(Geometry Aware) Deep Learning-based Omnidirectional Image Compression

Submitted on 17 Jan. 2020
Presented on 6 Feb. 2020
School of Engineering
Signal Processing Laboratory 4
École Polytechnique Fédérale de Lausanne
Master in Mechanical Engineering

By Yamin Sepehri

Prof. Pascal Frossard, Main Supervisor
Prof. Auke J. Ijspeert, Co-supervisor
Dr. Roberto Gerson de Albuquerque Azevedo, Supervisor
Dr. Francesca De Simone, External Expert of the Jury
Lausanne, EPFL, 2020

Abstract

Omnidirectional images are the spherical visual signals that provide a wide, 360°, view of a scene from a specific position. Such images are becoming increasingly popular in fields like virtual reality and robotics. Compared to conventional 2D images, the storage and badwidth requirements of omnidirectional signals are much higher, due to the specific nature of them. Thus, there is a need for image compression schemes to reduce the dedicated storage space of omnidirectional images.

Image compression algorithms can be broadly classified into two groups: lossless and lossy. Lossless schemes are able to reconstruct the exact original data but they cannot reduce the size beyond a specific criteria. Lossy methods are generally better solutions if they do not add a high visual distortion to the reconstructed image, as long as they provide a decent compression rate.

If a planar, lossy image compression scheme is applied on omnidirectional images, some problems show up. It is possible to apply a planar compression scheme on a projected version of a 360° image; however, in these projection schemes (such as equirectangular projection) the sampling rate is different in the poles and the center. Consequently, the filters of the planar compression schemes that do not consider this difference ends in suboptimal result and distortions in the reconstructed images.

Recently, with the success of deep neural networks in many image processing tasks, researchers began to use them for the image compression as well. In this study, we propose a deep learning-based method for the compression of omnidirectional images by combining some state of the art approaches from the deep learning-based image compression schemes and some special convolutional layers that take into account the geometry of the omnidirectional image. In comparison to the available methods, it is the first method that can be applied directly on the equirectangularly projected version of omnidirectional images and considers the geometry in the scheme and the layers themselves.

To propose this method, different geometry-aware convolutional layers have been tried. We exploited various methods of downsampling and upsampling, such as spherical pooling layers, strided or transposed convolutions, bilinear interpolation, and pixel shuffle. In the end, a method is proposed that benefits from specific spherical convolutional layers which contain sampling methods considering the geometry of omnidirectional images. The sampling positions differ in the different heights of the image based on the nature of the projected omnidirectional image. Additionally, as it benefits from an iterative training method that calculates the residual between the output and input and feeds it again to the network as input of the next iteration, it can provide different compression rates with just one pass of training. Finally, it benefits from a novel method of patching that is well-aligned with the spherical convolution layers and helps the method to run efficiently without a need for a high computational power. The model was compared with a similar architecture without spherical convolutions and spherical patching and showed some improvements. The architecture has been optimized and improved and it has the potential to compete with popular image compression schemes such as JPEG especially in terms of reconstructing the colors.

Contents

Abstract **2**

1 Introduction **5**

1.1 Omnidirectional Images . . . 5
1.2 Image Compression . . . 5
1.3 Classical Image Compression Methods . . . 7
1.3.1 Joint Photographic Experts Group (JPEG) . . . 7
1.3.2 Related Works . . . 9
1.4 Deep Learning-based Image compression methods . . . 9
1.4.1 Related Works . . . 10
1.5 Objective . . . 11
1.6 Thesis structure . . . 12

2 Theoretical Framework **13**

2.1 E nd-to-end Autoencoder . . . 13
2.2 Recurrent Neural Network-based Autoencoder . . . 14
2.3 Spherical Representation . . . 17
2.3.1 Convolution on Sphere . . . 17
2.3.2 Spherenet Approach . . . 18

3 Implementation and the Materials **21**

3.1 Spherical Layers . . . 21
3.1.1 SphereConv2d . . . 21
3.1.2 SphereMaxPool2d . . . 22
3.2 LSTM Layers . . . 23
3.2.1 SphereConvLSTM . . . 23
3.3 Upsampling Methods . . . 23
3.3.1 Strided SphereTransposedConv2d . . . 24
3.3.2 Bilinear Upsampling . . . 25
3.3.3 Pixel Shuffle . . . 25
3.4 Loss Functions and Metrics . . . 26
3.4.1 Mean Squared Error . . . 26
3.4.2 Weighted Mean Squared Error . . . 26
3.4.3 Weighted L1 Loss . . . 27
3.4.4 Peak Signal to Noise Ratio (PSNR) . . . 28
3.4.5 Weighted-to-Spherically-Uniform PSNR . . . 28

3.4.6 MS-SSIM 28
3.4.7 Weighted to Spherically Uniform SSIM 30
3.5 Viewports 30

4 Building the Model **32**
4.1 Version 1: Geometry Aware Recurrent Residual-based Trained Autoencoder 32
4.2 Version 2: Spherically Patched Geometery Aware Recurrent Residual-based Trained Autoencoder 39
4.2.1 SpherePatch: A Method to Perform Patching on the Omnidirectional Images. . . . 39
4.2.2 Version 2.1 and 2.2. 40

5 Performance Evaluation **44**
5.1 The Proposal 44
5.2 The Dataset 45
5.3 Results of Experiments 46

6 Conclusion **51**

7 Further Improvements **52**

8 Acknowledgments **52**

9 Bibliography **53**

10 Appendix A1: Version 0, Preliminary Study Based on end-to-end Autoencoders **59**

1 Introduction

1.1 Omnidirectional Images

Omnidirectional images are the spherical image signals that provide a wide, 360° view and they are mainly captured by specific type of cameras that use special lenses or multiple lenses, or by rotation of a normal camera in a static scene. They are becoming more popular and gaining more traction due to the growth of virtual reality goggles, robotics, and drones [1]. They provide a wide view of the scene in front of them and they are efficient in terms of data storage, in comparison to taking normal planar images in different directions. Companies like Samsung [2], Nikon [3], and Nokia [4] are active in the production of the omnidirectional cameras and the hardware, and technology giants such as Google [5] and Facebook [6] are developing the needed processing schemes.

In order to simplify processing tasks and for instance, reuse standard compression methods, a projection method is commonly used on 360° images. There are different projection methods available. In *equirectangular* projection, the coordinates of projected image show the longitude (ϕ) and latitude (θ) of a sphere, which respectively change from 0 to 2π and $-\frac{\pi}{2}$ to $\frac{\pi}{2}$. In *equal-area* projection, the vertical coordinates are multiplied by $cos(\theta)$ to reduce the shape distortion. Finally, in the cube-map projection, a cube surrounds the image sphere and the omnidirectional content is projected on each side of the box. Then the six generated surfaces can be arranged near each other in different ways [7]. The most popular method of projection is the equirectangular projection and it is used mainly in the available datasets.

Figure 1: A schematic view of equirectangular projection of the map of world. The circles show the deformations that happen in the surface area of different sections of image after the projection[8].

1.2 Image Compression

In signal processing, compression algorithms represent an encoded version of data which occupies less bits in comparison to the original representation [9].

Lossless compression algorithms reduce the unwanted redundancy whereas *lossy* methods might remove some of the information, which is less important, for the sake of occupied space reduction [10]. Compression algorithms can also be applied on image data. In image compression, a combination of both methods —lossless and lossy— are currently in use. In fact, most of the available algorithms, try

to provide the lowest possible bit-rate together with a low distortion that cannot be detected easily by human visual system [11]. The goal of an image compression algorithm is to optimize this rate-distortion trade-off. Distortion can be found using methods such as Mean Squared Error (MSE) and the rate of image compression can be calculated by measuring the generated size on storage in terms of bits per pixel (bpp) [12].

Lossy compression schemes were traditionally implemented using signal processing-based approaches, but after booming and success of deep neural networks in various image processing tasks, researchers developed lossy compression models based on them that are known as deep learning-based methods. Both of these image compression frameworks benefit from the following steps [11]:

- **Dividing into Blocks** The first idea that comes to the mind in order to quantize an image, is to perform the quantization for each pixel. This idea, however, will result in sub-optimal situation as it does not take into account the correlation among neighboring image pixels and the rate will become high. On the other hand, the compression can be done on block levels that take into account the structures among neighboring pixels. By allocating a controlled amount of pixels to each block, the rate can be selected more efficiently. The blocks can be non-overlapping (mostly in classical, signal processing-based methods) or can have overlap areas (mostly in the deep learning-based methods).

- **Transform Coding** A transformation is then applied on each block of data, in order to contain the important information and features in a few coefficients. Such a transformation should be easy to handle and efficient to use. In the classical methods, it is often based on Fourier Transform and in novel, deep learning based methods, it is based on different neural network layers such as convolutions.

- **Quantization** The core of lossy image compression is the quantization step and it is the place that the loss of information happens. Generally, in quantization, the range of a signal is mapped on a set of finite values. After this irreversible loss of data, a quantization error will be added to the data that can be calculated using a simple equation like this:

$$e[n] = Q(x[n]) - x[n] = \hat{x}[n] - x[n] \tag{1}$$

  where x is the input image and $\hat{x}$ is the reconstructed version of it. The method of quantization can be implemented in a more complex and smart way, based on the visual impact of each block on human vision.

- **Entropy Coding** The final, lossless step is to use methods based on special symbols to encode the data. These symbols are made such that the frequent values have the shortest possible symbols. In fact, the methods try to minimize the effort which is needed to encode a piece of information [11].

In the next two sections, the classical image compression based methods (which are used for comparison of the results) and deep learning based methods (which our model is one of them) are proposed briefly with examples.

1.3 Classical Image Compression Methods

The term "classical" image compression indicates that these method are based on traditional signal processing and most of them use approaches based on Fourier Transform for the transform coding step. Two of the most successful of such classical methods are Joint Photographic Experts Group (JPEG) that uses the Discrete Cosine Transform (DCT) [13] and JPEG 2000 (JP2) that benefits from Discrete Wavelet Transform (DWT) [14]. Here, we will have a brief look on JPEG algorithm which most of the new proposed methods are compared with it.

1.3.1 Joint Photographic Experts Group (JPEG)

JPEG was proposed on 1992 by Joint Photographic Experts Group. It has the ability to adjust the compressed image quality and the rate-distortion trade-off by changing the number of allocated bits for the quantization. Very few of the classical compression methods that were proposed afterwards got the popularity of JPEG. Here, we will have a brief look on the 4 different steps of JPEG image compression [15]:

- **Dividing into Blocks** JPEG performs the compression on 8×8 image blocks without overlap. In ultra-low compression rates, the so-called *block-artifacts* show-up. It is a type of distortion that happens due to the independent block processing and makes the blocks visible. (Figure 2)

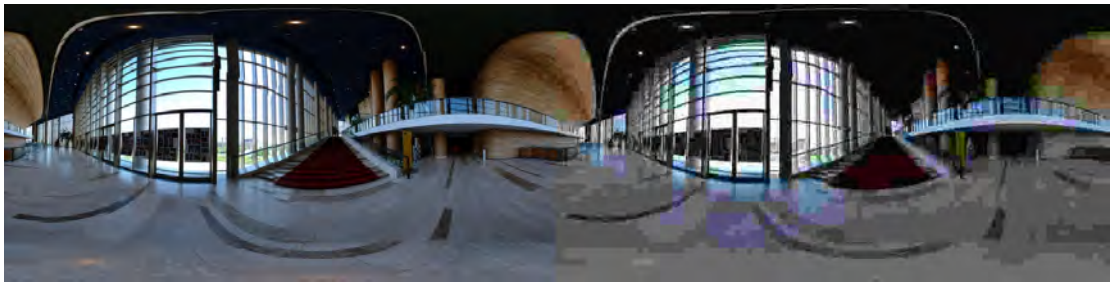

Figure 2: Blocking artifacts on an omnidirectional image from SUN360 dataset [16] processed by JPEG method in low rate. Left: Original Image with 380.7 kilobytes size, Right: JPEG compressed version of it with quality of 5 with 9.0 kilobytes size.

- **Transform Coding** JPEG applies 2D Discrete Cosine Transform on the image blocks. A set of basis vectors for this transform has been shown in Figure 3.

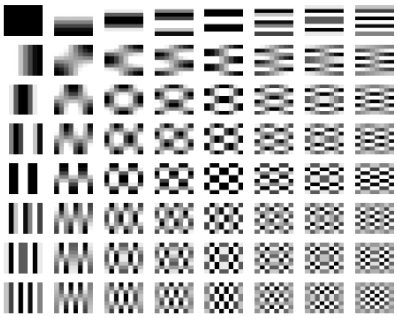

Figure 3: The 8×8 DCT blocks which are used by JPEG, from low frequency on top-left, to high frequency on bottom-right[17].

- **Quantization** JPEG uses smart, psychovisually-tuned quantization coefficients. In fact, after performing human experiments, the coefficients have been selected. Higher number of bits and smaller quantization intervals have been dedicated to the important points.
- **Entropy Coding** JPEG benefits from *prefix-free* symbolizing method that can parse a bit-stream sequentially without the need for look-ahead. Additionally, it benefits from Huffman algorithm to make an optimal code for a set of symbol probabilities. For more information, have a look on [18].

JPEG image compression scheme may also use sub-sampling methods to increase the rate of compression. This sub-sampling is done in the *YCbCr color space* which is different from the normal Red-Green-Blue (RGB) space that has three channels for these colors. YCbCr has one component for the luminance (Y), and two for the chrominance (Cr and Cb) that represent the color. In these type of sub-samplings, the chrominance sampling rate is lowered in comparison to luminance. These methods are shown by a code like $A : B : C$ where A shows the number of sampling for luminance in each row and it is used as the reference. B shows the number of chrominance sampling in each row with reference to A. And C shows the number of changes in chrominance sampling between the first two rows in comparison to A [19][20]. To clearly show it, Figure 4 shows a schematic view of $4 : 4 : 4$ method that uses the whole information and does not perform downsampling, and $4 : 2 : 0$ that performs the downsampling.

Figure 4: Left: a schematic view of $4 : 4 : 4$ method. Right: A schematic view of $4 : 2 : 0$ method that benefits from chroma subsampling [19][20].

1.3.2 Related Works

Omnidirectional image compression is a new and active research area. There are available methods that modify the traditional signal processing schemes for new geometries. De Simone et al.[21], for instance, showed that a shift happens in the frequency domain of equirectangularly projected images in comparison to normal images. Consequently, they generate a modified version of quantization table of JPEG scheme that considers this and they improved the quantized signal quality.

Rizkallah et al. [22] proposed a method based on a graph representation [23] that can be imposed on the sphere. Based on that representation, they implemented a partitioning strategy for the graph in order to make it efficiently running. Finally, they made a Graph Fourier Transform-based scheme for their method and in general, their method provides better results in comparison to DCT-based JPEG.

The adaptation of classical schemes for images based on new geometries is still an active topic and researchers are still working on it. For instance, one of the newest and impressive ones is a Catadioptric image compression scheme using a special shape-adavptive discrete cosine transform by Alouache et al. [24]. They mainly focused on reduction of block artifacts in methods like JPEG for catadioptric images by dividing the catadioptric image to the object neighborhoods and then applying the shape adaptive DCT on the these adapted neighborhood. Their results showed an improvement in comparison to JPEG and H264 [25] in terms of PSNR[12] and MS-SSIM metrics[26].

1.4 Deep Learning-based Image compression methods

The image compression world was dominated by Fourier Transform-based methods until the emerging and booming of neural networks. Convolutional Neural Networks (CNNs) brought a great achievement in image processing and computer vision tasks. For example, Krizhevsky et al. [27] CNN won the image classification challenge in 2012 with a pretty low error in comparison to the previous works and object detection methods improved surprisingly by Girshick et al. [28] in 2014. After all these happenings, researchers returned back to the old image compression problem and tried to use the deep learning as a tool for it [12]. Since 2015, thanks to the powerful computation systems and GPUs, deep learning-based compression methods demonstrated decent results and researchers validated the feasibility of such approach. Nowadays, researchers are trying to improve the results, simplify the approaches, and developing image compression methods for specific applications.

Currently, the deep learning image compression methods can be classified into two main groups: *deep tools* and *deep schemes* (see Figure 5). In the deep tools, the deep learning blocks are used inside a classical scheme in order to improve it. For example, intra-picture prediction can be used to find some similar structures between the different blocks inside the image for traditional methods. Cross-channel prediction can also be done using deep learning for color images. Additionally, some trained filters can be used to improve the appearance of the quantized image. Recently, some new deep tools work better even in comparison to the most powerful traditional schemes; however, we do not focus

on these methods in this study [12].

Deep schemes contain completely new schemes, with new deep learning-based transform coding, that perform the image compression using an end-to-end approach. Most of these methods are based on convolutional neural networks or recurrent networks. There are two main divisions in deep schemes: pixel probability models and autoencoder-based models [12].

The lossless entropy coding methods try to decrease the probability of each symbol of the image data down to Shannon's criteria [29]. The pixel probability models try to provide a method to find this probability. These methods's goal is to train a deep model, that finds the probability of each symbol (here the pixel), based on the context (the other pixels around) of that pixel [12]. There are methods like PixelRNN and PixelCNN [30] in this group that are working decently on natural image. The pixel probability methods are also not the main focus of this study.

The second division of image compression deep schemes are the famous autoencoders. Autoencoders are able to encode a piece of data to a latent space with lower dimension and reconstruct it again by a decoder. They provide the possibility of training the models in an unsupervised manner [31] and at first glance, they seem to be appealing for the image compression problem. However, as it has been mentioned before, the main goal of image compression schemes is to optimize the result on the rate-distortion trade-off and just providing an autoencoder that is able to reconstruct the image with low distortion and deformation is not enough. Indeed, the rate parameter should also be added to the autoencoder using some tricks. Additionally, it should be taken into account that the core step of image compression, the quantization step, is generally non-differential and a method is needed to model it. Almost all the different autoencoder image compression schemes are trying to provide, new and smart approaches to solve these two issues. Two of the famous autoencoders that had a high impact on this study will be reviewed on Chapter 2 [32].

