter is a scalar controlling the trade-off between rate (BPP) and Distortion (MSE). It is a fixed value, set when creating the model, and does not

Figure 5.4: BPP values of the base model, for all values of λ



Source: The Author

change. The BPP calculation is better defined at (BALLÉ; LAPARRA; SIMONCELLI, 2016).

Since the model needs to be trained to a specific value of λ, it is not possible to set a fixed value for the BPP to define a desired compression rate. Instead, four values of λ were chosen to approximate different compression qualities, and also based on the BPP values presented on (BALLÉ; LAPARRA; SIMONCELLI, 2016), together with higher BPP values.

The chosen λs are: 0.001, 0.004, 0.01, 0.03. These correspond to the desired target BPP (Bits Per Pixel) approximated values of 0.15, 0.35, 0.7 and 1.2 respectively. The λ values were obtained by trial and error, by training the model until the BPP validation value was relatively close to the target values defined. It is important to notice that the BPP of the compressed images is not the same for every image, so that is why these values are only an approximation.

In Figure 5.4, it is clear that the BPP values of the model are not constant, and that the variance in BPP is even more noticeable at higher λ values, i.e., lower compression ratio. On the other side, JPEG 2000 is able to maintain a fixed rate for the images, since the rate (BPP) can be explicitly set when compressing the images.

The amount of filters on the convolution layers are 192. This value is the default value on the base model code (TENSORFLOW, 2022).

Tests were made with models using models with 256 filters and inputs of 256 by 256 images. While the quality of the images for these models were better, their inference

time were not ideal in comparison with the base model, so we did not include them in the experiments results.

We opted to train the model only with colored images, i.e., three channels in the RGB color model, to have more meaningful results and for a better comparison with (BALLÉ; LAPARRA; SIMONCELLI, 2016) results, since that is what was used in his work.

For the learning rate, the Adam training algorithm was used, with a learning rate of 10^{-4}, same as on the base model (TENSORFLOW, 2022). This value was constant throughout all the models, since it would not improve the inference times.

5.3 Quality and Complexity Analysis

To evaluate the quality of the trained models, all the images of the Kodak dataset were compressed and uncompressed, and the values of BPP, MSE, MS-SSIM, PSNR (dB) and LPIPS were gathered. The same metrics were gathered for JPEG 2000 as well. The results were then saved on a Comma-Separated Values (CSV) file to facilitate plotting and data summarization using Python libraries like Seaborn and Pandas.

To evaluate the processing time, all images from the Kodak dataset were compressed and decompressed, utilizing either the GPU or the CPU. It is important to mention that the decompression and compression does not include saving or loading the image from the disk, but it includes the entropy coding and decoding, so the performance calculation is not only the inference time of the network.

To gather meaningful timings, for each image, we did 10 warm-up runs, to discard the possible overhead operations that happen at the first time the GPU is used, then, compressed or decompressed the same image 100 times, and calculated the mean time of these 100 runs as a result. Although the warm-up runs were mostly due to the GPU, we also did them on CPU to guarantee comparable results.

The inference time of the decompression on the model was calculated using the python *process_time()* method from the default python *time* library. We tried searching for better methods to calculate the inference time within the TensorFlow API, but it seems that there is not an implementation for that at the time of writing.

5.3.1 Complexity Analysis

Table 5.2 shows a table with compression and decompression times for both GPU and CPU using the base model, that will be called *Ballé, 2016* in the following figures. The processing time between CPU and GPU, on the base model, shows a time difference of two decimal places, for both compression and decompression, which is already an incredible difference. This is due to the GPU being specially designed to do parallel processing, much better than the CPU, when working with DNNs (GYAWALI, 2023).

Table 5.2: Mean compression and decompression time, for both GPU and CPU, of Ballé's base model for all λ values, and JPEG 2000, on all Kodak images. It also includes the mean BPP values for each λ value of the base model, and the fixed BPP for JPEG 2000.

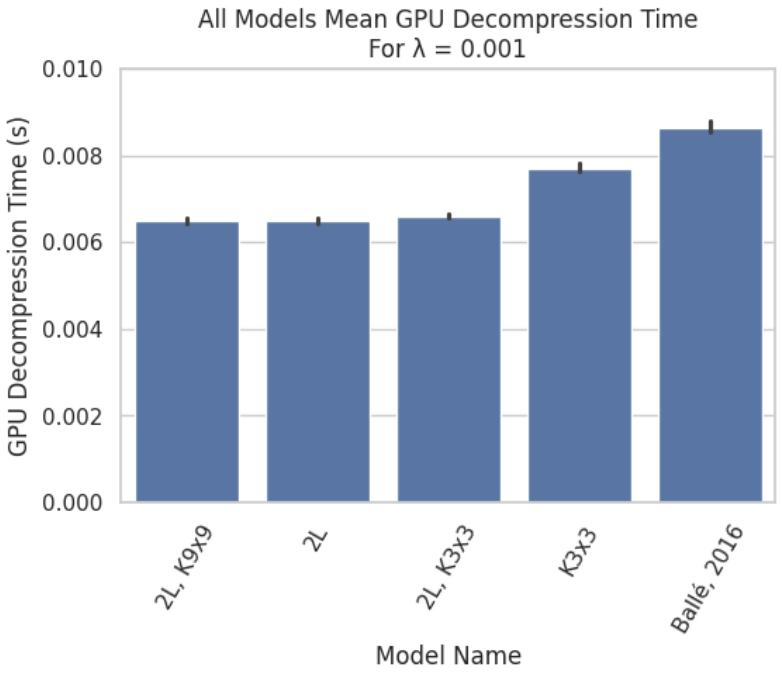
Model Name	*Platform*	*BPP*	λ	*Mean Compression Time (s)*	*Mean Decompression Time (s)*
Ballé, 2016	CPU	0.104	0.001	0.611	0.661
		0.292	0.004	0.594	0.635
		0.555	0.010	0.586	0.629
		1.031	0.030	0.584	0.627
			Average	**0.594**	**0.638**
	GPU	0.104	0.001	0.008	0.009
		0.292	0.004	0.008	0.009
		0.555	0.010	0.008	0.009
		1.031	0.030	0.008	0.010
			Average	**0.008**	**0.009**
JPEG2K	CPU	0.150	-	0.005	0.005
		0.350	-	0.004	0.004
		0.700	-	0.004	0.004
		1.200	-	0.004	0.004
			Average	**0.005**	**0.005**

Source: The Author

Comparing the compression and decompression times with JPEG 2000, in Table 5.2, we see that the GPU processing time of the base model is roughly twice as slow than JPEG 2000, at similar BPP values, but is already a small enough difference to compete with JPEG 2000, although the same can not be said about the CPU processing time of the base model.

Figures 5.5 and 5.6 show a processing time comparison between the optimized

Figure 5.5: Mean GPU Decompression time for all optimized models and the base model, for $\lambda = 0.001$.



Source: The Author

models and the base models, for $\lambda = 0.001$, using the GPU. The models with two layers show significant improvement over the base model, while the model with three layers, but with a smaller kernel of 3 by 3, shows only a slight improvement.

