Autoenkoderi za kompresiju slike

Kada, Karlo

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:107786

Rights / Prava: In copyright/Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-14**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 454

AUTOENCODERS FOR IMAGE COMPRESSION

Karlo Kada

Zagreb, June 2024

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 454

AUTOENCODERS FOR IMAGE COMPRESSION

Karlo Kada

Zagreb, June 2024

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 04 March 2024

MASTER THESIS ASSIGNMENT No. 454

Student: **Karlo Kada (1191244222)**

Study: Computing

Profile: Data Science

Mentor: assoc. prof. Tomislav Petković

Title: **Autoencoders for image compression**

Description:

Autoencoders are used for learning efficient representations of any data, including images. In image compression, autoencoders compress the image by transforming it into a lower-dimensional latent space through their encoder part. Some of the open problems in using autoencoders for image compression are: (i) during learning SNR is used as a quality measure instead of perceptual measures such as SSIM; (ii) the latent space is not structured in a way that allows for easy determination of compression quality as is possible with JPEG compression; and (iii) when compressing large images autoencoders will process image in blocks disregarding the boundary effects. In the thesis, an overview of autoencoders with a special emphasis on their use in image compression shall be provided. Then, one of the existing autoencoder architectures which is suitable for image compression shall be selected. Next, the selected autoencoder shall be extended to provide a viable solution for at least one of the aforementioned three open problems. The improved autoencoder shall be quantitatively compared with the original solution and with the baselines of JPEG compression and of Karhunen-Loeve transform (principal component analysis).

Submission date: 28 June 2024

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 454

Pristupnik:	**Karlo Kada (1191244222)**
Studij:	Računarstvo
Profil:	Znanost o podacima
Mentor:	izv. prof. dr. sc. Tomislav Petković

Zadatak: **Autoenkoderi za kompresiju slike**

Opis zadatka:

Autoenkoderi se koriste za učenje efikasne reprezentacije podataka kao što su slike. U kompresiji slike autoenkoderi kroz svoj enkoderski dio sliku sažimaju iz ulaznog u niže-dimenzionalni latentni prostor. Neki otvoreni problemi korištenja autoenkodera za kompresiju slike su: (i) učenje koristi SNR kao mjeru kvalitete umjesto perceptualnih mjera kao što je SSIM; (ii) latentni prostor nije strukturiran na način da omogućava jednostavno određivanje kvalitete kompresije kao što je moguće kod JPEG kompresije; i (iii) za kompresiju velikih slika autoenkoderi obrađuju blokove slike bez uzimanja u obzir što se događa na granicama blokova. U diplomskom radu je potrebno dati pregled autoenkodera s posebnim osvrtom na korištenje autoenkodera u kompresiji slike. Zatim je potrebno odabrati neko od postojećih rješenja autoenkodera za kompresiju slike te ga je potrebno proširiti tako da se ponudi rješenje za barem jedan od prethodno navedena tri otvorena problema. To poboljšanje rješenje je potrebno kvantitativno usporediti s polaznim rješenjem te s temeljnim rješenjima JPEG kompresije i Karhunen-Loeveove transformacije (analiza glavnih komponenti).

Rok za predaju rada: 28. lipnja 2024.

I would like to express my deepest gratitude to my kind and motivated mentor, assoc. prof. Tomislav Petković, for his understanding, patience, and feedback. Special thanks go to my family for their great support during my studies.

Contents

1 Introduction **3**

2 Types of Autoencoders **5**

2.1 Undercomplete Autoencoders 5

2.2 Sparse Autoencoders 6

2.3 Contractive Autoencoders 8

2.4 Slimmable Compressive Autoencoders 9

2.5 Autoencoders for Image Compression 12

3 Materials and Methods **14**

3.1 Encoder 14

3.2 Decoder 18

3.3 Training 20

4 Results **24**

4.1 Used Metrics 24

4.1.1 Mean Squared Error 25

4.1.2 Peak Signal to Noise Ratio 25

4.1.3 Multi-scale Structural Similarity Index Measure 26

4.2 Quantitative Results 27

4.3 Qualitative Results 27

5 Discussion **37**

6 Conclusion **40**

References **41**

Abstract **45**

Sažetak **46**

1 Introduction

Autoencoders are neural networks that are trained to map input $\mathbf{x}$ to output $\mathbf{r}$ (which we call the reconstruction) using an internal representation or code $\mathbf{h}$. Although conceptually simple, they are very important in machine learning. The concept of autoencoders was first mentioned in the 1980s [1] as a response to the problem of "backpropagation without a teacher", where only input data was used for learning. Traditionally, autoencoders have been used for dimensionality reduction or feature learning, and today they are considered one of the fundamental methods of unsupervised learning [2]. Autoencoders can be considered a special case of feedforward neural networks and can be trained using all the same techniques, usually gradient descent with mini-batches following gradients calculated by backpropagation. Unlike general feedforward neural networks, autoencoders can also be trained using recirculation [3], a learning algorithm based on comparing the activations of the network on the encoder with the activations on the decoder. An autoencoder consists of two components: an encoder f that maps the input $\mathbf{x}$ to the code $\mathbf{h}$, and a decoder g that maps the code $\mathbf{h}$ to the output $\mathbf{r}$. The aforementioned structure is represented in the figure 1.1. If an autoencoder merely learns to set $g(f(\mathbf{x})) = \mathbf{x}$ universally, it is not particularly valuable. Instead, autoencoders are intentionally designed to avoid perfect copying. They are typically constrained so they can only copy approximately and only for inputs similar to the training data. By forcing the model to prioritize certain aspects of the input for copying, it often discovers useful features of the data [4].

In this thesis, an overview of autoencoders with a special emphasis on their use in image compression is provided. Image compression is the application of data compression on digital images. In effect, the objective is to reduce redundancy of the image data in order to be able to store or transmit data in an efficient form [5]. During this pro-

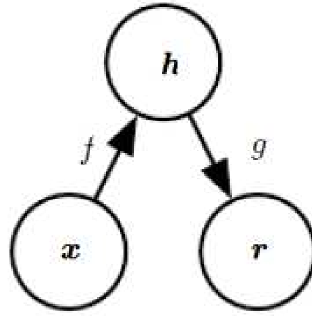

Figure 1.1: The general structure of an autoencoder (reproduced from [4]), mapping an input **x** to an output (called reconstruction) **r** through an internal representation or code **h**. The autoencoder has two components: the encoder f (mapping **x** to **h**) and the decoder g (mapping **h** to **r**).

cess, a slimmable compressive autoencoder [6] was used. We wanted to see how will the autoencoder perform when it is trained using a metric which approximates the level of perceptual image quality, such as multi-scale structural similarity index measure (MS-SSIM) [7], instead of more traditional metrics that don't take the perceptual image quality into consideration, like the mean squared error (MSE). The quantitative results obtained using this method were then evaluated using metrics such as MS-SSIM, MSE and peak signal to noise ratio (PSNR). Qualitative results were also presented in the form of original and reconstructed images, shown side-by-side. The quantitative and qualitative results indicate that the MS-SSIM is a suitable metric for training slimmable compressive autoencoders and that the image reconstructions were of good quality.

2 Types of Autoencoders

In modern literature, various types of autoencoders can be found, each designed to solve a different problem. This chapter explains some of the types of autoencoders in more detail, presenting their architectures and modes of operation. Sections 2.1, 2.2 and 2.3 were in large part reproduced from [4].

2.1 Undercomplete Autoencoders

Reconstructing the input to the output may sound trivial, but we are typically not interested in the output of the decoder. Instead, we hope that training the autoencoder to perform the input copying task will result in $\mathbf{h}$ taking on useful properties. One way to obtain useful features from the autoencoder is to constrain $\mathbf{h}$ to have a smaller dimension than $\mathbf{x}$. An autoencoder whose code dimension is less than the input dimension is called undercomplete. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data. The learning process is described simply as minimizing a loss function

$$\min \mathcal{L}[\mathbf{x},\ g(f(\mathbf{x}))], \tag{2.1}$$

where $\mathcal{L}$ is a loss function penalizing $g(f(\mathbf{x}))$ for being dissimilar from $\mathbf{x}$, such as the mean squared error. When the encoder and the decoder are linear and $\mathcal{L}$ is the mean squared error, an undercomplete autoencoder learns to span the same subspace as PCA (Principal Component Analysis) [8]. In this case, an autoencoder trained to perform the reconstructing task has learned the principal subspace of the training data as a side effect. Autoencoders with nonlinear encoder functions f and nonlinear decoder functions g can thus learn a more powerful nonlinear generalization of PCA. Unfortunately, if the encoder and decoder are allowed too much capacity, the autoencoder can learn to per-

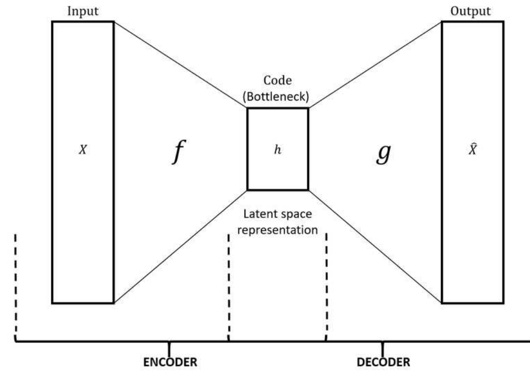

Figure 2.1: The general structure of an undercomplete autoencoder (reproduced from [9]). $\mathbf{X}$ represents the input, function f is the encoder, $\mathbf{h}$ represents the internal representation or code, function g is the decoder, and $\hat{\mathbf{X}}$ the output.

form the copying task without extracting useful information about the distribution of the data. In the figure 2.1 which represents the structure of an undercomplete autoencoder we can clearly see that the dimension of the internal representation $\mathbf{h}$ is smaller than the dimension of the input $\mathbf{x}$.

2.2 Sparse Autoencoders

A sparse autoencoder is simply an autoencoder whose training criterion involves a sparsity penalty $\Omega(\mathbf{h})$ on the code layer $\mathbf{h}$, in addition to the reconstruction error:

$$\mathcal{L}[\mathbf{x},\ g(f(\mathbf{x}))] + \Omega(\mathbf{h}), \tag{2.2}$$

where $\mathbf{h} = f(\mathbf{x})$ is the encoder output and $g(\mathbf{h})$ is the decoder output. Sparse autoencoders are typically used to learn features for another task, such as classification. An autoencoder that has been regularized to be sparse must respond to unique statistical features of the dataset it has been trained on, rather than simply acting as an identity function. In this way, training to perform the copying task with a sparsity penalty can yield a model that has learned useful features as a byproduct. We can think of the penalty $\Omega(\mathbf{h})$ simply as a regularizer term added to a feedforward network whose primary task is to copy the input to the output (unsupervised learning objective). Rather than thinking of the sparsity penalty $\Omega(\mathbf{h})$ as a regularizer for the copying task, we can think of the entire sparse autoencoder framework as approximating maximum likelihood training of a generative model that has latent variables. Suppose we have a model with visible

variables $\mathbf{x}$ and latent variables $\mathbf{h}$, with an explicit joint distribution

$$p_{\text{model}}(\mathbf{h}, \mathbf{x}) = p_{\text{model}}(\mathbf{h}) p_{\text{model}}(\mathbf{x} \mid \mathbf{h}). \tag{2.3}$$

We refer to $p_{\text{model}}(\mathbf{h})$ as the model's prior distribution over the latent variables, representing the model's beliefs prior to observing $\mathbf{x}$. The log-likelihood can now be decomposed as

$$\log p_{\text{model}}(\mathbf{x}) = \log \sum_{\mathbf{h}} p_{\text{model}}(\mathbf{h}, \mathbf{x}). \tag{2.4}$$

We can think of the autoencoder as approximating this sum with a point estimate for just one highly likely value for $\mathbf{h}$. From this point of view, with this chosen $\mathbf{h}$, we are maximizing

$$\log p_{\text{model}}(\mathbf{h}, \mathbf{x}) = \log p_{\text{model}}(\mathbf{h}) + \log p_{\text{model}}(\mathbf{x} \mid \mathbf{h}). \tag{2.5}$$

The $\log p_{\text{model}}(\mathbf{h})$ term can be sparsity inducing. For example, the Laplace prior,

$$p_{\text{model}}(h_i) = \frac{\lambda}{2} e^{-\lambda |h_i|}, \tag{2.6}$$

corresponds to an absolute value sparsity penalty. Expressing the log-prior as an absolute value penalty, we obtain

$$\Omega(\boldsymbol{h}) = \lambda \sum_i |h_i|, \tag{2.7}$$

$$-\log p_{\text{model}}(\boldsymbol{h}) = \sum_i \left(\lambda |h_i| - \log \frac{\lambda}{2} \right) = \Omega(\boldsymbol{h}) + \text{ const }, \tag{2.8}$$

where the constant term depends only on λ and not $\mathbf{h}$. We typically treat λ as a hyperparameter and discard the constant term since it does not affect the parameter learning. Other priors, such as the Student t prior, can also induce sparsity. From this point of view of sparsity as resulting from the effect of $\log p_{\text{model}}(\mathbf{h})$ on approximate maximum likelihood learning, the sparsity penalty is not a regularization term at all. It is just a consequence of the model's distribution over its latent variables. This view provides a different motivation for training an autoencoder: it is a way of approximately training a generative model. It also provides a different reason for why the features learned by the autoencoder are useful: they describe the latent variables that explain the input.