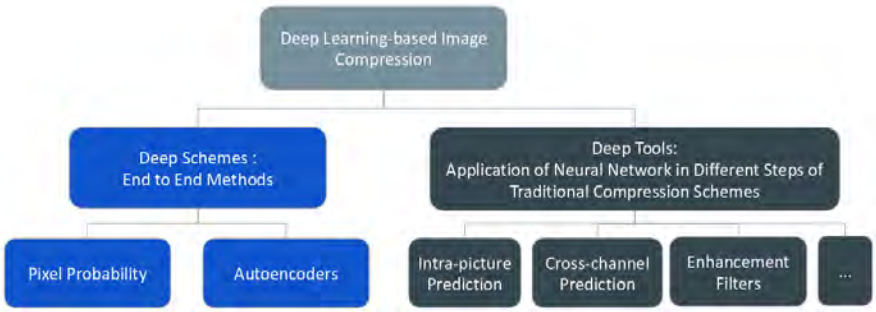

Figure 5: An overview of the different divisions in deep learning-based image compression[12].

1.4.1 Related Works

Compression of omnidirectional images using deep learning-based methods is still an underexplored topic, and most of the available methods for it are based on cube-map representation.

For example, Khasanova et al. [33], proposed a method based on graph-based convolutional filters that can be adopted to the geometry of the projected omnidirectional images. They have used the deep learning based-model for planar images which is presented by Ballé et al.[32] and changed the original convolutional layers to isotropic and anisotropic graph-based convolutional layers. They have shown that the method that is implemented with anisotropic graph-based convolutions that takes into account the geometry can provide similar PSNR quality in comparison to Ballé et al. model. However, they only implemented their method on cube-map projection of the 360° images and in fact, they did not apply it on the equirectangular projection. Additionally, although they substituted the planar convolution filters of Ballé et al. model with their graph-based convolutions, the results did not improve in comparison to their model. Finally, methods based on Ballé et al. model need to be retrained for every different rates of compression which is hard to use.

Moreover, Wang et al. [34] proposed a deep learning based method based on densely connected convolutional blocks to perform the image compression. They benefited from denseblock layers and convolutional layers in both encoder and decoder and they used MaxPool layers for the downsampling and depth to space layers for the upsampling. Although their method works well in comparison to the traditional models such as JPEG and deep learning based models such as Toderici et al.[35], they do not benefit from the layers that consider the omnidirectional nature of these images. In fact, they use the cube-map projection of images to train the network which by itself already makes them better and they just defined a specific loss for the cube-map projections in order to train. Additionally, although they did not elaborate about the method of considering the rate in their article, it seems like there is a need to retrain their model for every different rate.

Different from these two works, in this study, we propose a method that considers the geometry of omnidirectional images inside the deep learning-scheme and the convolutional layers. It is directly applied on the equirectangular representation of image, without the need to use projection methods like cube map. Additionally, it is able to provide different compression rates, in just one pass without the need of retraining it.

1.5 Objective

In this study, we develop a deep learning-based model that benefits from the unsupervised learning approach of autoencoders, to compress a set of omnidirectional images. For the first time, the method can be applied directly on the available equirectangularly projected images without a need for additional type of projections such as cube-map. The proposed method considers the specific geometry of these images inside the scheme itself by applying a geometry-aware sampling in the convolution layers. It is able to provide the possibility of changing the compression rate without a need for retraining, and should provide a low final distortion, based on the available metrics. Also, it should benefit from methods like patching which are well-aligned with the spherical layers to provide the ability of training it efficiently without a need for high computational power.

1.6 Thesis structure

Chapter 2 presents the theoretical background and details previous studies in the fields of deep learning-based planar image compression methods and methods spherical convolutions that have a direct effect on our proposed method. Chapter 3 discusses all the modules and materials that we have tried in the study. It contains the specific sampling and convolution layers, the loss functions and metrics that we implemented. Afterwards, in Chapter 4, we design and build our model and the architecture of autoencoder and provide the reasoning for the modules that we used. We test each of these versions, show the pros and cons, and then improve them step by step by modifying them. Finally, we select the final model based on these preliminary results, train it on more number of images, and compare it with JPEG method in Chapter 5 using different metrics. At the end, Chapter 7 proposes possible improvements to this work.

2 Theoretical Framework

In this chapter, the theoretical background that has been used in our proposed method is shown. First, we show two methods of autoencoder-based image compression for normal, planar images and elaborate them with examples of available models. End-to-end autoencoders (Section 2.1) and Recurrent neural network-based autoencoders (Section 2.2) use two different methods of considering the rate inside the autencoder and they benefit from different modules that can be useful for our omnidirectional model. Second, as we would like to implement specific convolutional layers that considers the geometry of omnidirectional images in our model, we discuss about the available spherical convolution approaches using specific available examples. In order to build our model (in Section 4), a selected spherical convolution method will be combined with a planar autoencoder to make our geometery-aware model for the compression of omnidirectional images.

2.1 E nd-to-end Autoencoder

The first type of autoencoder-based compression schemes are based on an end to end approach. These models are mainly based on a specific type of loss function that not only considers the distortion between the input and reconstructed images, but also takes into account the compression rate. In these methods, there is generally a need for retraining the model for different rates.

J. Ballé et al. [32] propose an end-to-end autoencoder-based method that benefits from 3 strided convolution layers in the encoder, and 3 transposed, output-strided convolution layers in the decoder. In the encoder, after each convolutional transform, a Generalized Divisive Normalization (GDN) nonlinearity is implemented and they benefit from the inverse version of GDN for the decoder. This layer, can be considered as a normalizaton similar to BatchNorm together with a nonlinearity and it has been used for other problems such as Gaussianization of images before this study [36]. (Figure 6)

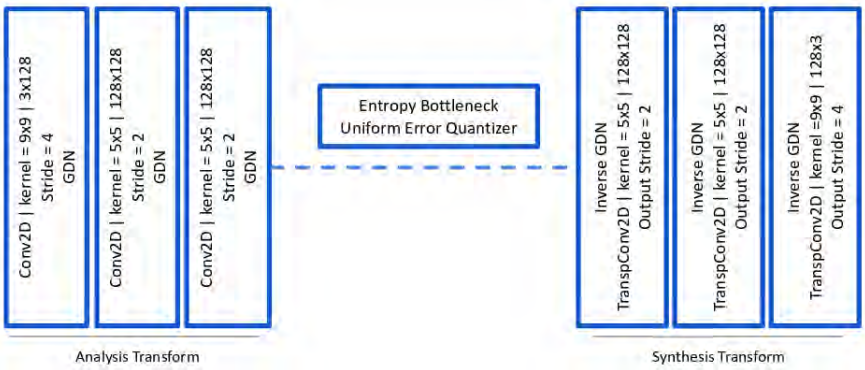

Figure 6: A schematic view of the main layers of Ballé et al. image compression scheme [32].

Ballé et al. [32] added a uniform noise instead of the usual quantization, in order to solve the non-differentiability problem of this step in the training phase. For the testing phase, a normal quantization

step has been exploited. It has been shown that this type of modeling for the quantization step can work well, in terms of the final results, after training the autoencoder.

In order to optimize the model for the rate-distortion trade-off, they defined a specific type of loss function that contains two different terms:

$$TotalLoss = -E(\log_2 P_q) + \lambda \cdot MSE(x, \hat{x}) \tag{2}$$

where the term on the left shows the rate, that is modeled by the entropy of the probability of quantization and the the right term shows the distortion between the input image (x) and the reconstructed image ($\hat{x}$) by taking the mean squared error. λ, is a parameter that should be set every time you want to train the model, and it controls the position of the result on the rate-distortion curve. With higher λ values, the reconstructed image will have a higher rate and will use more number of bits.

Although Ballé et al. [32] method provides very good results in terms of rate and distortion, it has some disadvantages as well. The training of the model is not easy since it should be performed for every different rate separately. Additionally, the number of epochs should be high in order to make sure that it learns all the different scale structures in different places of the image, due to the fact that it performs random cropping in order to patch down the images, as it is trained on 256x256 patches of image.

During some preliminary experiments, we tried to modify Ballé et al. model to be optimized for omnidirectional images, but these problems prevented us from achieving decent results. we will discuss more about it in the Chapter 4 and Appendix A1.

There are other available methods that can be trained once for all the needed rates. T. Dumas et al. [37] proposed a quantization independent method based on Ballé et al. that can be trained once for all the different rates. However, the method that we used for our omnidirectional compression model is mainly based on *Recurrent Neural Network-based Autoencoders* that is shown in the next section. In fact, although end-to-end autoencoder model could also be very useful in our proposal, in our point of view, the recurrent neural network-based autoencoder, is more intuitive, at least in terms of coding.

2.2 Recurrent Neural Network-based Autoencoder

Recurrent neural network-based autoencoder is an iterative approach that generates the loss between the input and the reconstructed images, and feed it again to the model as the input of the next iteration. In this iterative model, this autoencoder is able to be trained in one pass, for different needed image compression rates. In order to keep the information from previous iterations and propagate it to the next iterations, they mainly use recurrent neural network blocks that save the information as hidden states. One of the impressive examples of these models, are the work of Toderici et al.

Toderici et al. [35] proposed a deep learning based image compression method, that can issue different image compression rates, without a need for re-training the network again. They benefit from

an RNN-based enoder and decoder, and a binarizer for their method. They have tested different types of RNN blocks such as Long Short Term Memory (LSTM) [38], Gated Recurrent Units (GRU) [39], and associative LSTM [40] for their model and compared their results. In this study, we benefit from their LSTM-based model. During the training phase, a few iterations of encoding and decoding is done and each time, the residual between the input and and output is calculated using an L1 loss, defined as:

$$L1\ Loss_n = \frac{1}{B \times H \times W \times C} \sum_n |r_n| \tag{3}$$

where r_n shows the residual for each step that comes from output minus input for the first step and output minus residuals for the next steps. H and W are respectively the size of height and weight, and B and C are the batch size and the number of channels. From the back-propagation of the L1 loss, the network that looks like the equations below is trained.

$$\hat{x}_n = D_n(B(E_n(r_{n-1}))) \tag{4}$$

Where E_n shows the encoder (analysis transform), B shows the binarizer, and D_n shows the decoder (synthesis transform). The residuals are initialized and calculated as:

$$r_n = x - \hat{x}_n \tag{5}$$

$\hat{x}_0$ can be initialized by 0 or a low value in the beginning. After each iteration, the model is allowed to use more pixels and the rate is increased. By the aid of this trick, residual networks can provide the ability of training just once for the different rates which is good from the usage point of view. The architecture of the network looks like Figure 29:

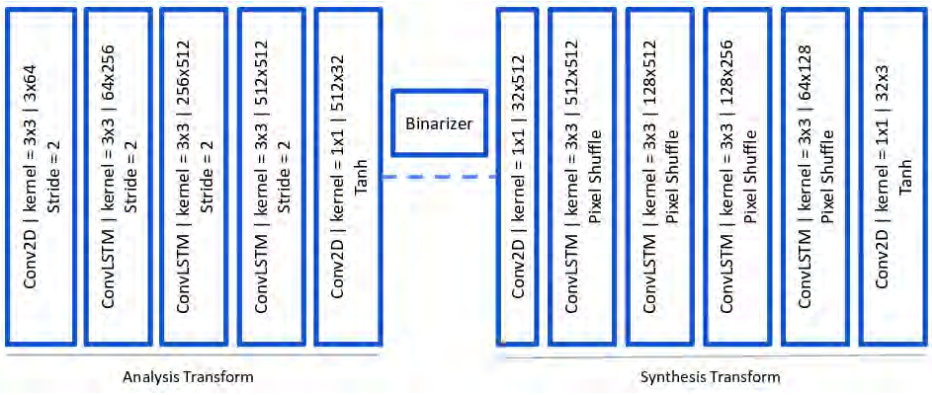

Figure 7: A schematic view of the main layers of Toderici et al. image compression scheme [35].

They benefit from an asymmetric autoencoder with three residual blocks in the encoder and 4 residual blocks in the decoder. There is an “input” convolution layer to increase the channels and there is a binarizer convolution layer to decrease the number of channels in order to decrease the rate of the

encoded latent space and to improve the model on rate-distortion trade-off. On the other side, there is a convolution layer to increase the number of channels before the decoder residual blocks and there is a final convolution layer with 1×1 kernel to decrease them again. There are just two non-linearity layers before the binarizer and at the end of decoder in addition to the built-in nonlinearities of the residual blocks. For the downsampling, they use strided convolutions and residual blocks but for the upsampling, depth to space, pixel-shuffle units have been used [41]. Inside each of the residual blocks, looks like the Figure 8:

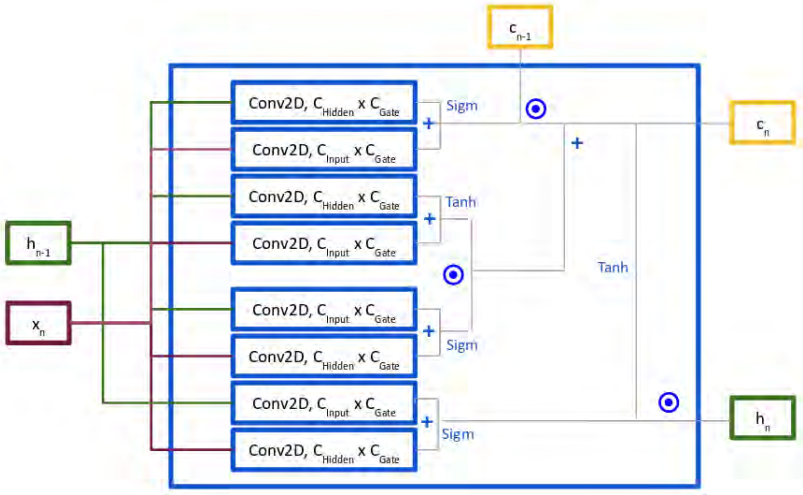

Figure 8: A schematic view of the LSTM cell that can be used in deep learning based image compression methods with inputs and outputs [42, 35].

In Figure 8, x_n shows the new input, h_{n-1} is the hidden state of the previous step and c_{n-1} is the cell state of the previous step. Sigmoid and Tanh non-linearities exist in the LSTM cell and the most important job it does is to affect the state of the previous step on the current step which is needed when you train a model based on the residuals[35].

The quantizer they use is a binarizer based on sign function. During the training phase, the forward pass is computed as [43]:

$$s[n] = (E_n(x_{n-1})) \tag{6}$$

$$s[(1 - s[n])/2 <= P] = 1, \;\; s[(1 - s[n])/2 > P] = -1 \tag{7}$$

where P is a uniform noise probability function with the same size as $s[n]$ and E_n shows the encoding transform. During the testing, the forward pass is just a normal sign function. In the backward pass, the gradient of input simply is set to be equal to gradient of output and the sign function is bypassed.

This model, is the base of our proposed model and we will come back to it in Chapter 4. In the next section, the topic will be changed and we will have a look on the problems of applying a convolutional

kernel on a spherical image and the available solutions.

2.3 Spherical Representation

2.3.1 Convolution on Sphere

The convolutional neural networks have had great achievements for the problems such as image classification and object detection related to the planar images. The main core of these architectures are the convolution filters which benefit from weight sharing and provide the ability to capture specific structures in the images on different places and with different sizes. Figure 9 illustrates the convolution operation.

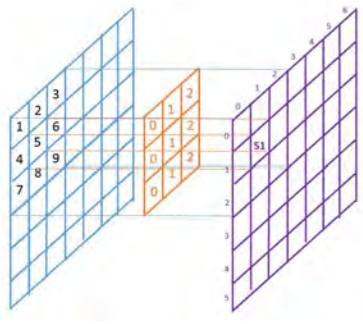

Figure 9: A schematic view of a 3×3 kernel applied on point [1,1] of a tensor.

As it is clear in Figure 9, a 3×3 kernel of planar convolution slides on input tensor and applies a dot product between the value of the input and the weights of it and it provides the so-called weight-sharing. For the color images, the convolution kernel usually has a depth equal to three as well. Now if a simple 3×3 convolution kernel is applied on equirectangularly projected omnidirectional image, the situation is illustrated in Figure 10.

Figure 10: Implementation of planar 3×3 kernel on an omniderectional image from SUN360[16] dataset. The main problem happens when the kernel is applied on the lower parts or upper parts of the image.

As you can see in Figure 10, if the kernel is applied on the middle part of an equirectangularly projected image, as the sampling rate is almost equal to the planar images, there will not be a significant problem for the weights of the kernel. However, when it is applied on the lower parts or higher parts of a projected omnidirectional image, as it takes samples more than enough from a position that has lower amount of information per area, the weights of the kernel will be affected more by these parts after the back-propagation and this will degrade the quality of model and results in suboptimal response.

Researchers implemented different methods to solve this issue. Khasanova et al. [33] proposed a method based on the graph signal processing framework [23] and used a weighted graph for the equirectangularly projected representation of the image. They selected the weights so that the effect of filter is as similar as possible on the different positions. Cohen et al. [44] proposed a method called spherical CNN that considers the rotation of the kernel in the model. They defined spherical cross-correlation (convolution) as the inner product between the input and a filter kernel which is rotated by $R \in SO(3)$ where $SO(3)$ is the set of rotations which can be shown as a 3×3 matrix in a three dimensional space.

Although the methods proposed by Khasanova et al. [33] and Cohen et al. [44] provide impressive results, in applied problems such as compression of omnidirectional images they might not be effective enough. The graph-based method of Khasanova et al. works well in low image resolutions and little graphs. For the images we have in our scope (e.g., the Sun360 dataset[16] which is a set of real scene images) it might not be easy to implement from a computational power point of view. On the other hand, the spherical CNNs of Cohen et al. benefits from a full rotational invariance; however, in real omnidirectional images that are captured by omnidirectional cameras (such as Samsung Gear 360[2] that benefits from two 8.4-megapixel image sensors) there is usually a main direction and a clear idea of up (usually the sky) and down (usually the floor). Such images statistics should also be explored by image compression methods.

In order to avoid these application-based problems, we have used a method which is called Spherenet [45] in our proposal. The next section details this approach.

2.3.2 Spherenet Approach

Proposed by Coors et al. [45], Spherenet is based on projecting the position of sampling locations, for a convolutional filter kernel. The model benefits from small batches tangent to the sphere (see Figure 11), and it it is mainly based on distortion invariance instead of deformation invariance.