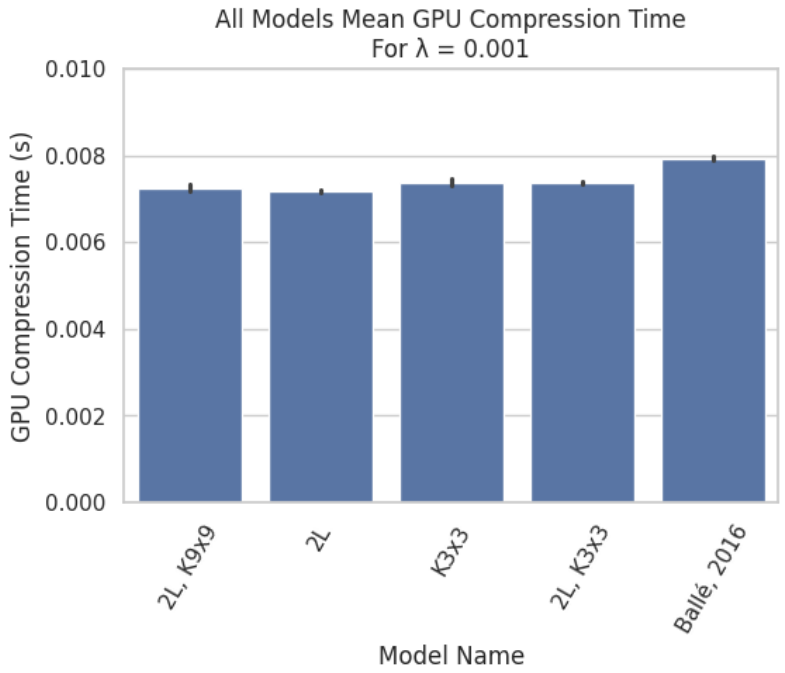
In Figure 5.5, the models with two layers have approximately 2ms less time for decompression, a 25% improvement over the base model. This shows that the number of layers on the model has significantly more impact to the performance than the changes on the kernel size.

The GPU average compression time was similar throughout all models, as seen on Figure 5.6, since the encoder layers of the base model were not changed. Although, these times did lower a little, on the optimized models, even though the encoder layers were the same. This is probably due to the AE model being trained end-to-end, thus, changes on the decoder also changes the model as a whole.

5.3.2 Quality Analysis

After analyzing the performance of the optimized models, the quality analysis is needed to evaluate which models maintain quality, since we do not want to sacrifice quality to gain in performance. The Kodak test images were used in these experiments as

Figure 5.6: Mean GPU Compression time for all optimized models and the base model, for $\lambda = 0.001$.



Source: The Author

well.

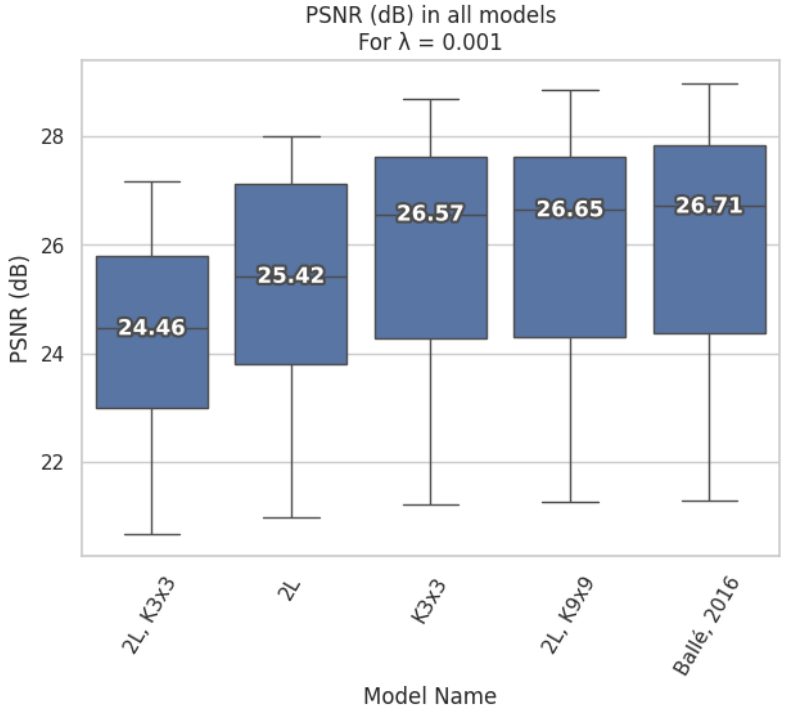
Figure 5.7 shows the PSNR values in dB, for every model with $\lambda = 0.001$. From the optimized models, the models K3x3 and 2L,K9x9 are the closest to the original PSNR quality from the base model, while the 2L, K3x3 and 2L fails to reach that same level of quality. Although, when comparing the MS-SSIM metric, the same is not true, as the Figure 5.7 shows that all the models seem to have maintained similar levels of quality, although *2L, K9x9* has slightly less variation, as seen in the box plot superior and inferior limit. Still, in MS-SSIM, even though by a small difference, the 2L, K3x3 model is still the worst of them all in terms of quality.

Figure 5.9 shows the LPIPS metric results for all models. Here, the LPIPS metric did not provide much useful information, since all the models obtained similar results, making it hard to decide which model performed better.

Table 5.3 shows the average PSNR (db) values obtained for each learning-based model, as well as JPEG 2000. The values inside parentheses represent the difference between the average PSNR obtained with the *Ballé, 2016* base model and each of the other models presented. The λ values for JPEG represent the same BPP range as the models.

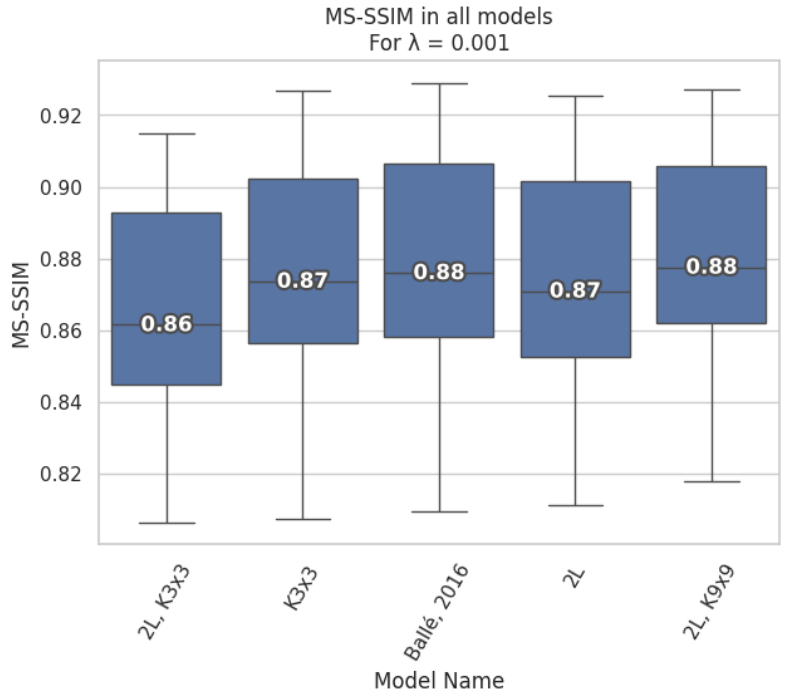
We can observe that the *2L, K9x9* model has a small PSNR loss of up to 0.07 dB

Figure 5.7: PSNR values for all models, with $\lambda = 0.001$, in dB. Higher values mean better quality. Ordered from lowest to highest.