2.3 Contractive Autoencoders

The contractive autoencoder [10] introduces an explicit regularizer on the code $\mathbf{h} = f(\mathbf{x})$, encouraging the derivatives of f to be as small as possible:

$$\Omega(\mathbf{h}) = \left\| \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right\|_F^2 . \tag{2.9}$$

The penalty $\Omega(\mathbf{h})$ is the squared Frobenius norm (sum of squared elements) of the Jacobian matrix of partial derivatives associated with the encoder function. There is a connection between the denoising autoencoder and the contractive autoencoder [11]: research showed that in the limit of small Gaussian input noise, the denoising reconstruction error is equivalent to a contractive penalty on the reconstruction function that maps $\mathbf{x}$ to $\mathbf{r} = g(f(\mathbf{x}))$. In other words, denoising autoencoders make the reconstruction function resist small but finite-sized perturbations of the input, while contractive autoencoders make the feature extraction function resist infinitesimal perturbations of the input. When using the Jacobian-based contractive penalty to pretrain features $f(\mathbf{x})$ for use with a classifier, the best classification accuracy usually results from applying the contractive penalty to $f(\mathbf{x})$ rather than to $g(f(\mathbf{x}))$. The name contractive arises from the way that the contractive autoencoder warps space. Specifically, because the contractive autoencoder is trained to resist perturbations of its input, it is encouraged to map a neighborhood of input points to a smaller neighborhood of output points. We can think of this as contracting the input neighborhood to a smaller output neighborhood. To clarify, this type of autoencoder is contractive only locally, all perturbations of a training point $\mathbf{x}$ are mapped near to $f(\mathbf{x})$. Globally, two different points $\mathbf{x}$ and $\tilde{\mathbf{x}}$ may be mapped to $f(\mathbf{x})$ and $f(\tilde{\mathbf{x}})$ points that are farther apart than the original points. It is plausible that f could be expanding in-between or far from the data manifolds. When the $\Omega(\mathbf{h})$ penalty is applied to sigmoidal units, one easy way to shrink the Jacobian is to make the sigmoid units saturate to 0 or 1. This encourages the autoencoder to encode input points with extreme values of the sigmoid, which may be interpreted as a binary code. It also ensures that the contractive autoencoder will spread its code values throughout most of the hypercube that its sigmoidal hidden units can span.

2.4 Slimmable Compressive Autoencoders

For the content presented in this section, I have predominantly relied on the materials from Yang et al. [6] as the primary source. General autoencoder structure includes an encoder f with parameters θ which maps input $\mathbf{x} \in \mathbb{R}^N$ to code $\mathbf{h} \in \mathbb{R}^D$ and a decoder g with parameters ϕ which maps the code to the output $\mathbf{r} \in \mathbb{R}^N$. The name of the slimmable autoencoders originates from their slimmable layers [12], which allow for dynamic control over memory and computational costs. A slimmable layer enables part of the layer's parameters to be discarded (often by setting them to zero) while still performing valid operations. This trade-off reduces memory and computational costs at the expense of expressiveness. We consider slimmable autoencoders that contain K subautoencoders, where each of the subautoencoders is parametrized by a pair $\psi^{(k)} = (\theta^{(k)}, \phi^{(k)}) \in \Psi = \{(\theta^{(1)}, \phi^{(1)}), \ldots, (\theta^{(K)}, \phi^{(K)})\}$, with assuming that the following conditions are met in every layer: $\theta^{(1)} \subset \cdots \subset \theta^{(K)} = \theta$ and $\phi^{(1)} \subset \cdots \subset \phi^{(K)} = \phi$. We also define the loss for the subautoencoder $k,\ (1 \leq k \leq K)$ as $\mathcal{L}^{(k)}(\theta^{(k)}, \phi^{(k)}; \mathcal{X})$ and train the slimmable autoencoder with the joint loss or a weighted average $\mathcal{L}(\Psi; \mathcal{X}) = \Sigma_k\, \mathcal{L}^{(k)}(\theta^{(k)}, \phi^{(k)}; \mathcal{X})$.

A compressive autoencoder is a type of autoencoder where the encoder's output is a binary stream (bitstream), which is usually stored or sent through a communications channel. The goal is to achieve the highest possible quality of the reconstructed image (minimize distortion) while reducing the number of bits transmitted (minimize the rate). If we look at the structure of a compressive autoencoder, the encoder is followed by a quantizer $\mathbf{q} = Q(\mathbf{h})$, with $\mathbf{q} \in \mathbb{Z}^D$ being a discrete-valued symbol vector. This vector is then processed by a lossless entropy encoder, which converts it into a bitstream $\mathbf{b}$, utilizing its statistical redundancy to produce code lengths close to its entropy. The decoder reverses these operations to reconstruct the original image. Compressive autoencoders are trained by solving a *rate-distortion optimization (RDO)* problem with loss defined as

$$\mathcal{L}(\theta, \psi; \mathcal{X}, \lambda) = D(\theta, \psi; \mathcal{X}) + \lambda R(\theta; \mathcal{X}), \tag{2.10}$$

with $\mathcal{X}$ being the training dataset and λ the fixed tradeoff between rate and distortion. To enable end-to-end optimization with backpropagation, non-differentiable operations like quantization and entropy coding are replaced by differentiable substitutes during

training, such as additive noise and entropy estimation. The compressive autoencoder framework proposed by Balle et al. [13] integrates convolutional layers, generalized divisive normalization (GDN) and inverse GDN (IGDN) layers, scalar quantization to the nearest neighbor ($\mathbf{q} = \lfloor \mathbf{h} \rceil$), and arithmetic coding. During training, quantization is substituted with additive uniform noise (denoted as $\tilde{\mathbf{h}} = \mathbf{h} + \Delta\mathbf{h}\, ; \Delta\mathbf{h} = \mathcal{U}\left(\frac{-1}{2}, \frac{1}{2}\right)$). Similarly, arithmetic coding is bypassed, and the rate is approximated by the entropy of the quantized symbol vector $R(\mathbf{b}) \approx H[P_{\mathbf{q}}] \approx H[p_{\tilde{h}}(\tilde{h}; \nu)]$, with ν representing the parameters of the entropy model used in [13]. Distortion is measured by the reconstruction mean square error $||\mathbf{x} - \mathbf{r}||^2$. Therefore, the compressive autoencoder is parameterized by $\psi = (\theta, \phi, \nu)$.

To create a slimmable compressive autoencoder (SlimCAE), all operations within the CAE must be non-parametric, slimmable, or efficiently switchable. In this implementation, quantization is non-parametric, and the convolutional layers are implemented as slimmable [12]. For the GDN/IGDN layers, several variants can be used. Additionally, we use switchable entropy models, where each subautoencoder k has its own parameters $\nu^{(k)}$ that can easily be switched because their size is insignificant in comparison to the other parameters $\theta^{(k)}$ or $\phi^{(k)}$ [6].

While GDN [14] was proposed to Gaussianize the local joint statistics of natural images, Balle et al. [13] proposed an approximate inverse operation (IGDN), and showed that GDN/IGDN layer pairs are highly beneficial in learned image compression, and since then have been adopted by many compressive autoencoder frameworks. Both GDN and IGDN are parametrized by $\gamma \in \mathbb{R}^{w \times w}$ and $\beta \in \mathbb{R}^{w}$, where w is the number of input (and output) channels.

In the case of a SlimCAE with K subCAEs, the input to the GDN layer has the following possible channel dimensions $w^{(1)}, \ldots, w^{(K)}$. We consider three possible variants:

1. **Switchable GDNs.** We use independent sets of parameters $\gamma^{(k)} \in \mathbb{R}^{w^{(k)} \times w^{(k)}}$ and $\beta^{(k)} \in \mathbb{R}^{w^{(k)}}$ for every subGDN k. The normalized representation for an input $y^{(k)} \in \mathbb{R}^{w^{(k)}}$ is then

$$\tilde{y}_i^{(k)} = \frac{y_i^{(k)}}{\left(\beta_i^{(k)} + \sum_j \gamma_{ij}^{(k)} \left|y_j^{(k)}\right|^2\right)^{\frac{1}{2}}}. \tag{2.11}$$

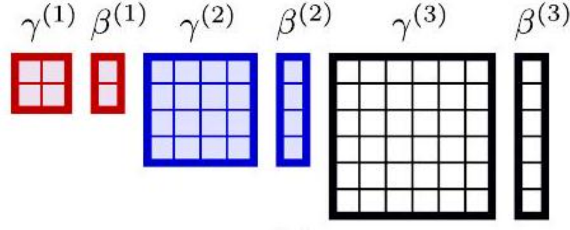

Figure 2.2: The general structure of a switchable GDN (reproduced from [6]).

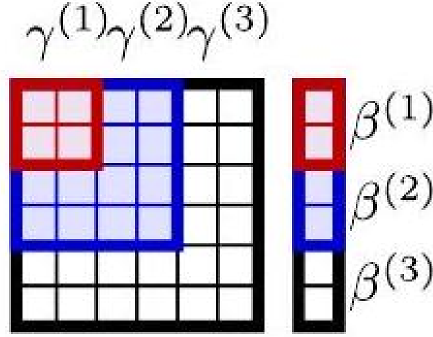

Figure 2.3: The general structure of a slimmable GDN (reproduced from [6]).

While flexible, the total number of parameters is relatively high $\sum_{k=1}^{K}\left(w^{(k)}+1\right)w^{(k)}$, and switching may be not very efficient. An illustration of this GDN variant can be seen in the figure 2.2 [6].

2. **Slimmable GDN (SlimGDN)**. A more compact option is to reuse parameters from smaller subGDNs by imposing $\gamma^{(1)} \subset \cdots \subset \gamma^{(K)}$ and $\beta^{(1)} \subset \cdots \subset \beta^{(K)}$. Now the total number of parameters in a SlimGDN layer is $\left(M^{(K)}+1\right)w^{(K)}$. An illustration of a slimmable GDN can be seen in the figure 2.3 [6].

3. **Slimmable GDN with switchable parameter modulation.** SlimGDNs usually performs worse than switchable GDNs, since they are less flexible to adapt to the statistics of the different $\mathbf{y}^{(k)}$. We propose a variant using switchable parameter modulation, where a global scale and bias are learned separately for every subGDN (i.e. switchable), i.e. $\gamma_{ij}^{(k)} = s_{\gamma}^{(k)}\gamma'^{(k)}_{ij} + b_{\gamma}^{(k)}$ and $\beta_{i}^{(k)} = s_{\beta}^{(k)}\beta'^{(k)}_{i} + b_{\beta}^{(k)}$, where $\gamma'^{(k)}$ and $\beta'^{(k)}$ are shared and slimmable and $s_{\gamma}^{(k)}, b_{\gamma}^{(k)}, s_{\beta}^{(k)}$ and $b_{\beta}^{(k)}$ are switchable scalars specific for the subGDN k. This variant requires only 4 additional parameters per subGDN, for a total number of parameters $\left(w^{(K)}+1\right)w^{(K)}+4K$. The illustration of a slimmable GDN with switchable parameter modulation can be seen in the figure 2.4 [6].