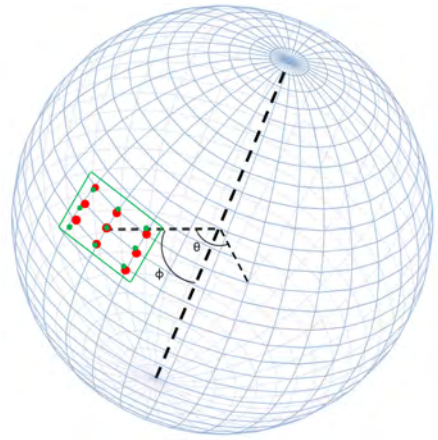

Figure 11: A kernel which is tangent to the sphere, is projected on the sphere surface using θ and ϕ. θ and ϕ are selected based on the height and weight of the input image [45].

In order to implement the Spherenet method, firstly, the increments for polar angle ϕ and the azimuthal angle θ should be selected [46]:

$$\phi_i = -\frac{(i+0.5)\cdot\pi}{h} + \frac{\pi}{2} \tag{8}$$

$$\theta_j = \frac{(j+0.5)\cdot 2\cdot\pi}{w} - \pi \tag{9}$$

where h is the height of the input image and w is the weight of it. i is the vertical increment and j is the horizontal increment. The distance between two increments can be found simply from the Equations 8 and 9 [46].

$$\Delta\phi = \frac{\pi}{h} \tag{10}$$

$$\Delta\theta = \frac{2\pi}{w} \tag{11}$$

Afterwards, in order to find the projected coordinates of the kernel, gnomonic projection [47] based on the middle point (that is the intersection point of the kernel and the sphere surface) is applied. The matrix that is able to perform this transformation is:

$$I^{x,y}_{\theta,\phi} = \begin{bmatrix} (-tan(\Delta\theta), (1/cos(\Delta\theta))\cdot tan(\Delta\phi)) & (0, tan(\Delta\phi)) & (tan(\Delta\theta), 1/cos(\Delta\theta)\cdot tan(\Delta\phi)) \\ (-tan(\Delta\theta), 0) & (0,0) & (tan(\Delta\theta), 0) \\ (-tan(\Delta\theta), -(1/cos(\Delta\theta))\cdot tan(\Delta\phi)) & (0, -tan(\Delta\phi)) & (tan(\Delta\theta), -(1/cos(\Delta\theta))\cdot tan(\Delta\phi)) \end{bmatrix} \tag{12}$$

This matrix shows the position of the 9 sampling points of kernel on the real, projected input image for x and y of each point. The radius of the projected kernel can be found from the result of the above transform using the next equation:

$$\rho = \sqrt{x^2 + y^2} \tag{13}$$

And the ϕ and θ of each pixel of the image can be found using the inverse gnomonic transform [47]:

$$\phi(x, y) = \sin^{-1}(\cos(\tan^{-1}(\rho)\sin(\phi_i) + \frac{y\sin(\tan^{-1}(\rho))\cos(\phi_i)}{\rho}) \tag{14}$$

$$\theta(x, y) = \theta_i + \tan^{-1}(\frac{x\sin(\tan^{-1}(\rho))}{\rho\cos(\phi_i)\cos(\tan^{-1}(\rho)) - y\sin(\phi_i)\sin(\tan^{-1}(\rho))}) \tag{15}$$

In addition to the projection of kernel sampling points using the above equations, Spherenet adds another feature to the normal convolution. In the normal planar convolutions, when the kernel reaches the boundaries of the image and a part of it may go outside, zero padding is the solution that is in use. However, Spherenet exploits the fact that omnidirectional images cover the whole scene. Consequently, if the kernel reaches the right boundary of the image, samples from the left boundary are used for the kernel that cannot be covered completely inside the image. On the other hand, when the kernel goes to the top or bottom, again it exploits features of $360°$ images and more samples are used from the top or bottom instead of naive padding (Figure 12).

Figure 12: When the kernel reaches the boundaries, instead of naive zero-padding, Spherenet benefits from the features of omnidirectional images and use the samples from the other side [45].

In general, we implement the effective sampling method of Spherenet in a recurrent neural networks to propose our own model. In fact, this model combines the power of training on residuals, with the power geometry-aware sampling. In order to this, different layers of a recurrent neural network such as LSTM cell and convolution should be changed by spherical version of them. The implementation of these spherical layers have some important points and details that will be discussed in the next chapter. Additionally, the next chapter will cover the different modules that we tried, like the different methods of upsampling, in order to make the autoencoder more optimized. Generally, it covers the implementation of all modules and materials that have been used in our model.

3 Implementation and the Materials

This chapter details the implementation of the layers and functions that we use in our method or for performance evaluation. Firstly, it contains implementation details about the modules that have been tried to build our own model containing spherical (Section 3.1) and LSTM (Section 3.2) layers, and different upsampling methods (Section 3.3). The spherical convolution and LSTM modules are important in our model since they provide the possibility of sampling on the sphere and taking into account the geometry. About the upsampling methods, they help to reconstruct the image from the compressed latent space and there are different types of them which are used in the state of art autoencoders. We try them to find the best one with lowest distortion for our spherical model. Because of this, the details about the implementation of some of the upsampling methods which have been tried are described here.

After that, we show the evaluation methods that we tried as loss functions of our autoencoder, or as the metrics to compare the results (Section 3.4). This part contains details about L1 and L2 functions and the spherically weighted version of them which have been used as loss functions, and PSNR, SSIM, and the weighted version of them that have been used as metrics. The spherical metrics, provide the possibility of evaluating our reconstructed images considering the sampling rate difference in the different positions of the equirectangularly projected image. Finally, we show the *viewports* method that will also be used in our metrics (Section 3.5). This is a method that provides the possibility of using the planar metrics, on omnidirectional images.

3.1 Spherical Layers

In this section we will have a look on the spherical layers that we have used in our model. The layers are based on the SphereNet article by Coors et al. [45] and the "SphereNet-pytorch" implementation by C. W. Hsiao [46].

3.1.1 SphereConv2d

The SphereConv2d layer has two different parts. First, there is a sampling part based on the gnomonic projection [47] of Spherenet. In this part, for each pixel of the input image, a 3×3 tensor is created. Then, the value of these generated points are found using the neighbor points. Nearest Neighbor (NN) approach or the Bilinear [48] interpolation approach can be used to find these values. The NN approach works faster in comparison to the bilinear method, but the correct selection of the neighbor point depends on the preciseness of the sampling positions that we selected. On the other hand, bilinear approach needs more computational power and uses more points, but in the image compression application, as it performs a type of weighted linear averaging on different signal points, it might result in a blurred view of the reconstructed image (see Figure 15) [45][46]. We tested both methods and show them in the Chapter 4.

After performing the sampling and interpolation, the total size of the signal will be changed from $w \times h$ to $3w \times 3h$. The next step is to perform a normal convolution that can be applied based on the built-in neural network functions of Tensorflow [49] or PyTorch [50] API frameworks. The point that should be noticed here is that the stride of this convolution should be equal to the size of generated matrices by the Spherenet sampling (in our model, it is 3×3.). In fact, if a stride is required in the model (which is actually the case in the image compression problem and generally all the autoencoders.), it should be implemented in the sampling step. Here is the place that one of the disadvantages of Spherenet method for this application shows up. If the stride is applied on the sampling and before the convolution, there will be loss of the data and it might have effect in the images which are reconstructed by the autoencoder. In fact, when a 3×3 kernel of a normal, planar convolution is applied, if the stride is equal to or less than 3, there is still no loss of information and all the pixels of the image are used. However, in the Spherenet implementation, due to the fact that the stride is implemented before the convolution in the sampling step, there will be loss of data. We will come back to this in the next chapters. The SphereConv2d layer has been tried in the different versions of our model (see Chapter 4).

3.1.2 SphereMaxPool2d

MaxPool is a layer that finds the maximum value of the blocks which has the same size as its kernel. (Figure 13). It has been reported that MaxPooling layer adds pseudo-invariance to deformation and because of this, it is popular especially in image classification deep learning schemes [51].

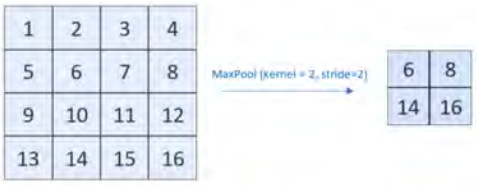

Figure 13: $2x2$ kernel Maxpool layer is applied with a stride equal to 2.

One of the other layers which is proposed by Coors et al.[45] is the SphereMaxPool2d. The idea is the same. Firstly, a 3×3 grid is generated for each pixel. Then using a type of interpolation (Nearest Neighbor or Bilinear) the value of those points will be found. Afterwards, a normal, planar MaxPool layer will be applied with a kernel size and stride equal to the size of the generated grid (here 3×3). Again, if a different stride should be applied on the image, it will be implemented in the grid generation step and there will be some loss of information. It is worth mentioning that as a $3w \times 3h$ tensor is generated here again and the MaxPool is applied on it, it is much slower in comparison to the normal, planar MaxPool layers [45][46].

3.2 LSTM Layers

This section is dedicated to the spherical Recurrent, LSTM cells that we have benefited from it in some of our models. The implementation of this layer is based on the work of Toderici et al. and Coors et al. [35] [45].

3.2.1 SphereConvLSTM

The LSTM layers are made mainly for the task of natural language processing. In the article of Toderici et al., they have used ConvLSTM cells that are using convolution inside an LSTM. In the Pytorch implementation of 1zb [43], the input state and hidden state are sent to different convolution layers and they generate gate channels which have a number of channels equal to 4 times the input. Then, the gate channels are separated into 4 chunks and, by applying nonlinearities, the new states are generated according to [51][42]:

$$f_n = sigmoid(W_{(x,f)}x_n + W_{(h,f)}h_{n-1} + b_{(f)}) \tag{16}$$

$$i_n = sigmoid(W_{(x,i)}x_n + W_{(h,i)}h_{n-1} + b_{(i)}) \tag{17}$$

$$g_n = tanh(W_{(x,c)}x_n + W_{(h,c)}h_{n-1} + b_{(c)}) \tag{18}$$

$$c_n = f_n \odot c_{n-1} + i_n \odot g_n \tag{19}$$

$$o_n = sigmoid(W_{(x,o)}x_n + W_{(h,i)}h_{n-1} + b_{(o)}) \tag{20}$$

$$h_n = o_n \odot \tanh(c_n) \tag{21}$$

where h, c, and x show the hidden state, the cell state, and the input, respectively. F corresponds to the forget-gate, i corresponds to input gate, and o corresponds to the output gate, all three are generated by sigmoid nonlinearity after convolution. Each of the Ws shows weight of a convolution filter and bs show the biases. In fact, there are 8 different convolutions in each LSTM cell that makes it computationally expensive. The most important point about LSTM cell is providing the effect of the previous iterations to the current iteration which is important when you are training a network on residuals.

The SphereConvLSTM cell is made when all these 8 convolutions are replaced by Spherical convolutions. In our work, we used Spherenet convolutions with bias and 3×3 kernel for the SphereConvLSTM. This layer provides the power of previous iterations together with the power of sampling on the sphere for our image compression model.

3.3 Upsampling Methods

In this project, we would like to propose a method based on the autoencoders for image compression. In image compression autoencoders, (like the ones that have been showed in Section 2.1 and 2.2) the

image is firstly mapped and downsampled to a latent space (using strided convolution or MaxPool) in order to achieve a low rate for compression, and then it should be upsampled again in order to reconstruct the omnidirectional image. From the available methods, we try to find the best ones for our autoencoder. Here we show the methods we have tried for upsampling:

3.3.1 Strided SphereTransposedConv2d

In autoencoders for planar data, a method which is in use for upsampling is to perform Strided Transposed Convolutions. Transposed convolutions are made by transposing the weight tensor and they can be looked as weighted sum of transposed kernels. If there is an input image with a size $W \times H$ and there is transposed convolution kernel with a size of $w \times h$ and finally a stride of $s_w \times s_h$, the output image will have a size of $[s_w(W-1)+w] \times [s_h(H-1)+h]$ [51] :

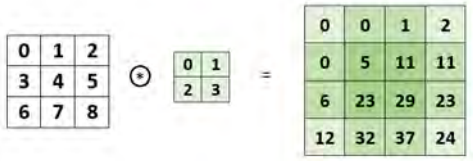

Figure 14: The result of transposed convolution on a 3×3 tensor by a 2×2 kernel with stride$=1\times1$. The color of the blocks in the output shows the number of times that the transposed convolution affected them.

The main problem of the transposed covolution is clear on the Figure 14. Due to the fact that the number of times that the kernel meets each point is different, if you apply a transposed convolution on an image data, there will be grid-structure anomalies in the final reconstructed image. This grid structute anomaly is more severe when a strided transposed convolution is implemented. This is the main reason that the strided transposed convolutions are not popular in many of the state of art reconstruction autoencoders [51]. They are more in use without stride and together with an interpolation method. We come back to this in the next subsection.

When it comes to a spherical transposed convolution layer, new problems show up. In Section 3.1, there has been shown that in the spherical convolutions, firstly a sampling and an interpolation are done and then a normal planar convolution will be applied there. The main trick was to apply the normal planar convolution with a stride equal to the size of the generated grid in order to use fast, built in convolutional functions of neural network coding API frameworks and to avoid the for loops which are awfully slow. However, this trick cannot be applied in the transposed spherical convolutions. Here, firstly a transposed planar convolution with a kernel equal to the size of the grid should be applied. After that, the generated pixels should be sent to the correct places according to the omnidirectional sampling. However, this second step cannot be done by simple tricks and it needs loops in the codes which are slow. Our trials for this layer did not end up with satisfying speed and results and it made

the training time long. At the end, we have decided not to put this layer inside our autoencoder and to use the more popular methods that provide better results.

3.3.2 Bilinear Upsampling

Another method, that is considered as an alternative to strided transposed convoluvition is the bilinear upsampling. Bilinear upsampling is a method to find the value of the points which are inside 4 other points using a linear estimation[48]:

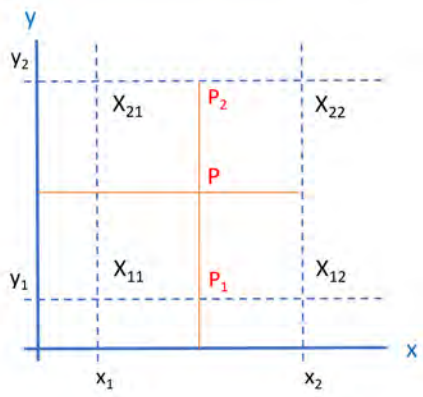

Figure 15: Bilinear upsampling method in order to find the value of point P [48].

When the value of points x_{11}, x_{12}, x_{21}, and x_{22} of an image are known, the value of point P can be found easily using the middle values of P_1 and P_2:

$$P = \frac{1}{(x_2 - x_1)(y_2 - y_1)} [x_2 - x, x - x_1] \begin{bmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix} \tag{22}$$

It is better to apply bilinear upsampling after a covolution or a transposed convolution in the decoder part of the autoencoder in order to decrease the blurring of the result. It is a well known method that is used in some state of art autoencoders and the result looks less distorted in comparison to the result from the transposed convolution[51].

3.3.3 Pixel Shuffle

Pixel Shuffle or Depth to Space, is another popular method of upsampling in deep autoencoders. It simply increases the number of pixels for the output image by reducing the number of channels. If an input from the size of $[C \times H \times W]$ is given to a pixel shuffle layer with an upsacle factor equal to R, then the output will be from the size of $[\frac{C}{R^2} \times RH \times RW]$[52]. The figure below shows it visually:

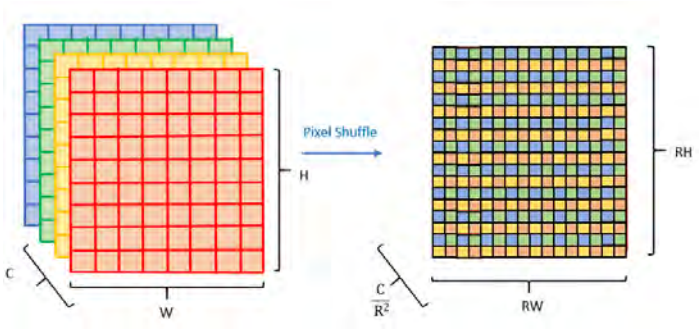

Figure 16: Pixel shuffle increases the height and weight by reducing the number of channels[52].

This method is also popular in the state of art autoencoders and we have tried to build our model in Chapter 4.

3.4 Loss Functions and Metrics

The loss function is one of the most important parts of every deep learning model and it is where the back-propagation begins. In unsupervised autoencoders for image compression, the loss function shows the difference between the reconstructed image and the input image. It mainly stands for the distortion on the rate-distortion trade-off. For omnidirectional images, specific loss functions are available that contain the different rate of sampling on the poles and on the center.

3.4.1 Mean Squared Error

One of the most popular metrics, which has been used widely as the loss function which is friendly with the deep neural networks [53], is Mean Squared Error or MSE. It is defined as the squared root of the difference between the input and the result. In the image signal schemes, it is defined as:

$$MSE = \frac{||x - \bar{x}||^2}{h \times w} \tag{23}$$

where x is the input image, $\bar{x}$ here shows the reconstructed image after compression. w and h are the number of columns and the number of rows in the image.

3.4.2 Weighted Mean Squared Error

In the omnidirectional framework, MSE can be weighted in a way that it contains the geometry of $360°$ images. The first step is to find these weights. We have used a method based on cosines and the work of Sun et al. [54]. Firstly, the stretching relationship of an equirectangular projection should be found.

