Lossy Compression of Climate Data using Convolutional Autoencoders

Master Thesis of

Silke Teresa Donayre Holtz

At the Department of Informatics

Reviewers:	Prof. Dr. Peter Sanders
	Prof. Dr. Peter Braesicke
Advisors:	Dr. Uğur Çayoğlu
	Prof. Dr. Pascal Friederich

Time Period: 25th January 2021 - 25th August 2021

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, August 25, 2021 Silke Teresa Donayre Holtz

Acknowledgments

Although this work lists a single author, it would not have been possible without more people's involvement. It is my pleasure to dedicate these few lines to thank them.

First of all, I would like to thank Dr. Uğur Çayoğlu for his excellent advisory. His interest and enthusiasm on me at every stage of my thesis helped me to a very great extent to accomplish this task.

I would also like to acknowledge my deepest thanks to Prof. Dr. Pascal Friederich for teaching me more about neural networks and also for inviting me to be part of the Artificial Intelligence for Materials Sciences (AiMat) group. Without his guidance, this work wouldn't have been possible.

Furthermore, I would like to thank Prof. Dr. Peter Sanders and Prof. Dr. Peter Braesicke for their kind advice and help in completing this project.

In addition, I would like to thank my friends. Even though many of them are not physically near, they always listened and encouraged me to keep going.

Last but not least, I would like to thank my family, my mother Sabine, my dad Javier and my sister Katia, for their love, support and encouragement. Without them, I would not have been able to pursue my master's. Moreover, I would like to take this opportunity to give a special thanks to my father, even though we couldn't end this journey together, remembering his teachings and words of support allowed me to finish this thesis.

Abstract

Climate model simulations, coupled with the increasing computing power of high-performance computing (HPC) centres, are generating huge amounts of climate data. Preserving such data presents a challenge as storage resources are mostly limited. Techniques like limiting the simulation length have been used to reduce the amount of climate data. However, such techniques can have a negative impact on scientific goals. This problem underlines the importance of data compression, the process in which the size of the data is reduced by removing redundant information.

In this work, we present a novel lossy compression algorithm. Lossy compression, unlike lossless compression, does not allow an exact reconstruction of the original data. However, this type of compression has the advantage that data volumes can be reduced further. As long as no statistically significant effects are caused, lossy compression can effectively reduce climate data without compromising scientific conclusions.

The proposed method uses convolutional autoencoders to compress selected ERA5 weather data from the European Centre for Medium-Range Weather Forecasts (ECMWF). The algorithm is compared with current state-of-the-art lossy compression algorithms. Our results show that convolutional autoencoders in combination with lossless residual compression are superior to state-of-the-art algorithms for lossy compression at almost all error-bounds tested. We show that for absolute errors between 0.3 and 1.0 Kelvin, the compression algorithm achieves 1.3 to 4 times higher compression factors than state-of-the-art.

Kurzfassung

Mit der zunehmenden Rechenleistung von Hochleisungsrechenzentren (HPC) und neuartigen Klimamodell-Simulationen, erzeugen Klimawissenschaftler riesige Mengen an Klimadaten. Die Aufbewahrung dieser Daten stellt eine große Herausforderung dar, weil die Speicherressourcen meist begrenzt sind. Aktuell wurden rudimentäre Techniken, wie die Begrenzung der Simulationsdauer eingesetzt, um die Menge der Klimadaten zu reduzieren. Solche Techniken können sich jedoch negativ auf die wissenschaftlichen Ziele auswirken. Dieses Problem unterstreicht die Bedeutung der Datenkomprimierung, d.h. des Prozesses, bei dem die Größe der Daten durch die Entfernung redundanter Informationen reduziert wird.

In dieser Arbeit stellen wir einen neuartigen verlustbehafteten Komprimierungsalgorithmus vor. Bei der verlustbehafteten Komprimierung können die ursprünglichen Daten im Gegensatz zur verlustfreien Komprimierung nicht fehlerfrei rekonstruiert werden. Dafür hat diese Art der Komprimierung den Vorteil, dass Datenmengen stärker reduziert werden können. Solange keine statistisch signifikanten Effekte hervorgerufen werden, kann die verlustbehaftete Komprimierung Klimadaten effektiv reduzieren, ohne wissenschaftliche Schlussfolgerungen zu beeinträchtigen.

Die vorgestellte Methode verwendet einen Convolutional-Autoencoder, um ausgewählte ERA5-Klimadaten des Europäischen Zentrums für mittelfristige Wettervorhersage (ECMWF) zu komprimieren. Der Algorithmus wird mit dem aktuellen Stand der Technik von verlustbehafteten Komprimierungsalgorithmen verglichen. Die Ergebnisse dieser Arbeit zeigen, dass Convolutional-Autoencoder in Kombination mit verlustfreier Residualkompression bei nahezu allen getesteten Fehlergrenzen dem Stand der Technik für verlustbehaftete Kompression überlegen sind. Es wird gezeigt, dass der Kompressionsalgorithmus bei absoluten Fehlern zwischen 0.3 und 1.0 Kelvin, einen um den Faktor 1.3 bis 4 höheren Kompressionsfaktor erreicht.

Contents

1 Introduction

With the increasing computational power of high-performance computing (HPC) centres, climate simulations have been increasing enormously, generating vast amounts of data by about 233 TB per day [1]. Climate models simulate more than 100 climate variables, and the simulation spans often decades. Moreover, the data presented in scientific publications must be stored for at least ten years. Therefore, climate research centres must make great investments regarding storage and bandwidth [2]. However, it is not always possible to cover such costs. Asch et al. [3] raise concerns about the growing need for storage and bandwidth. The authors state that there is a need of storing data efficiently through data-reduction methods like data compression.

Data compression represents information in a compact form which is created by identifying and using patterns that are in the data [4]. By using some extra computation, data can be compressed and decompressed, reducing the number of transferred bytes and improving I/O performance reducing storage requirements. [5]. There are many different compression techniques with different approaches. One of the approaches that is lately being encouraged to be used is the Machine Learning (ML) approach. [6].

ML methods have become very popular and essential areas of research [7], with more than 100 papers published per day [8]. There has been a growing interest in the application of these data-driven techniques to scientific research. ML methods have already been used in climatology for precipitation forecasting [9], improving subseasonal forecasting in the western U.S.A. [10] and other applications [11] [12] reaching promising results. Even though many of the applications are about forecasting, ML methods can also be employed in other fields like data compression.

In this thesis we employ a ML method called deep convolutional autoencoder to perform data compression of large netCDF [13] climate datasets. We specifically compress data from the ERA5 hourly data on pressure levels from 1979 to present dataset provided by the European Centre for Medium-Range Weather Forecasts (ECMWF).

1.1 Motivation

During the last decades, the popularity of ML has been increasing. They have achieved human-level performance in many challenging applications due to increased data volume, processing power, and understanding. Many ML models have outperformed traditional models, like in the field of computer vision. We are motivated to test if a ML-based compressor could also outperform traditional lossy compression algorithms used in climatology.

1.2 Thesis Purpose

The purpose of the present thesis is to develop a lossy compression algorithm using ML to achieve the greatest possible impact on climate data's compression factor. This algorithm aims to surpass the compression factors gotten with state-of-the-art floating-point compressor libraries like SZF and ZFP.

1.3 Thesis Structure

The thesis is organized as follows. Chapter 2 provides the fundamentals of data compression, neural networks, convolutional neural networks, autoencoders and climate data. Chapter 3 discusses related work in the field of climate data compression using autoencoders and ML and other techniques used to achieve data compression in general. Chapter 4 presents the proposed compression algorithm, the architecture, the data, the convolutional autoencoder, and the processing of the residuals. Chapter 5 presents the results obtained with the proposed compression algorithm. We detail the training process and the hyperparameter optimization; we analyze the models' behavior and compare the models' compression factor with state-of-the-art algorithms. Finally, chapter 6 includes the conclusions and further recommendations that should be implemented in the future.

2 Fundamentals

This chapter gives a brief introduction to the basics of data compression, neural networks, and climate data. The goal of this chapter is to provide basic knowledge to the reader about the aforementioned topics. The chapter starts by describing data compression, its classification and different performance metrics used to evaluate the quality of a compression algorithm. We also talk about the scheme of a compression algorithm and give a brief introduction to four compression algorithms that will be used in this work. Then, we introduce the concept of neural networks and their components. Moreover, we introduce convolutional neural networks and convolutional autoencoders. Finally, since we are working with climate data, a brief introduction about its structure and properties are given.

2.1 Data Compression

Data compression is the process of encoding, restructuring, or otherwise altering data to reduce its size. A data compression algorithm consists of two entities, the encoder, and the decoder. The encoder takes an input $\mathcal{X}$ and generates a representation $\mathcal{X}_c$ that tries to need less storage space by storing the information in a different way. The decoder does the opposite, it takes as input a representation $\mathcal{X}_c$ and generates a reconstruction $\mathcal{Y}$ which will be the same or almost the same as input $\mathcal{X}$ [4].

2.1.1 Types of Compression

Data compression algorithms can be classified into two broad classes: lossless compression and lossy compression. The chosen category will depend on the requirements of the user [4].