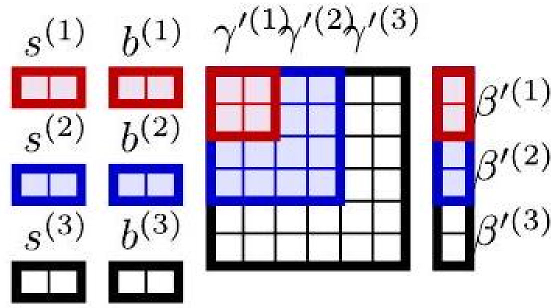

Figure 2.4: The general structure of a slimmable GDN with switchable parameter modulation (reproduced from [6]).

2.5 Autoencoders for Image Compression

Uncompressed multimedia, such as graphics, audio, and video, requires significant storage space and bandwidth. Despite advances in mass-storage density, processor speeds, and digital communication systems, the demand for data storage and transmission can exceed the capabilities of current technologies. The surge in data-heavy multimedia web applications has not only underscored the necessity for efficient encoding methods for signals and images but has also made compression a crucial aspect of storage and communication technology [5].

Image compression can be categorized into two main types: lossless and lossy. Lossless compression reduces the file size by encoding all the original image information, ensuring that the decompressed image is identical to the original. Examples of lossless compression formats include PNG and GIF [15]. In contrast, lossy compression results in some loss of information. The compressed image resembles the original but is not an exact replica, as certain details are discarded during compression. This method is typically suited to images, with JPEG being the most well-known example of lossy compression [16]. Autoencoders can be used for both lossless and lossy compression, depending on the dimension of the internal representation $\mathbf{h}$. Lossless compression can be achieved by setting the dimension of the code to be equal to the dimension of the input, and lossy compression can be achieved by limiting the code dimension to be smaller than the input.

We can compress an image using methods such as Principal Component Analysis (PCA), where we form a hyperplane of a lower dimension than the original while retain-

ing the variance among the data. However, PCA can only model linear relationships. Undercomplete autoencoders, on the other hand, can learn nonlinear relationships and thus perform dimensionality reduction better. This type of dimensionality reduction is also known as manifold learning. If we were to remove all nonlinear activations from an undercomplete autoencoder and use only linear layers, we would obtain a reduced undercomplete autoencoder that works in the same way as PCA [17].

The existing methods of using autoencoders for image compression include convolutional autoencoders [18], variational autoencoders [19], conditional autoencoders [20] and compressive autoencoders [21].

3 Materials and Methods

This masters thesis follows the work of Yang et al. [6]. One of the open problems in using autoencoders for image compression is that during learning, SNR is often used as a quality measure instead of perceptual measures such as MS-SSIM.

The changes made to the code [22] were to the training function of the slimmable compressive autoencoder. Precisely, the mean squared error (MSE) metric wasn't used to train the autoencoder. Instead, the used metric was the multi-scale structural similarity index measure (MS-SSIM) [7]. This was done because MS-SSIM is perceptual image quality metric, which means it aligns more closely with human visual perception. It considers structural similarity, luminance, and contrast at multiple scales, leading to reconstructions that are more visually pleasing and perceptually accurate. MSE, on the other hand, minimizes the pixel-wise squared difference, which does not necessarily correlate well with perceptual image quality. It treats all errors equally, which can lead to ringing artifacts and loss of important structural details in the image reconstructions.

To better understand the architecture of the slimmable compressive autoencoder, we will go through the Python code, which was run on the Python 2.7.12 version. In the following sections, we will describe the used functions, starting with the encoder.

3.1 Encoder

In the encoder architecture, compression is achieved using strided convolutions. Conventional convolution uses a step size (or stride) of 1 meaning that the sliding filter moves 1 pixel at a time. On the contrary, strided convolution introduces a stride variable that controls the step of the folder as it moves over the input. So, for example, when the stride is equal to 2, the filter skips one pixel every time it slides over the input sample, resulting

```
def slimmable_analysis_transform(tensor_in, switch_list,
    total_filters_num):
    with tf.variable_scope("analysis"):
        tensor_encoder = list()
        for i, _switch in enumerate(switch_list):
            with tf.variable_scope("layer_0",reuse=(i>0)):
                layer = SignalConv2D_slim(
                    total_filters_num, (9, 9), corr=True,
                    strides_down=4, padding="same_zeros",
                    use_bias=True, activation=None)
                tensor = layer(tensor_in, 3, _switch)
            with tf.variable_scope("gdn_an_0_{:1d}".format(i)):
                tensor_gdn_0 = tfc.GDN()(tensor)
            tensor_gdn_0 = tf.pad(tensor_gdn_0, [[0,0], [0,0], [0,0],
                [0,(total_filters_num - _switch)]], "CONSTANT")
            with tf.variable_scope("layer_1",reuse=(i>0)):
                layer = SignalConv2D_slim(
                    total_filters_num, (5, 5), corr=True,
                    strides_down=2, padding="same_zeros",
                    use_bias=True, activation=None)
                tensor = layer(tensor_gdn_0, _switch, _switch)
            with tf.variable_scope("gdn_an_1_{:1d}".format(i)):
                tensor_gdn_1 = tfc.GDN()(tensor)
            tensor_gdn_1 = tf.pad(tensor_gdn_1, [[0,0], [0,0], [0,0],
                [0,(total_filters_num - _switch)]], "CONSTANT")
            with tf.variable_scope("layer_2",reuse=(i>0)):
                layer = SignalConv2D_slim(
                    total_filters_num, (5, 5), corr=True,
                    strides_down=2, padding="same_zeros",
                    use_bias=False, activation=None)
                tensor = layer(tensor_gdn_1, _switch, _switch)
            with tf.variable_scope("gdn_an_2_{:1d}".format(i)):
                tensor_gdn_2 = tfc.GDN()(tensor)
            tensor_encoder.append(tensor_gdn_2)
        return tensor_encoder
```

Code 3.1: Encoder function.

in a smaller output feature map. During strided convolution, the filter skips some pixels as it moves over the given input, performing a downsampling on the input data. This inherent downsampling comes with several advantages in the learning process. Importantly, downsampling pushes the network to focus on the most discriminative features, ignoring redundant information [23]. This encoder utilizes three convolutional layers. The first one uses a stride of 4, and the second and third use a stride of 2. This implies that if a image of height H, width W and color channels C is provided as the input of the encoder and if n_f is the number of filters used, the output of the first convolutional layer will be a tensor of size

$$\left\lceil \frac{H}{4} \right\rceil \times \left\lceil \frac{W}{4} \right\rceil \times n_f, \tag{3.1}$$

the output of the second convolutional layer will be a tensor of size

$$\left\lceil \frac{\left\lceil \frac{H}{4} \right\rceil}{2} \right\rceil \times \left\lceil \frac{\left\lceil \frac{W}{4} \right\rceil}{2} \right\rceil \times n_f \tag{3.2}$$

and the output of the third convolutional layer will be a tensor of size

$$\left\lceil \frac{\left\lceil \frac{\left\lceil \frac{H}{4} \right\rceil}{2} \right\rceil}{2} \right\rceil \times \left\lceil \frac{\left\lceil \frac{\left\lceil \frac{W}{4} \right\rceil}{2} \right\rceil}{2} \right\rceil \times n_f. \tag{3.3}$$

If we apply this procedure to an image of size $512 \times 512 \times 3$ and use 192 filters, input size will be equal to $512 \times 512 \times 3 = 786432$ bytes or 768 kibibytes. For a neural network implemented in TensorFlow, the latent space tensor elements are of type *float32*, which means each element takes up 4 bytes (since *float32* is a 32-bit floating point number). This implies that if we use 192 filters per layer, the size of the latent space will be equal to $32 \times 32 \times 192 \times 4 = 786432$ bytes or 768 kibibytes. We can see that the size of the latent space is equal to the original image. However, if we use 48 filters per layer, the size of the latent space will be equal to $32 \times 32 \times 48 \times 4 = 196608$ bytes or 192 kibibytes, which is 4 times smaller than when using 192 filters. The reverse process is present in the decoder function, where we apply upsampling by using strides of 2, 2 and 4 in each of the deconvolutional layers, respectively.

Code of the encoder function can be seen in the code listing 3.1. The input of the encoder is a tensor of size $H \times W \times 3 \times 4$ bytes, where H is the height of the image, W is the width, 3 is the number of color channels and 4 is the number of bytes required to store a *float32* number. The output is a tensor of size $\left\lceil \frac{H}{16} \right\rceil \times \left\lceil \frac{W}{16} \right\rceil \times n_f \times 4$ bytes, where n_f is the number of filters per layer. The `slimmable_analysis_transform` function builds the encoder part of the slimmable autoencoder [22]. It processes an input tensor through a series of convolutional layers and generalized divisive normalization (GDN) layers, adapting the number of filters per layer dynamically based on the `switch_list` variable. In this thesis, the used `switch_list` variable was an array `[192, 144, 96, 72, 48]`, and the total number of filters was set to 192. Function parameters include:

1. `tensor_in` - input tensor, which represents a batch of images,
2. `switch_list` - list of integers specifying the number of filters per layer to use at each step, and
3. `total_filters_num` - total number of filters used in the layers before switching.

First, we initialize the `tensor_encoder` list in the line 3, in which the output tensors are stored in. We then enter a `for` loop which iterates over all number of filters per layer. Then we define `layer_0` - the first convolutional layer with the `SignalConv2D_slim` function (line 6). The first convolutional layer was initialized with 192 total filters used, a kernel size of 9×9, downsampling was defined by a factor of 4 and the padding parameter was set to `same_zeros`, which ensures the output size matches the input size after convolution. In the line 10, the aforementioned layer then processes the input tensor with 3 input channels and `_switch` output channels - number of filters per layer previously specified in the `switch_list` array. We then apply a generalized divisive normalization (GDN) layer (line 12), which helps with reducing redundancy in the activation maps, and finish by padding the given tensor to maintain a consistent number of filters across different stages. In the lines $16 - 19$, the second convolutional layer is built. The differences between it and the first layer are that it uses a smaller kernel size of 5×5, downsamples by a factor of 2, and uses the `_switch` variable for both input and output channels. The GDN and padding stay identical. The third convolutional layer (lines $26 - 29$) is the same as the second one, but without a bias term. We then append the

output variable of the third convolutional layer `tensor_gdn_2` to the `tensor_encoder` list (line 33), by which we are storing the bottleneck features for each filter width. After repeating this process for all the number of filters per layer specified, in the line 34 we return the `tensor_encoder` list.

3.2 Decoder

The code of the decoder function can be seen in the code listing 3.2. The input of the decoder is a tensor of size $\left\lceil \frac{H}{16} \right\rceil \times \left\lceil \frac{W}{16} \right\rceil \times n_f \times 4$ bytes, where *nf* is the number of filters per layer and 4 is the number of bytes required to store a *float32* number. The decoder output is an tensor of size $H \times W \times 3 \times 4$ bytes. The `slimmable_synthesis_transform` function builds the decoder part of a slimmable autoencoder [22]. It transforms the encoded tensors back into their original form, processing them through a series of transposed convolutional (deconvolutional) layers and inverse generalized divisive normalization (IGDN) layers, while dynamically adjusting the number of filters based on the `switch_list` given. Function parameters include:

1. `tensor_encoder` - list of encoded tensors from the encoder, each corresponding to different widths,
2. `switch_list` - list of integers specifying the number of filters per layer to use at each step, and
3. `total_filters_num` - total number of filters used in the layers before switching.