The latitude (θ) and longitude (ϕ) of a spherical surface are projected on the Cartesian coordinates (x and y) on the planar surface. If we assume each points on the Cartesian coordinates is on the center of a differential unit surface, the surface area of the unit is equal to $dxdy$. On the other hand, for the

sphere surface, the unit is equal to $d\theta d\phi$. According to [55], the Jacobian matrix can connect these two differential surface areas:

$$dxdy = J(\theta,\phi)cos(\phi)d\phi d\theta = \left| \begin{bmatrix} \frac{\partial x}{\partial \theta} & \frac{\partial x}{\partial \phi} \\ \frac{\partial y}{\partial \theta} & \frac{\partial y}{\partial \phi} \end{bmatrix} \right| \cdot cos(\phi)d\phi d\theta \tag{24}$$

where "||" is the determinant. The stretching ratio, which indicates the weights we are looking for, can be found by dividing the differential area on the sphere and the differential area of the equirectanularly projected image:

$$Stretching\ Ratio = \frac{cos(\phi)|d\phi d\theta|}{|dxdy|} = \frac{cos(\phi)|d\phi d\theta|}{|J(\theta,\phi)||d\phi d\theta|} = \frac{cos(\phi)}{|J(\theta,\phi)|} \tag{25}$$

In an equirectanularly projected image, if w is the number of columns and h the number of rows, the weights for each of the rows can be found using the following equation[54]:

$$w(j) = cos(\frac{(j + 0.5 - \frac{h}{2})\pi}{h}) \tag{26}$$

This value, should be multiplied to the MSE of the pixels for each row and then the whole number should be devided by sum of the weights, in order to provide a correct loss function. It is called WMSE or Weighted Mean Squared Error:

$$WMSE_{total} = \frac{\sum_{j=1}^{h}(w(j) \cdot \sum_{i=1}^{w}(\hat{x}_{i,j} - x_{i,j})^2)}{\sum_{j=1}^{h} w(j)} \tag{27}$$

We have tested this type of loss function for the deformation term of the loss in our preliminary model (look at Section 4 and Appendix A1) and we have used it in our metrics (look at Section 3.4.5).

3.4.3 Weighted L1 Loss

The L1 loss function is another common loss in deep learning models. It uses the absolute value of error instead of using the second order of it. In comparison to MSE, L1 loss has the advantage that it is more robust to the outlier inputs as it does not use the squared value of the error, unintentionally making the outlier anomalies important [53]. In our image compression problem, it is important to consider that the omnidirectional image dataset that we use (SUN360 [16]) is not a clear dataset and in some of the images, there are letters and watermarked characters, that they were added to the image after taking it (look at the Section 5.2), so benefiting from a method that is more resistant against the outliers, can be helpful.

The weighted L1 loss has the same structure as the WMSE loss and the same weight increments have been used. The only difference is that the weights are applied on the first order residuals. The effect of weights are higher here as they are in the same order as the residuals themselves:

$$Weighted\ L1_{total} = \frac{\sum_{j=1}^{h}(w(j) \cdot \sum_{i=1}^{w} |(\hat{x}_{i,j} - x_{i,j})|)}{\sum_{j=1}^{h} w(j)} \tag{28}$$

3.4.4 Peak Signal to Noise Ratio (PSNR)

Peak Signal to Noise Ratio (PSNR) is one the most famous metrics that is used frequently to show the quality of a reconstructed signal. It is defined by:

$$PSNR = 10\ log_{10}\frac{max(S)^2}{MSE} \tag{29}$$

where S is the maximum pixel value which is feasible. Using a binary representation, S is equal to $2^B - 1$ where B shows the number of available bits per sample. ($S = 255$ for 8 bits per sample). PSNR has been used as one of our metrics for performance evaluation of our model (look at Chapter 5).

3.4.5 Weighted-to-Spherically-Uniform PSNR

Weighted to Spherically Uniform Peak Signal to Noise Ratio (WS-PSNR) is an evaluation method and meteric that is made specifically for omnidirectional images [54]. It is made based on the definition of PSNR by substitution of the MSE loss function with WMSE loss function (look at Section 3.4.2).

$$WS - PSNR = 10\ log_{10}\frac{max(S)^2}{WMSE} \tag{30}$$

where for the equirectangularly projected images with 8 bits per sample becomes:

$$WS - PSNR = 10\ log_{10}\frac{255^2}{\frac{\sum_{j=1}^{h}(w(j)\cdot\sum_{i=1}^{w}(\hat{x}_{i,j}-x_{i,j})^2)}{\sum_{j=1}^{h}w(j)}} \tag{31}$$

This method has been used frequently in our tests.

3.4.6 MS-SSIM

Structural Similarity, or SSIM, is an image quality metric or loss function based on the human visual perception [56]. Researchers believe that human visual perception is based on recognizing specific structures inside the image [57] and this metric is also focuses on similarity of structures. SSIM can be calculated based on luminance, contrast, and structure components of two images:

$$l(x,\hat{x}) = \frac{2\mu_x\mu_{\hat{x}} + c_l}{\mu_x^2 + \mu_{\hat{x}}^2 + c_l} \tag{32}$$

$$c(x,\hat{x}) = \frac{2\sigma_x\sigma_{\hat{x}} + c_c}{\sigma_x^2 + \sigma_{\hat{x}}^2 + c_c} \tag{33}$$

$$s(x,\hat{x}) = \frac{\sigma_{x,\hat{x}} + c_s}{\sigma_x\sigma_{\hat{x}} + c_s} \tag{34}$$

where σ_x is the standard deviation of x that can be considered as an estimation of contrast in image signal, μ_x is the mean of x that can be looked as an estimation of luminance in the images, and $\sigma_{x,\hat{x}}$ is the covariance and it can be interpreted as the tendency of the input image and the reconstructed

image to vary together. c_l, c_c, and c_s are the constants of each value and they can be found using the following equations:

$$c_l = k_l^2 B^2, \;\; c_c = \frac{c_s}{2} = k_c^2 B^2 \tag{35}$$

where B is the number of available bits per sample ($B = 255$ for 8 bits per sample), and k_l and k_c are both scalar constants which are ultra low in comparison to 1. Then, the SSIM value can be computed as [56]:

$$SSIM(x, \hat{x}) = (l(x, \hat{x}))^{\alpha} (c(x, \hat{x}))^{\beta} (s(x, \hat{x}))^{\gamma} \tag{36}$$

By changing α, β, and γ the contribution of each of these parts can be selected. In case they are equally important, the SSIM metric can be found using:

$$SSIM(x, \hat{x}) = \frac{(2\mu_x \mu_{\hat{x}} + c_l)(2\sigma_{x,\hat{x}} + c_c)}{(\mu_x^2 + \mu_{\hat{x}}^2 + c_l)(\sigma_x^2 + \sigma_{\hat{x}}^2 + c_c)} \tag{37}$$

Wang et al. [26], proposed a type of SSIM metric that is able to compare structures in the image in the different scales and they called it Multi Scale SSIM or MS-SSIM. In their method, the input image and the reconstructed one are evaluated using a normal SSIM approach. After that, both are low-pass filtered and downsampled by a factor of 2 and the SSIM is calculated again. They continue this up to N times and the final metric is calculated using the following equation:

$$MS - SSIM(x, \hat{x}) = l_N(x, \hat{x}))^{\alpha_N} \prod_{n=1}^{N} (c_n(x, \hat{x}))^{\beta_n} (s_n(x, \hat{x}))^{\gamma_n} \tag{38}$$

where n is the iteration number. They have used a set of grayscale images and the distorted version of them using white Gaussian noise. They trained the values of β_n and γ_n using gradient decent that is applied on MSE to find them for N=5. In order to make the problem simpler, they set $\beta_n = \gamma_n$. Table 1 shows the values they found.

Table 1: The power values of MS-SSIM for 5 iterations found by Wang et al. [26].

Iteration (n)	$\gamma_n = \beta_n$
1	0.0448
2	0.2856
3	0.3001
4	0.2363
5	0.1333

They have shown that MS-SSIM can perform a better evaluation for the distortions which are visible to human visual system in comparison to normal visual distortions. We have benefited from MS-SSIM as a metric (look at Chapter 5).

3.4.7 Weighted to Spherically Uniform SSIM

Similar to WS-MSE, to apply Multi Scale SSIM on an omnidirectional image, specific weights can be used to take into account the geometry. Firstly, a pixel is selected on the equirectangular projected version of the input image and the reconstructed image and it is mapped on the sphere using the equations we showed in Section 2.3.2. On the sphere, for an area around this pixel, SSIM is calculated using the local version of Equation 37 on the sphere domain [58]:

$$S - SSIM(i,j) = \frac{(2\mu_x\mu_{\hat{x}} + c_l)(2\sigma_{x,\hat{x}} + c_c)}{(\mu_x^2 + \mu_{\hat{x}}^2 + c_l)(\sigma_x^2 + \sigma_{\hat{x}}^2 + c_c)} \tag{39}$$

where i is the point from input image on spherical domain and j is the spherical domain representation of the pixel on output image. In the implentation of Chen et al.[58], they made an 11×11 patch around that specific pixel. For an equirectangular projection, this should be weighted with the weights we showed in Equation 26 as:

$$WS - SSIM(i,j) = \frac{\sum_{j=1}^{h}(\sum_{i=1}^{w}(S - SSIM(i,j) \cdot w(j))}{\sum_{j=1}^{h} w(j)} \tag{40}$$

3.5 Viewports

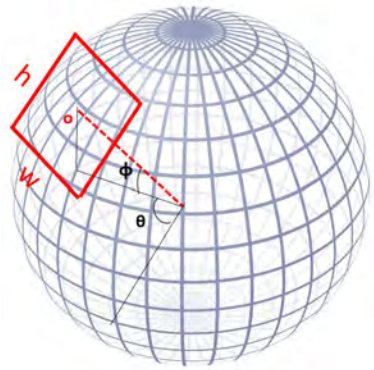

Figure 17: A schematic view of a viewport which is tangent to the sphere on point o with size $h \times w$.

A viewport, is defined as a part of the spherical surface which is projected on a tangent plane.[21] The center of it (point o in Figure 17), is the tangent point. The projected signal on the viewport can be considered as almost non-omnidirectional, so the normal, planar metrics such as PSNR and SSIM can be applied on it. We will use this method just for the performance evaluation of our results and comparison (look at Chapter 5).

In this study, in order to find the positions of the viewports on the spherical domain (which is represented by centers of them), we benefited from the work of Xu et al.[59]. In their study, they proposed

a method that considers the spherical geometery of omnidirectional images to find the centers of the viewports that should be generated (Figure 18):

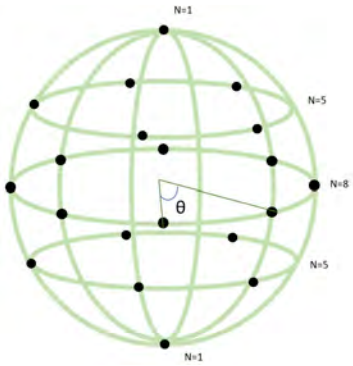

Figure 18: Position of the centers of the viewports when $N_0 = 8$ [59].

The points which are shown are selected using the equations proposed by Xu et al. Firstly, the number of points on the equator is selected as N_0. Then Θ which is the angle between each two neighboring points on the equator is selected as:

$$\Theta = \frac{360}{N_0} \tag{41}$$

Then the number of point on the other orbits are selected as:

$$N_1 = N_0 * \lceil cos(\Theta) \rceil \ , \ N_2 = N_0 * \lceil cos(2\Theta) \rceil \ , \ ... \tag{42}$$

After this, based on the size of viewports that should be selected and using gnomonic projection, the planar image signals are made (look at Section 2.3.2 for information about this type of projection).

4 Building the Model

In this chapter, we build our proposed model and we show, step by step, how we ended up with current final architecture. They are important points in each step and experiment that resulted in some of our conclusions we point out in Chapter 6. **However, in case you would like to see the final model, you can skip this chapter and go directly to Chapter 5**.

First, we show how to make an omnidirectional image autoencoder that can get trained for one specific rate. After that, we show the iterative method of training on residuals that provides the ability to train for different rates in just one pass. This method that benefits from planar and spherical recurrent neural networks blocks, is called Version 1. Then, we propose the method of patching the image in spherical domain and we propose models that benefit from it to be computationally efficient and better in terms of distortion which are called Version 2. Each of these versions include some different sub-versions that show the effect of changing the details and modules of models.

It is worth mentioning that as the first step, we have performed a preliminary test based on end-to-end autoencoder model that has been presented in Chapter 2. Due to the fact that training the model was time consuming and computationally demanding, and it should be trained on every different compression rate, we decided not to move forward and to propose our model based on another model that can be modified and trained in a more feasible way. The results of these preliminary experiments (which is called Version 0) and the details are provided in Appendix A1.

4.1 Version 1: Geometry Aware Recurrent Residual-based Trained Autoencoder

Generally, the main ideas for architectures and modules that we propose are based on the planer image compression method of Toderici et al. [35]. and spherical methods of Coors et al. [45]. The first model we made is a simple autoencoder that uses spherical convolutions and spherical MaxPooling in the encoder, and benefits from bilinear interpolation for the upsampling in the decoder (see Figure 19). Between each of the upsampling layers, there are transposed convolution layers to reduce the bluring effect (as we discussed, we did not use spherical transposed layers as they could not be efficiently implemented). We modeled the quantization step using a simple binarizer that uses a uniform noise probability in the forward pass and it is bypassed in the backward pass similar to the implementation of Toderici et al. that we showed in equations 6 and 7. It generates the compressed version of the input image and the reconstruction of it. The compressed signal is sent to a lossless compression bottleneck (in our project, we use NPZ format that uses the “gzip” lossless compression scheme[60]). Figure 19 shows the architecture of this model and the reconstructed version of image No. 936 from the test set (look at Section 5.2 for original input image).

Figure 19: Left: Image compression autoencoder architecture. Right: Reconstructed image.

The reconstructed image looks similar to the input, but the quality is clearly degraded. In fact, this model is unable to control the dedicated number of bits for each pixel and it is not possible to increase the rate that might increase the quality. In order to add the ability to provide the different compression rates without the need to retrain the model for every different rate, we benefit from training on residuals. We can perform the training and testing in an iterative way that is shown in Figure 20 [35].

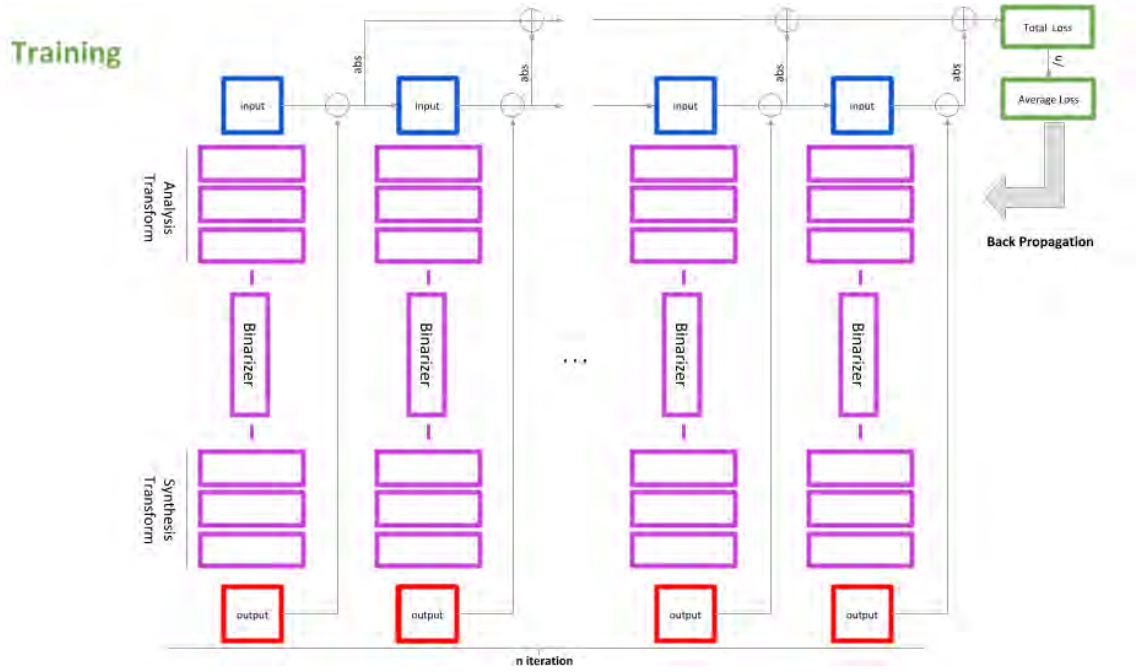

Figure 20: Training of our autoencoders in an iterative way based on the residuals[61]. It provides all the needed compression rates with one pass of training.

Training: In every epoch, a few number of iterations are done. In the forward pass, the autoencoder works on input and reconstructs the output image. The residual of the output image with respect to the input is calculated and it becomes the input of the next step. Also, the absolute value of it should

be stored. Again the forward pass is performed on the residual and another output image is reconstructed. Afterwards, the output is subtracted from the input (which, itself, is actually the residual of the previous step). This process can be done on a few iterations and the sum of the absolute values of the residuals should be calculated (L1 total loss). After that, this value is divided by the number of iterations to produce the average loss. It is then backpropagated through the model to update the weights and biases of all layers. The weights and biases are the same for the different iterations and provide a huge gain in terms of training time and the required computational power in comparison to the end-to-end autoencoders.

After performing the training on the dataset and finding the weights and biases, now it is time to perform the testing in a similar iterative way. The process of testing is shown in Figure 21.

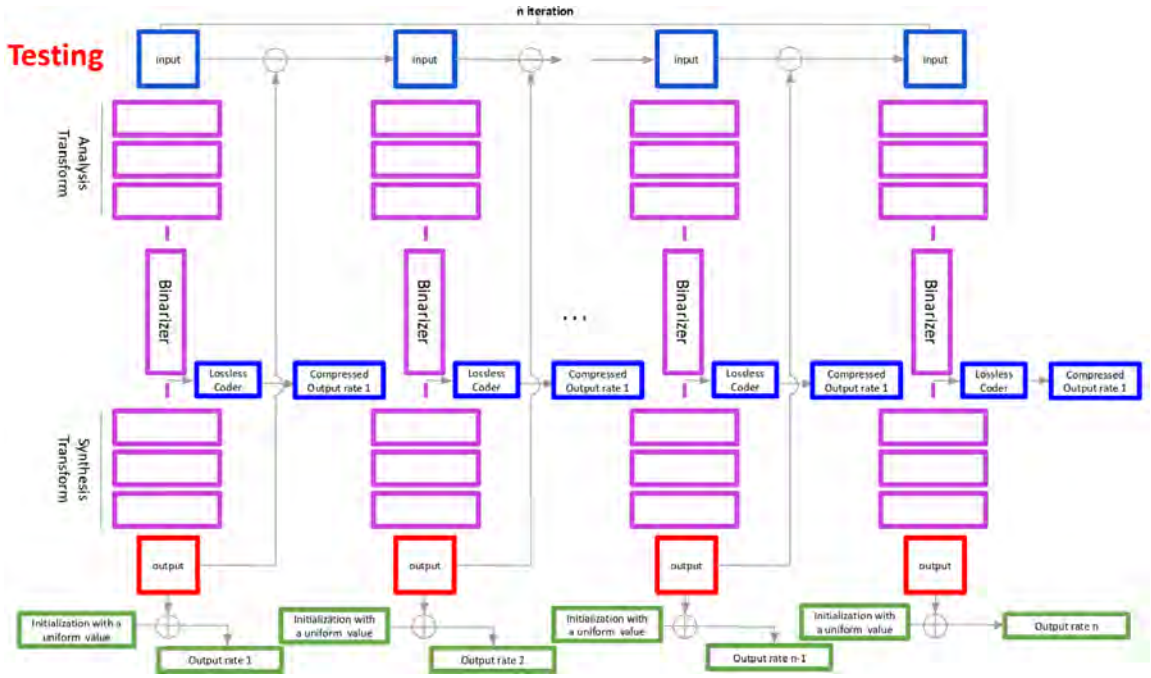

Figure 21: Testing phase of our autoencoders in an iterative way based on the residuals[61]. It provides all the needed compression rates with one testing epoch and a rate distortion curve can be generated easily afterwards.