Lossless techniques do not present any loss of information when compressing and decompressing data. In other words, when decompressing $\mathcal{X}_c$, $\mathcal{Y}$ will be identical to $\mathcal{X}$ [4]. There are many cases where the decompressed data is required to be identical to the original, e.g., bank records. If there are compression errors, it could lead to the erroneous addition or substraction of money from a customers account. Nonetheless, there are other situations where compression errors are allowed to further compress

the data. In these situations, lossy compression is used.

As the name suggests, lossy compression algorithms present a certain amount of information loss. This means that the reconstruction $\mathcal{Y}$ will not be identical to $\mathcal{X}$ [4]. An excellent example of this is multimedia. As long as there are no strong artifacts in the decompressed sound, image or video, we will not be bothered if the sound or images are not exactly the same as in the original version.

Compression algorithms can be further classified as following [14]: symmetric or assymetric, universal or non-universal, and block or streaming mode.

- **Symmetric vs. Assymetric**
  The architecture of the encoder and the decoder in a symmetric compression algorithm are the same but differ in direction. In an assymetric architecture one of the entities has more workload than the other.

- **Universal vs. Non-universal**
  A universal compression algorithm has no statistical information about the original data. A non-universal compression algorithm has a predefined knowledge about the data to be compressed e.g. compression of climate data.

- **Block vs. Streaming Mode**
  A block mode compression algorithm divides the data into blocks and compresses them independently. A streaming mode compression algorithm process the data value by value. After reading and encoding one value, the following value is processed until finished.

2.1.2 Compression Metrics

Since there are many different applications, distinct ways have been developed to describe and measure the performance of a compression algorithm, e.g, time and memory complexity, amount of compression, and how close the reconstruction is to the input [4]. In this work, we will be using three measurements, compression factor, distortion and peak-signal-to-noise-ratio.

Compression factor

One of the most straightforward ways to compare different compression algorithms is by comparing the number of bytes of an input file with the number of bytes of the compression algorithm's output. There are two metrics: compression factor (CF) and compression ratio (CR).

The CF is given as follows [4]:

$$\text{CF} = \frac{\mathcal{X}}{\mathcal{X}_c}$$

where CR is the inverse of the CF. A percentage can also be given as:

$$\text{CF}(\%) = (1 - \frac{\mathcal{X}}{\mathcal{X}_c}) \cdot 100$$

Suppose we have a file of size of 98,592 bytes, and its compressed version a size of 24,648 bytes. The CR would be 4, the CF 0.25, and the CF percentage 75%, meaning that the compressed file is four times smaller than the original file.

Distortion

As mentioned in 2.1.1, lossy compression loses information in the process. To measure the efficiency of a compression algorithm with such characteristics, the difference between $\mathcal{X}$ and $\mathcal{Y}$ is calculated. This difference is called distortion. There are two popular approaches, the mean squared error (MSE) and the mean abolute error (MAE) [4]. Let $x \in \mathbb{R}^n$ be the input of the compression algorithm and $y \in \mathbb{R}^n$ the output, where n is the number of elements in the vector. The MSE and the MAE are defined as [4]:

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

$$\text{MAE}(x, y) = \frac{1}{n} \sum_{i=1}^{n} |x_i - y_i|$$

Another measure is the peak-signal-to-noise-ratio (PSNR). This metric measures the similarity between the input and output of the compression algorithm, and is defined as [15]:

$$\text{PSNR}(x, y) = 20 \log_{10}(range(x)) - 10 \log_{10}(\text{MSE}(x, y))$$

where range$(\cdot)$ represents the value range of the input i.e., the absolute difference between its highest value and lowest value. The PSNR is expressed in terms of decibels indicating higher values to have a higher similarity.

2.1.3 Modeling and Coding

Up to now, we have seen that a compression algorithm can be lossless or lossy. In this subsection, we will describe how a compression algorithm is developed. For the development of a compression algorithm, two phases are generally needed: modeling and coding [4]. The modeling phase tries to extract and describe redundant information in the data in order to get a compact representation of it. The compression algorithm's performance will depend on how well this compact representation was formed. The coding phase is the description of the compact representation. For example, given the series of numbers: 4, 5, 6, 7, 8, we could model it as a straight line and code it with the equation $x = n + 4$ where $0 \leq n \leq 4$.

Next, we will introduce the compression algorithms used in this thesis, bzip2, fpzip, zfp and SZ.

2.1.4 Relevant Compression Algorithms

bzip2 and fpzip (lossless)

bzip2 [16] is an open-source lossless universal file compression algorithm. During compression, bzip2 passes the data through several stacked compression algorithm layers. The amount of layers can be set by the user in order to make the compression algorithm faster. The CF increases with the amount of layers used [17].

fpzip [18] is an open-source library for lossless compression of large multidimensional floating-point arrays. fpzip is a prediction-based compression algorithm. It traverses the data in some order, e.g., row-by-row, and predicts every value from a subset with already encoded data. The predicted data and actual values are then transformed to an integer representation which are used to compute the residuals [18].

zfp and SZ (lossy)

zfp [19] is an open-source library for compression of integer and floating-point arrays that support high throughput when encoding and decoding [19]. zfp is a transformation-based compression algorithm, this means, to compress the data, a series of transformations are done in order to eliminate redundancies. It works well for correlated arrays, e.g., physics simulations. To achieve high compression factors, zfp compresses the data lossy, with error bounds set by the user. zfp also supports lossless compression by setting that error to zero [19].

SZ [20] is an open-source modular parametrizable lossy compression library for integer and floating-point data. Like zfp, SZ is, among others, used in physics simulations, and it does error-bounded compression. There is not a single compressor which is universally better than the other, since the performance varies depending on the type of application. However, SZ and ZFP are considered to be two of the best error-bounded lossy compressors currently available [21].

2.2 Artificial Neural Networks

ML is the science of programming computers so they can "learn from data without being explicitly programmed" [22]. Using the data, ML algorithms build models in order to solve specific tasks such as prediction and regression, image and text classification, speech recognition, anomaly detection and natural language processing. ML algorithms are often categorized based on whether or not they are trained with supervision. The two main categories are supervised and unsupervised learning [22]. In supervised learning, the data x fed to the algorithm includes labels y [22], and the model learns a model $p(y|x)$. In unsupervised learning, no labels are provided to the model. In other words, the model tries to learn $p(x)$ to solve tasks such as dimensionality reduction or clustering [22]. In dimensionality reduction, the model tries to find low dimensional data representations without losing too much information by, e.g., by merging several correlated features into one. There are different ML algorithms used to solve the aforementioned tasks. One of them are artificial neural networks (ANN).

An ANN is a ML model that aims to mimic the capabilities of biological neural networks found in human brains using artificial neurons. The information that an artificial neuron receives as input will be either transmitted or inhibited depending if the input to the artificial neuron is excitatory or not. Like the human brain, ANNs need to learn from examples. For a long time, this task was computationally difficult because of the limited amount of data and processing power. However, ANNs have recently become very popular, thanks to the growing amounts of data and increasing computing performance [7]. ANNs were even shown to outperform human experts e.g., in the ImageNet Challenge [23], encouraging researchers to keep using and improving ANNs over different fields.

To get a better understanding on how ANNs work, we will start by introducing the simplest architecture, the perceptron.

2.2.1 Perceptron

The perceptron is a linear classifier that consists of a single layer of artificial neurons. Let j be a neuron with input $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$ and weights $w = (w_1, \ldots, w_n)^T \in \mathbb{R}^n$, with n being the number of features. The output is calculated by the following function [22]:

$$y = \sigma(\sum_{i=1}^{n} x_i w_i + w_0)$$

where σ is an activation function (see section 2.2.5), normally the Heavyside step function [22], w_0 the bias weight of neuron j and $y \in \mathbb{R}$ the output of the network. An illustration of a perceptron with the respective parameters can be seen in Figure 2.1.

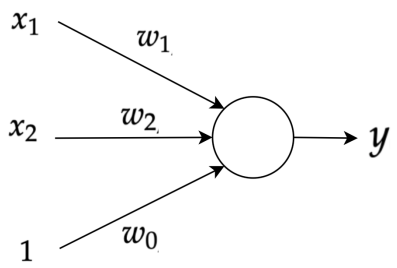

Figure 2.1: *Perceptron with two inputs and one output.*

The major drawback of the perceptron is that it can only be used for binary or multilabel classifications with linearly separable output [22]. In order to be able to solve complex tasks, a multi-layer architecture can be build by stacking several layers. This architecture is called multi-layer perceptron (MLP) network.

2.2.2 Multi-layer Perceptron

An MLP consists of an input layer, multiple layers of artificial neurons also called hidden layers and an output layer. All neurons are mostly fully connected with the ones from the previous and next layer, the flow goes from the input to the output without back loops and it contains at least one hidden layer [22]. Every layer, except the output layer, includes a bias neuron. MLPs with non-linear activation functions solve perceptrons major drawback, their output is non-linearly dependent given the input.

An MLP has τ layers (the input layer does not count), where $\tau \geq 2$. Let C be a set of neurons that is split into mutually disjunct subsets called layers $L_1, \ldots, L_\tau$. The

input layer would be then L_1, the hidden layers $L_2, \ldots, L_{\tau-1}$ and the output layer L_τ. Each neuron in layer L_i is connected to every neuron in layer L_{i+1} with $i < \tau - 1$ with a set of weights W and a set of bias b. That is to say; all neighboring layers form complete bipartite graphs [24]. To have a better understanding, Figure 2.2 provides a representation of an MLP with only one hidden layer.