The `switch_list` and `total_filters_num` were identical to the ones used in the encoder function, i. e. the `switch_list` variable was an array `[192, 144, 96, 72, 48]`, and the total number of filters was set to 192. First, we initialize the `tensor_decoder` list in the line 3, in which the output tensors are stored in. We then enter a `for` loop which iterates over all number of filters per layer. Then we define the first inverse GDN layer (lines 6–7) with the `tensorflow_compression.GDN` function (inverse parameter is set to `True`). The aforementioned layer is applied to the element at the position i of the encoded tensor. We then pad the output tensor in the lines $8-9$ to maintain a consistent number of filters across different stages. After doing so, we define the first deconvolutional layer using the `SignalConv2D_slim` function (lines $11-14$). The first deconvolutional layer

```python
def slimmable_synthesis_transform(tensor_encoder, switch_list,
    total_filters_num):
    with tf.variable_scope("synthesis"):
        tensor_decoder = list()
        for i, _switch in enumerate(switch_list):
            with tf.variable_scope("gdn_sy_0_{:1d}".format(i)):
                tensor_igdn_0 = tfc.GDN(inverse=True)
                    (tensor_encoder[i])
            tensor_igdn_0 = tf.pad(tensor_igdn_0, [[0,0], [0,0],
                [0,0], [0,(total_filters_num - _switch)]], "CONSTANT")
            with tf.variable_scope("layer_0",reuse=(i>0)):
                layer = SignalConv2D_slim(
                    total_filters_num, (5, 5), corr=False,
                    strides_up=2, padding="same_zeros",
                    use_bias=True, activation=None)
                tensor = layer(tensor_igdn_0, _switch, _switch)
            with tf.variable_scope("gdn_sy_1_{:1d}".format(i)):
                tensor_igdn_1 = tfc.GDN(inverse=True)(tensor)
            tensor_igdn_1 = tf.pad(tensor_igdn_1, [[0,0], [0,0],
                [0,0], [0,(total_filters_num - _switch)]], "CONSTANT")
            with tf.variable_scope("layer_1",reuse=(i>0)):
                layer = SignalConv2D_slim(
                    total_filters_num, (5, 5), corr=False,
                    strides_up=2, padding="same_zeros",
                    use_bias=True, activation=None)
                tensor = layer(tensor_igdn_1, _switch, _switch)
            with tf.variable_scope("gdn_sy_2_{:1d}".format(i)):
                tensor_igdn_2 = tfc.GDN(inverse=True)(tensor)
            tensor_igdn_2 = tf.pad(tensor_igdn_2, [[0,0], [0,0],
                [0,0], [0,(total_filters_num - _switch)]], "CONSTANT")
            with tf.variable_scope("layer_2",reuse=(i>0)):
                layer = SignalConv2D_slim(
                    3, (9, 9), corr=False,
                    strides_up=4, padding="same_zeros",
                    use_bias=True, activation=None)
                tensor = layer(tensor_igdn_2, _switch, 3)
            tensor_decoder.append(tensor)
        return tensor_decoder
```

Code 3.2: Decoder function.

was initialized with 192 total filters used, a kernel size of 5×5, upsampling by a factor of 2 and the padding parameter was set to `same_zeros`, which ensures the output size matches the input size after convolution. The aforementioned layer then processes the input tensor in the line 15 with the same number of input and output channels - number of filters per layer in the current iteration. The second inverse GDN layer (identical to the first one) is then built (line 17) and the tensor is passed through it. The output tensor is then padded as described above (lines $18-19$) and passed through a second deconvolutional layer in the line 25, identical to the first one. Lastly, the tensor is processed by the third inverse GDN layer (line 27), padded and then processed by the third deconvolutional layer (lines $28-35$). The only distinction to the first two processes is that the third deconvolutional layer has a kernel size of 9×9 and features upsampling by a factor of 4. It outputs the tensor with 3 channels, which represent the RGB channels of the image. We then append the output tensor to the `tensor_decoder` list in the line 36, by which we are storing the reconstructed features for each filter width. After repeating this process for all the number of filters per layer specified, we return the `tensor_decoder` list.

3.3 Training

One of the main parts of this thesis was implementing the MS-SSIM as the training metric. The train loss for each of the reconstructed images is defined as a rate-distortion optimization (RDO) problem $\lambda^i\,(1 - MS\text{-}SSIM(\mathbf{x}^i, \mathbf{r}^i)) + bpp^i$, where λ^i is one of the predefined regularization constants for each iteration, MS-SSIM is the multi-scale structural similarity index measure of the original image $\mathbf{x}$ and the reconstruction $\mathbf{r}$. bpp^i represents the bitrate measured by total number of bits divided by number of pixels or simply *bits per pixel*. Instead of multiplying the λ^i with the mean square error, it is multiplied with the term $(1 - MS\text{-}SSIM(\mathbf{x}^i, \mathbf{r}^i))$. The goal is to maximize the MS-SSIM, so subtracting it from 1 and then minimizing this value effectively maximizes the MS-SSIM. The number of iterations is defined by the number of filters per layer of the autoencoder.

To explain how multi-scale SSIM works, we must first describe the single-scale SSIM method. The following part of this section was in large part reproduced from [7]. Let $\mathbf{x} = \{x_i \mid i = 1, 2, \cdots, N\}$ and $\mathbf{y} = \{y_i \mid i = 1, 2, \cdots, N\}$ be two discrete non-negative signals that have been aligned with each other (e.g., two image patches extracted from the

same spatial location from two images being compared, respectively), and let μ_x, σ_x^2 and σ_{xy} be the mean of $\mathbf{x}$, the variance of $\mathbf{x}$, and the covariance of $\mathbf{x}$ and $\mathbf{y}$, respectively. Approximately, μ_x and σ_x can be viewed as estimates of the luminance and contrast of $\mathbf{x}$, and σ_{xy} measures the the tendency of $\mathbf{x}$ and $\mathbf{y}$ to vary together, thus an indication of structural similarity. In [24], the luminance, contrast and structure comparison measures were given as follows:

$$l(\mathbf{x},\mathbf{y}) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}, \tag{3.4}$$

$$c(\mathbf{x},\mathbf{y}) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}, \tag{3.5}$$

$$s(\mathbf{x},\mathbf{y}) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}, \tag{3.6}$$

where C_1, C_2 and C_3 are small constants given by

$$C_1 = (K_1\, L)^2,\ C_2 = (K_2\, L)^2 \text{ and } C_3 = \frac{C_2}{2}. \tag{3.7}$$

L is the dynamic range of the pixel values ($L = 255$ for 8 bits/pixel gray scale images), and $K_1 \ll 1$ and $K_2 \ll 1$ are two scalar constants. The general form of the structural similarity (SSIM) index between signal $\mathbf{x}$ and $\mathbf{y}$ is defined as:

$$\text{SSIM}(\mathbf{x},\mathbf{y}) = [l(\mathbf{x},\mathbf{y})]^\alpha \cdot [c(\mathbf{x},\mathbf{y})]^\beta \cdot [s(\mathbf{x},\mathbf{y})]^\gamma, \tag{3.8}$$

where α, β and γ are parameters to define the relative importance of the three components. Specifically, if we set $\alpha = \beta = \gamma = 1$, the resulting SSIM index is given by

$$\text{SSIM}(\mathbf{x},\mathbf{y}) = \frac{(2\mu_x\mu_y + C_1)\,(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)\,(\sigma_x^2 + \sigma_y^2 + C_2)}, \tag{3.9}$$

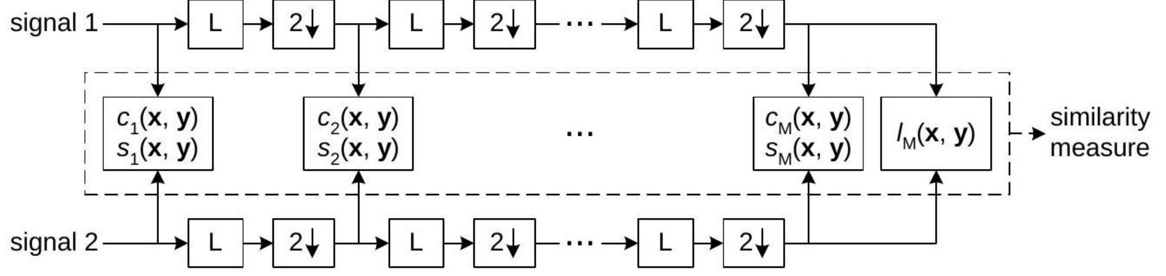

Figure 3.1: Multi-scale structural similarity measurement system (reproduced from [7]). L: low-pass filtering; $2\downarrow$: downsampling by 2.

which satisfies the following conditions:

1. symmetry: $\text{SSIM}(\mathbf{x}, \mathbf{y}) = \text{SSIM}(\mathbf{y}, \mathbf{x})$;
2. boundedness: $\text{SSIM}(\mathbf{x}, \mathbf{y}) \leq 1$;
3. unique maximum: $\text{SSIM}(\mathbf{x}, \mathbf{y}) = 1$ if and only if $\mathbf{x} = \mathbf{y}$.

Now we are ready to explain the multi-scale SSIM index. While a single-scale may be appropriate only for specific settings, multi-scale method is a convenient way to incorporate image details at different resolutions. In the figure 3.1 we can see an illustration of the multi-scale SSIM index system diagram. Taking the reference and distorted image signals as the input, the system iteratively applies a low-pass filter and downsamples the filtered image by a factor of 2. We index the original image as Scale 1, and the highest scale as Scale M, which is obtained after $M-1$ iterations. At the j-th scale, the contrast comparison Eq. (3.5) and the structure comparison Eq. (3.6) are calculated and denoted as $c_j(\mathbf{x}, \mathbf{y})$ and $s_j(\mathbf{x}, \mathbf{y})$, respectively. The luminance comparison Eq. (3.4) is computed only at Scale M and is denoted as $l_M(\mathbf{x}, \mathbf{y})$. The overall SSIM evaluation is obtained by combining the measurement at different scales using

$$\text{MS-SSIM}(\mathbf{x}, \mathbf{y}) = [l_M(\mathbf{x}, \mathbf{y})]^{\alpha_M} \cdot \prod_{j=1}^{M} \left[c_j(\mathbf{x}, \mathbf{y})\right]^{\beta_j} \left[s_j(\mathbf{x}, \mathbf{y})\right]^{\gamma_j} \tag{3.10}$$

Similar to Eq. (3.8), the exponents α_M, β_j and γ_j are used to adjust the relative importance of different components. This multi-scale SSIM index definition satisfies the three conditions given in the last section. It also includes the single-scale method as a special case. In particular, a single-scale implementation for Scale M applies the itera-

tive filtering and downsampling procedure up to Scale M and only the exponents α_M, β_M and γ_M are given nonzero values. To simplify parameter selection, we let $\alpha_j = \beta_j = \gamma_j$ for all j's. In addition, we normalize the cross-scale settings such that $\sum_{j=1}^{M} \gamma_j = 1$. This makes different parameter settings (including all single-scale and multi-scale settings) comparable.

When training the model, the batch size is set to 8, and the size of image patches is set to 240 pixels. There are 5 different numbers of filters per layer used: $192, 144, 96, 72$ and 48. The model's training is limited by the training steps, which were set to 2000. In the code, a training step refers to one update of the model's parameters using a single batch of training images. The number of training steps is determined by the batch size and the total number of training images. On the other hand, an epoch is defined as one complete pass through the entire training dataset. The number of epochs indicates how many times the model will see each training image during training. The relationship between training steps and epochs depends on the batch size and the total number of training images. Specifically, if we define spe as the number of training steps in one epoch, tnfi as total number of training images and b as batch size, we can calculate the number of training steps in one epoch as follows:

$$\text{spe} = \frac{\text{tnfi}}{b} \tag{3.11}$$

Then we can calculate the number of epochs by dividing the number of training steps by the number of training steps in one epoch. In this case, the number of training steps in one epoch was 22.125 and the number of training epochs was 90.

4 Results

For the purposes of this thesis, a subset of images taken from a Kaggle "Human Faces" image dataset [25] was used. The original dataset contained about 7200 images of PNG and JPEG format. Being limited by available resources, the subset contained only 225 PNG images. The subset was divided into training and test parts. 177 images were used for training of the autoencoder and the remaining 48 were used for testing the model. To broaden the data diversity and test the robustness and versatility of the autoencoder, another 4 hand drawn images of various artists ([26], [27], [28], [29]) were added to the evaluation dataset, which meant that the number of test images was 52, and the total number of images was 229. The autoencoder was trained using as NVIDIA TITAN Xp graphics card, with 11.91 gibibytes of total memory. Machine learning libraries included *tensorflow* (version 1.13) and *tensorflow_compression* (version 1.1). As mentioned before, there are 5 different numbers of filters per layer used: 192, 144, 96, 72 and 48. Respective λ^i are set to 2048, 1024, 512, 256 and 128. The higher the number of filters, the bigger the regularization expression, to avoid overfitting of the autoencoder. Adam optimizer was used for optimization of the training process, with the learning rate of 10^{-4}.