Testing: The number of iterations in the testing phase should be the same of the ones in the training phase. Each of the iterations stands for one of the rates, on the final rate-distortion curve. In the testing phase, firstly, a tensor with the same size as the input image is generated. This tensor should be initialized with a uniform value (for example, equal to 0 or 0.5 in our case as the model runs on the values between -0.5 and 0.5. Indeed, the input images are firstly normalized from 0 to 255 interval to $[-1, 1]$. It is done due to the fact that we benefit from the $Tanh$ nolinearities and they generaly generates values between -1 and 1.). After that, the input image is given to the model and the output of the first iteration is generated and should be added to the initialized tensor. It becomes

the output image with the lowest rate. Then, the output of the next iteration is generated by giving the residual of the previous step as input. Again, this should be added to the initialized tensor and it becomes the reconstructed image with the next rate value. This process can be done again until the nth iteration which generates the reconstructed image with the highest rate. The compressed version of images with the different rates is generated from the output of the binarizer and it is then compressed by a lossless scheme. If there is a need to generate the loss of testing, the same method as the training can be used to store the losses and to perform averaging on them.

An important point about the image compression schemes which are trained on residuals is that the statistics of the first iteration is differnet in comparison to the next iterations. Indded, the input of the first iteration is the image itself, but in the next layers the input is residual and training the same weights and biases based on these different inputs is theoretically wrong. In order to avoid this problem and have the effects of the first iteration in the next ones, recurrent neural network blocks should be used. Theses layers broadcast the effect of previous iterations to the next iteration and by using them. In this study we benefit from the LSTM blocks and Spherical LSTM blocks (see Section 3.2.1).

So we propose the Version 1 of our model. This model is inspired by Toderici et al. autoencoder (look at Figure 29). The ConvLSTM cells are replaced by SphericalConv LSTM cells and the model is trained with these parameters:

Table 2: The parameters which are usedd for training of the model. These parameters are used for all the models in this chapter.

Feature	Value
Number of Epochs	200
Learning Rate	10^{-4}
Weight Decay for L2 Penalty	10^{-6}
Number of Iterations per Epoch	10
Dataset	256 Images Subset of SUN360

Figure 22 shows a schematic view of the Version 1.1 and Version 1.2 that we made. One of them benefits from MaxPooling in the encoder for the downsampling and the other one benefits from strided spherical LSTM cells:

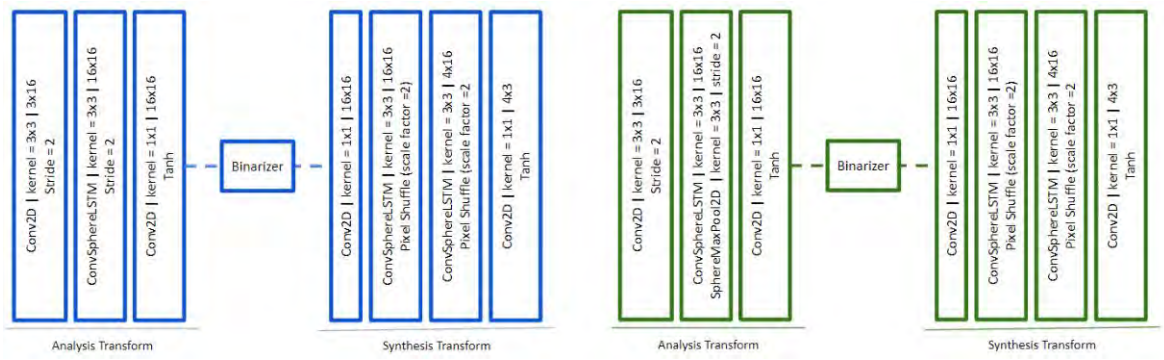

Figure 22: Left: Model version 1.1. Right: Model version 1.2. Both are autoencoders benefiting from recurrent neural network blocks and trained on residuals.

Similar to the work of Toderici et al. we have convolutional filters before and after spherical layers. We have exploited spherical LSTM layers in between and there are three hidden states in these models for the three LSTM cells. In the decoder, we have used the pixel shuffle layer that is in use in the state of art autoencoders and we did not use transposed convolutional layers that may add grid structure anomaly (look at Section 3.3.1). The only difference between these two models are the method of downsampling. Version 2.1 uses strided spherical LSTM and method 2.2 benefits from Spherical Maxpool layer. Figures 23 and 24 show the reconstructed image No. 936 and the metric plots .

Figure 23: Reconstruction of image 936 from the test set. Up-left: Version 1.1 with rate=0.2442, Up-right: Version 1.1 with rate=6.6878, Bottom-left: Version 1.2 with rate=0.2243, Bottom-right: Version 1.2 with rate=5.8719 (all in bpp).

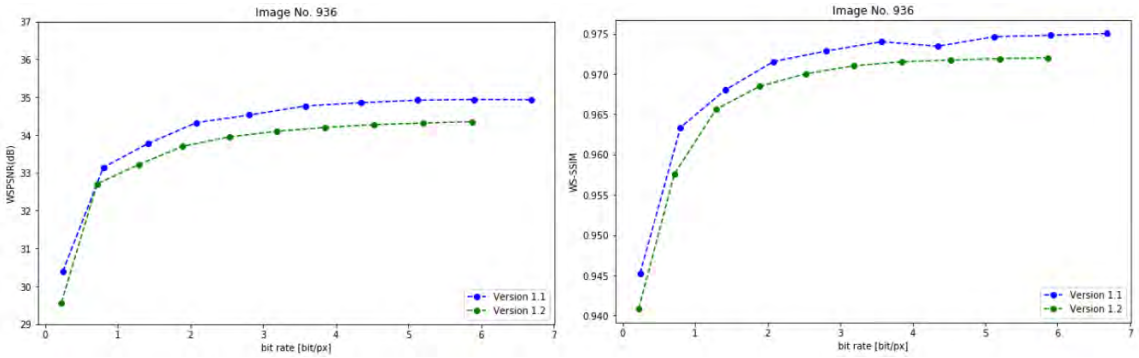

Figure 24: WSPSNR and WS-SSIM metrics for image 936 reconstructed with version 1.1 and 1.2.

It has been shown that in the high rates, the reconstructed images are almost clear; however, in the low rates, they are blurred. One of the reasons can be the downsampling method. In Version 1.1 and Version 1.2, we have used 2 downsampling methods that may not be completely effective. As we mentioned in Section 3.1.1, in the strided Spherical convolution or LSTM layers, the stride is performed on the grid, before doing the convolution. Indeed, even if the size of the convolution kernel is bigger in comparison to the stride, there is still some loss of information and it may have effect in the reconstructed images. Also about the MaxPool layer, some researchers [62] proposed that it is better to avoid it for the downsampling in image reconstruction autoencoders. As it is clear in the plots, it is even worse in comparison to the strided spherical convolution that does not benefit from the whole available information.

To improve the model, we have made a few changes in Version 1.3. In this model, the downsampling is done using regular strided convolutions instead of MaxPool or strided spherical layers. Also, spherical LSTM cell has been moved to the first layer. Indeed, we thought it might be better to perform the spherical sampling in the beginning before doing any other operations on the signal so the whole model receives the correctly sampled input before any processes. This time, we have also made a similar model without spherical layers in order to compare.

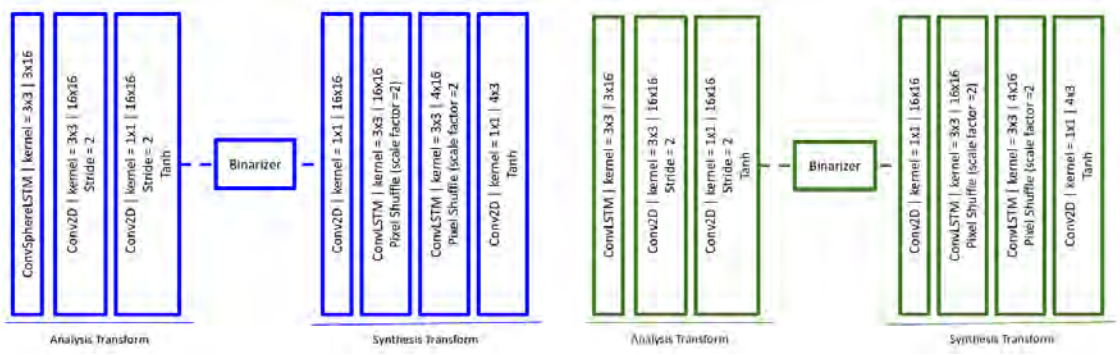

Figure 25: Left: Model version 1.3. Right: Similar model without spherical modules in order to compare.

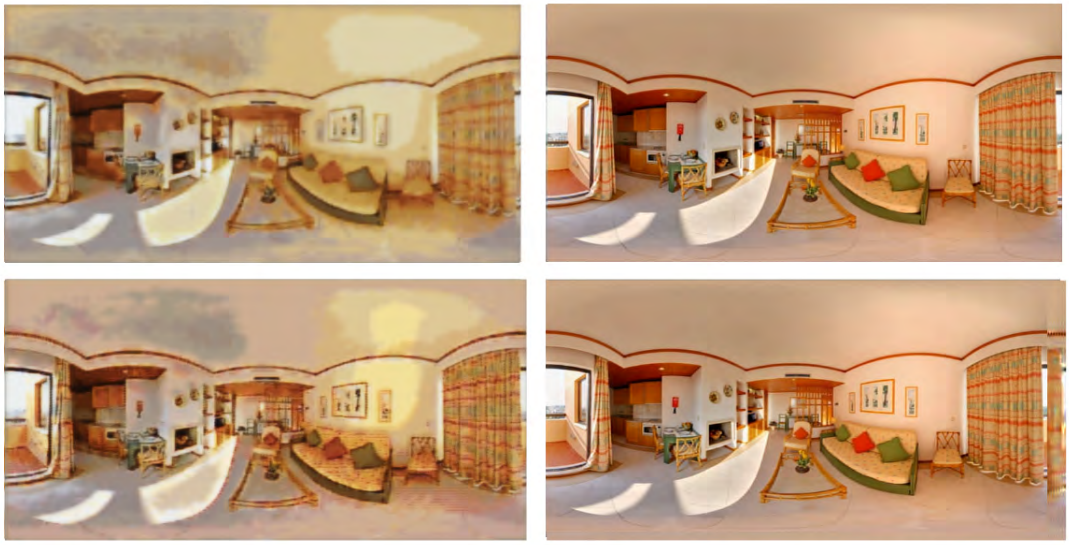

Figure 26: Reconstruction of image 936 from the test set. Up-left: Version 1.3 with rate=0.4156, Up-right: Version 1.3 with rate=7.5602, Bottom-left: Similar model without spherical modules with rate=0.4031, Bottom-right: Similar model without spherical modules with rate=8.0800.

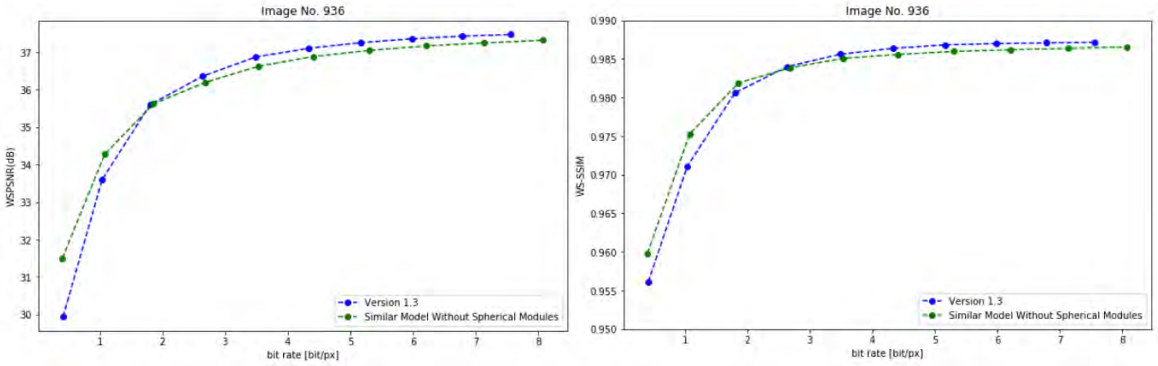

Figure 27: WSPSNR and WS-SSIM metrics for image 936 reconstructed with version 1.3 and a similar model to it without spherical modules.

It is clear in the Figures 26 and 27 that this method generally performs a nice image compression and the figures are improved. Additionally, the WS-PSNR and WS-SSIM plots show that performing the spherical sampling at the beginning in a spherical LSTM layer helps to improve the model especially for the high rates; however, in the low rates it is still a bit lower in comparison to the similar model without spherical layers. Here, we decided to increase the number of filters or layers for both models to see if it improves in comparison to the planar model. We had this intuition that as the spherical model is more complex, using it with few number of filters degrades it more in comparison to normal planar models.

In this position, we faced a problem related to computational power. When we ran the spherical

LSTM cells with a higher number of filters, we ran out of GPU computational power; although we were using an NVIDIA RTX Titan with 12 GB of memory. The main reason behind this problem is the fact that in the omnidirectional models that benefit from spherical layers, it was not feasible for us to perform the normal random crop patching on the images which is available in the deep learning APIs. As a result, the size of the input images were much higher in comparison to the planar deep learning-based methods (for instance, Toderici et al.[35] model uses just 32×32 size but our inputs were from the size of 768×384.) Additionally, as it has been discussed in Section 3.1.1 , spherical convolutional operations are practically done using a spherical grid generation and then a planar convolution. A grid is generated for each of the points and then a planar convolution with a kernel size equal to grid is used. Consequently, they are more demanding in terms of computational power at least from an order of the kernel size (3×3 here) in comparison to a planar conovolution. Moreover, a SphereConvLSTM block that we have shown has 8 spherical convolutions inside that makes this effect even more significant. In order to solve this problem, a method of batching for the omnidirectional models have been proposed and it is used in the next version.

4.2 Version 2: Spherically Patched Geometery Aware Recurrent Residual-based Trained Autoencoder

In this section, firstly we propose the spherical patching method and then we use it in the Version 2.1 of our model. Also, in order to compare, we increase the number of spherical LSTM layers in Version 2.2 to see if it results in improvement. The methods are compared with similar models without spherical LSTM layers and spherical patching.

4.2.1 SpherePatch: A Method to Perform Patching on the Omnidirectional Images.

Most of the novel, deep learning-based image compression schemes are benefiting from a type of input image patching for the training phase. It helps the method to be less computationally demanding and to run faster. Additionally, if patching is done randomly on different places of the training set, it virtually increases the size of input dataset and it is the secret that these methods are able to work, after they were being trained on few number of images.

If the simple, random patching is done on the methods that benefits from spherical layers, it significently decreases the accuracy of the result as these methods generate a projection grid based on the size of input image. For example, if there is a patch on the higher parts of an image, as the polar angle (ϕ) is higher, the sampling positions are significently different and using the naive method of applying the same spherical sampling positions is wrong.

In our method, instead of using the RandomCrop built-in classes of deep learning API frameworks that does not provide data about the position of patching, we made a cropping function that keeps the position data of the generated random patch. It can be made by generating a random number based on the size of the patch and the images. For instance, if the size of image is $H \times W$ and the size of

patch is $b_H \times b_w$ a random number for the top-left position of patch can be selected in the interval of $y_t \in [b_h, B]$ and $x_l \in [0, W - b_w]$.

After that, the top-left position of the patch, together with the total height and total weight of the image should be sent to each of the spherical layers of the model. In order to find the distance between every two neighbor increments of spherical grid, instead of the size of input image (which is $b_h \times b_w$), the total size should be used:

$$\Delta\phi = \frac{\pi}{H} \qquad \Delta\theta = \frac{2\pi}{W} \tag{43}$$

The increments of polar angle and spherical angle should also be selected based on the total weight and the total height and $[i, j]$ should be generated in the specific interval of the patch:

$$\phi_i = -\frac{(i + 0.5) \cdot \pi}{H} + \frac{\pi}{2}, \qquad i \in [y_t - b_h, y_t] \tag{44}$$

$$\theta_j = \frac{(j + 0.5) \cdot 2 \cdot \pi}{W} - \pi, \qquad j \in [x_l, x_l + b_w] \tag{45}$$

Then the rest of calculations would be the same as Equation 12 and the equations after it. It is worth mentioning that as each of the patches are not a complete $360°$ image, the method has been used by Coors et al. [45] to avoid padding on the sides by rotating around the image cannot be implemented here. Instead, when the grid is generated, if it goes out of the patch, we simply use the available pixels around.