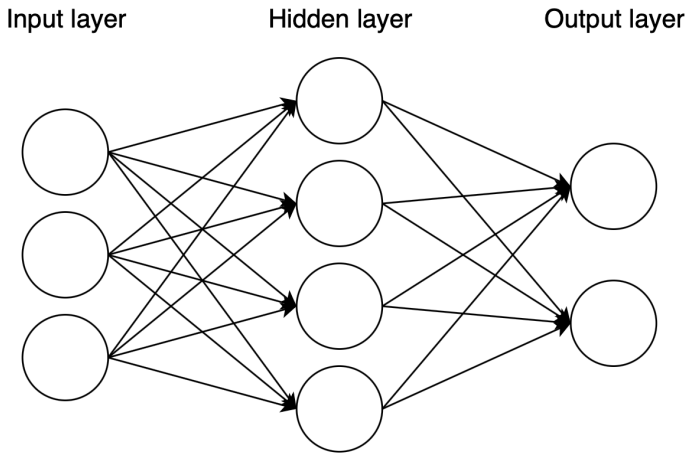

Figure 2.2: *Multi-layer Perceptron with one hidden layer.*

Now that we are more familiar with neural networks, the next step is to understand how such ANNs are trained.

2.2.3 Training Process

The training process aims to find the optimal set of network parameters, i.e., weights and biases, to solve a given problem by using a training dataset. A training dataset is a set of data used only during the training process. An optimization algorithm searches through a space of possible weights and bias values that provide a good performance. One of the most common optimization algorithms is called gradient descent (GD). It starts by initializing the network's parameters with random values. Inputs are then iteratively fed to the network, which calculates an output given the iteration's input. The output is analyzed, the error backpropagated, and the network's parameters are adjusted to minimize the error, thus achieving better results. For example, let $\theta \in (W \cup b)$ be a parameter to be optimized, and the network's loss function L be defined as [22]:

$$L = \frac{1}{n} \sum_{i=1}^{n} (Net_\theta(x_i) - y_i)^2$$

where n is the number of samples in the training data, $Net(x_i)$ the output of the network given data x_i, and y_i the observed label in the training dataset. To update the parameters, a learning rule is followed defined by the following equation [22]:

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}$$

where η is the learning rate deciding the magnitude of each update. For each parameter of the network, the gradient of the loss function with respect to the parameter is calculated and updated by changing the parameters value to the opposite direction of the gradients $\frac{\partial L}{\partial \theta}$.

GD aims to find a global minimum for the loss function through many iterations with the given data and parameters. After the training process, the network is evaluated with another set of data called validation dataset. The evaluation process provides information about how well the network performs with data not used in the training process, i.e, it evaluates how well the network generalizes.

2.2.4 Loss Function

The loss function is used to evaluate the performance of a set of network parameters. When training neural networks, we typically want to minimize the error. Even though it may sound simple, the selection of the loss function can be challenging since we need to capture the goal of the search with it [25]. One of the most used loss functions is the MSE:

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

2.2.5 Activation Function

We mentioned in Section 2.2.1 the use of activation functions. In a perceptron, we normally use the Heaviside step function, defined as [22]:

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

or the sign function [22]:

$$\text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

Nevertheless, perceptrons can use any activation function, thus a perceptron and single neurons in ANNs are the same. Other activation functions are mostly used in MLPs being one of them the rectified linear unit (ReLU). ReLU is defined as [26]:

$$\text{ReLU}(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

There are other activation functions such as Sigmoid and Tanh [22] that are often used in MLPs. ReLUs are often less expensive than other activation functions since it can be implemented by thresholding a matrix of activations at zero. ReLU also leads to sparsity [27], which is desirable since it avoids overfitting by doing the weights of each layer more precise by making some of the features insignificant. However, if there is a large gradient flowing through the network, it could update the weights in a way so that some inputs to the ReLU always remain negative, meaning that the output of some neurons will always be zero independently from the input. This results in a network with many inactive neurons that will be not afected by GD. Nevertheless, it can be mitigated by reducing the learning rate or using a variant of ReLU called LeakyReLU, which allows a small positive gradient when the unit is not active.

2.2.6 Regularization

Once we are finished training a network, we might have the problem that the results were a lot better with the training set than the validation set. The problem is called overfitting, and it occurs when a model fits exactly against its training set, thus, being unable to generalize. To prevent overfitting, regularization methods are used. One common way of regularization is to influence the weights in specific ways by adding a new term to the networks loss function. We can control how much of this term will influence by setting a parameter λ. If λ is set too high, we might end with a non-optimal solution since it will influence the gradient more than the data itself. Two of the most common are L1 and L2 regularization, we will briefly introduce both of them.

L1 Regularization

Also called Lasso regression, tends to eliminate the weights of the least important features, i.e., set them to zero. Let the MSE be the networks loss function which takes as input the set of parameters θ, the L1 regression penalizes the weights $w \in \theta$ but not the bias [22]. The L1 regression loss term is defined as [22]:

$$J_{\text{L1}}(\theta) = \lambda \cdot \sum_{w \in \theta} |w|$$

L2 Regularization

Also called ridge regression, forces the network to keep the model weights as small as possible [22]. The L2 regression loss term is defined as [22]:

$$J_{\text{L2}}(\theta) = \lambda \cdot \sum_{w \in \theta} w^2$$

2.3 Convolutional Neural Networks

The ANNs we have seen so far required the input to be 1-dimensional (see 2.2). Let's say we want to process a greyscale image with pixel size $640x480$ with an MLP. To feed the MLP with the image, we would need to flatten the values into a 1-dimensional array. This means we would have $307,200$ values in the input layer. Now, let us suppose we have only one neuron in the hidden layer; since every input value is connected to the hidden neuron and each connection has a weight, the number of parameters would already be larger than $300,000$. Adding more neurons to the hidden layer would significantly increase the number of parameters. Moreover, since every neuron represents a pixel with a fixed position in the image, we would not be able to detect a common feature in different images since it will likely not be at the same position because there is no generalization. However, there is an architecture that tackles this problem. In contrast to MLPs, convolutional neural networks (CNN) learn that it does not matter where in the image they detect certain pattern. CNNs take advantage of the local connectivity of image data [28].

CNNs are multi-layer feed-forward networks that process data that has a known grid-like structure [7]. Instead of working with every single value, the neurons in a CNN layer, also called convolutional layer, consider a small portion of the N-dimensional data instead of being fully-connected. This portion of data is analyzed

through filters for features that might be important. These features are then captured in feature maps, passed through an activation function layer and then passed to the next layer. The neurons from the next layer will be connected to a small portion from the previous layer as well. In this way, the network can concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on [22]. To get a better visualization, Figure 2.3 shows the architecture of a CNN network. The architecture consists of an input, convolutional layers, pooling layers and fully-connected layers. The pooled maps in the figure are generated by pooling layers. Pooling layers do not contain any parameters to be optimized, they only reduce the size of feature maps.

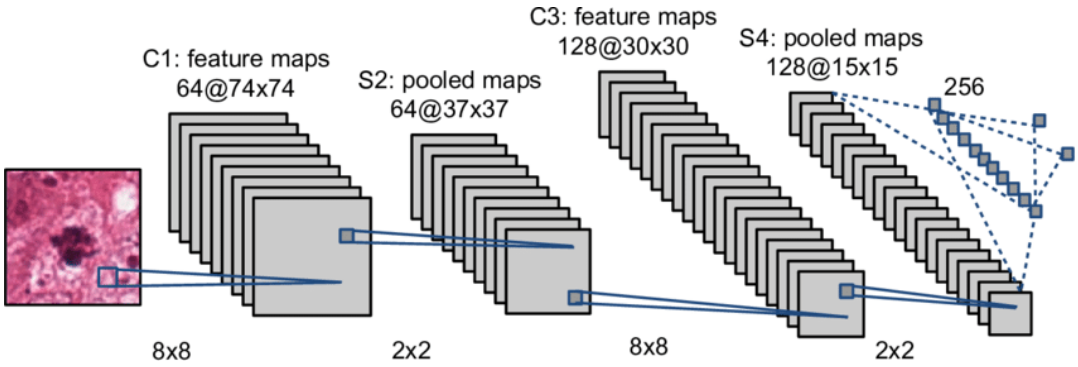

Figure 2.3: *Example of a CNN architecture [29].*

CNNs have been very successful in practical applications such as image classification and segmentation, object detection, video processing, natural language processing, and speech recognition [30].

2.3.1 Convolutional and Deconvolutional Layers

The name of the convolutional layer comes from a mathematical operation called convolution, which is used to extract features. For brevity reasons, we will only explain 2-dimensional convolutional layers which can be easily extended to N-dimensional layers.

A convolutional layer's parameters consist of a set of learnable filters. A filter is small along width and height (spatial size) but extends through the full depth of the input volume. For example, RGB images have a depth of three since they are composed by three channels: red, green and blue. Taking them as input would mean that the filter would also have a depth of three. To compute a feature, the filter is convolved across the width and height of the input volume and the dot products between the entries of the filter and the input at any position are computed [31].