4.1 Used Metrics

In the evaluation, the used metrics include mean squared error (MSE), peak signal-to-noise ratio (PSNR), multi-scale structural similarity index measure (MS-SSIM) and information content in bits per pixel. These metrics were calculated for every image reconstruction. At the end of the evaluation process, we calculated the mean of these metrics for each of the number of filters per layer used. As expected, we got the best results for the highest number of filters. All of the used metrics will explained in more detail in the following subsections of this chapter. Subsections 4.1.1 and 4.1.2 were in large part

reproduced from [30], and the subsection 4.1.3 from [7].

4.1.1 Mean Squared Error

Mean squared error (MSE) is a widely used metric for assessing image quality. As a full-reference metric, lower MSE values indicate better image quality. MSE represents the second moment of the error, encompassing both the estimator's variance and bias. For an unbiased estimator, MSE equals the variance. It shares the same units as the square of the measured quantity, similar to variance. MSE also introduces the root-mean-square error (RMSE) or root-mean-square deviation (RMSD), often referred to as the standard deviation of the variance.

Additionally, MSE can be termed as mean squared deviation (MSD) of an estimator. An estimator, in this context, is a method for measuring an unobserved image quantity. MSE or MSD calculates the average of the squared errors, where the error is the difference between the estimator and the actual outcome. This metric considers the risk as the expected value of the squared error loss, also known as quadratic loss.

Mean squared error (MSE) between two images such as $g(x, y)$ and $\hat{g}(x, y)$ is defined as [31]

$$MSE = \frac{1}{MN} \sum_{n=1}^{M} \sum_{m=1}^{N} [\hat{g}(n, m) - g(n, m)]^2 , \tag{4.1}$$

where $g(n, m)$ represents the pixel value at position (n, m) in the first image and the $\hat{g}(n, m)$ represents the pixel value at position (n, m) in the second image. M is the number of rows (height) in the images and N is the number of columns (width) in the images. From the aforementioned Eq. 4.1, we can see that MSE is a representation of absolute error.

4.1.2 Peak Signal to Noise Ratio

Peak signal to noise ratio (PSNR) measures the ratio between the maximum possible signal power and the power of distorting noise affecting the signal quality. This ratio, expressed in decibels (dB), is calculated to compare two images. PSNR uses a logarithmic scale due to the wide dynamic range of signals, which varies between the smallest and largest possible values based on their quality.

PSNR is a widely used technique for assessing the quality of reconstructed images in lossy image compression codecs. In this context, the signal represents the original data, and the noise signifies the error introduced by compression or distortion. The PSNR is the approximate estimation to human perception of reconstruction quality compared to the compression codecs.

For image and video compression, PSNR values typically range from 30 to 50 dB for 8-bit data and from 60 to 80 dB for 16-bit data. In wireless transmission, an accepted quality loss range is around 20 to 25 dB [30].

Peak signal to noise ratio (PSNR) is defined by the formula

$$PSNR = 10\log_{10}\left(\frac{peakval^2}{MSE}\right). \tag{4.2}$$

Here, peakval (peak value) is the maximum possible pixel value of the image in the image data.

From Eq. 4.2, we can see that it is a representation of absolute error in dB.

4.1.3 Multi-scale Structural Similarity Index Measure

Structural Similarity Index Measure (SSIM) offers an alternative and complementary approach to image quality assessment by leveraging the human visual system's adaptation to structural information, and therefore a measure of structural similarity should be a good approximation of perceptual image quality [32]. However, the SSIM index algorithm introduced in [24] is a single-scale approach, which can be limiting because the appropriate scale depends on viewing conditions, such as display resolution and viewing distance. To address this limitation, a multi-scale structural similarity (MS-SSIM) method was introduced [7]. In MS-SSIM, parameters are calibrated to weigh the relative importance of different scales, providing a more robust assessment of image quality across various viewing conditions. By considering multiple scales, MS-SSIM enhances the accuracy and reliability of image quality measurements, making it a significant improvement over the single-scale SSIM method.

The MS-SSIM was calculated as it is specified in Eq. 3.10.

	Number of filters per layer				
	192	**144**	**96**	**72**	**48**
MSE	188.2	215.1	432.4	476.8	684.3
PSNR	26.47	25.81	22.71	22.28	20.62
MS-SSIM	0.9639	0.9592	0.9412	0.9334	0.9136

Table 4.1: Quantitative results of the autoencoder on the train subset.

	Number of filters per layer				
	192	**144**	**96**	**72**	**48**
MSE	215.5	243.2	521.6	574.4	773.0
PSNR	25.88	25.25	21.67	21.25	19.88
MS-SSIM	0.9582	0.9535	0.9276	0.9179	0.8976

Table 4.2: Quantitative results of the autoencoder on the test subset.

4.2 Quantitative Results

The quantitative output of the autoencoder is given in the form of three different previously described metrics - mean squared error (MSE), peak signal to noise ratio (PSNR) and multi-scale structural similarity index measure (MS-SSIM). The mean value of aforementioned metrics is calculated for every value of number of filters per layer used in the autoencoder. In the table 4.1 we can see the results for the train subset and in the table 4.2 we can see the results for the test subset.

In the figures 4.1 and 4.2 we can see the distribution of the MS-SSIM metric on the train and test subsets for each number of filters per layer used. The histograms of the train subset are located on the left, while the histograms of the test subset are located on the right of the figures.

4.3 Qualitative Results

A side-by-side overview of the original image and the reconstructed images is given, along with the number of filters per layer used and the multi-scale structural similarity index measure (MS-SSIM) metric, which is written beneath every reconstructed image. The goal was to show how the autoencoder performs on images of people of diverse ages, races and sexes. These results are presented in Figs. 4.3, 4.4, 4.5, 4.6, 4.7, 4.8 and 4.9.

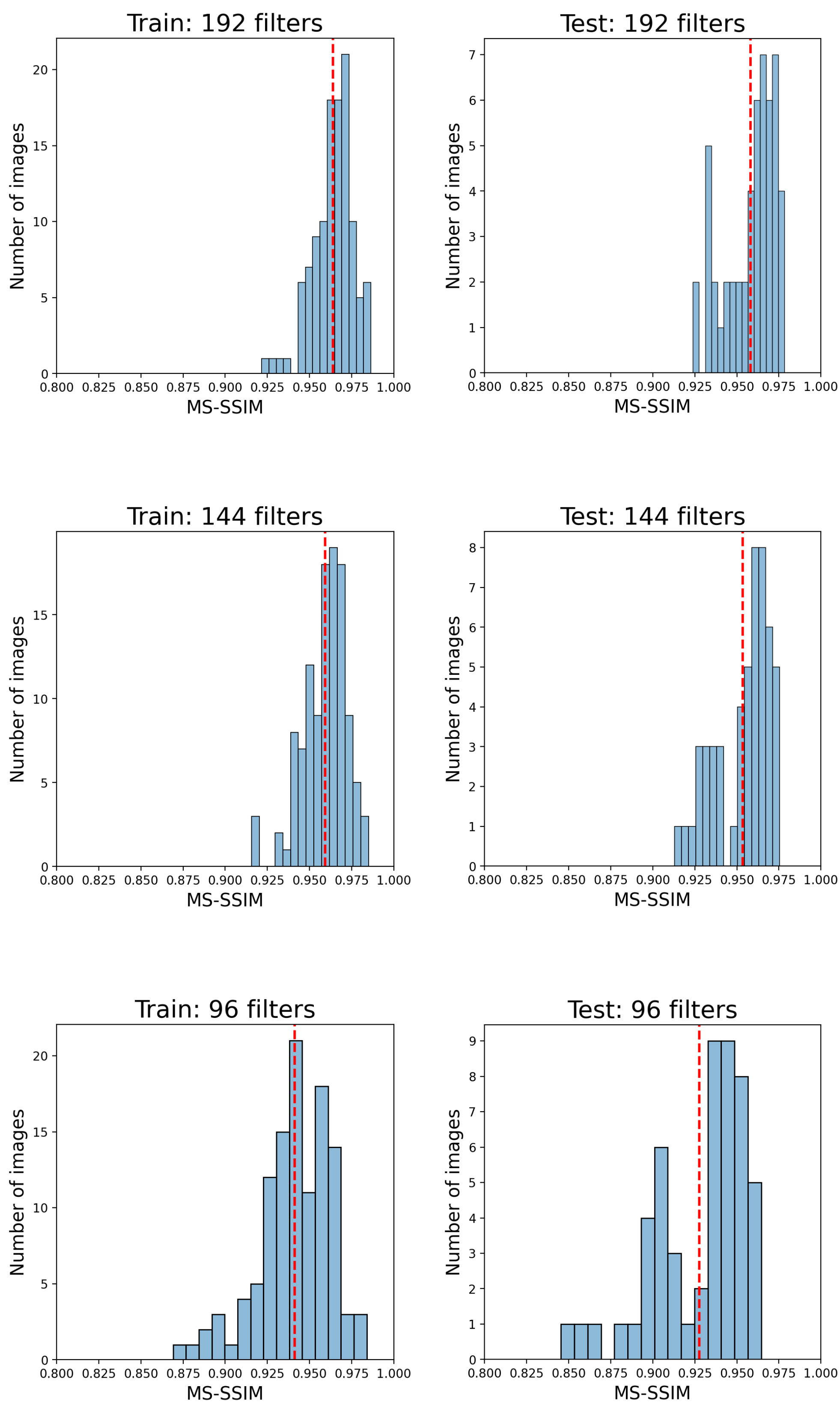

Figure 4.1: Histograms of the MS-SSIM metric for train (left) and test (right) subsets

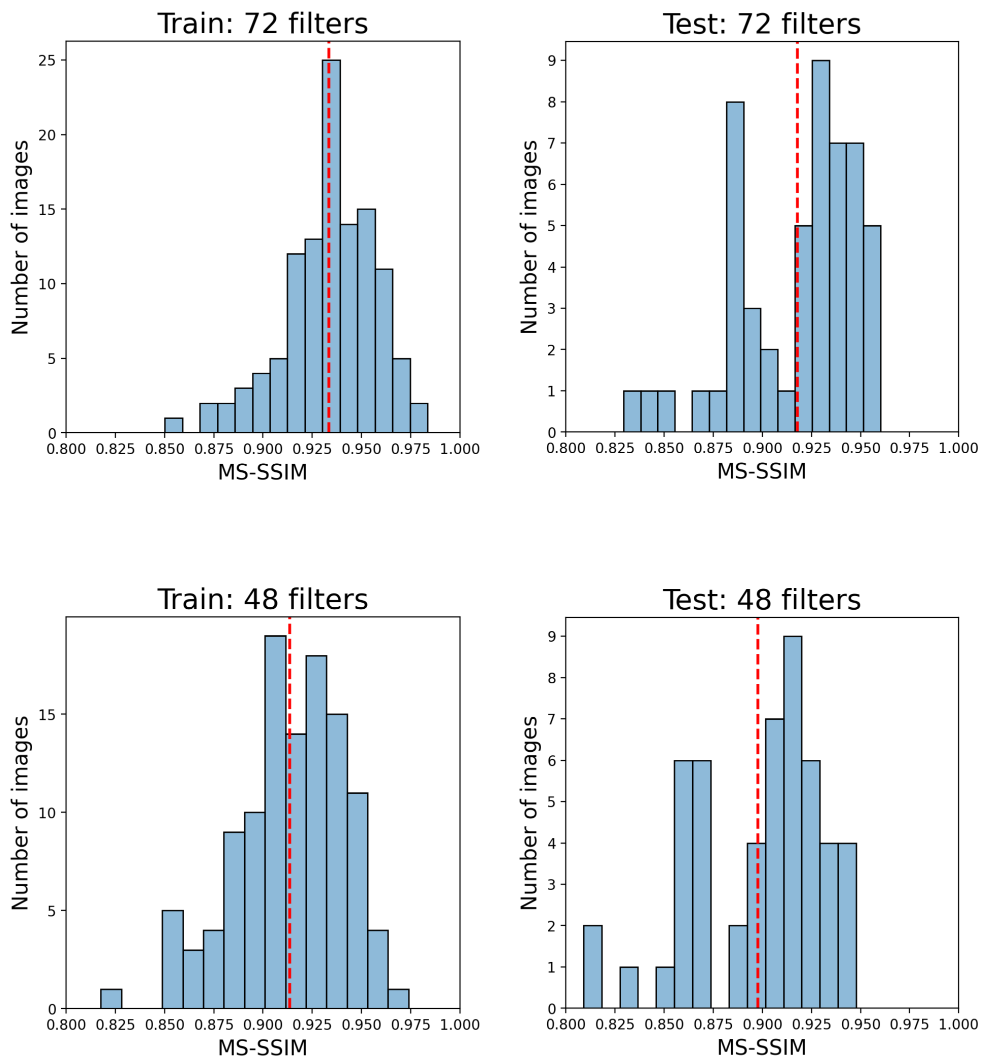

Figure 4.2: Histograms of the MS-SSIM metric for train (left) and test (right) subsets