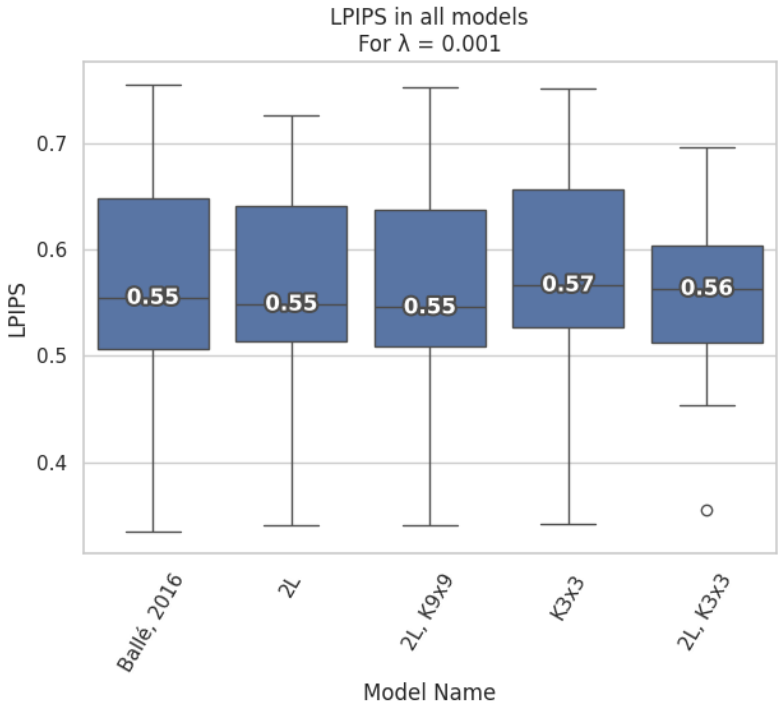
Source: The Author

Figure 5.8: MS-SSIM values for all models, with $\lambda = 0.001$. Values range from 0 to 1. Higher values mean better quality. Ordered from lowest to highest.



Source: The Author

Figure 5.9: LPIPS values for all models, with $\lambda = 0.001$. Values range from 0 to 1. Higher values mean better quality. Ordered from lowest to highest.



Source: The Author

($\lambda = 0.03$) for the tested lambda values. In some cases, PSNR gains are observed (up to 0.2 dB for $\lambda = 0.01$). The *K3x3* presented event higher gains of 0.29 dB for $\lambda = 0.01$. We can also observe that JPEG2K is considerably less efficient.

The models *2L, K9x9* and *K3x3* obtained the best quality results compared to the other optimizations. *2L, K9x9* had the best performance time from all the models, and although the *K3x3* did not have the same performance level as *2L, K9x9*, it still did better than the base model.

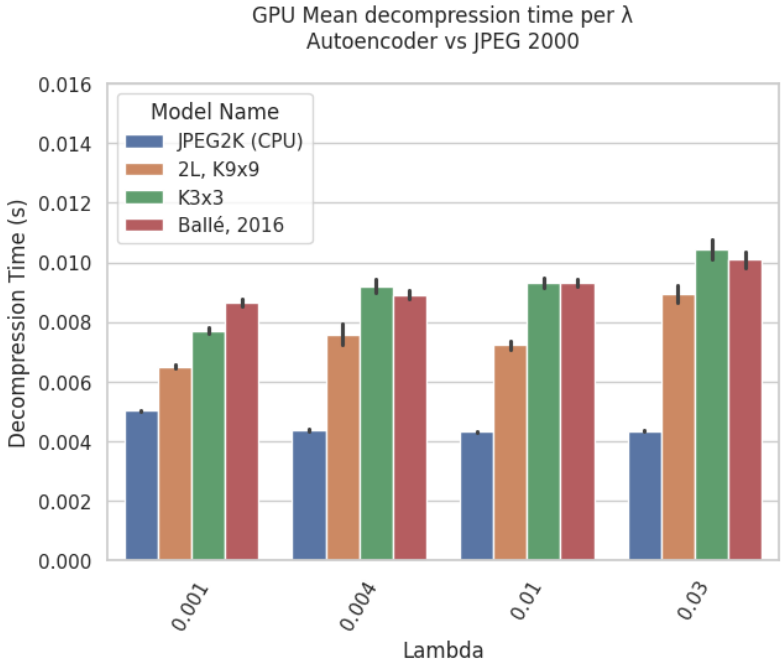
While the *2L* and *2L, K3x3* models had good decompression times, they were obtained with a great penalty to image quality, specially for *2L, K3x3*.

Table 5.3: Average PSNR values for each λ and PSNR difference with (BALLÉ; LAPARRA; SIMONCELLI, 2016)

λ	0.001	0.004	0.01	0.03
Ballé, 2016	23.83 dB	26.49 dB	29.31 dB	33.14 dB
2L, K9x9	23.82 dB (-0.01)	26.57 dB (0.08)	29.50 dB (0.20)	33.08 dB (-0.07)
K3x3	23.76 dB (-0.07)	26.57 dB (0.08)	29.60 dB (0.29)	33.13 dB (-0.02)
JPEG2K	23.40 dB (-0.43)	25.19 dB (-1.31)	27.40 dB (-1.91)	30.88 dB (-2.27)

Source: The Author

Figure 5.10: Mean decompression time of the optimized models, the base model, and JPEG 2000, for all λs. The decompression time of the JPEG 2000 on CPU, while the decompression time of the models are on GPU. The λ on JPEG 2000 represents the same level of BPP as those of the models.



Source: The Author

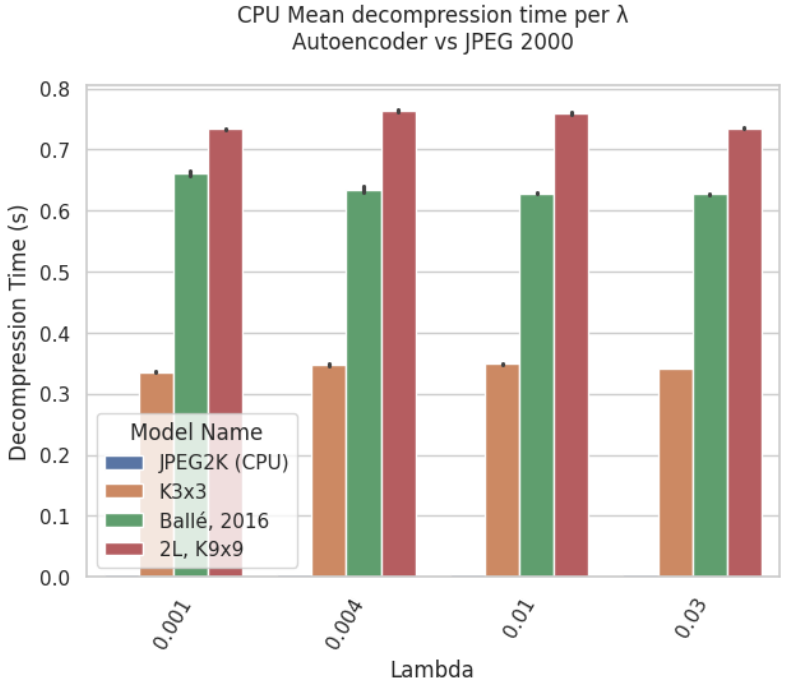
Seeing that the models *2L, K9x9* and *K3x3* obtained the best quality and performance results compared to the other optimizations, we will focus only on them for the in-depth analysis.