The output will be a 2-dimensional feature map which will be passed through an activation function layer, resulting in an activation map. In this way, the network will learn filters that activate when a visual feature is detected.

The size of the output after the convolution operation will depend on the number of filters, the stride, and the padding [31]. The stride parameter indicates the amount of shifting at a time. If the stride is set to one, the filter will be moved one pixel at a time. The bigger the stride, the smaller will be the output size. The zero-padding parameter defines how many zero pixels should be added around the border of the input, letting us control the spatial size of the output volume.

Suppose we have a convolutional layer l with a set of feature maps $Q_l = (\phi_1, \phi_2, \ldots, \phi_k)$ with k denoting the number of filters. Each feature map has a size of $m^l \times n^l$ since all feature maps in the same layer have the same size. We also define the size of the set of filters F_l in layer l to be $u^l \times v^l$, the stride as $s^l > 0$ and the padding as $p^l \geq 0$. To calculate the output size of a convolutional layer, the following equation is used:

$$(m^l, n^l) = \left(\frac{m^{l-1} - u^l + 2 \times p^l}{s^l} + 1, \frac{n^{l-1} - v^l + 2 \times p^l}{s^l} + 1 \right)$$

the padding and stride can also differ among dimensions, the equation above assumes that the padding and stride are equal for both dimensions.

To calculate the feature map values in layer $_l$, we need to compute the output of the neurons in each feature map given Q_{l-1} and F_l. We will denote the output of a neuron (i, j, ϕ_k, l), at position row i and column j in feature map ϕ_k in layer l as $y_{i,j}^{\phi_k,l}$, where $0 \leq i < m^l$ and $0 \leq j < n^l$. To be more specific, the neuron (i, j, ϕ_k, l) will be connected to the outputs of neurons located in rows $i \times s^l$ to $i \times s^l + u^l - 1$ and columns $j \times s^l$ to $j \times s^l + v^l - 1$ from feature map ϕ_k in Q_{l-1}. The computation of the output of a neuron in a convolutional layer is defined by [22]:

$$y_{i,j}^{\phi_k,l} = b^{l,\phi_k} + \sum_{\phi' \in Q_{l-1}} \sum_{p=0}^{u^l-1} \sum_{q=0}^{v^l-1} y_{i \times s_r^l + p, j \times s^l + q}^{\phi',l-1} \times w_{p,q}^{\phi_k,\phi',l}$$

where b^{l,ϕ_k} is the feature map's ϕ_k bias and $w_{p,q}^{\phi_k,\phi',l}$ the connection weight (filter value) between a neuron in feature map ϕ_k in layer l and a field in feature map ϕ'. Figure 2.4 provides a visualization of the convolutional layer.

The deconvolutional layer is the inverse of the convolutional layer. Instead of reducing the input's size, the deconvolutional layer increases it. For that, it transposes

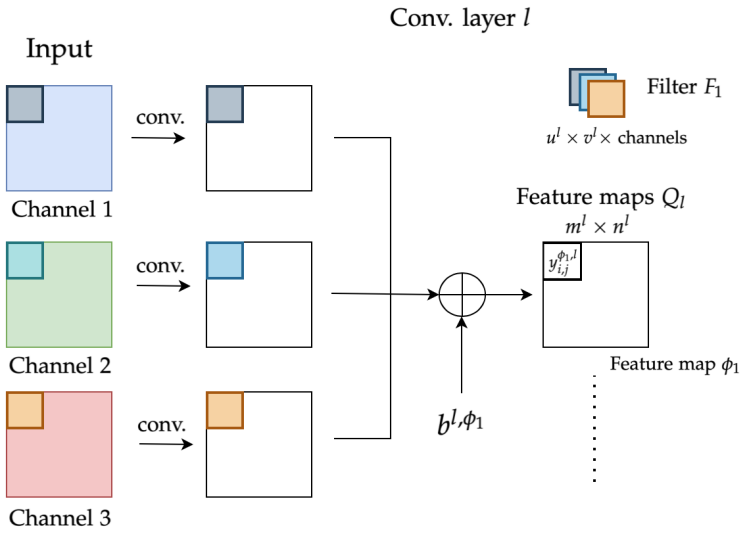

Figure 2.4: *Convolutional layer.*

the convolution operation. It has a similar stucture as the convolutional layer, but the calculation of the outputs is slightly different.

2.3.2 Convolutional Autoencoder

The goal of this work is to compress climate data as much as possible using ANNs. We look for a network that learns a lower dimensional representation of a training set and reconstructs it, like a compression algorithm. There is a ML architecture that has those properties, the autoencoder (AE).

AEs are ANNs capable of learning dense representations of input data, called latent representations. These latent representations can have a lower dimensionality as the input data and therefore act as a compression algorithm. An AE consists of an encoder function $y = f_\theta(x)$ and a decoder function $\widehat{x} = g_\phi(y)$, where x is the original data, $\widehat{x}$ the reconstructed data and y the latent representation. θ and ϕ are optimized parameters in the encoder and the decoder function respectively [32]. In order to learn the dense representations of the input data, the AE needs to be parametrized in such way that it minimizes a reconstruction loss e.g., MSE and can be defined as [15]:

$$\theta, \phi = \underset{\theta,\phi}{\operatorname{argmin}} \left\| x - g_\phi(f_\theta(x)) \right\|_2^2$$

where the squared l_2-norm is used as loss function. Note that there are other loss functions that can be used in the equation.

AEs will not only be able to compress the training set, but also data similar to what they have been trained on, which is a useful property since it means that the algorithm will perform well on a specific type of input [33].

To train an AE, we define the targets of the AE to be the same as the input. The AE will try to map the inputs to the outputs while generating a latent representation. Even though the task may sound easy, the constraints that affect the network can make this task challenging. By limiting the size of the latent representation, it will force the network to learn efficient ways of representing the data [22].

The data we are working with has a grid-like topology. This means we will use convolutional layers in our autoencoder to obtain latent representations. An AE using convolutional layers is named convolutional autoencoder (CAE). A CAE with three convolutional layers with stride 2 is presented in Figure 2.5. By setting the stride to two, we automatically halve all dimensions three times since there are three convolutional layers. The latent representation is defined as h and it can also be represented as a 1-dimensional vector. Afterwards, the input is is reconstructed from the latent representation by using deconvolutional layers with also stride 2.

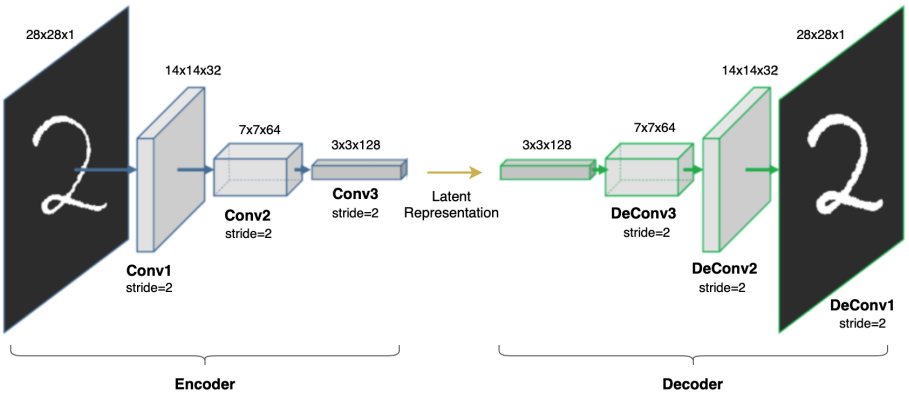

Figure 2.5: *Example of a convolutional autoencoder. Adapted from [34].*

By constraining the network by setting latent representation dimensions and the number of convolutional layers with $s_l \geq 2$ or pooling layers, we can approximate the CF of the model. Let d be number of dimensions in our data, k the compression factor in every dimension and t the number of times the dimensions are being reduced.

An upper bound for the CF can be calculated as following:

$$\mathrm{CF}_{\mathrm{model}} \approx \frac{(k^d)^t}{|Q_e|}$$

where $|Q_e|$ is the number of feature maps of the last layer in the encoder function.

After having seen an introduction about compression and ANNs, we will proceed to introduce the properties of the climate dataset.

2.4 Climate Data

Even though there are many different ways to collect climate data, they can be divided into two groups: observational data and simulation data [14]. Observational data is data that is collected using measurement instruments. These can be constrained regarding its spatial and temporal resolution due to hardware and/or software specifications. Simulation data is data obtained through climate and weather models, which are not constrained like observed data. Since the data we will be working with is based on simulations, the next section will provide a brief introduction to the topic.

2.4.1 Climate and Weather Model

Climate and weather models simulate the transfer of energy and materials through the climate system [35]. These models are systems of differential equations that describe how energy and matter behave in different parts of the ocean, atmosphere, and on land. To compute these mathematical equations, climate and weather models mostly break the Earth's surface into three-dimensional grid boxes through time. The results of the mathematical equations in each box are passed to the neighboring boxes to model the exchange of energy and matter over time. There are two types of variables that models simulate: prognostic and diagnostic variables [14]. The prognostic variables are directly calculated by the differential equations. Diagnostic variables are derived from prognostic variables at specific time steps, e.g., humidity from temperature. However, to achieve more accurate data, climate reanalyses are done. Reanalyses combine past observations with models to generate consistent time series for a large set of climate variables. Reanalyses are among the most-used datasets in the geophysical sciences [36].