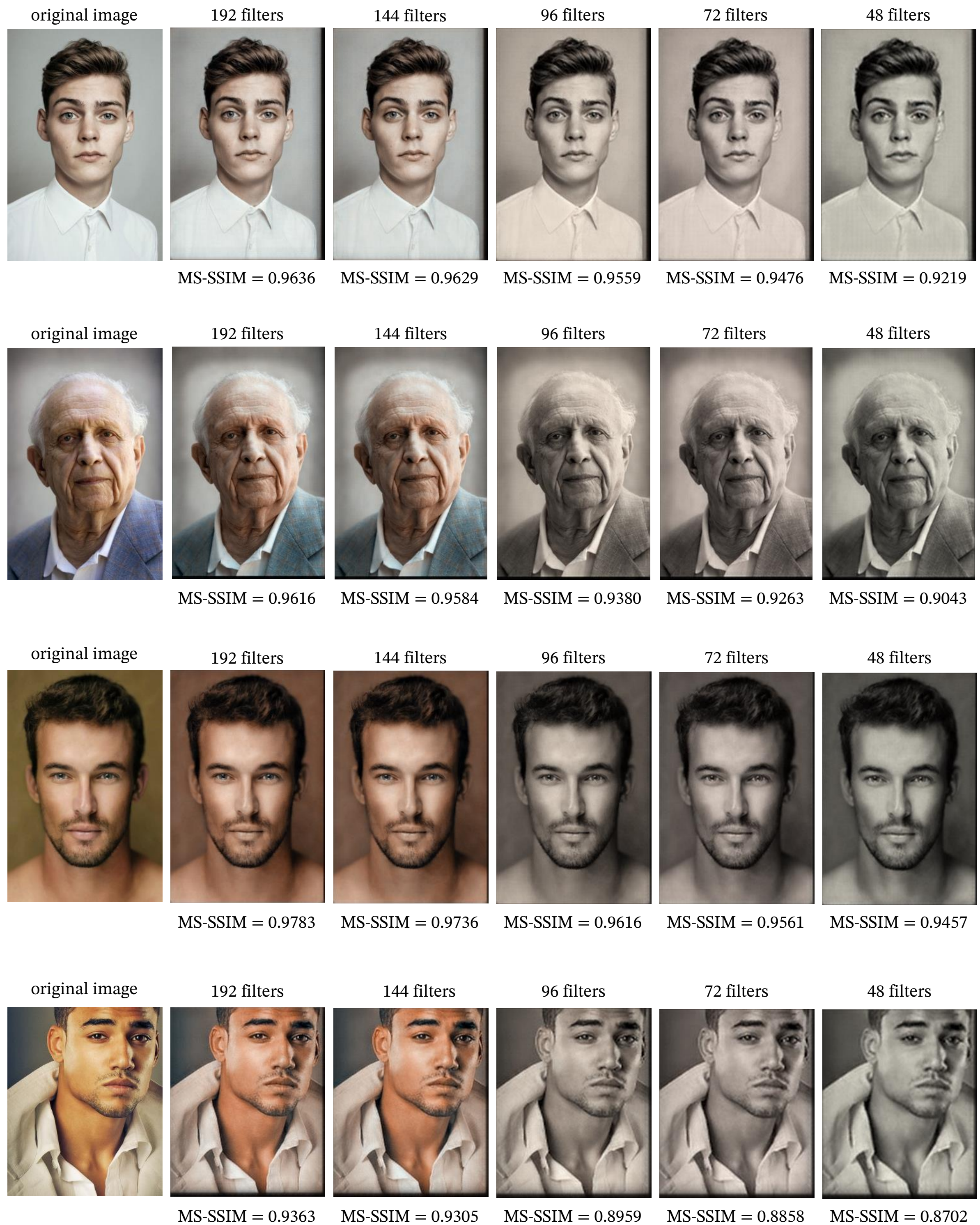

Figure 4.3: Original and reconstructed images of various male faces.

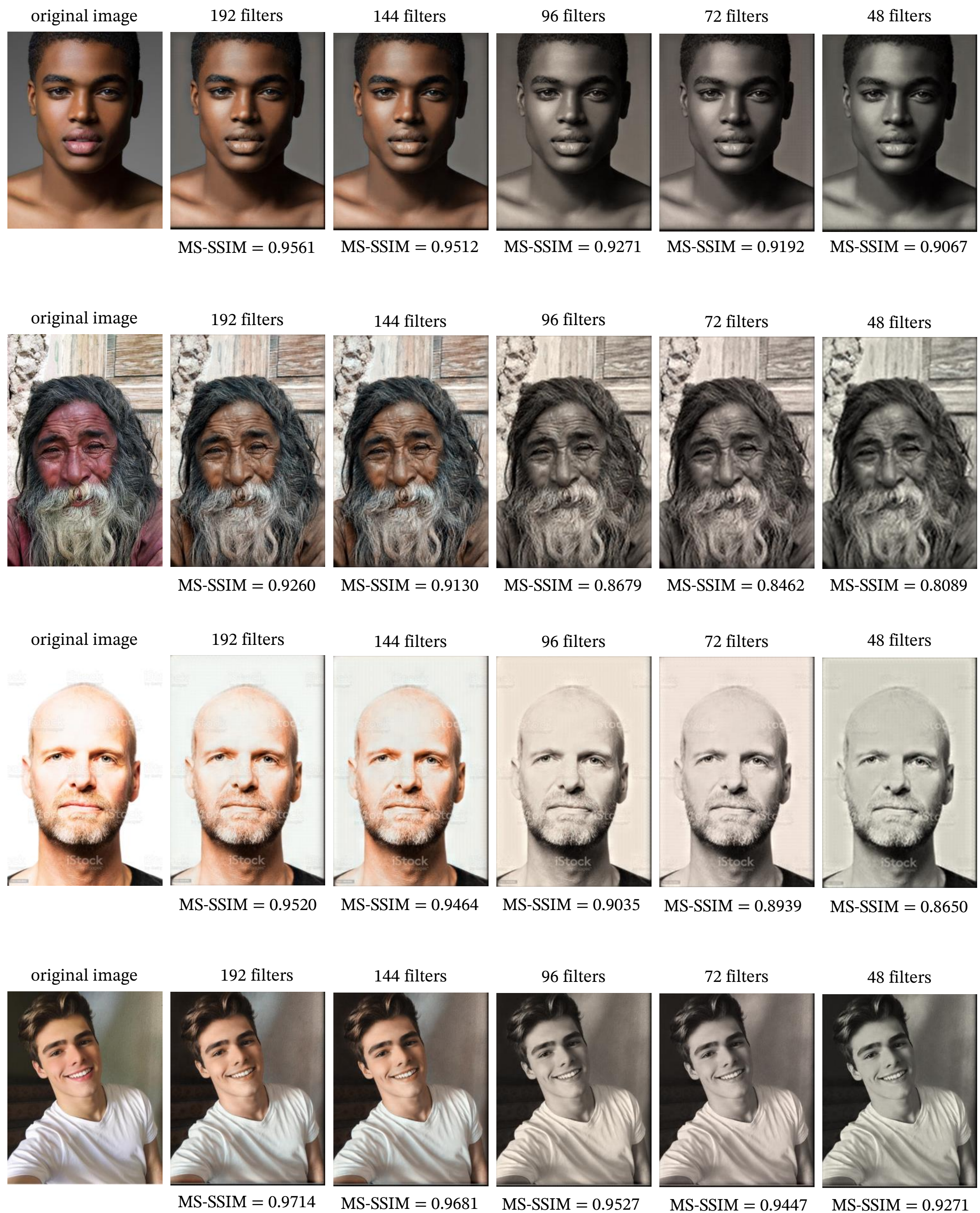

Figure 4.4: Original and reconstructed images of various male faces.

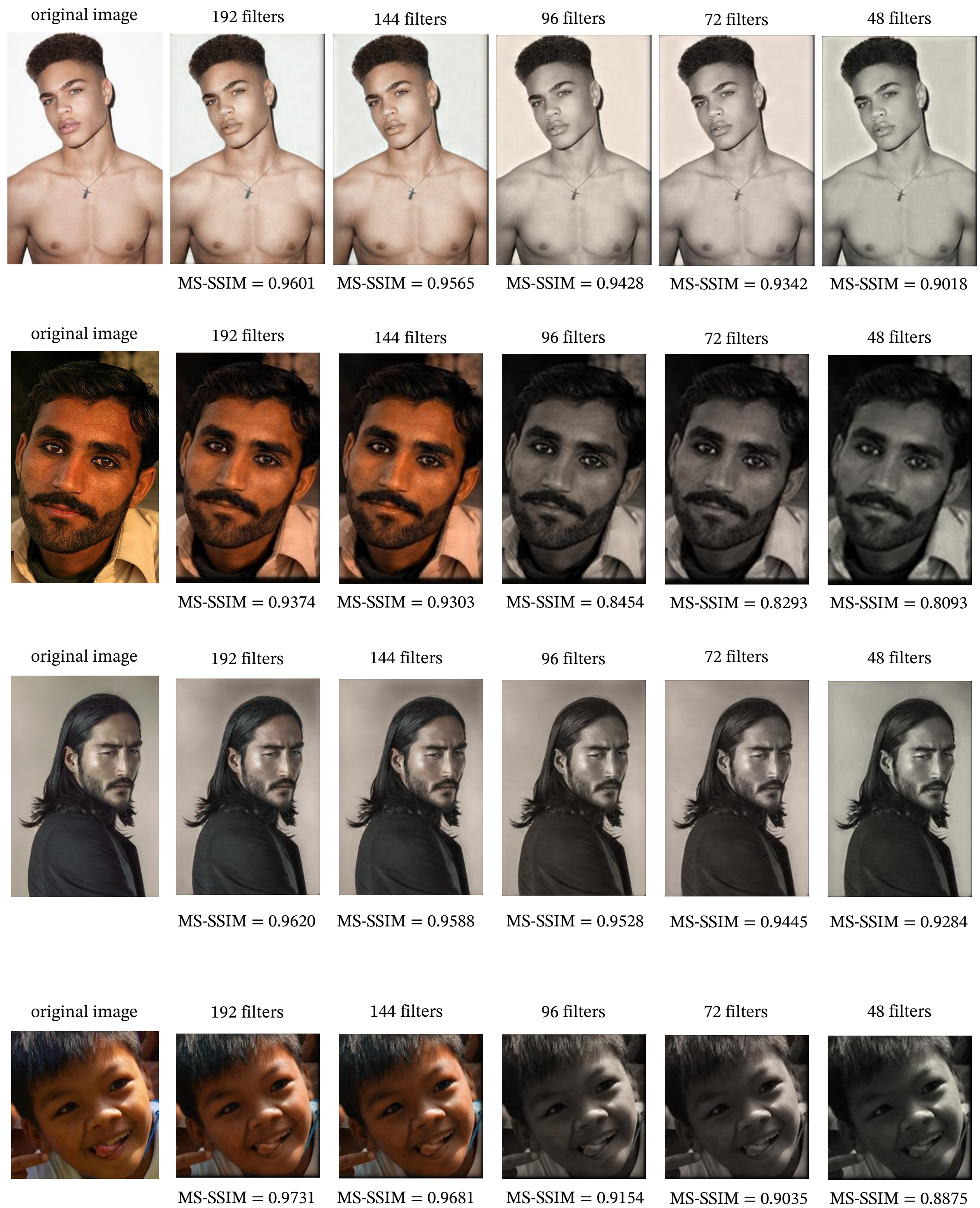

Figure 4.5: Original and reconstructed images of various male faces.