5.3.3 In-depth Evaluations

In this section the chosen optimized models will be compared with the base model and the JPEG 2000 codec. First, comparing GPU times and quality between them, then comparing GPU and CPU times, to check if the same performance gain of the optimized models were also reflected when using the CPU.

Figure 5.10 compares the best optimized models, the base model, and the JPEG 2000 codec in terms of decompression time, for all λ values, using the GPU for the models and CPU for JPEG 2000. It is interesting to note that the models kept the same level of speed between each other in all λs, except *K3x3*, that got worse results as the λ got higher, being better than the base model only at the lowest λ. We also see that the *2L, K9x9* model is by far the fastest of the AEs, proving to be an effective optimization when using GPU

Figure 5.11: Mean decompression time of the optimized models, the base model, and JPEG 2000, for all λs, using the CPU. The λ on JPEG 2000 represents the same level of BPP as those of the models.



Source: The Author

due to the parallel processing, as mentioned on Section 5.3.1.

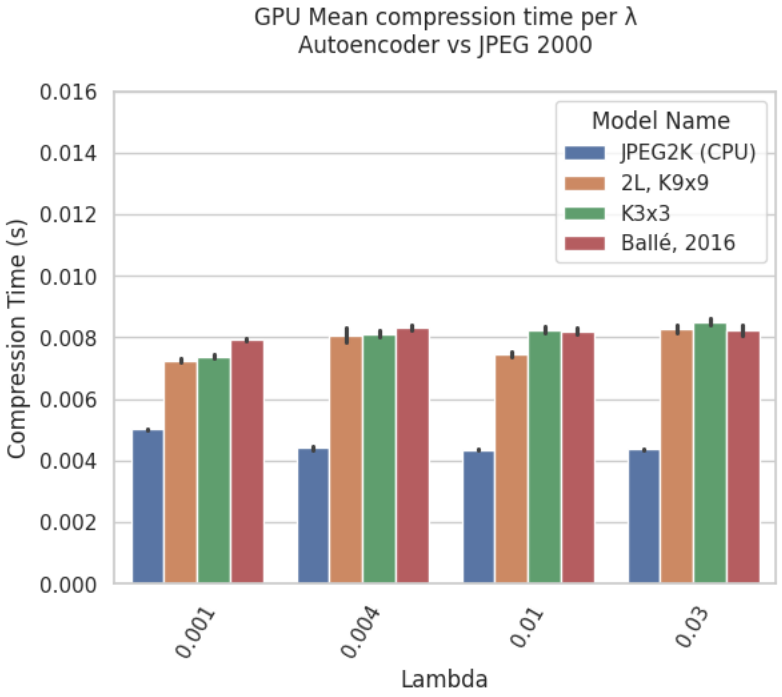
Figure 5.11 shows the mean decompression time, same as Figure 5.10, but this time using the CPU. Here we see the same level of speed throughout all λ. Although, contrary to Figure 5.10, here we see that the model *K3x3* performed at an almost three times less speed than the fastest model on GPU, *2L, K9x9*, that performed even worse than the base model. The cause of that is definitely due to the kernel size being way smaller on *K3x3*, thus, having less parallel operations, while *2L, K9x9*, even though having one less layer, had an increased cost in performance.

Though the model *K3x3* had better performance time than the base model, all of them can not even compare to the performance time of JPEG 2000, being so faster than the AE models that it did not even appear in the plot.

Figures 5.12 and 5.13 shows that, as was expected, the compression times for both the CPU and GPU of the optimized models did not have significant differences, when compared to the decompression times, as we did not change the layers of the encoder.

Although, both optimized models, *2L, K9x9* and *K3x3*, had slightly lower compression times for $\lambda = 0.001$ and $\lambda = 0.004$, and for $\lambda = 0.03$ *2L, K9x9* was almost a milissecond faster than the other models on GPU.

Figure 5.12: Mean compression time of the optimized models, the base model, and JPEG 2000, for all λs. The compression time of the JPEG 2000 on CPU, while the decompression time of the models are on GPU. The λ on JPEG 2000 represents the same level of BPP as those of the models.



Source: The Author

On CPU the *2L, K9x9* model was faster for all λ, while *K3x3* was the slowest overall, except for $\lambda = 0.001$.

Further analyzing the CPU and GPU compression and decompression times, the Table 5.4 shows the compression and decompression times, as well as their average through all λs, for both GPU and CPU of the optimized models and base model, together with JPEG 2000 times.

5.3.4 Effects of λ on model performance

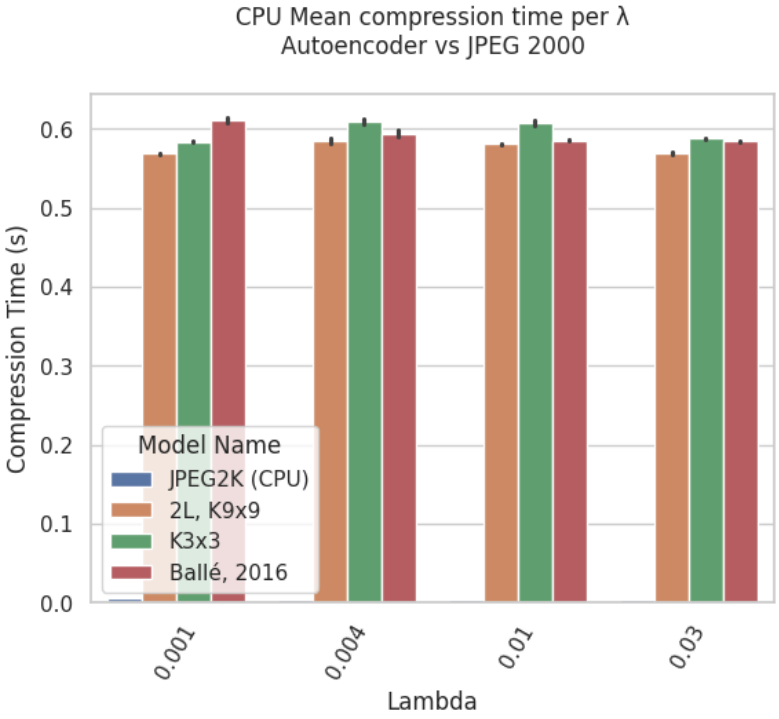
To have an idea on how much theλ values affect the model performance, Figure 5.14 shows how the decompression time on the GPU varies for each model, for all values of λ. As shown in Figure 5.4, we see that the BPP values increases in variance the higher the λ is. This same behavior is reflected in Figure 5.14 as we see that the variance in decompression time is minimal for $\lambda = 0.001$ when compared to $\lambda = 0.03$.