2.4.2 ERA5 Dataset

In this thesis, we will be working with the ERA5 [36] dataset from the European Centre for Medium-Range Weather Forecasts (ECMWF) [37]. ERA5 is the fifth generation ECMWF reanalysis for the global climate and weather for the past 4 to 7 decades. The data is available from 1979 within 3 months of real-time, i.e. if today is March 2021, the available data would be up to December 2020, which assimilates as many observations as possible in the upper air and near earth surface. The ERA5 atmospheric model is coupled with a land surface model and a wave model [38]. We will train and test our model with the ERA5 hourly data on pressure levels from 1979 and 1980 respectively. We chose these years as test since we are developing a compression algorithm prototype for climate data. Table 2.1 provides a description of the dataset. Note that 1000hPa is closest the earth's surface and 1hPa the furthest.

ERA5 hourly data on pressure levels from 1979 to present	
Data type	Gridded
Projection	Regular latitude-longitude grid
Horizontal coverage	Global
Horizontal resolution	Reanalysis: $0.25^\circ \times 0.25^\circ$
Vertical coverage	1000 hPa to 1 hPa
Vertical resolution	37 pressure levels
Temporal coverage	1979 to present
Temporal resolution	Hourly
File format	GRIB; NetCDF

Table 2.1: *ERA5 dataset characteristics [39].*

The data structure of the ERA5 dataset is the hypercube, one of the most common in climate data [14]. The hypercube presents four dimensions called coordinates which are: longitude, latitude, altitude, and time as shown in Figure 2.6. Longitude is represented by degrees east, latitude by degrees north and altitude by hPa.
The number of data variables in the dataset is 16 and are single-precision floating-point values.

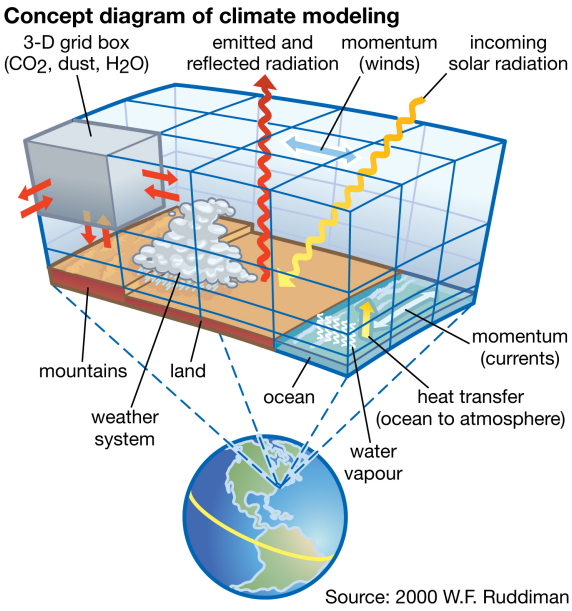

Figure 2.6: *Hypercube data structure [40].*

3 Related work

Attempting to compress vast volumes of data, such as climate data, is not new. This chapter provides an overview of work related to the compression of climate data using machine learning and non-ML based approaches. The first section presents work directly related to the topic of this thesis, climate data compression using ML. Then, we present work related to the compression of other data types, such as images or text. The third section provides information about non-ML based compression algorithms used in the compression of climate data. Finally, the last section presents state-of-the-art non-ML based compression algorithms for floating-point data.

3.1 Compression of Climate Data with Machine Learning

The excellent results obtained with machine learning in other fields [23] [41], have spurred research into using such models for climate data compression. Liu et al. [42] present a fully connected AE with three layers for encoding and decoding climate data. Every layer reduces the data by a factor of eight, having a theoretical CF boundary of 512. To bound the relative error, the authors calculate the difference between the input data and the reconstructed one and store the differences that are greater than the error bound. After storing the differences, they lossy compress them with SZ with an error bound of 0.1. They also store the indices of the numbers where the difference is outside the error bound. The indices are stored as a bitmap where a set bit indicates a difference outside the error bound and an unset bit otherwise. Such bitmap is then losslessly compressed with bzip2. The authors compare their results with SZ and ZFP obtaining two to four times the CF of SZ and 10 to 50 times of ZFP. The code is available to the public[1]; however, the model only works with 1-dimensional data. Moreover, experiments are based on small-scale scientific data (4.8MB), questioning its performance on Big Data.

The authors of SZ compression library, Liu et al. [43], present a modified version of SZ called AE-SZ, an ML-based lossy compression algorithm. In AE-SZ, the authors

[1] `https://github.com/tobivcu/autoencoder`

replace the linear regression predictor of SZ with a Sliced Wasserstein AE (SWAE) [44]. First, the data is split into 2-dimensional or 3-dimensional blocks, and then fed to the SWAE and the mean-Lorenzo predictor [45]. Each block is then compressed by both compression algorithms. The one with the lowest L1-loss is used to compress the data. The reason why they use two different compression algorithms and not only the AE-based one is that the mean-Lorenzo predictor works better when compressing data for really small error bounds(1E-4) [43]. After choosing the best compression algorithm, the authors apply a linear-scale quantization to the difference between the input and the reconstruction. Finally, the quantized values are encoded using Huffman coding [46] and Zstd [47]. For CFs above 100, the AE-SZ has about 100% to 800% CF improvement with the same PSNR achieved by SZ and ZFP. This is because the AE-SZ works better with higher error-bounds, thus, reaching higher CFs. The disadvantage is that the AE-SZ is about 10%-40% slower than SZ. The authors conclude that the block and the latent space size in the AE play an essential role for the CF. Nevertheless, the latent space and the block size depend on the dataset, meaning that there is no general network architecture that provides the best CF for all climate datasets. The authors mention that their approach outperforms any other AE-based error-bounded lossy compression algorithm. However, they only compare their method with the previously mentioned work [42]. Moreover, they do not give a proper reason about the usage of the SWAE over a simple AE. The authors only compare the PSNR gotten with different AEs when compressing a single field of a specific dataset. The authors also use a layer called generalized divisive normalization, which has been used for image compression; however, there is no research about its effectiveness on climate data.

Pan et al. [48] compress climate data using an AE; however, they do not guarantee an error-bound. The proposed AE does not have convolutional layers but fully connected layers. The authors fed the AE with chunks of data. Each chunk is flattened into a 1-dimensional vector and normalized. Afterwards, the authors quantize the predictions and use an adaptive compression algorithm. They introduce an adaptive compression mechanism which penalizes the use of more significant bits than needed by defining a custom loss function. One problem the authors faced was visible artifacts along the boundaries of each chunk. To solve that, they modify the model to compress each value within the chunk with the distance to the chunk boundaries, allowing a more precise compression of boundary values. The authors measure the effectiveness of their approach by comparing the RMSE and PSNR achieved by other models by setting a CF of 100. The authors do not compare the results with SZ. The authors achieve 9 to 30 times less RMSE compared to other models.

Not all climate data compression algorithms are lossy. There are other works that losslessly compress climate data using ML. One of them is presented in Mummadisetty et al. [49]. The authors build a lossless compressor using a fully connected ANN with four layers with ten nodes each to compress solar radiation data. First, the ANN predicts the solar radiation values with other information from variables that are related to solar radiation. After predicting the values, the algorithm checks for the leading zeros and stores the count and the same process is done for the trailing zeros. Next, it checks for the non-zero data points and performs differential encoding, keeping the first non-zero digit as it is required during decoding operation. Finally, Huffman coding is applied to the count of leading zeros, trailing zeros and the differentially encoded data. As a result, the authors get a maximum CF of 5.81.

Saenz et al. [50] show that convolutional AEs are suitable to encode climate data. The authors compare the reconstruction error with the one gotten with principal component analysis (PCA). They analyze several convolutional AEs with different settings to encode two temperature fields from pre-industrial climate model simulation datasets. The results show that the reconstructed temperature fields preserve the large-scale features of the global temperature patterns, but small-scale features are filtered out. They also show that AE outperforms PCA.

3.2 Compression of Other Data with Machine Learning

While there are only a handful of ML-based compression algorithms for climate data, other types of data that share structural similarities have been also tried to be compressed using ML. Glaws et al. [15] compress turbulence flow simulation data. The authors use an AE with 12 residual blocks and three compression layers with a maximum CF of 64. The results show that AE outperformed singular value decomposition (SVD) and was able to restart the simulations and keep them in the correct trajectory. In Choi et al. [51], physics plasma simulation data is being compressed. The authors use a variational AE with physics-informed optimization functions and refinement layers. Further, the authors quantize the latent space and compress it with Gzip [52]. They achieve a CF 1.5 times higher than ZFP.