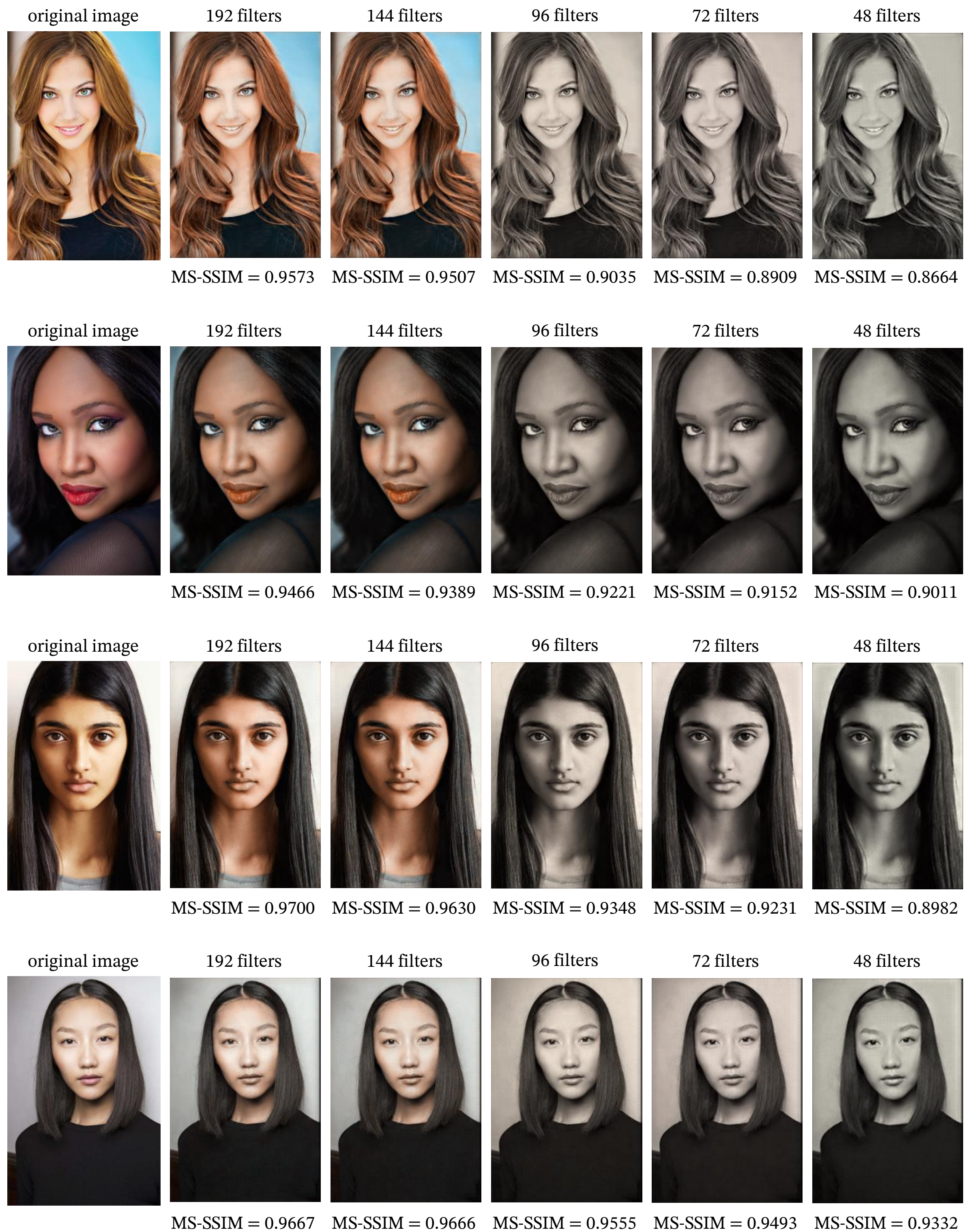

Figure 4.6: Original and reconstructed images of various female faces.

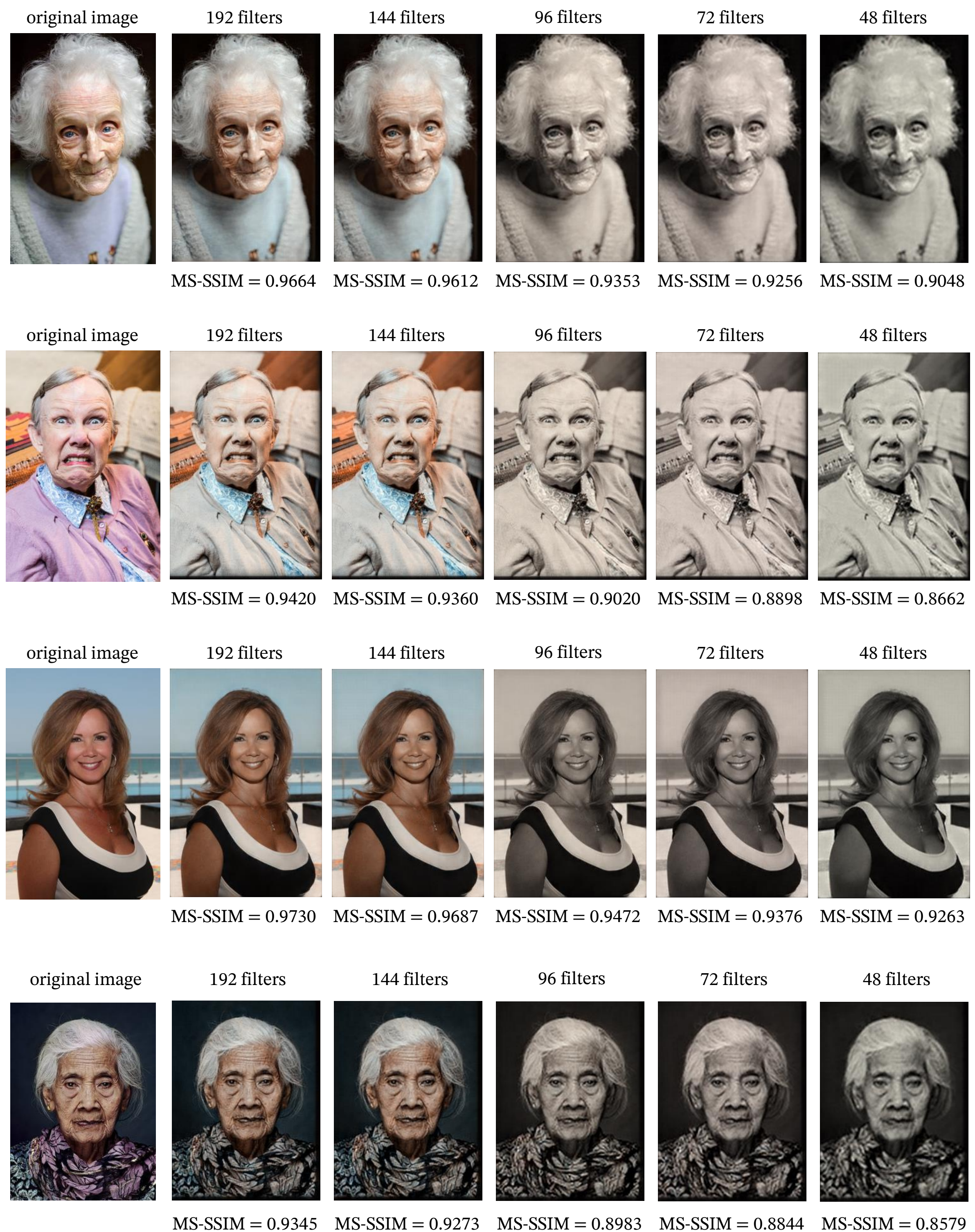

Figure 4.7: Original and reconstructed images of various female faces.

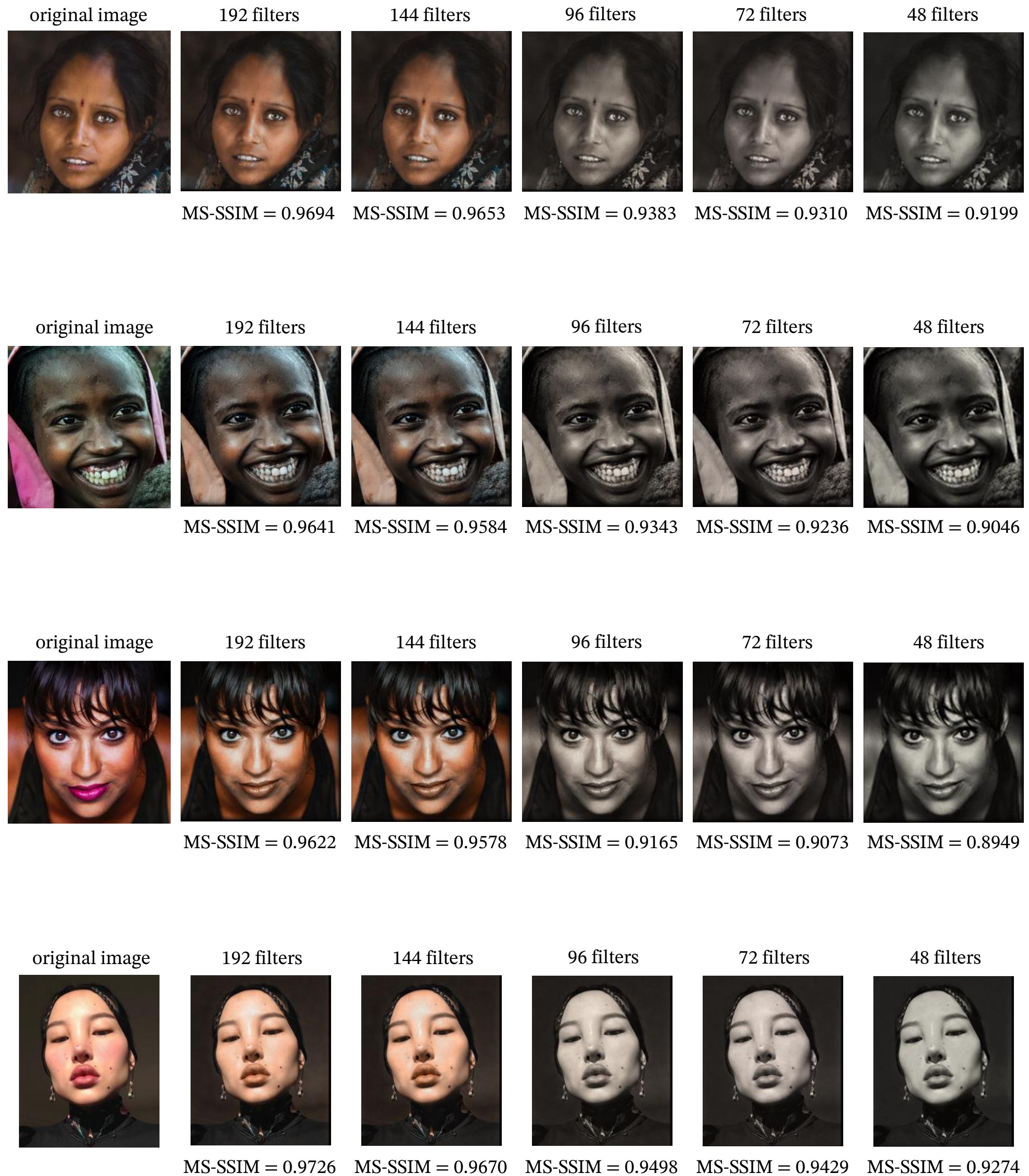

Figure 4.8: Original and reconstructed images of various female faces.