Table 5.4: Mean decompression and decompression times for the best optimized models and the base model for all λ values, and JPEG 2K, on GPU and on CPU (Except JPEG 2000, that only runs on CPU).

Model Name	λ	*Platform*	*Mean Compression Time (s)*	*Mean Decompression Time (s)*
Ballé, 2016	0.001	CPU	0.611	0.661
Ballé, 2016	0.004	CPU	0.594	0.635
Ballé, 2016	0.010	CPU	0.586	0.629
Ballé, 2016	0.030	CPU	0.584	0.627
		Average	**0.594**	**0.638**
Ballé, 2016	0.001	GPU	0.008	0.009
Ballé, 2016	0.004	GPU	0.008	0.009
Ballé, 2016	0.010	GPU	0.008	0.009
Ballé, 2016	0.030	GPU	0.008	0.010
		Average	**0.008**	**0.009**
2L, K9x9	0.001	CPU	0.568	0.734
2L, K9x9	0.004	CPU	0.585	0.764
2L, K9x9	0.010	CPU	0.580	0.759
2L, K9x9	0.030	CPU	0.569	0.735
		Average	**0.576**	**0.748**
2L, K9x9	0.001	GPU	0.007	0.006
2L, K9x9	0.004	GPU	0.008	0.008
2L, K9x9	0.010	GPU	0.007	0.007
2L, K9x9	0.030	GPU	0.008	0.009
		Average	**0.008**	**0.008**
K3x3	0.001	CPU	0.584	0.336
K3x3	0.004	CPU	0.609	0.348
K3x3	0.010	CPU	0.608	0.349
K3x3	0.030	CPU	0.588	0.341
		Average	**0.597**	**0.344**
K3x3	0.001	GPU	0.007	0.008
K3x3	0.004	GPU	0.008	0.009
K3x3	0.010	GPU	0.008	0.009
K3x3	0.030	GPU	0.009	0.010
		Average	**0.008**	**0.009**
JPEG2K	0.001	CPU	0.005	0.005
JPEG2K	0.004	CPU	0.004	0.004
JPEG2K	0.010	CPU	0.004	0.004
JPEG2K	0.030	CPU	0.004	0.004
		Average	**0.005**	**0.005**

Source: The Author

Figure 5.13: Mean compression time of the optimized models, the base model, and JPEG 2000, for all λs, using the CPU. The λ on JPEG 2000 represents the same level of BPP as those of the models.



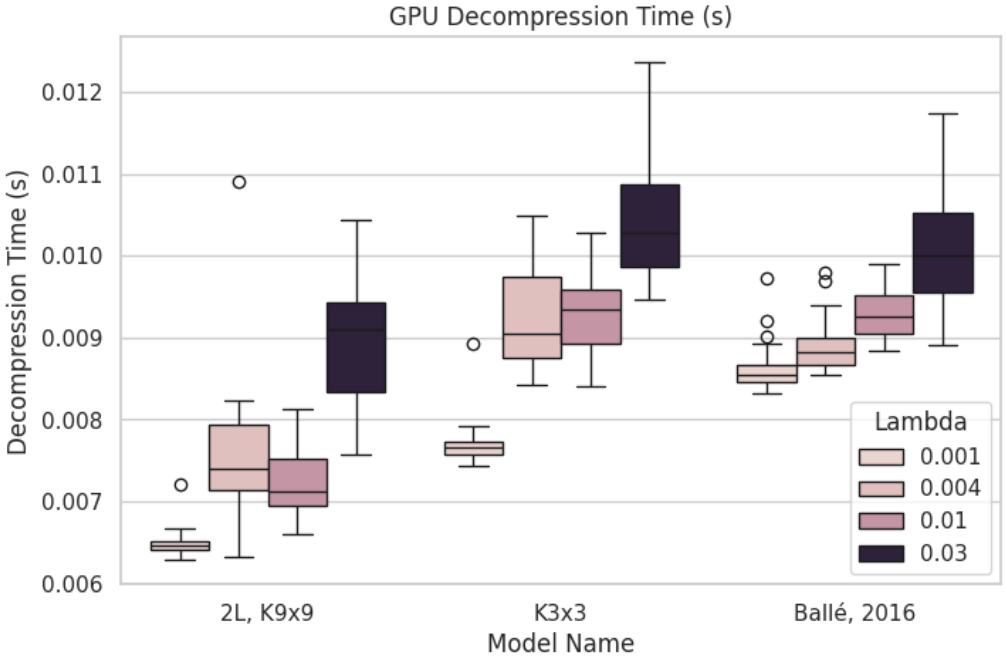
Source: The Author

5.3.5 Rate-distortion performance

To prove that the rate-distortion trade-off is maintained for the optimized models we have Figures 5.15 and 5.16. Each sample on the figures are the results obtained for an image from the Kodak dataset. The samples that are grouped forming a curve represents the images for a specific λ value, increasing from left to right.

We can see, on both figures, that both PSNR and MS-SSIM are maintained in the optimized models. Figure 5.15 shows that, while the optimized models have slightly better PSNR values per sample, they also balance that by having more BPP on the images. For MS-SSIM, the *2L, K9x9* model has slightly higher samples than the other two models, which shows that this model has better MS-SSIM quality overall, for the same values of BPP.

Figure 5.14: GPU decompression time for the optimized models and the base model, for all values of λ.



Source: The Author

5.3.6 Visual Analysis

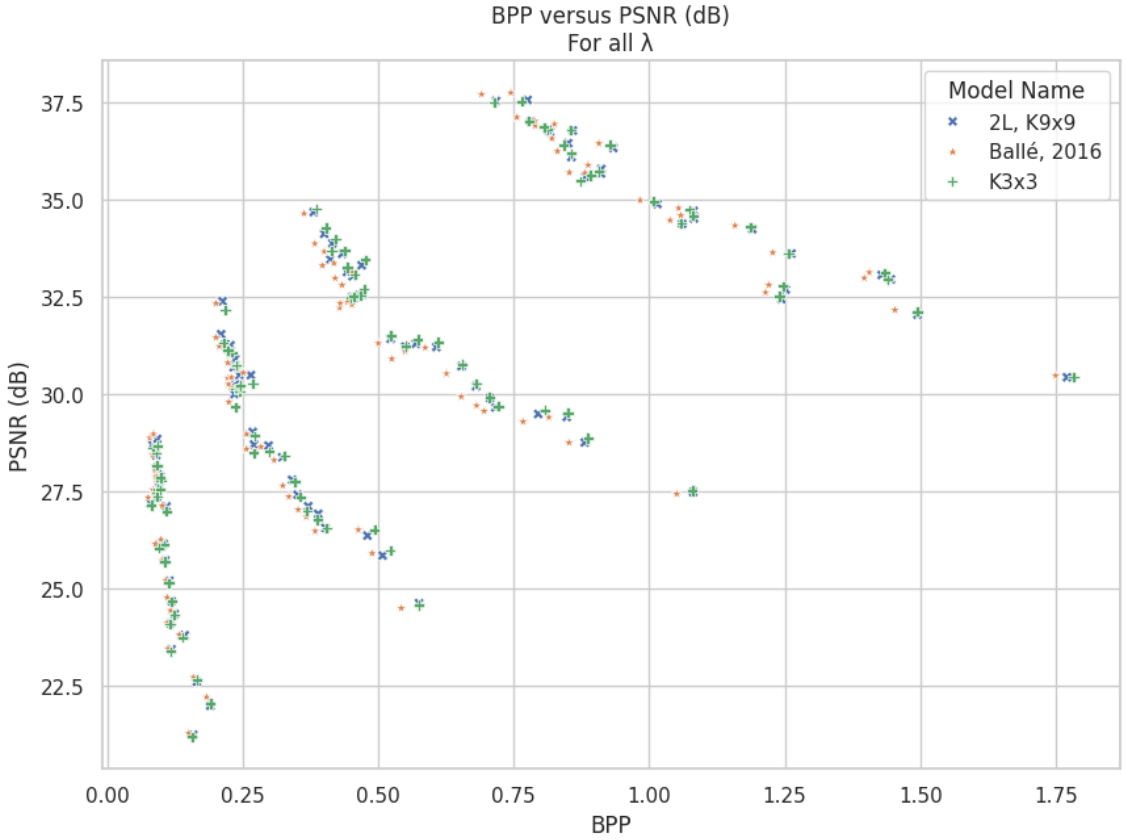
Even though the optimized models obtained great results in both quality and performance, the metrics presented are still not the only measure that should be taken into account, since image quality is often subjective.