Not only floating-point datasets have been compressed with ML but also images. For brevity, we will only introduce a few relevant works. Ballè et al. [53] introduce a transformation called generalized divisive normalization (GDN) to AE. DN represents a multivariate generalization of a particular type of sigmoidal function. The authors indicate that GDN and ReLU achieve qualitatively similar results but that ReLUs generally require a substantially larger number of model stages to achieve the same

performance as the GDN. The authors claim that GDN provides better results for natural image compression. This transformation is used in the AE's architecture instead of activation functions, e.g., ReLU. Theis et al. [54] modify the AE by adding a sub-pixel convolutional layer [55] which is a convolution followed by reshaping and reshuffling of the feature map values. The sub-pixel layer allows the images to be better reconstructed and achieve a higher resolution when upsampling. The authors also add residual layers to learn features of the image. Both works quantize the output of the encoder by modifying the loss function, Ballè et al. [53] add additive uniform noise to simulate the quantizer noise, and Theis et al. [54] use a rounding function for the quantization. Li et al. [56] use an importance map for the coding of the quantized values. The authors present an AE with an aditional residual network, called an importance map, that calculates the importance each pixel in the feature maps. After quantizing the feature maps, the importance map identifies the areas with more details and structural information in order to assign them more bits to encode. The idea behind this, is to use more space for essential features from the image and save space when storing less important features. Thus, generating better visually decoded images. To achieve this, the authors integrate the quantization and importance map to the loss function. Mentzer et al. [57] also use an importance map for the quantization; however, they quantize the values differently. They use a relaxation of the rounding function for quantizing the values called soft quantization. This function is derivable, making it suitable for calculating the AE gradients. All of the above-mentioned authors compare their approach with JPEG [58] and JPEG2000 [59]. They conclude that the use of AEs looks promising for lossy image compression.

AEs are not the only ML-based method to compress data. Generative adversarial networks (GAN) [60] and recurrent neural networks (RNN) [61] are also used for data compression. Agustsson et al. [62] present a lossy image compression algorithm based on GANs. The difference between a GAN and other compression algorithms, is that the decoder in a GAN is a generator. Instead of following a deterministic process like the AE, the GAN tries to generate new data consistent with the input data distribution and its latent space. This approach works really well on images since the importance lies in the images' visual quality and not the actual pixel values. Lossless compression can also be achieved with other ML methods. Goyal et al. [63] and Bellard et al. [64] use RNNs to compress sequences like text and genomic datasets losslessly. Both works use a variant of RNN called LSTM, which is capable of learning long short-term dependencies. In this way, the authors compress text and genomic datasets by predicting the sequence to be compressed. Goyal et al. [63] further combine the LSTM with an arithmetic coder to achieve a higher CF.

So far, we have seen different ML methods to achieve lossless and lossy compres-

sion of different types of data. The following section will cover methods that use non ML-based models to compress climate data.

3.3 Compression of Climate Data with non ML-based Compression methods

Many authors have expressed their interest in the effectiveness of non ML-based compression algorithms on climate data. Baker et al. [65] and Hübbe et al. [2] examine the effects of lossy compression on climate data. In Baker et al. [65], the authors employ different quality metrics, e.g., mean and standard deviation of the reconstructed data, to compare the effects of lossy compression. The authors conclude that the compression of climate data must be done variable-by-variable since the data is very diverse. Hübbe et al. [2] also compare different lossy compression methods and evaluate them according to compression ratio, signal-to-residual ratio, and processing time. Hübbe et al. [2] conclude that climate data could probably be lossy compressed; however, intermediate results like checkpoints should be compressed losslessly since numerical stability and control play an important role in the restart of simulations.

Besides comparing different lossy compression algorithms on climate data, Hammerling et al. [66] analyze the behavior of a lossy compression algorithm i.e., ZFP when compressing climate data. To tackle artifacts that appear by compressing climate data and are not typically picked up by standard compression metrics, the authors modify the compression algorithm. As a result, the authors provide a new compression algorithm that improves the compression ratio in climate data analysis.

In the case of lossless compression, Huang et al. [67] propose a lossless compression algorithm called Czip. Czip eliminates data redundancy through several new methods, including adaptive prediction, eXclusive OR (XOR) differencing, multiway compression, and static regions. The authors show that Czip can achieve outstanding compression ratios as well as higher throughput since it runs in parallel.

Methods to improve lossless prediction-based compression algorithms used in climate data were presented by Cayoglu et al. [68] [69]. After analyzing the performance of different prediction-based compression algorithms on climate data, the authors introduce the concept of Information Spaces (IS) [68]. With the introduction of IS, better predictions and more consistent compression ratios were achieved. Furthermore, Cayoglu et al. [69] propose an encoding scheme that outperforms the current

state-of-the-art scheme used in lossless prediction-based compression algorithms. The authors use the shifted XOR calculation to move the data into another range to improve compression since compression performance is dependent on the value range covered by the data.

A modular software framework for compression of structured climate data was developed by Cayoglu et al. [70]. The authors present a framework that provides all necessary components to design, test, and grade various prediction-based compression algorithm. Moreover, the framework provides additional features such as the execution of benchmarks and validity tests for sequential as well as parallel execution of compression algorithms.

3.4 Compression Data with non ML-based Compression methods

There are many non-ML based compression algorithms for data compression. For reasons of brevity, this subsection will only mention the most recent work. For floating-point data, two lossy compression algorithms stand out.

SZ is a prediction-based compression algorithm First presented in Di and Capello [71] and improved over the years. The latest version of SZ is presented in Liang et al. [72]. SZ works with three predictors: a linear-regression-based predictor, the Lorenzo predictor [73] and the mean-Lorenzo predictor. SZ chooses the best predictor based on the reconstruction loss to compress the data. After predicting the data, the errors are calculated, quantized, and coded using Huffman coding [46].

The second lossy compression algorithm is zfp, introduced in Lindstrom [74]. zfp is a lossy transformation-based compression algorithm that decorrelates the data by an applying orthogonal block transformation. The data is divided the data into blocks, which are then aligned by a common exponent. Next, they convert the floating-point values to a fixed-point representation. Afterwards, the values are decorrelated using an orthogonal block transform. Finally, the transform coefficients are ordered by the expected magnitude, sorted by bit plane, and coded.

The performance of the previously mentioned compression algorithms vary according to the data. For that reason, Tao et al. [21] developed a hybrid compression algorithm by choosing between SZ and zfp regarding their performance.

In the case of lossless compression, a lossless compression algorithm called fpzip was introduced in Lindstrom and Isenburg [5]. fpzip traverses the data in some coherent order, e.g., row-by-row, and predicts each value from a subset of already coded data. The predicted and actual values are then transformed to an integer representation. Next, the residuals are computed, partitioned into entropy codes and raw bits, which are then transmitted by an entropy coder.

4 Methodology

This chapter describes the architecture of the ML-based compression algorithm. First, we look at the overall architecture and discuss the convolutional AE as the selected ML model. Second, we describe the core steps of the architecture in more detail. We start by explaining the pre-processing of the data to be compressed. Then, we explain and provide details of the convolutional AE's architecture. Finally, we explain the calculation of the residuals and discuss the use of the lossless coding methods used for the residuals and the convolutional AE's latent representation compression.

4.1 Architecture

We aim to develop a prototype of a prediction-based lossy compression algorithm for climate data using ML. A prediction-based lossy compression algorithm compresses data by predicting the values given a latent representation and later comparing its predictions with the actual data [14]. Afterwards, the comparison is lossy encoded and given as output of the encoder together with the latent representation used for predicting the values. In this way, the data can be later reconstructed. Thus, for the prediction of values, we will be using an ML model.

We will start by giving an insight into the climate dataset since it plays an essential role in the architecture's design. As mentioned in Section 2.4, this work focuses on the compression of the ERA5 dataset, specifically the temperature variable. This variable is a four-dimensional array (time, level, latitude, longitude) of floating-point numbers. The first challenge we tackle is the processing of data for compression. We mentioned in Section 2.1.1 that there are two ways to process the data, value by value (streaming mode) or block-wise (block mode). Climate data contains spatio-temporal information that should be exploited. Processing the data value by value would require looking at neighbouring data in order to get as much information as possible about the value to be processed. In fact, this is how the compression framework SZ [20] works. Otherwise, we could take 4-D blocks of data and process them as a whole, as zfp [19] does.

We first find and use an ML model that can exploit the spatio-temporal information in climate data to get good predictions. The better the predictions the less residuals

are going to be stored, thus, achieving greater compression. Previous work (see Section 3.2) have compressed N-dimensional data with two different ML models: GANs and convolutional AEs.

A GAN is a generative model which reconstructs the compressed data by generating new data according to a latent representation. GANs achieve significant compression factors [32]. However, they only work well for subjective quality reconstructions like images. The reconstructed data by the GAN present great visual quality. Nevertheless, the reconstruction error is high since it only generates similar data to the actual data. For this reason, we decided against GANs.

Convolutional AEs reconstruct the data by learning a compact representation of the data, thus, getting a more accurate output than GANs. AEs have also been proved to be better than traditional methods for lossy compression [32]. Therefore, we decided to choose a convolutional AE as the ML model for our compression algorithm. The input of an AE is N-dimensional which means that the input of the model needs to be N-dimensional as well. If we choose to process the data in streaming mode i.e., receiving multi-dimensional data sequentially, the compression and decompression time would be high since the data would be processed one by one. If we process the data in block mode, the blocks could be processed in parallel which fits in our case since convolutional AE operations can be done using GPUs which are very good at data-parallel computing. Therefore, we decided our compression algorithm to work in block mode. Because of the symmetry of AEs, our compression algorithm will also be symmetric.