4.2.2 Version 2.1 and 2.2.

In this section, we add the SpherePatch method that we proposed in the previous section to the Version 1.3. We implement a 32×32 spherical patch. Consequently, the model can now easily be trained even on an average NVIDIA GTX 1050 Ti with 4 Gb of GDDR5 memory. As a result, we were able to increase the number of filters and we increased it to 64. Figure 28 shows the implemented architecture (Version 2.1):

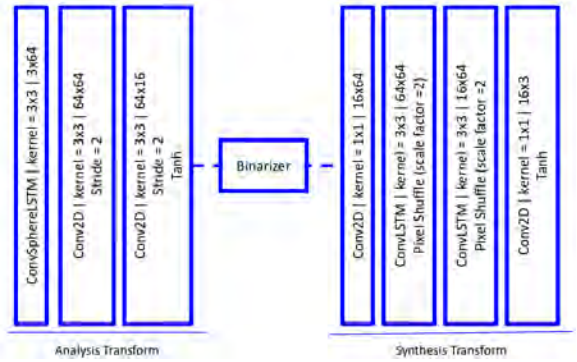

Figure 28: Version 2.1 autoencoder that benefits from spherical patching and spherical LSTM block and is trained on residuals.

As it is shown in Figure 28, spherical LSTM cell performs the sampling on the sphere and applies convolution. The number of filters are reduced just one layer before the binarizer in order to improve the model in terms of the rate and it is increased again in the decoder. For the decoder, we have implemented pixel shuffle upsampling. This method has been trained on two different sampling interpolation methods (nearest neighbor and bilinear) for the spherical grid and we compare them in terms of WSPSNR and WS-SSIM. Also, in addition to the L1 loss function that we used for all the models up to here and it is from the work of Toderici et al. (Equation 3), we also tried the WL1 loss as well (Equation 28) to see if it provides an improvment for the model or not.

After that, a similar method without spherical patching and sampling has been implemented which is based on the work of Toderici et al.[35]. This model exploits the popular RandomCrop approach for the patching, We made this model in order to compare with our proposed model. Figure 29 shows this model:

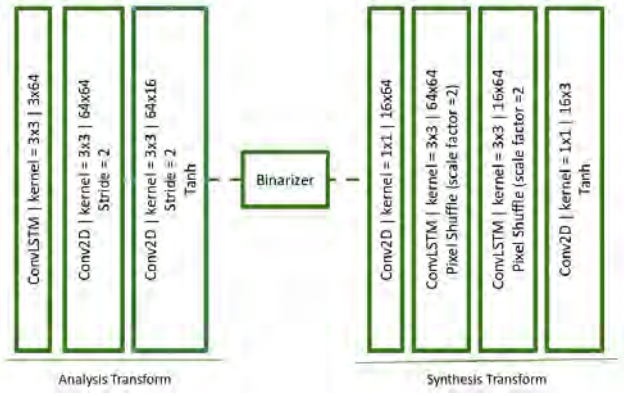

Figure 29: Autoencoder made without use of spherical patching and spherical layer with an architeccture similar to V2.1 in order to compare.

Finally, we added a second spherical LSTM cell to see if it improves the model or not. Figure 30 shows this architecture that benefits from two spherical layers (Version 2.2):

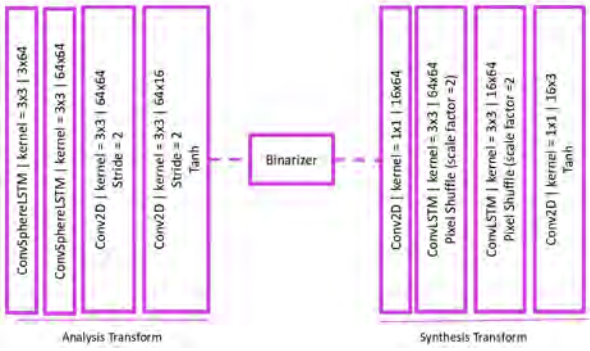

Figure 30: Version 2.2 autoencoder that benefits from spherical patching and two spherical LSTM block and is trained on residuals.

The reconstructed images and the results of WSPSNR and WS-SSIM metrics are shown in Figures 31, 32, and 33:

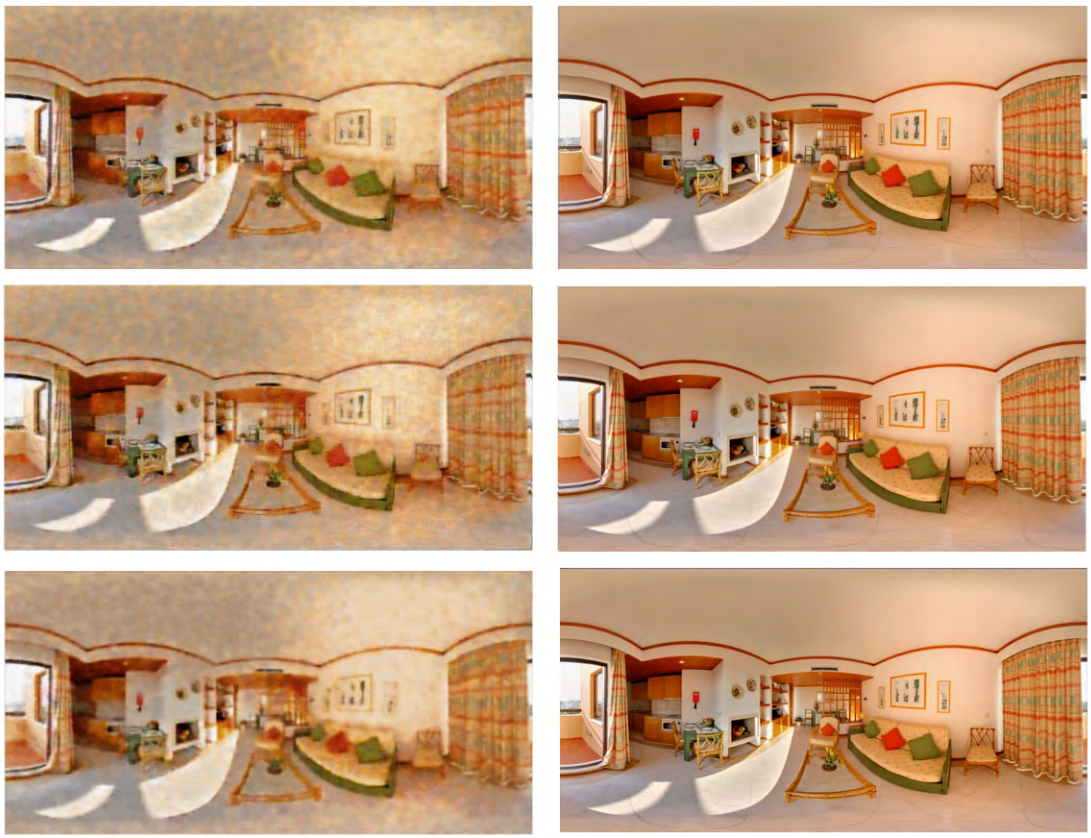

Figure 31: Reconstruction of image 936 from the test set in version 2. Up-left: Version 2.1 with rate=0.6564, Up-right: Version 2.1 with rate=9.5075, Middle-left: Version 2.1 with bilinear sampling with rate=0.6605, Middle-right: Version 2.1 with bilinear sampling with rate=9.5432, Bottom-left: Version 2.2 with rate=0.6144, Bottom-right: Version 2.2 with and rate=9.1638 (all in bpp)

Figure 32: Reconstruction of image 936 from model similar to version 2 without spherical patching or layers. Left: rate=0.6343 bpp, right: rate=9.5361 bpp

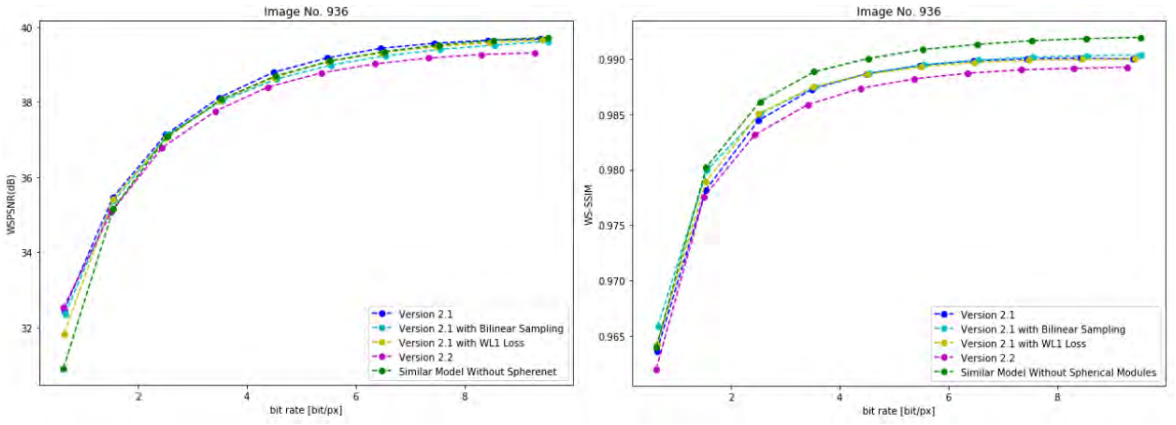

Figure 33: WS-PSNR and WS-SSIM metrics test for Version 2.1 and 2.2

In Figure 39 it has been shown that after applying the spherical patching method, the reconstructed images improved and now even in the low rates, the image structures are mainly clear (compare them to Figure 26). In the WS-PSNR plot we can see that Version 2.1 model with nearest neighbor interpolation performs a better job in comparison to the model that has the similar architecture without the spherical modules. In the WS-SSIM metric, the results from Version 2.1 with bilinear interpolation is slightly better; however, in the high rates model without the spherical modules passes it and performs a better reconstruction. About this metric, it should be noticed that WS-SSIM parameters are not trained for this omnidirectinal dataset and it might have some effects in the result. When we compare the result from nearest neighbor sampling method and bilinear sampling, it has been shown that nearest neighbor does a slightly better job in WS-PSNR, but for the WS-SSIM, bilinear is generally better. The reason is that WS-PSNR is more sensitive to pixel-wise difference and as the bilinear performs a type of linear averaging, it blurs down some parts which reduces the result from this metric. On the other hand, WS-SSIM is more sensitive to the structures in the image and this linear interpolation provides a better structure-wise view especially in the low rates. After that, as it is clear in the Figure 33, adding a second spherical LSTM layer did not improve the model and we think that it is due to the fact that it ignores the learned spherical structures from the first layer which causes the degradation of the model. Indeed, the signal after the first spherical layer is not completely spherical anymore and it is better not to use more than one spherical layer. Finally, the model with WL1 loss does not show a significant improvement in comparison to other models that they benefit from simple L1 loss. The reason is not compeletly clear but it may be because of the specific nature of this recurrent neural network-based model and it cannot easily generalized to all possible solutions for omnidirectional image compression autoencoders.

We selected Version 2.1 with nearest neighbor sampling and simple L1 loss as our final method and in the next chapter, we increase the number of channels even more and we train it on the 776 images subset we have from SUN360 dataset to compete with JPEG.

5 Performance Evaluation

5.1 The Proposal

In this chapter, we show our proposal and compare it with JPEG standard, classical model. This model, is an improved version of Version 2.1 (look at Section 4.2.2) that we selected as our proposal. As JPEG rates are generally in the interval of $(0, 5]$ bpp, another downsampling layer has been added to the model to shift it more to the left on rate-distortion plot. Also, the number of filters has been increased to 128.

This model, has one spherical LSTM cell layer in the beginning that performs the sampling on the sphere and considers the geometry of the omnidirectional image (to see why just one spherical layer has been used, look at 4.2.2). It benefits from spherical patching method that is well-aligned with this spherical LSTM cell and makes it able to run more efficiently with lower computational power and also improves the result (for more information, look at Section 4.2). It uses a simple averaged L1-loss on the residuals as it is more robust to outliers and anomalies in comparison to MSE and it does not use spherical weighted loss functions. (In our model, spherically weighted loss functions did not show a significent improvement in the results, look at the experiment at Section 4.2.2). Finally, it benefits from training on residuals that provides different compression rates with just one pass of training, similar to the work of Toderici et al. (for details about this method of training, look at Section 4.1). Figure 34 shows the proposed model architecture.

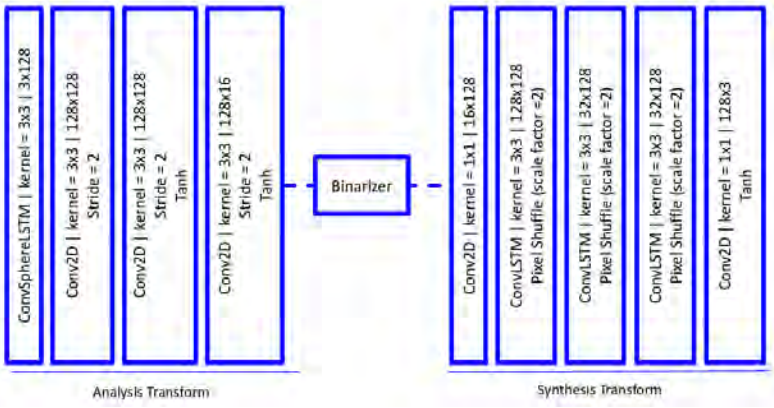

Figure 34: Final model which is made for performance evaluation based on Version 2.1.

The training is done using the parameters which are shown in Table 3.

Table 3: The parameters which are used for training of the proposed model.

Feature	Value
Number of Epochs	300
Size of Spherical Patching	32×32
Learning Rate	10^{-4}
Weight Decay for L2 Penalty	10^{-6}
Number of Iterations per Epoch	10
Dataset	776 Images Subset of SUN360

As it is clear in Table 3, we have used 776 images from SUN360 dataset to train the model. In the next section, we talk about the features of this dataset.

5.2 The Dataset

In the field of planar images, there are many different standard datasets available with different resolutions; however, in the field of omnidirectional images, the situation is different and the number of available datasets is low. Some available omnidirectional datasets include 360-Indoor [63], 3D60[64], Salient 360[65], and SUN360[16]. In this study, our choice is the SUN360 datasets that contains a high number of images, from outdoor and indoor. The images are complex and contain a lot of details (Sometimes they contain human in the images too). Additionally, it has some varieties and anomalies in some images that make our model more robust. First, The source images had different resolutions (1024×512 and 9104×4552) and were possibly taken by different cameras. Second, some of the images, like the images in Figure 35, contain some anomalies and some objects (like text) were added to them afterwards. These additional objects are very common on the available omnidirectional images on the Web and it is helpful to make our model more robust to them. In general, it seems around 20 % of the images from the SUN360 dataset contain these type of anomalies.

Figure 35: Left:Texts and other unnatural objects that were added to the images after capturing. Right: Unnatural anomaly in the lower part of image that might be from the method of capturing or added to the image afterwards in order to keep the resolution uniform.

To prepare the dataset for training, firstly, the images were converted from the JPEG format to PNG in order to show the uncompressed size and also to be able to compress it again to JPEG images with

different qualities for comparison. Then, since we wanted to have the same resolution in all images and to reduce the training size, all the images were downsampled to the size of 768×384. After that, three subsets were made from the dataset containing 24 (used in Appendix A1), 256 (used in Chapter 4 to build the model), and 776 images (used in this Chapter 5 for performance evaluation).

Three images were selected to test the results from the different methods. One image is from inside of a building, the other one from the outside, and the third one is from a train station that contains a part inside and a part outside, with some complex structures on the ceiling. The three test images are shown in Figure 36.

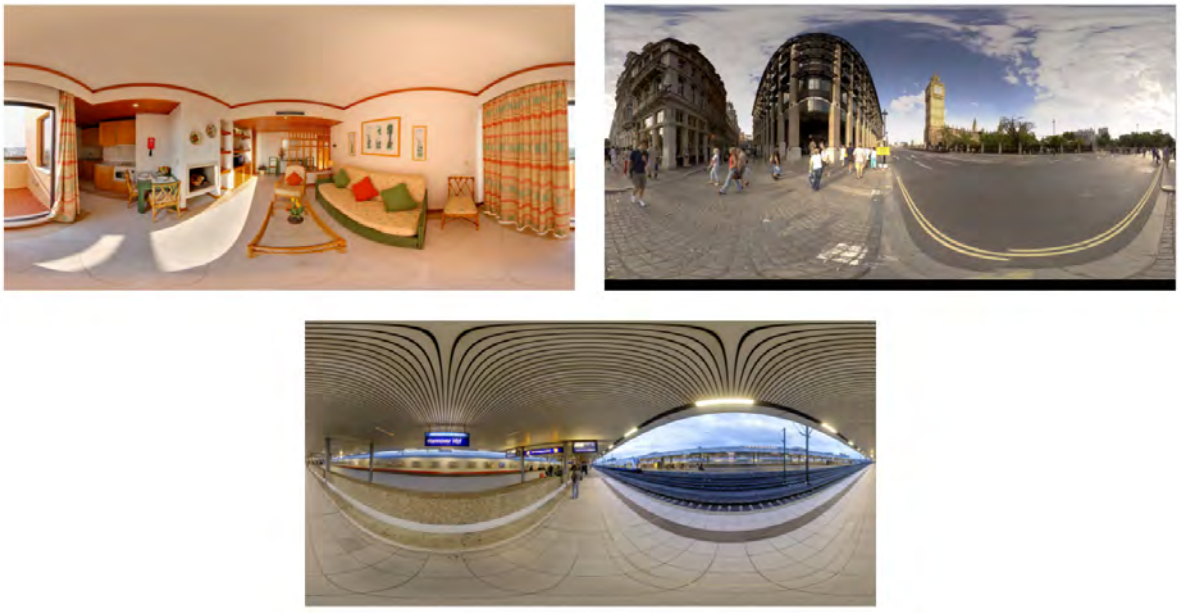

Figure 36: The three test images that have been used. All three are downsampled to 768×384. One indoor, one outdoor, and one in between from a train station [16]. Top-left: image No. 936, top-right: image No. 528, and bottom: image No. 1231.

5.3 Results of Experiments

After training and the testing of the model, the test images were reconstructed. Here, the reconstructed images are shown for the all three images of the test set. Different spherical metrics like WSPSNR and WS-SSIM are plotted to help us compare the results. WSPSNR metric is applied on the average of R, G, and B components to contain the effects of both luminance and color. The reconstructed images from JPEG with $4:2:0$ subsampeling method is shown and the metrics' results are also available for them to compare. The selected qualities of JPEG points are from the set of $Q = [5, 10, 20, 30, 40, 50, 60, 70, 80, 90]$.