The following images were compressed by the AEs, then saved on a Portable Network Graphics (PNG) format. The JPEG 2000 images had to be saved on the Bitmap (BMP) format due to incompatibilities with PNG on the Kakadu software.

On Figure 5.17, one of the main differences between the models and JPEG 2000 is overall smoothness of the image. On JPEG 2000 there seems to be a lot of noise on the bushes and trees, while on the models these parts are mostly blurred. Even though the images have visible differences between them, their quality metrics values are really similar, with only the LPIPS value of the JPEG 2000 compressed image being a little lower than the others.

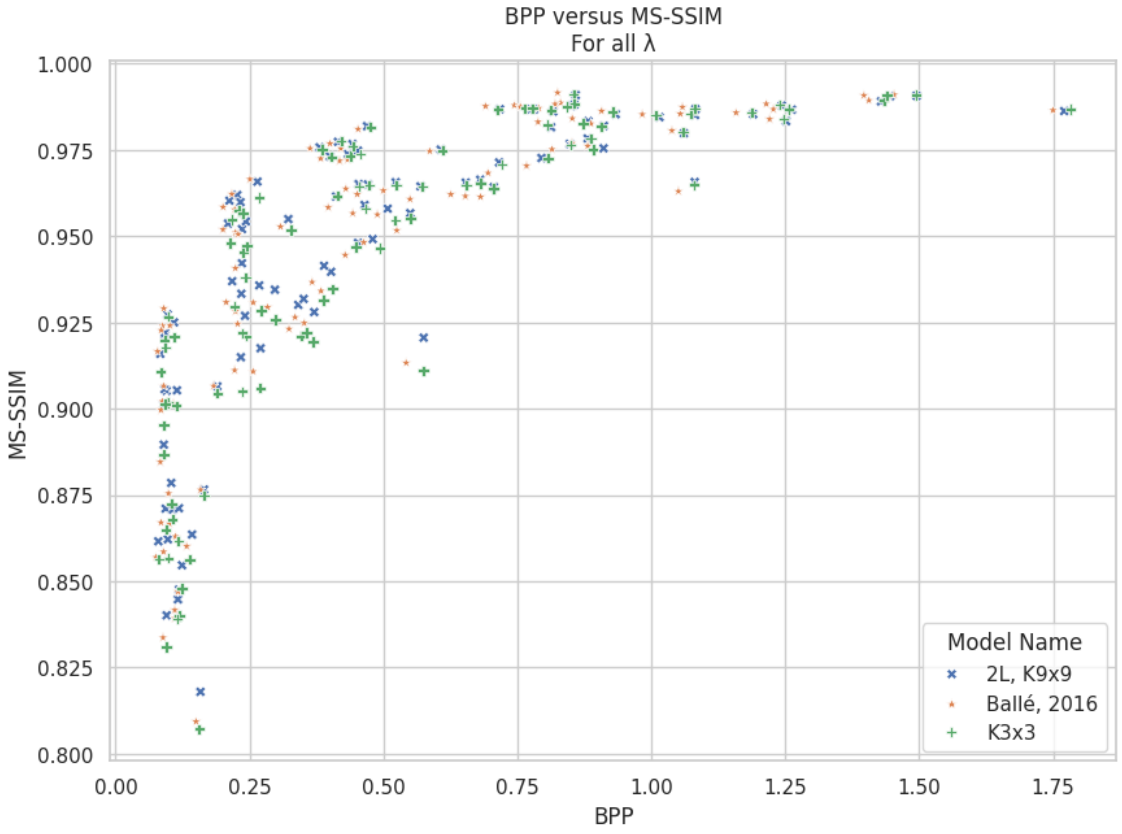
One interesting thing, present on the images compressed by the models, is the presence of blocking artifacts, usually seen in both JPEG and JPEG2000, on the images compressed by the model, which is something that is not present on the images presented on (BALLÉ; LAPARRA; SIMONCELLI, 2016). This is probably due to the fact that the

Figure 5.15: PSNR distribution by BPP value.



Source: The Author

Figure 5.16: MS-SSIM distribution by BPP value.



Source: The Author

Figure 5.17: Image from Kodak dataset compressed for BPP = 0.15.

JPEG2K, BPP: 0.15, PSNR: 21.28 dB, MS-SSIM: 0.817, LPIPS: 0.321.

Ballé, 2016, BPP: 0.15, PSNR: 21.29 dB, MS-SSIM: 0.809, LPIPS: 0.335.

2L, K9x9, BPP: 0.158, PSNR: 21.26 dB, MS-SSIM: 0.817, LPIPS: 0.341.

K3x3, BPP: 0.155, PSNR: 21.20 dB, MS-SSIM: 0.807, LPIPS: 0.342.

Source: (KODAK, 1999)

models presented here are all trained on compressed images, from the (SAMET; HICSONMEZ; AKBAS, 2020) dataset.

Although the blocking artifacts are present on the images compressed by the AE models, this is not very noticeable on the *2L, K9x9* model, while on the *K3x3* model it is very noticeable, specially on images with low BPP. This shows a direct correlation between the size of the blocking artifacts and the kernel size of the convolutional layers.

Figure 5.18 shows images compressed to a BPP value of 0.35. Here we still see some differences between JPEG 2000 and the models. On the images compressed by the *2L, K9x9*, and the *Ballé, 2016* model, we can notice that the details on the faces of the persons are a little better defined. The text at the front of the raft is also slightly more defined on *2L, K9x9* than the others. Overall, the compression of the models have better outlines around the edges of objects, as you can see on the edges of the raft, as well as the words written at the front of the raft.

Figure 5.19 shows images compressed to a BPP value of 0.7. Contrary to the previous images, that had similar metric values, the image compressed by JPEG 2000, in this figure, has a significantly lower PSNR value than the other two, while the perceptual

Figure 5.18: Image from Kodak dataset compressed for BPP = 0.35.