Scientific data, like climate data, go through extensive scientific analysis. In consequence, its compression and decompression must be done without negatively affecting scientific conclusions. That being the case, our compression algorithm bounds the error of the decompressed data. The error in the reconstructed data will not pass a previously specified error. For bounding the error, the absolute error is used, and its unit is the same as the data to be compressed.

The proposed compression algorithm is a non-universal, block mode, symmetric lossy compression algorithm. It takes as input an N-dimensional array to be compressed and the maximum allowed absolute error. The output of the compression algorithm is the reconstructed data. Figure 4.1 illustrates the architecture of our compression algorithm which consists of two parts: an encoder that compresses data and a decoder that reconstructs the data.

The encoder of the compression algorithm starts by normalizing and chunking

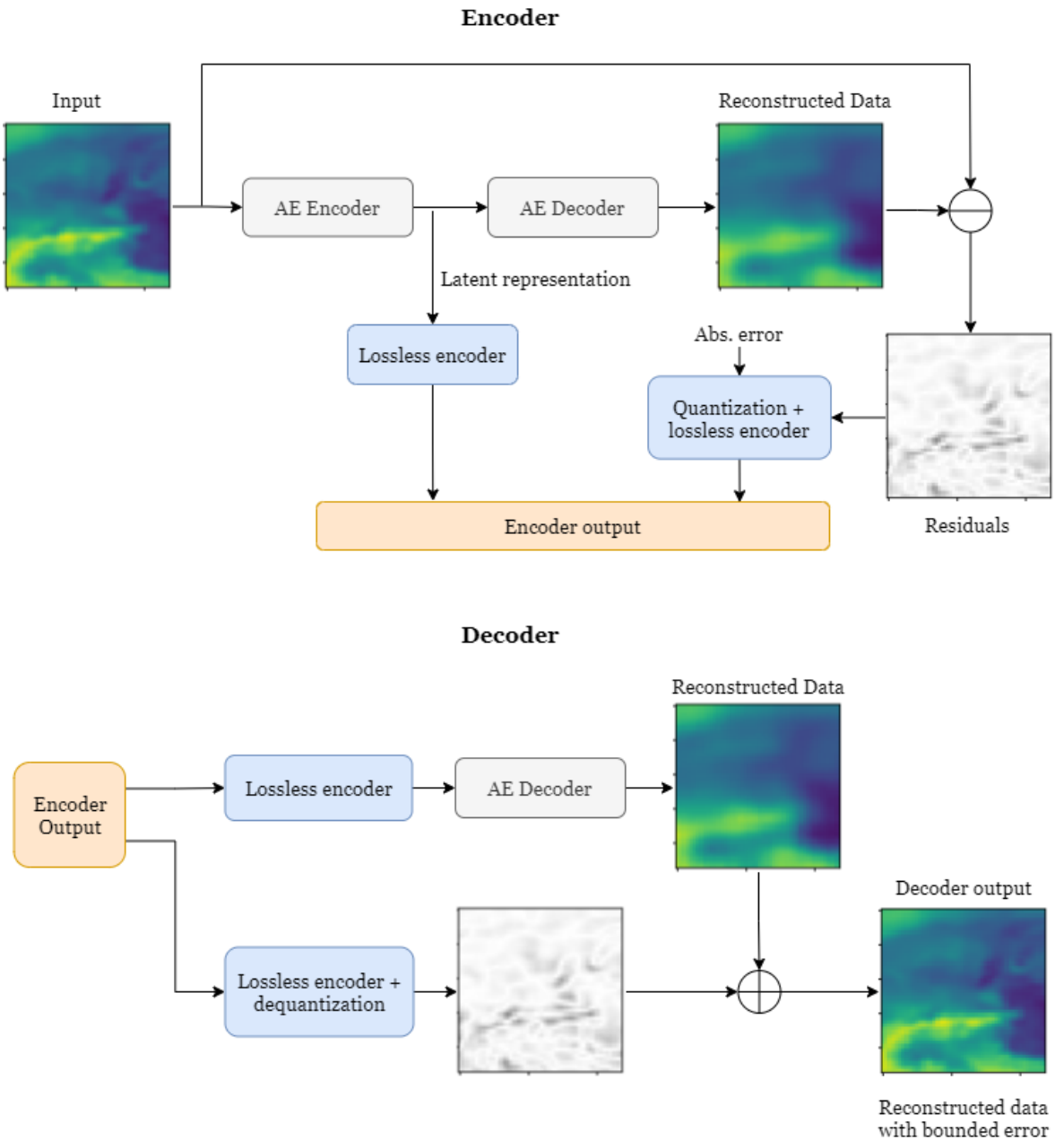

Figure 4.1: *Compression algorithm architecture.*

the data into blocks. This is the pre-processing of the data. Afterwards, the blocks are fed into the convolutional AE, which outputs a more compact representation of the data (i.e., a latent representation), and the predicted data (i.e., reconstructed data). Even though the AE compresses the data, we can still further compress it using an encoding algorithm. The reconstructed data provided by the convolutional AE may present a low average prediction error. However, it is not enough for the lossy compression of climate data. Given an average prediction error of 0.003 in the convolutional AE's reconstructed data, and a set absolute error of 0.5. Not all the reconstructed values will respect the set absolute error, the majority of the them, e.g. 90%, will lie under the set error bound but the remaining 10% will not respect it, which is unacceptable. Therefore, we work with residuals.

Residuals are the values obtained by differentiating the input data from the reconstructed data. In order to do the differentiation, the values must have the same scale. Since the convolutional AE outputs a normalized version of the data, we first pre-process the reconstructed data, i.e., we merge the data blocks and denormalize them. After calculating the residuals, we compare the values with the set absolute error. The values that are outside the allowed absolute error are stored in an array. Since we only store the values outside the allowed absolute error independently of their position in the data, we need to store another array that lets us know where such values are located. After storing the positions and residuals, we quantize the residuals and use an approach called differential coding in together with bzip2 [16], to compress the quantized residuals and position array further.

The encoder includes extra information that helps the decoder reconstruct the data. The decoder only receives as input the compressed data, meaning that that user does not need to specify the absolute error as in the encoder. For this reason, the absolute error is included in the compressed data as it is needed to dequantize the residuals. We also store the mean and the standard deviation for the post-processing of the values. The input data and the residuals are also stored because they are needed for reconstructing the arrays. The output elements of the encoder are illustrated in Figure 4.2. The colour denotes the type of each element, green represents a floating-point value, yellow a tuple, and violet bytes.

As previously mentioned, our compression algorithm is symmetric. The same steps will be done in the decoder but in reverse order. The decoder of our compression algorithm gets as input the output of the encoder. The decoder reads the input elements and starts decompressing the latent representation using fpzip [18]. With the latent representation the data is reconstructed using the convolutional AE and processed to go back to the original scale (see Section 4.1.1). The residuals and

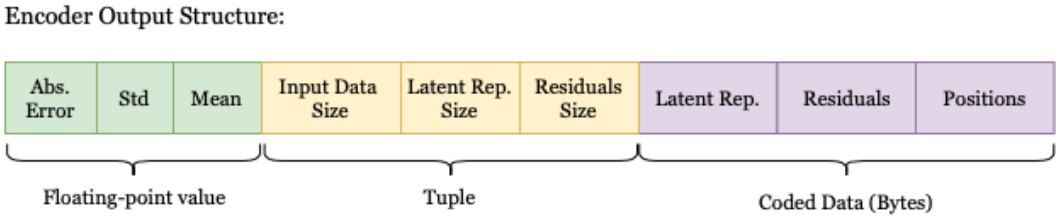

Figure 4.2: *Encoder output.*

the position arrays are decoded as well using differential decoding together with bzip2 [16]. Then, the residuals are dequantized and assigned to their respective position. Finally, we sum up the residuals and the reconstructed data and return the result as the output of the decoder.

The following section will provide a detailed description of the procedures.

4.1.1 Data Pre-processing and Post-processing

The pre-processing of the data has two steps: normalization and division of data into chunks. The first step consists of normalizing the data using the standard score normalization by subtracting the mean from the data and dividing the difference by the standard deviation of the data. This helps us to keep the input data consistent. By normalizing the data, each feature contributes approximately proportionate to the training of the convolutional AE, thus resulting in a faster convergence of the network [75]. The standard score normalization is defined as [76]:

$$z = \frac{x - \overline{x}}{S}$$

where x are the input data values, $\overline{x}$ the mean of the data, and S the standard deviation of the data.

As mentioned in Section 2.3.2, a convolutional AE takes as input N-dimensional data. Due to constraints of the programming API, the prototype uses a 3-dimensional convolutional AE. Since climate data is 4-dimensional, we had to choose which dimension to set aside. Previous work [77] has shown that a higher CF is achieved when using the time, longitude, and latitude dimensions. Therefore, we decided to desist from using pressure levels. Nevertheless, there are 37 pressure levels in the dataset. We decided to choose the highest pressure level (1000hPa), i.e., values nearest to the surface because they have greater complexity. This is the reason why the second step is to divide the data into 3-dimensional fix-sized chunks. The size of the chunk will

depend on the convolutional AE's architecture and available computational power. The convolutional AE reduces the dimensions of the input data depending on the number of convolutional layers and stride in the encoder. Therefore, the size of the block must be bigger than the amount of reduction the encoder achieves since a small chunk would reach a state of not being reducible anymore when reaching a minimum size of $[1, 1, 1]$ in a 3-dimensional data block. The blocks can not be too big either due to increased training time since a bigger input means more mathematical operations. Moreover, a bigger block size would mean that the input data has to be big enough to cover the size of at least one block. Figure 4.3 illustrates the chunking along the longitude and latitude dimensions for temperature.