Figure 4.9: Original and reconstructed images of hand drawn male and female faces.

5 Discussion

The quantitative results presented in the tables 4.1 and 4.2 are in accordance with the expectations. It can clear that the number of filters in each layer impacts the performance of the autoencoder.

The MS-SSIM values, which measure the perceptual quality of the images, also decrease as the number of filters is reduced. For 192 filters, the MS-SSIM average value is 0.9639 on the training set and 0.9582 on the test set. With only 48 filters, these values drop to 0.9136 on the training set and 0.8976 on the test set. The positive aspect is that the difference in MS-SSIM mean values is not substantial and that they remain acceptable, even for the model with 48 filters per layer. For most of the images across all number of filters, the MS-SSIM values stay above 0.9, which correlates to good image reconstruction quality. Only for 72 and 48 filters per layer can we see an increase in number of images with MS-SSIM values below 0.9, which could be an indicator of poorer perceptual quality.

Overall, the trend across all metrics (MSE, PSNR, and MS-SSIM) clearly indicates that using a higher number of filters (192) in the autoencoder leads to better image reconstruction quality. Conversely, reducing the number of filters to 48 compromises the model's ability to reconstruct the images accurately. This suggests that the model with more filters has a greater capacity to capture and preserve the essential features of the images during the compression process.

As mentioned before, the difference in latent space size when using 48 filters per layer vs. using 192 filters was significant. For 48 filters per layer, the latent space was 4 times smaller in size, which is a good indicator of achieved compression.

As for the qualitative results presented in the figures 4.3, 4.4, 4.5, 4.6, 4.7, 4.8 and 4.9,

it can be clearly seen that the quality of the image reconstruction diminishes with the decrease in the number of filters per layer. The images reconstructed using 192 and 144 filters per layer are structurally similar to the original images in luminance, contrast and structure. There are no significant changes in image blur across all number of filters. Also, another positive aspect is that even when 48 filters per layer are used, we can evidently recognize the structural similarity of the reconstructed and the original images. In the autoencoder's architecture, the number of filters per layer impacts the ability to capture and reproduce the detailed features of the input images. The reconstructions with 192 and 144 filters per layer preserve the color information, while those with 96, 72 and 48 filters lose this information, resulting in grayscale images. Possible explanations of this could be the capacity of the model, size of the training dataset or missing of the chrominance parameter in the MS-SSIM metric.

With a higher number of filters per layer, the autoencoder has a greater capacity to learn and retain the complex patterns and features necessary to reconstruct the color information in the images. The additional filters provide more parameters and a richer representation of the input data. However, when the number of filters is reduced, the model's capacity to capture and retain detailed color information diminishes. Another factor is the size of the training data. If the model was trained on bigger image datasets, it could improve it's ability to reconstruct images.

Further improvements in model's ability to preserve the color information could be made by incorporating a metric which measures chrominance in the training loss function. One such metric is the chroma difference derived from the CIELAB color space [33]. The CIE has developed a uniform color space where the perceived color difference can be estimated between any two colors (L_1^*, a_1^*, b_1^*) and (L_2^*, a_2^*, b_2^*). Chroma difference is then calculated as

$$\Delta C_{ab}^* = C_{ab,2}^* - C_{ab,1}^*, \tag{5.1}$$

where

$$C_{ab}^* = \sqrt{(a^*)^2 + (b^*)^2}. \tag{5.2}$$

The training loss function could then be equal to

$$\lambda_1^i\,(1 - MS\text{-}SSIM(\mathbf{x}^i, \mathbf{r}^i)) + bpp^i + \lambda_2^i\,\Delta C^*_{ab}. \tag{5.3}$$

6 Conclusion

In this thesis, we investigated the effectiveness of slimmable compressive autoencoders for image compression, particularly focusing on their performance when trained using the multi-scale structural similarity index measure. Our experiments utilized a subset of human face images from a publicly available Kaggle “Human Faces” dataset [25], and we assessed the autoencoders with different numbers of filters per layer. The results indicate that the use of MS-SSIM as a training metric improves the perceptual quality of the reconstructed images compared to traditional metrics like mean squared error.

We observed that higher filter counts per layer generally led to better image reconstructions, both in terms of quantitative metrics and visual assessment. The use of 192, 144, 96, 72 and 48 filters per layer provided a comprehensive understanding of the trade-offs between model complexity and reconstruction quality.

Possible improvements of the model include increasing the number of filters per layer, training the model on a larger dataset and the inclusion of a metric which measures chrominance (e.g. chroma difference) in the training loss function.

Our findings suggest that slimmable compressive autoencoders hold substantial promise for practical image compression applications, offering a flexible and effective approach to balancing compression rate and image quality. Future work could explore the application of these autoencoders to different types of images and further refine the architectures to enhance performance.

References

[1] J. L. McClelland, D. E. Rumelhart, P. R. Group *et al.*, *Parallel distributed processing*. MIT press Cambridge, MA, 1986, vol. 2.

[2] P. Baldi, "Autoencoders, unsupervised learning, and deep architectures," in *Proceedings of ICML workshop on unsupervised and transfer learning*. JMLR Workshop and Conference Proceedings, 2012, pp. 37–49.

[3] G. E. Hinton and J. McClelland, *Learning representations by recirculation*, 1987.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[5] S. Dhawan, "A review of image compression and comparison of its algorithms," *International Journal of electronics & Communication technology*, vol. 2, no. 1, pp. 22–26, 2011.

[6] F. Yang, L. Herranz, Y. Cheng, and M. G. Mozerov, "Slimmable compressive autoencoders for practical neural image compression," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 4998–5007.

[7] Z. Wang, E. P. Simoncelli, and A. C. Bovik, "Multiscale structural similarity for image quality assessment," in *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, vol. 2. Ieee, 2003, pp. 1398–1402.

[8] A. Maćkiewicz and W. Ratajczak, "Principal components analysis (pca)," *Computers & Geosciences*, vol. 19, no. 3, pp. 303–342, 1993.

[9] G. N. Karagoz, A. Yazici, T. Dokeroglu, and A. Cosar, "Analysis of multiobjective algorithms for the classification of multi-label video datasets," *IEEE Access*, vol. 8, pp. 163 937–163 952, 2020.

[10] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: Explicit invariance during feature extraction," in *Proceedings of the 28th international conference on international conference on machine learning*, 2011, pp. 833–840.

[11] Y. Bengio, L. Yao, G. Alain, and P. Vincent, "Generalized denoising auto-encoders as generative models," *Advances in neural information processing systems*, vol. 26, 2013.

[12] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang, "Slimmable neural networks," *arXiv preprint arXiv:1812.08928*, 2018.

[13] J. Ballé, V. Laparra, and E. P. Simoncelli, "End-to-end optimized image compression," *arXiv preprint arXiv:1611.01704*, 2016.

[14] ——, "Density modeling of images using a generalized normalization transformation," *arXiv preprint arXiv:1511.06281*, 2015.

[15] R. H. Wiggins, H. C. Davidson, H. R. Harnsberger, J. R. Lauman, and P. A. Goede, "Image file formats: past, present, and future," *Radiographics*, vol. 21, no. 3, pp. 789–798, 2001.

[16] G. Vijayvargiya, S. Silakari, and R. Pandey, "A survey: various techniques of image compression," *arXiv preprint arXiv:1311.6877*, 2013.

[17] H. Bandyopadhyay, "Autoencoders in deep learning: Tutorial & use cases [2023]," https://www.v7labs.com/blog/autoencoders-guide#:~:text=Undercomplete%20Autoencoders,-An%20undercomplete%20autoencoder&text=The%20primary%20use%20of%20autoencoders,of%20the%20network%20when%20needed., accessed: 2023-05-01.

[18] Z. Cheng, H. Sun, M. Takeuchi, and J. Katto, "Deep convolutional autoencoder-based lossy image compression," in *2018 Picture Coding Symposium (PCS)*. IEEE,

2018, pp. 253–257.

[19] L. Zhou, C. Cai, Y. Gao, S. Su, and J. Wu, "Variational autoencoder for low bit-rate image compression," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, pp. 2617–2620.

[20] Y. Choi, M. El-Khamy, and J. Lee, "Variable rate deep image compression with a conditional autoencoder," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 3146–3154.

[21] L. Theis, W. Shi, A. Cunningham, and F. Huszár, "Lossy image compression with compressive autoencoders," in *International conference on learning representations*, 2022.

[22] F. Yang, L. Herranz, Y. Cheng, and M. G. Mozerov, "Slimmable compressive autoencoders for practical neural image compression," https://github.com/FireFYF/SlimCAE, 2021.

[23] P. Antoniadis, "Neural networks: Strided convolutions," https://www.baeldung.com/cs/neural-nets-strided-convolutions, accessed: 2024-05-16.

[24] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[25] A. Gupta, "Human faces," https://www.kaggle.com/datasets/ashwingupta3012/human-faces/data, accessed: 2024-05-01.

[26] Dreameraddict, "Portrait of a woman," https://in.pinterest.com/pin/617978380117829646/, accessed: 2024-05-15.

[27] V. Rio, "Portrait of Georges Brassens," https://www.pinterest.com/pin/34480753390639484/, accessed: 2024-05-15.

[28] Unknown, "Portrait of a man," https://www.pinterest.com/pin/34480753390639471/, accessed: 2024-05-15.

[29] H. Le Fur, "Portrait of Serge Gainsbourg," https://www.pinterest.com/pin/344807533390639471/, accessed: 2024-05-15.

[30] U. Sara, M. Akter, and M. S. Uddin, "Image quality assessment through fsim, ssim, mse and psnr—a comparative study," *Journal of Computer and Communications*, vol. 7, no. 3, pp. 8–18, 2019.

[31] J. Søgaard, L. Krasula, M. Shahid, D. Temel, K. Brunnström, and M. Razaak, "Applicability of existing objective metrics of perceptual quality for adaptive video streaming," *Electronic Imaging*, vol. 28, pp. 1–7, 2016.

[32] Z. Wang and A. C. Bovik, "A universal image quality index," *IEEE signal processing letters*, vol. 9, no. 3, pp. 81–84, 2002.

[33] J. Penczek, P. A. Boynton, and J. D. Splett, "Color error in the digital camera image capture process," *Journal of digital imaging*, vol. 27, pp. 182–191, 2014.

Abstract

Autoencoders for Image Compression

Karlo Kada

This thesis explores the use of slimmable compressive autoencoders for the compression of human face images. The thesis investigates the performance of autoencoders with varying numbers of filters per layer, focusing on training using the multi-scale structural similarity index measure (MS-SSIM) instead of traditional metrics like the mean squared error (MSE). By utilizing a subset of the "Human Faces" dataset published on Kaggle, the autoencoder's ability to produce high-quality image reconstructions is evaluated both quantitatively and qualitatively, with a particular emphasis on perceptual image quality. The findings demonstrate that MS-SSIM is an adequate metric for training autoencoders for image compression. Possible enhancement of the model could be the inclusion of a chrominance term in the training loss function to facilitate better color reproduction.

Keywords: autoencoders; image compression; convolutional neural networks

Sažetak

Autoenkoderi za kompresiju slike

Karlo Kada

Ovaj rad istražuje upotrebu stanjivajućih autoenkodera za kompresiju slika ljudskog lica. Rad istražuje performanse autoenkodera s različitim brojevima filtera po sloju, fokusirajući se na trening autoenkodera korištenjem više-razinskog indeksa strukturalne sličnosti (MS-SSIM) umjesto tradicionalnih metoda poput srednje kvadratne pogreške (MSE). Korištenjem testnog podskupa uzetog iz skupa "Human Faces" objavljenog na Kaggleu, evaluira se sposobnost autoenkodera da proizvede rekonstrukcije slika visoke kvalitete, s posebnim naglaskom na percipiranu kvalitetu slike. Rezultati pokazuju da je MS-SSIM adekvatna metrika za treniranje autoenkodera za kompresiju slika. Jedno od mogućih poboljšanja ovog modela je proširivanje funkcije gubitka za učenje modela mjerom krominancije kako bi se pospješila reprodukcija boje.

Ključne riječi: autoenkoderi; kompresija slike; konvolucijske neuronske mreže