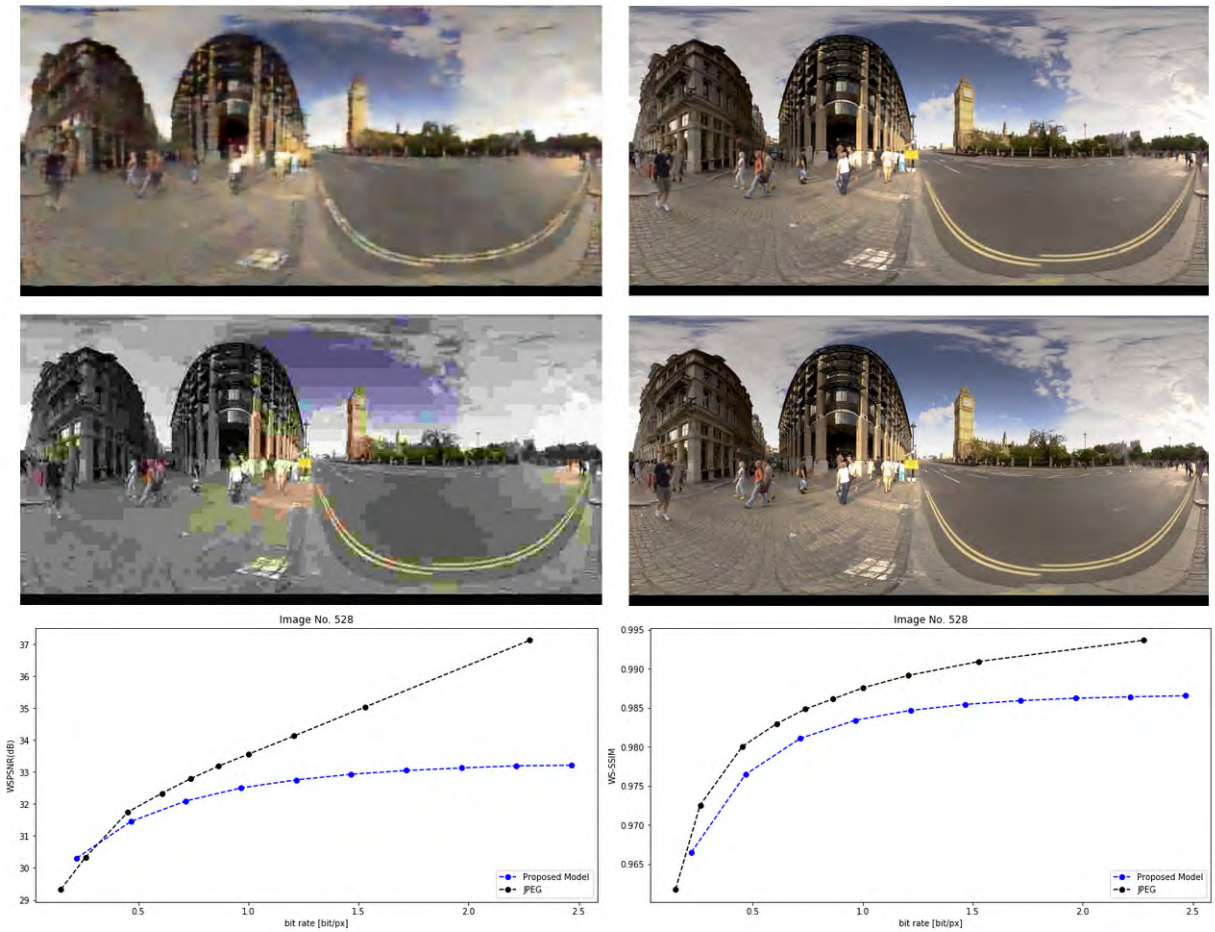

Figure 37: Reconstruction of image 528. Up-left: Proposed model with rate=0.2206, Up-right: Proposed model with rate=2.4673, Middle-left: JPEG with rate=0.1490, Middle-right: JPEG with rate=2.2800 (all in bpp) Bottom-left: WS-PSNR metric results, Bottom-right: WS-SSIM metric results

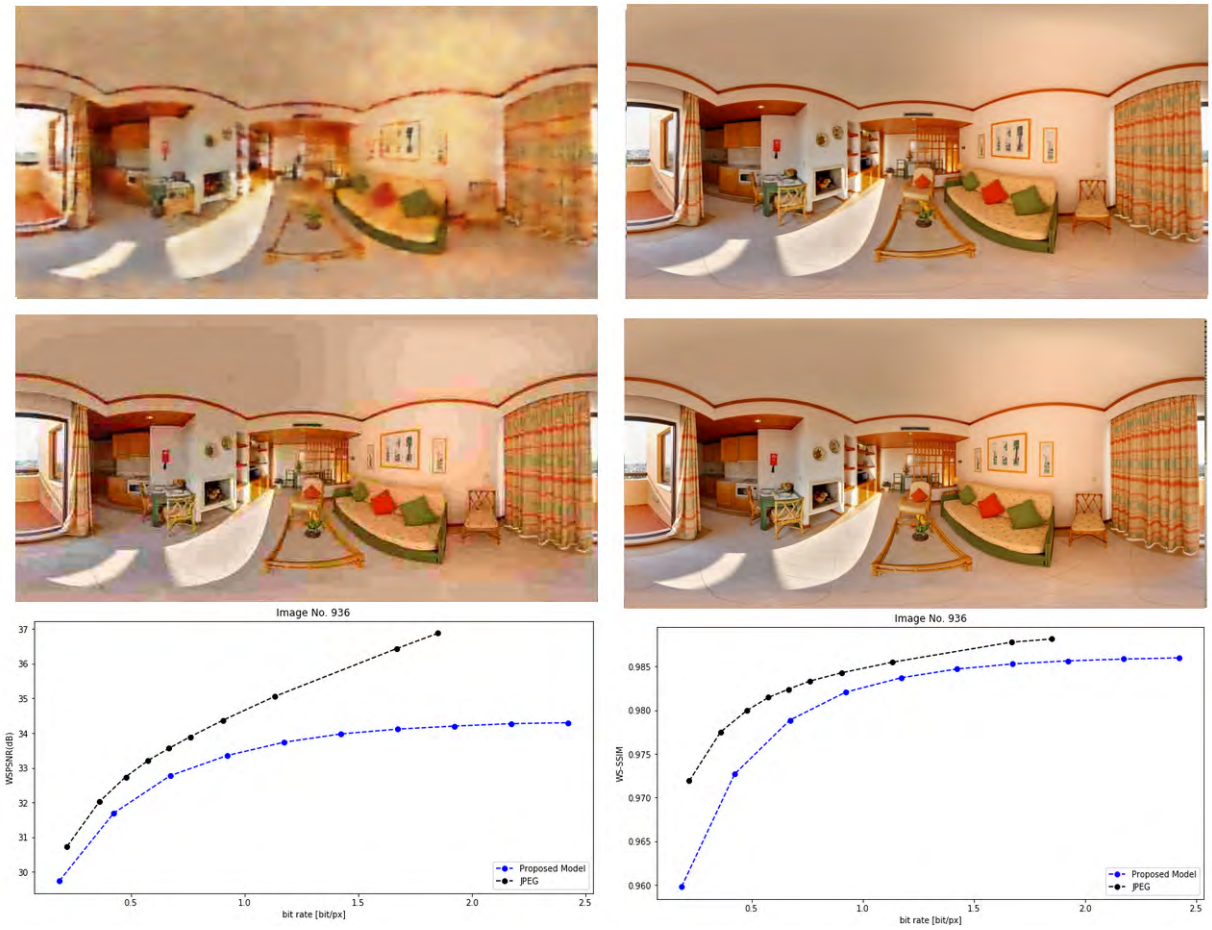

Figure 38: Reconstruction of image 936. Up-left: Proposed model with rate=0.1861, Up-right: Proposed model with rate=2.4237, Middle-left: JPEG with rate=0.2200, Middle-right: JPEG with rate=1.8514 (all in bpp), Bottom-left: WS-PSNR metric results, Bottom-right: WS-SSIM metric results

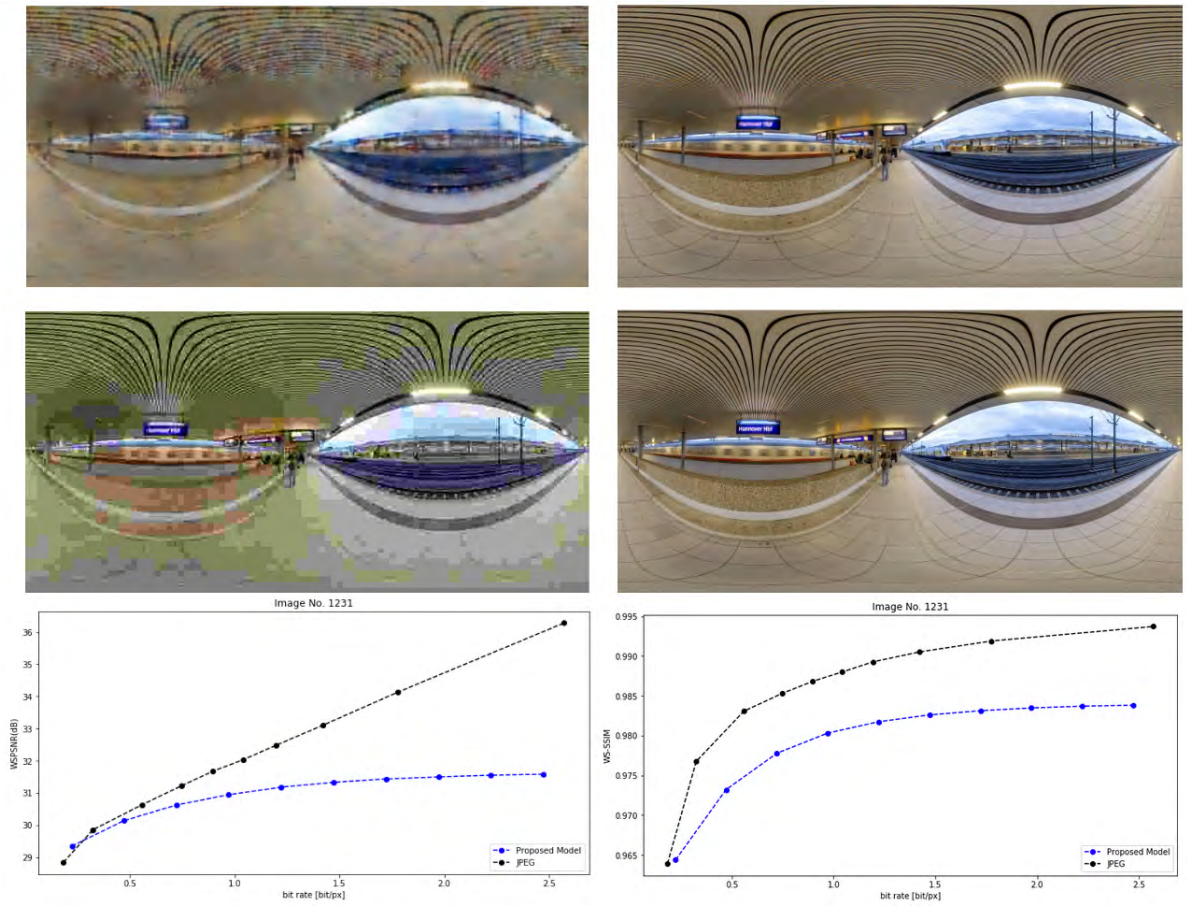

Figure 39: Reconstruction of image 1231. Up-left: Proposed model with rate=0.1842, Up-right: Proposed model with rate=2.4237, Middle-left: JPEG with rate= 0.2239, Middle-right: JPEG with rate=2.5700 (all in bpp), Bottom-left: WS-PSNR metric results, Bottom-right: WS-SSIM metric results.

As it is clear in the images, although the rates are generally low, our model can perform a nice reconstruction on the image and it shows the important structures. Especially in terms of showing the colors, it works well. The model can come close to JPEG in the metrics and even pass it for 2 of the images in the low rates, but in the high rates, it is still a bit lower. It is worth mentioning that due to the problems regarding the available computational power, we could not increase the number of filters more or to the train on the whole dataset. (As an example, the number of filters in the work of Toderici et al. that provides better results in comparison to JPEG is from the order of 512 and they trained on 1 million epochs. Also, the number of layers and training images are higher [35].) We think if we have access to higher compuational power and increase the number of filters, epochs, and the images in the training set, this model has the potential to provide better results in comparison to JPEG even for the higher rates. Meanwhile, it should be mentioned that JPEG benefits from a specific optimized lossless entropy coder, but the entropy coder that we use is not optimize for our model (it is one the possible further improvements), so it is not a completely fair comparison.

To show the effect of Omnidirectional layers on the results and the possible improvement they bring to the model, we performed an analysis based on the generated viewports from the test set. Viewports with a grid sampling size of 45×45 have been generated. N_0 that sets the position of the centers and number of viewports is selected to be equal to 8, so according to Equation 42, the total number of viewports is $8 + 2 \times 5 + 2 \times 1 = 20$. Planar meterics such as SSIM and PSNR have been implemented on the generated viewports. The same process have been done on JPEG images in order to compare. This time, in addition to JPEG $4:2:0$, we also made results based on JPEG $4:4:4$ which does not perform chroma subsampling. In fact, we did this as we have seen that our method performed a better job to find the colors in the low rates in comparison to JPEG and we wanted to see if it has an effect on the results. Figure 40 shows the results:

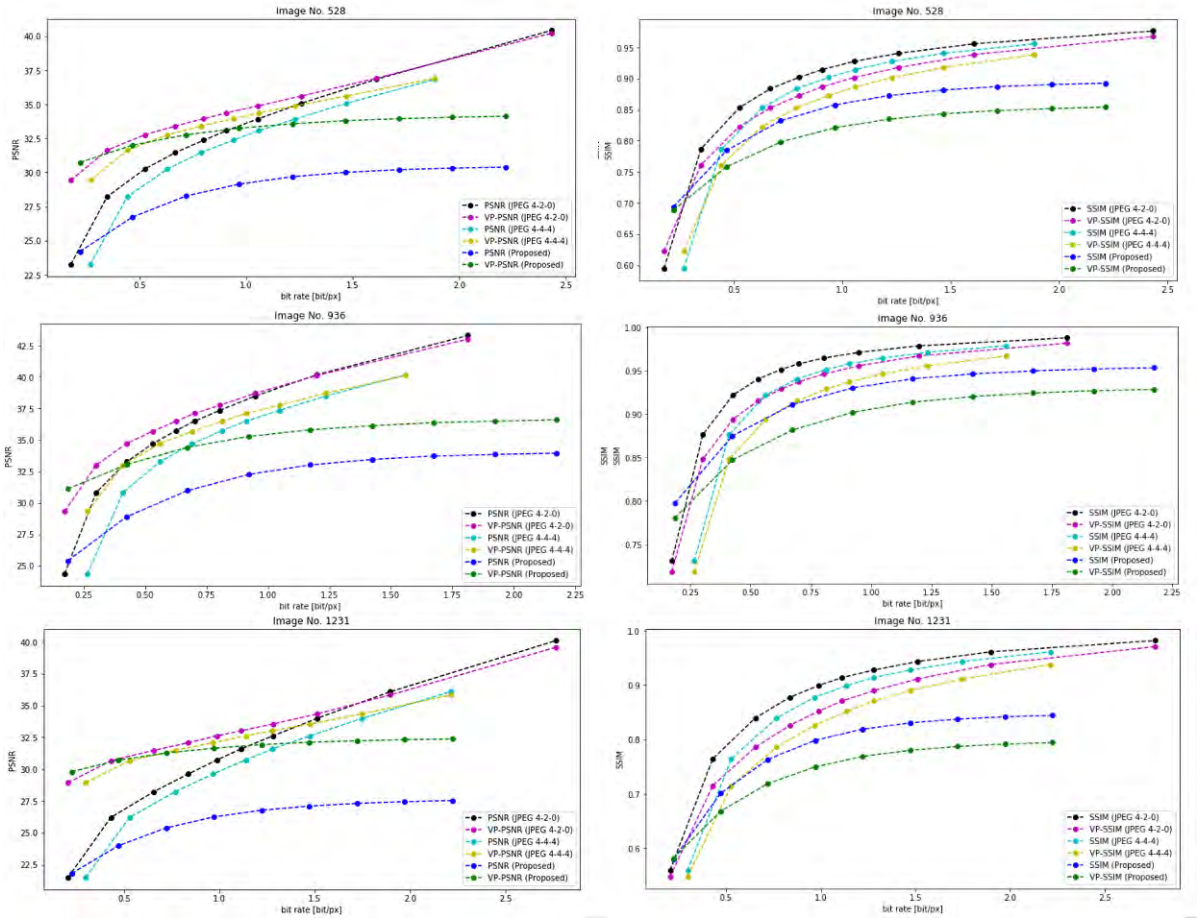

Figure 40: Metrics results on the viewports for JPEG and the proposed method. Up-left: Image No. 528 PSNR, Up-right: Image No. 528 SSIM, Middle-left: Image No. 936 PSNR, Middle-right: Image No. 936 SSIM, Bottom-left: Image No. 1231 PSNR, Bottom-right: Image No. 1231 SSIM

It is clear in Figure 40 that the gap of the PSNR and SSIM curves for the whole omnidirectional image is wide between the results from the proposed method and JPEG. However, after generation

of the view ports, the gap between these two curves reduces and it means that the proposed model becomes closer (or even passes) JPEG in the plots. It is also clear from the significant difference between the results of planer metrics on our proposed model when it is applied on the whole image and when it is applied on the viewports. This big change does not happen in JPEG. These are signs that this method considers the geometery of these specific images. Additionally, as we see in the results, our method performs better in comparison to JPEG $4:4:4$ rather than JPEG $4:2:0$ as we expected. In fact as JPEG is not perfect to find the color in low rates, when more samples are dedicated to chroma components, its quality generally degrades in that rate.