JPEG2K, BPP: 0.35, PSNR: 27.01 dB, MS-SSIM: 0.934, LPIPS: 0.690.

Ballé, 2016, BPP: 0.335, PSNR: 27.38 dB, MS-SSIM: 0.926, LPIPS: 0.658.

2L, K9x9, BPP: 0.351, PSNR: 27.43 dB, MS-SSIM: 0.932, LPIPS: 0.654.

K3x3, BPP: 0.356, PSNR: 27.37 dB, MS-SSIM: 0.922, LPIPS: 0.663.

Source: (KODAK, 1999)

Figure 5.19: Image from Kodak dataset compressed for BPP = 0.7.

JPEG2K, BPP: 0.7, PSNR: 28.56 dB, MS-SSIM: 0.970, LPIPS: 0.840.

Ballé, 2016, BPP: 0.695, PSNR: 29.58 dB, MS-SSIM: 0.969, LPIPS: 0.826.

2L, K9x9, BPP: 0.716, PSNR: 29.67 dB, MS-SSIM: 0.971, LPIPS: 0.845.

K3x3, BPP: 0.721, PSNR: 29.71 dB, MS-SSIM: 0.970, LPIPS: 0.855.

Source: (KODAK, 1999)

metrics have similar values.

Other than the differences mentioned before, in these images it is possible to notice that the images compressed by the models tend to blend some parts of the image that have similar colors. For example, at the balcony, and at the windows, on the JPEG 2000 image we see that there are flower pots with red flowers in them. Then, if you check the other images, we can barely see any color other than green. Looking closely, the flowers are still faintly visible, but they do not maintain their original colors.

Figure 5.20 shows images compressed to a BPP value of 1.2. As these images have a low compression ratio, compared to the previous images, they do not have any outstanding differences. The blocking artifacts present on the images compressed by the models are not even noticeable anymore.

Figure 5.20: Image from Kodak dataset compressed for BPP = 1.2.

JPEG2K, BPP: 1.2, PSNR: 34.14 dB, MS-SSIM: 0.986, LPIPS: 0.924.

Ballé, 2016, BPP: 1.158, PSNR: 34.35 dB, MS-SSIM: 0.986, LPIPS: 0.941.

2L, K9x9, BPP: 1.19, PSNR: 34.25 dB, MS-SSIM: 0.985, LPIPS: 0.944.

K3x3, BPP: 1.187, PSNR: 34.31 dB, MS-SSIM: 0.986, LPIPS: 0.946.

Source: (KODAK, 1999)

6 CONCLUSION AND FUTURE WORKS

Throughout this work, we stated the possibilities of image compression utilizing neural networks, with its advantages and disadvantages when compared to conventional codecs. While (BALLÉ; LAPARRA; SIMONCELLI, 2016) work propose an end-to-end compression model, with great compression quality, it did not delve deeper on evaluating its runtime performance. Thus, did (DICK et al., 2021) work gave us a general idea on the compression and decompression times of current learning-based models, showing that it was still not up to the same level in terms of runtime performance, as that of conventional codecs, such as JPEG 2000 and BPG.

Seeking to lower the decompression time of the AE proposed by (BALLÉ; LAPARRA; SIMONCELLI, 2016), we tried lowering the complexity of the decoder layer of its model, by experimenting with multiple architectures, and selecting the two models with the best results, the *K3x3*, fastest on the CPU, and *2L, K9x9*, fastest on GPU. These lower complexity models still managed to maintain the quality of the original model, with some small quality differences.

The quality of the images are certainly very different between the AE models and JPEG 2000, but this does not mean that one is better than the other. Quality in images can be subjective, and the quality metrics used in this work, like the PSNR, MS-SSIM and LPIPS, are not the only thing that needs to be considered when comparing images. Notice that some of the images presented in this work, have almost identical quality values, but the image are still not visually identical, specially when comparing the learning-based models with the JPEG 2000.

We managed to obtain, on average, approximately 25% reduction in decompression time for the *2L, K9x9* model, when decompressing the images with the GPU. This model achieved decompression times competitive enough to be compared with the JPEG 2000 codec, which is an amazing feat for such a small NN. These results were achieved utilizing one of the newest mid-range consumer GPU, which is not the most cheap GPU available on the market, but also not as expensive as a high-end GPU, or a datacenter-grade GPU.

On CPU, the decompression times did not manage to reach JPEG 2000 levels, but we also saw an incredible 50% lower average decompression time, while only a lowering the kernel size of the first two layers of *Ballé, 2016* model, from a 5 by 5 kernel size to a 3 by 3.

Future works have a lot of paths to choose for experimenting, both on quality and on complexity perspectives of learning-based models. As mentioned by (BALLÉ; LAPARRA; SIMONCELLI, 2016), different metrics could be used during training to possibly increase image quality. Unfortunately this did not work in our case, but with further improvements to these metrics implementations, and TensorFlow, they could provide interesting results.

We utilized a dataset with already compressed images, on the JPEG format, and identified the presence of blocking artifacts on the images compressed by the proposed optimized models, which could probably be avoided by using datasets with less compression and different compression algorithms.

The fact that the model was trained with COCOmini dataset, and then evaluated with the Kodak dataset, shows the power of generalization that this model has, since it still managed to obtain great results without necessarily testing it against the same dataset used in training

Learning-based models have an incredible potential for image compression, seeing how flexible they can be, and how new research can bring new improvements to complexity and quality, bringing them even closer to current conventional compression methods. Also, different complexity reduction techniques could be applied on these models, as the TensorFlow API receives updates, improving compatibility with custom models and adding new features.

To conclude this work, we can highlight how practical it is to test new architectures for an AE, after the initial coding setup is implemented, and how different the results can be with only small changes to the network, achieving high quality images, at high compression ratios, while lowering decompression times. Although the training time is still quite a challenge, we saw how much improvement on training time can have, by taking advantage of the GPU computational power. With the increase in AI technologies, machine learning is rising in popularity, and with Nvidia now becoming an "AI company" (ROACH, 2023), we can expect that even more machine learning focused hardware will be developed in coming years, possibly increasing the adoption of machine-learning-based solutions, like AE models, for example.

REFERENCES

AGGARWAL, C. C. **Neural Networks and Deep Learning, 2nd Edition**. [S.l.]: Springer, 2023.

BALLÉ, J. et al. Nonlinear transform coding. **CoRR**, abs/2007.03034, 2020. Available from Internet: <https://arxiv.org/abs/2007.03034>.

BALLÉ, J.; LAPARRA, V.; SIMONCELLI, E. P. End-to-end optimized image compression. **CoRR**, abs/1611.01704, 2016. Available from Internet: <http://arxiv.org/abs/1611.01704>.

BALLé, J. **Ballé Autoencoder model**. 2022. <https://github.com/tensorflow/compression/blob/81370246972866249 71adb8f6ee4dd3d35d83619/models/bls2017.py>. Accessed: 2024-01-26.

BALLé, J.; LAPARRA, V.; SIMONCELLI, E. P. **Density Modeling of Images using a Generalized Normalization Transformation**. 2016.