6 Conclusion

In this study, a method has been presented that is able to consider the specific geometry of omnidirectional images in the image compression scheme. The method is based on autoencoders and it can be trained easily for different rates in one pass in an unsupervised manner. As it benefits from spherical patching that is well-aligned with spherical layers, it can be trained easier faster and more efficiently. The model has the potential to compete with approaches like JPEG if it is trained on enough number images with enough filter kernels. Additionally, the following points have been noticed during the process of designing the model:

- Having one spherical convolution layer or spherical residual block that performs the sampling on the correct positions is enough to improve the model. In fact, we think that performing the correct sampling based on the type of projection at the beginning is enough to send the whole model to spherical space and adding more spherical layers (such as spherical convolutions and spherical LSTM cells) does not help. Indeed, after the first spherical layer, the signal is not completely spherical anymore and our results did not not show improvement by adding more spherical layers.

- Performing a precise spherical patching can help not only in reduction of the training time and the required computational power, but also in the quality of the reconstructed images.

- The layers that are benefiting from a kind of averaging, like MaxPool or bilinear interpolation modules are not good choices for the reconstruction of omnidirectional images. In fact, the position of sampling is from high importance in 360° image schemes and averaging reduces the preciseness of sampling and may cause blurring in the reconstructed image.

- Applying the weighted loss functions did not show a significant improvement in our model and it does not exist in our final proposal. However, this result might be because of the specific recurrent neural network method that we used and it cannot be easily generalized to all possible solutions for omnidirectional image compression.

- Pixel shuffle, depth to space is a decent model for upsampling in the decoder for omindirectional image compression.

7 Further Improvements

Although this study has shown the potential of convolutional layers that consider the geometry of the omnidirectional images and proposed a specific model of patching which is well aligned with these models, it is just the first step. The whole arechitucture and parameters like number of filter and layers, learning rate, weight decay, etc. can be fine-tuned to be completely optimized for the problem of omnidirectional image compression in order to improve the model. Also, although our method did not show a significant difference by using the weighted loss functions, this can also be studied more in similar models.

Additionally, a specific entropy coder can be made and trained for this autoencoder to improve the model. Moreover, instead of the simple binarizer that we used, learning for quantization algorithms can also be implemented. There are interesting works in these field such as [66] and [67].

Finally, there are new ideas currently emerging in the field of planar image compression that can be helpful in the field of 360° images as well. Region of Interset based image compression [68] and saliency-based coding [69] can be added that dedicates more bits to the specific defined RoI which is of high importance and less bits are allocated to the rest. For instance, there are novel approaches that are performing object detection before the image compression and allocate more bits to these positions[70]. Using an spherical object detection approach such as what is proposed on [45], these methods can also be added to an omnidirectional image compression schemes and they may provide improvements for the rate-distortion tradeoff.

8 Acknowledgments

I would like to thank Prof. Pascal Frossard for his guidance and supervision and Dr. Roberto Azevedo for all his help during the project and all the efforts he put to review the report. Also, I have to thank Prof. Auke Ijspeert for providing the opportunity of working on this project. Finally, I would like to thank my friends, Javier Morales and Harshita Machiraju for their help whenever I had problems.

9 Bibliography

References

[1] Jin Shin and Soo-Yeong Yi. An image processing for omnidirectional image based on the bresenham raster algorithm. 261:8–16, 01 2011.

[2] Samsung Inc. Gear 360 (retrieved-on-20.12.2019). https://www.samsung.com/global/galaxy/gear-360/.

[3] Nikon Inc. Keymission 360 camera (retrieved-on-20.12.2019). https://www.nikonusa.com/en/nikon-products/product/action-camera/keymission-360.html.

[4] Nokia Inc. Nokia ozo (retrieved-on-20.12.2019). https://ozo.nokia.com/.

[5] Google Inc. Build virtual world (retrieved-on-19.12.2019). https://developers.google.com/vr.

[6] Facebook Research Inc. Creating the future of personal and shared reality (retrieved-on-19.12.2019). https://research.fb.com/category/augmented-reality-virtual-reality/.

[7] Zhenzhong Chen, Yiming Li, and Yingxue Zhang. Recent advances in omnidirectional video coding for virtual reality: Projection and evaluation. *Signal Processing*, 146, 05 2018.

[8] VR and Fun. Equirectangular projection of the world (retrieved-on-19.12.2019). https://www.vrandfun.com/google-looks-to-sharpen-qualities-of-\vr-content-with-a-new-projection-technique-called-eac/.

[9] OMAR MAHDI, Mazin Mohammed, and Ahmed Mohamed. Implementing a novel approach an convert audio compression to text coding via hybrid technique. 11 2013.

[10] M.A. Alkhalayleh and Mohammed Otair. A new lossless method of image compression by decomposing the tree of huffman technique. 15:79–96, 01 2015.

[11] P. Prandoni and M. Vetterli. Digital signal processing course. https://www.coursera.org/learn/dsp.

[12] Dong Liu, Yue Li, Jianping Lin, Houqiang Li, and Feng Wu. Deep learning-based video coding: A review and A case study. *CoRR*, abs/1904.12462, 2019.

[13] Graham Hudson, Alain Léger, Birger Niss, István Sebestyén, and Jørgen Vaaben. JPEG-1 standard 25 years: past, present, and future reasons for a success. *Journal of Electronic Imaging*, 27(4):1 – 19, 2018.

[14] D. Taubman and M. Marcellin. *JPEG2000 Image Compression Fundamentals, Standards and Practice*.

[15] Graham Hudson, Alain Léger, Birger Niss, István Sebestyén, and Jørgen Vaaben. JPEG-1 standard 25 years: past, present, and future reasons for a success. *Journal of Electronic Imaging*, 27(4):1 – 19, 2018.

[16] J. Xiao, K. A. Ehinger, and A. Oliva, A.and Torralba. Recognizing scene viewpoint using panoramic place representation. *Proceedings of 25th IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[17] Abir Hussain, Ali Al-Fayadh, and Naeem Radi. Image compression techniques: A survey in lossless and lossy algorithms. *Neurocomputing*, 300, 03 2018.

[18] David Huffman. A method for the construction of minimum-redundancy codes. *Resonance*, 11:91–99, 02 2006.

[19] Telairity. Digital color coding (retrieved-on-21.12.2019). `https://web.archive.org/web/20140107171831/http://www.telairity.com/assets/downloads/Digital%20Color%20Coding.pdf`.

[20] the free encyclopedia Wikipedia. Chroma subsampling (retrieved-on-21.12.2019). `https://en.wikipedia.org/wiki/Chroma_subsampling`.

[21] Francesca De Simone, Pascal Frossard, Paul Wilkins, Neil Birkbeck, and Anil Kokaram. Geometry-driven quantization for omnidirectional image coding. In *Picture Coding Symposium (PCS), 2016*, pages 1–5. IEEE, 2016.

[22] Mira Rizkallah, Francesca De Simone, Thomas Maugey, Christine Guillemot, and Pascal Frossard. Rate Distortion Optimized Graph Partitioning for Omnidirectional Image Coding. page 5, 2018.

[23] Antonio Ortega, Pascal Frossard, Jelena Kovačević, José M. F. Moura, and Pierre Vandergheynst. Graph signal processing: Overview, challenges and applications, 2017.

[24] Djamal Alouache, Z. Ameur, and Djemaa Kachi. Catadioptric images compression using an adapted neighborhood and the shape-adaptive dct. *Multimedia Tools and Applications*, pages 1 – 17, 2019.

[25] International Telecommunications Union. H.264 : Advanced video coding for generic audiovisual services (retrieved-on-22.12.2019). `https://www.itu.int/rec/T-REC-H.264`.

[26] Z. Wang, E. P. Simoncelli, and A. C. Bovik. Multiscale structural similarity for image quality assessment. 2:1398–1402 Vol.2, Nov 2003.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. 2012.

[28] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.

[29] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(4):623–656, 1948.

[30] Aäron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *CoRR*, abs/1601.06759, 2016.

[31] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. 27:37–49, 02 Jul 2012.

[32] Johannes Ballé, Valero Laparra, and Eero P. Simoncelli. End-to-end optimized image compression. *CoRR*, abs/1611.01704, 2016.

[33] Renata Khasanova and Pascal Frossard. Geometry aware convolutional filters for omnidirectional images representation. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3351–3359, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[34] Shengwei Wang, Hongkui Wang, Sen Xiang, and Li Yu. Densely connected convolutional network block based autoencoder for panorama map compression. *Signal Processing: Image Communication*, 80:115678, 2020.

[35] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks. *CoRR*, abs/1608.05148, 2016.

[36] Johannes Ballé, Valero Laparra, and Eero P. Simoncelli. Density modeling of images using a generalized normalization transformation, 2015.

[37] Thierry Dumas, Aline Roumy, and Christine Guillemot. Autoencoder based image compression: can the learning be quantization independent?, 2018.

[38] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[39] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[40] Ivo Danihelka, Greg Wayne, Benigno Uria, Nal Kalchbrenner, and Alex Graves. Associative long short-term memory. *CoRR*, abs/1602.03032, 2016.

[41] Pytorch Documentation. Pixel shuffle (retrieved-on-30.12.2019). https://pytorch.org/docs/stable/nn.functional.html?highlight=pixel20shuffle#torch.nn.functional.pixel_shuffle.

[42] R. Jozefowicz, W. Zaremba, and I Sutskever. An empirical exploration of recurrent network architectures. *Journal of Machine Learning Research*, 2015.

[43] an anonymous programmer 1zb. Full resolution image compression with recurrent neural networks (retrieved-on-20.12.2019). https://github.com/1zb/pytorch-image-comp-rnn.

[44] Taco S. Cohen, Mario Geiger, Jonas Köhler, and Max Welling. Spherical cnns. *CoRR*, abs/1801.10130, 2018.

[45] Benjamin Coors, Alexandru Paul Condurache, and Andreas Geiger. Spherenet: Learning spherical representations for detection and classification in omnidirectional images. pages 525–541, 2018.

[46] C. W. Hsiao. Spherenet-pytorch (retrieved-on-21.12.2019). https://github.com/ChiWeiHsiao/SphereNet-pytorch.

[47] H. S. M. Coxeste. *Introduction to Geometry, 2nd. edition.* Wiley, 1989.

[48] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C: the art of scientific computing*. Cambridge University Press, 1992.

[49] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. pages 265–283, 2016.

[50] Facebook Inc. Pytorch, deep learning framework for python. https://pytorch.org/docs/stable/index.html.

[51] F. Fleuret. Ee-559 – epfl – deep learning course notes. https://fleuret.org/ee559/.

[52] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P. Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. *CoRR*, abs/1609.05158, 2016.

[53] P. Grover. 5 regression loss functions all machine learners should know (retrieved-on-20.12.2019). https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0.

[54] Y. Sun, A. Lu, and L. Yu. Weighted-to-spherically-uniform quality evaluation for omnidirectional video. *IEEE Signal Processing Letters*, 24(9):1408–1412, Sep. 2017.

[55] W. Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, 1976.

[56] Zhou Wang, Alan Bovik, Hamid Sheikh, and Eero Simoncelli. Image quality assessment: From error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13:600 – 612, 05 2004.

[57] Maximilian Riesenhuber and Tomaso Poggio. Riesenhuber, m. poggio, t. hierarchical models of object recognition in cortex. nat. neurosci. 2, 10191025. *Nature neuroscience*, 2:1019–25, 12 1999.

[58] S. Chen, Y. Zhang, Y. Li, Z. Chen, and Z. Wang. Spherical structural similarity index for objective omnidirectional video quality assessment. In *2018 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, July 2018.

[59] Jiahua Xu, Ziyuan Luo, Wei Zhou, Wenyuan Zhang, and Zhibo Chen. Quality assessment of stereoscopic 360-degree images from multi-viewports, 2019.

[60] Imageio. Npz numpy's compressed array format (retrieved-on-20.12.2019). `https://imageio.readthedocs.io/en/stable/format_npz.html`.

[61] C. Tu, E. Takeuchi, A. Carballo, and K. Takeda. Point cloud compression for 3d lidar sensor using recurrent neural network with residual blocks. pages 3274–3280, May 2019.

[62] Jeffrey Ede. Autoencoder reconstructed image (output) are not clear as i want (retrieved-on-20.12.2019). `https://stackoverflow.com/questions/54573221/autoencoder-reconstructed-image-output-are-not-clear-as-i-want`.

[63] Shih-Han Chou, Cheng Sun, Wen-Yen Chang, Wan-Ting Hsu, Min Sun, and Jianlong Fu. 360-indoor: Towards learning real-world objects in 360° indoor equirectangular images, 2019.

[64] Nikolaos Zioulis, Antonis Karakottas, Dimitrios Zarpalas, and Petros Daras. Omnidepth: Dense depth estimation for indoors spherical panoramas. pages 448–465, 2018.

[65] Y. Rai, P. Le Callet, and P. Guillotel. Which saliency weighting for omni directional image quality assessment? In *2017 Ninth International Conference on Quality of Multimedia Experience (QoMEX)*, pages 1–6, May 2017.

[66] J. Wang, X. Tao, X. Liu, N. Ge, and J. Lu. Quantization and entropy coding scheme for dictionary learning based image compression. In *2016 IEEE 84th Vehicular Technology Conference (VTC-Fall)*, pages 1–5, Sep. 2016.

[67] P. A. Chou, T. Lookabaugh, and R. M. Gray. Entropy-constrained vector quantization. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(1):31–42, Jan 1989.

[68] S. Han and N. Vasconcelos. Object-based regions of interest for image compression. In *Data Compression Conference (dcc 2008)*, pages 132–141, March 2008.

[69] H. Wei, X. Zhou, W. Zhou, C. Yan, Z. Duan, and N. Shan. Visual saliency based perceptual video coding in hevc. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2547–2550, May 2016.

[70] M. Moradi, F. Bayat, and M. Charmi. Concept-aware web image compression based on crowd-sourced salient object detection. In *2019 5th International Conference on Web Research (ICWR)*, pages 221–227, April 2019.

[71] Franzen. R. et al. Kodak lossless true color image suite (retrieved-on-20.12.2019). `http://r0k.us/graphics/kodak/`.

10 Appendix A1: Version 0, Preliminary Study Based on end-to-end Autoencoders

In this part, the preliminary experimants that have been done on Ballé et al.[32] model are presented. These experiments have been done as the first step in this study. Firstly, the Ballé et al. model was trained on the Kodak planar image dataset[71] and a 24 images subset of SUN360[16] omnidirectional dataset (look at Section 5.2). As this model should be trained for every different rate, to save the training time and computational power, Tensorboard API has been used and the rates were checked in terms of the number of training epochs. When the curves became almost flat, we finished the training. After this, the model was tested on the test set (look at Section 5.2). The Figure 41 shows the result of metrics on image No. 936:

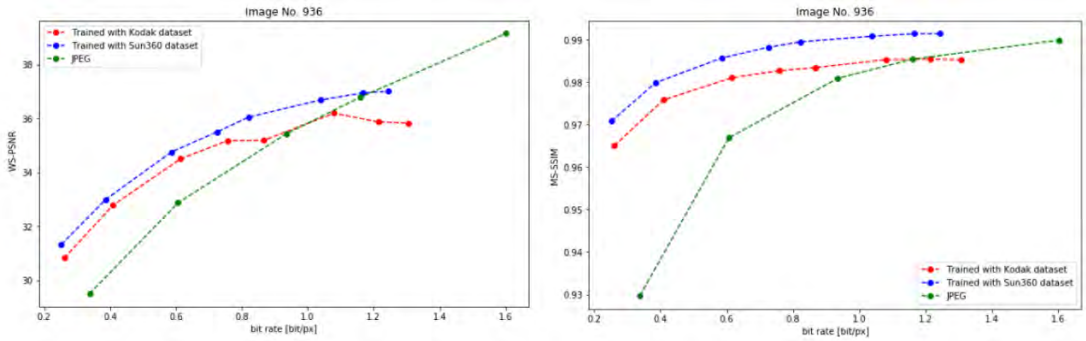

Figure 41: The results of Ballé et al. model trained on 24 images from Kodak and SUN360 datasets in comparison to JPEG.

As it is shown in the Figure 41, Ballé et al. method performs well in comparison to JPEG for low rates. Training it on omnidirectional images shows a bit of improvement in the results. As the next step, we changed the weight function of this model from the MSE to weighted-MSE (look at Section 3.4.2). As WMSE applies the weights based on the position of the specific pixel, it is better not to implement it together with a method of patching, so the patching has been removed. Additionally, in order to have a fair comparison between this model and the original Ballé et al. model, patching has been removed from their model too. Figure 42 shows the results in terms of the metrics for one of the images:

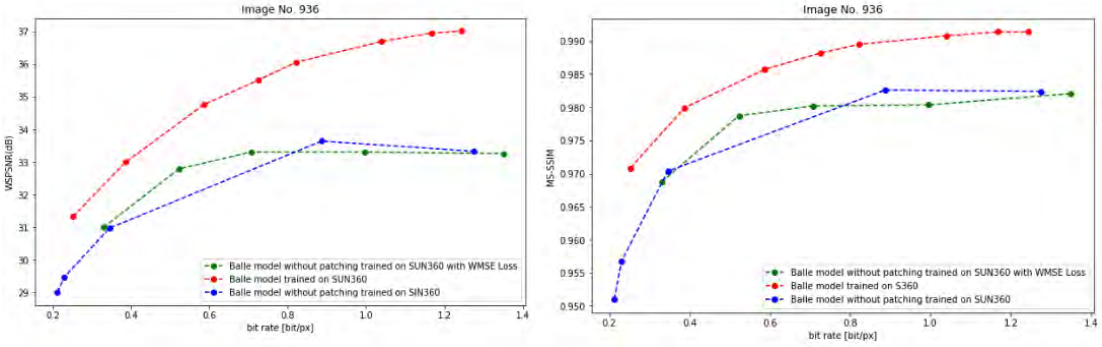

Figure 42: The results of Ballé et al. model trained on 24 images with WMSE loss (and without patching), in comparison to normal Ballé et al. model and normal Ballé et al. model without patching.

The above plots show that changing the loss function to WMSE just makes a little difference in comparison to the original Ballé et al. without patching and as we expected, the change is not significent. In the next step, we tried to change the convolutional transforms in the Ballé et al. model to spherical convolutions from Spherenet method[45] but due to the lack some of the important functions such as `nn.functional.grid_sample` in TensorFlow API, the code has some additional Python loops and consequently it was ultra-slow. Indeed, we were not able to train it with our current available computational power. So finally, due to this reason and the fact that we have to train this model for every different rate which was not feasible for us in computational power point of view, we decided to change the base model we used for our different versions to the work of Toderici et al.[35].