BANK, D.; KOENIGSTEIN, N.; GIRYES, R. Autoencoders. **CoRR**, abs/2003.05991, 2020. Available from Internet: <https://arxiv.org/abs/2003.05991>.

BELLARD, F. **Codecs demo and comparison**. 2014. <https://xooyoozoo.github.io/yolo-octo-bugfixes/#soccer-players&jpg=t&bpg=t>. Accessed: 2023-08-04.

BELLARD, F. **BPG Image format**. 2018. <https://bellard.org/bpg/>. Accessed: 2023-07-23.

BROSS, B. et al. Developments in international video coding standardization after avc, with an overview of versatile video coding (vvc). **Proceedings of the IEEE**, v. 109, n. 9, p. 1463–1493, 2021.

BUBOLZ, T. L. A. **Accelerating Intra Frame Partitioning in Versatile Video Coding (VVC) Encoder Using Deep Neural Networks**. 2021. Available from Internet: <http://guaiaca.ufpel.edu.br/handle/prefix/8073>.

BUDAGAVI, M. et al. Core transform design in the high efficiency video coding (hevc) standard. **IEEE Journal of Selected Topics in Signal Processing**, IEEE, v. 7, n. 6, p. 1029–1041, 2013.

CAI, Y. et al. Yolobile: Real-time object detection on mobile devices via compression-compilation co-design. In: **Proceedings of the AAAI conference on artificial intelligence**. [S.l.: s.n.], 2021. v. 35, n. 2, p. 955–963.

DICK, J. et al. Quality and complexity assessment of learning-based image compression solutions. In: **2021 IEEE International Conference on Image Processing (ICIP)**. [S.l.: s.n.], 2021. p. 599–603.

FLORES, S. **Variational Autoencoders are Beautiful**. 2019. <https://www.compthree.com/blog/autoencoder/>. Accessed: 2023-08-05.

GANESH, K. S. **What's The Role Of Weights And Bias In a Neural Network?** 2020. <https://towardsdatascience.com/whats-the-role-of-weights-and-bias-in-a-neural-network-4cf7e9888a0f>. Accessed: 2024-01-31.

GOOGLE. **Supervised vs. unsupervised learning: What's the difference?** n.d. <https://cloud.google.com/discover/supervised-vs-unsupervised-learning>. Accessed: 2024-01-31.

GROIS, D.; NGUYEN, T.; MARPE, D. Coding efficiency comparison of av1/vp9, h.265/mpeg-hevc, and h.264/mpeg-avc encoders. p. 1–5, 2016.

GYAWALI, D. **Comparative Analysis of CPU and GPU Profiling for Deep Learning Models**. 2023.

HASKELL, B. et al. Image and video coding-emerging standards and beyond. **IEEE Transactions on Circuits and Systems for Video Technology**, v. 8, n. 7, p. 814–837, 1998.

JOHNSON, J. **What's a Deep Neural Network? Deep Nets Explained**. 2020. <https://www.bmc.com/blogs/deep-neural-network/>. Accessed: 2023-08-20.

JORDAN, J. **Variational Autoencoders**. 2018. <https://www.jeremyjordan.me/variational-autoencoders/>. Accessed: 2023-08-05.

JPEG. **JPEG Site**. 1992. <https://jpeg.org/jpeg/index.html>. Accessed: 2023-07-23.

JPEG. **JPEG 2000 Site**. 2000. <https://jpeg.org/jpeg2000/index.html>. Accessed: 2023-07-23.

KODAK, E. **Kodak Lossless True Color Image Suite**. 1999. <https://r0k.us/graphics/kodak/>. Accessed: 2023-08-014.

KRIZHEVSKY, A. **Learning Multiple Layers of Features from Tiny Images**. 2009. <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>. Accessed: 2023-08-17.

LIN, T.-Y. et al. **Microsoft COCO: Common Objects in Context**. 2015.

LIU, Y. **AV1 beats x264 and libvpx-vp9 in practical use case**. 2018. <https://engineering.fb.com/2018/04/10/video-engineering/av1-beats-x264-and-libvpx-vp9-in-practical-use-case/>. Accessed: 2023-07-09.

MARCELLIN, M. et al. An overview of jpeg-2000. In: **Proceedings DCC 2000. Data Compression Conference**. [S.l.: s.n.], 2000. p. 523–541.

O'NEILL, J. An overview of neural network compression. **CoRR**, abs/2006.03669, 2020. Available from Internet: <https://arxiv.org/abs/2006.03669>.

RICHARDSON, I. The h.264 advanced video compression standard: Second edition. 04 2010.

RICHARDSON, I. E. **H. 264 and MPEG-4 video compression: video coding for next-generation multimedia**. [S.l.]: John Wiley & Sons, 2004.

ROACH, J. **Nvidia is 'no longer a graphics company'**. 2023. <https://www.digitaltrends.com/computing/nvidia-said-no-longer-graphics-company/>. Accessed: 2024-02-11.

ROSENBERG, J. **Introducing the Industry's Next Video Codec: AV1**. 2018. <https://blogs.cisco.com/collaboration/av1-video-codec>. Accessed: 2023-07-09.

SAMET, N.; HICSONMEZ, S.; AKBAS, E. **HoughNet: Integrating near and long-range evidence for bottom-up object detection**. 2020.

SANDVINE. **The Global Internet Phenomena Report**. [S.l.], 2023.

SULLIVAN, G. J. et al. Overview of the high efficiency video coding (hevc) standard. **IEEE Transactions on Circuits and Systems for Video Technology**, v. 22, n. 12, p. 1649–1668, 2012.

TENSORFLOW. **Tensorflow Compression GitHub page**. 2022. <https://github.com/tensorflow/compression>. Accessed: 2024-01-30.

THEIS, L. et al. Lossy image compression with compressive autoencoders. **ArXiv**, abs/1703.00395, 2017. Available from Internet: <https://arxiv.org/abs/1703.00395>.

TODERICI, G. et al. **Variable Rate Image Compression with Recurrent Neural Networks**. 2016.

TODERICI, G. et al. **Full Resolution Image Compression with Recurrent Neural Networks**. 2017.

WANG, F. et al. Joint activity recognition and indoor localization with wifi fingerprints. **IEEE Access**, v. 7, p. 1–1, 06 2019.

WANG, Z. et al. Image quality assessment: from error visibility to structural similarity. **IEEE Transactions on Image Processing**, v. 13, n. 4, p. 600–612, 2004.

WANG, Z.; SIMONCELLI, E.; BOVIK, A. Multiscale structural similarity for image quality assessment. In: **The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003**. [S.l.: s.n.], 2003. v. 2, p. 1398–1402 Vol.2.

YANG, L.; SONG, J. Rethinking the knowledge distillation from the perspective of model calibration. **arXiv preprint arXiv:2111.01684**, 2021.

YANG, Y. et al. An introduction to neural data compression. **Foundations and Trends® in Computer Graphics and Vision**, Now Publishers, Inc., v. 15, n. 2, p. 113–200, 2023. Available from Internet: <https://arxiv.org/abs/2202.06533>.

ZHANG, R. et al. The unreasonable effectiveness of deep features as a perceptual metric. In: **CVPR**. [S.l.: s.n.], 2018.