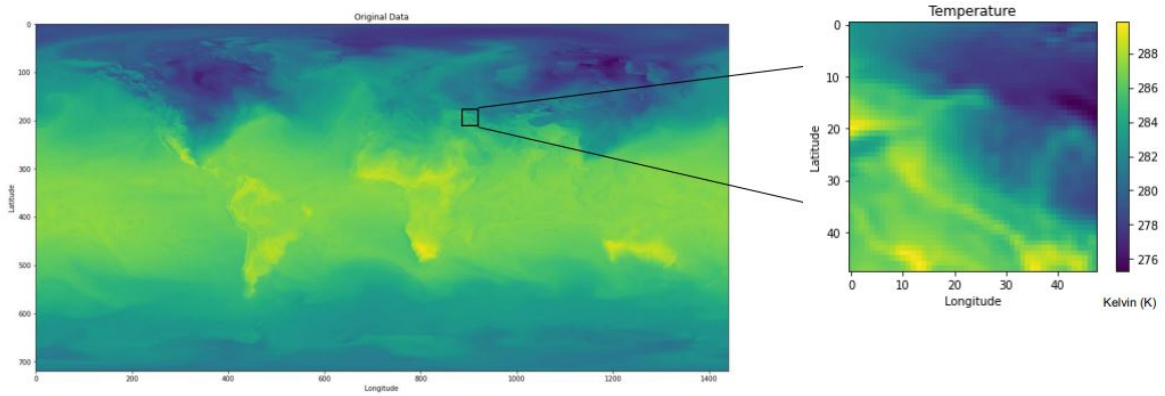

Figure 4.3: *Visualization of a 2-dimensional chunk.*

The data is split into same sized chunks and bounded to form an array of chunks. If the number of blocks resulting from the chunking is not a multiple of the chunk's size, an extra chunk of data that includes the data not covered by the other chunks is taken. Figure 4.4 shows the process when the data is not a multiple of the chunk's size.

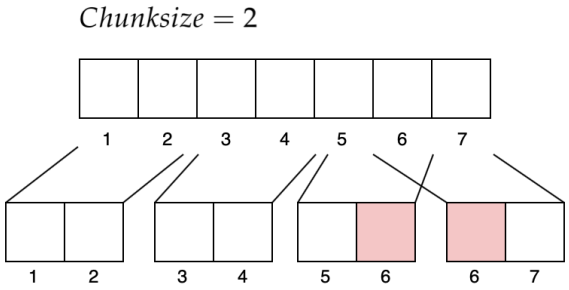

Figure 4.4: *Division of data into chunks.*

The post-processing of the data consists in reversing the steps made in the encoder's pre-processing part. After decompressing, we merge and denormalize the chunks to get the original shape and values of the data. The merging function requires the chunked data and the original size of the data to be reconstructed. To denormalize the data, the following function is used [76]:

$$z = x \times S + \overline{x}$$

the mean value $\overline{x}$ and standard deviation S are part of the encoder's output. Next section will provide a description of the convolutional AE's architecture.

4.1.2 Convolutional Autoencoder

The performance of the convolutional AE plays an essential role in the performance of the compression algorithm. The reason lies in the residuals. The better the convolutional AE reconstructs the data, the smaller the errors, thus, reducing the size of the residual array. If the size of the residual array is small, a high compression factor will be reached since there will be fewer data to store. We draw inspiration from the convolutional AE presented in [15] and propose a convolutional AE that consists of four 3-dimensional convolutional layers and 3-dimensional deconvolutional layers with one residual block each. The architecture of our convolutional AE is shown in Figure 4.5. The colour of the blocks denote the stride value of the convolution; blue represents a stride of one and yellow a stride of two. The convolutions (deconvolutions) with a stride of two halves (double) the input's size in all dimensions, thus reducing(increasing) the size of the data. This means that the data compression happens after the residual block and the decompression before the residual block. A residual block is a skip-connection block that copies the current state of the data and adds it back to the network further down the line. The use of skip connections reduces the amount of information that intermediate processing layers must encode, enabling the effective training of deeper networks with increased capacity to learn complex features [15]. There are different configurations of residual blocks; our implementation uses the first version of residual blocks which consists in two convolutional layers and an activation function [23]. We use 20 filters in each convolutional and deconvolutional layer. Only the last convolutional layer in the decoder has one filter since it outputs the reconstructed data. The padding varies in each convolutional and deconvolutional layer. The resulting feature map's size has the same size as the input. The convolutional layers inside the residual block have a size of three. The rest of the convolutional and deconvolutional layers have a kernel size of five.

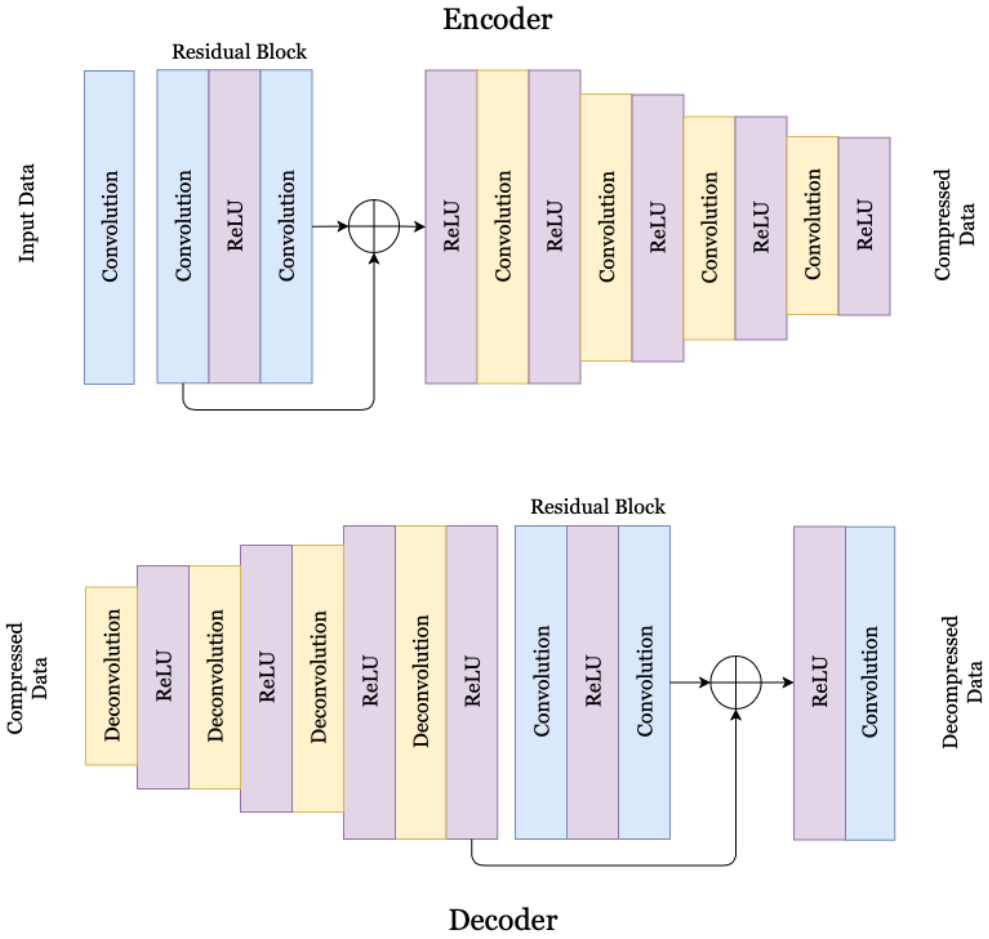

Figure 4.5: *Architecture of the AE.*

We use ReLU as activation function since it is more computationally efficient than other activation functions and reduces the likelihood of a vanishing gradient [27]. Notice that the last convolutional layer in the decoder is not followed by an activation function since it provides the reconstructed values. If we add a ReLU after this convolutional layer, values that are less than zero would stay at zero since ReLUs only return zero or positive values (see Section 2.2.5). We choose the convolutional AE parameters through hyperparameter optimization and as loss function the MSE (as defined in Section 2.2.4).

With the defined AE architecture, we can now calculate the maximum CF the convolutional AE can get if the user does not set any error bound. Note that the CF achieved by the AE is not the maximum CF that the compression algorithm can achieve since the latent representation gets further compressed in another step by fpzip. By halving three dimensions four times and expanding the latent representation across 20 filters, the maximum CF the AE achieves is (as defined in Section 2.3.2):

$$\mathrm{CF}_{\mathrm{model}} \approx \frac{(2^3)^4}{|20|} \approx 204,8$$

Now that we have the latent representation and reconstructed data from the AE, the next step is to compute the residuals in order to be able to set an error bound. The following section will provide the details of the process.

4.1.3 Residuals, Quantization and Lossless Coding

Once we got the reconstructed data from the AE, we calculate the residuals by taking the difference between the reconstructed data and the data to be compressed. The residuals allows us to control the error bound in the reconstructed data.

The error bound set has to be respected when reconstructing the data; however, the convolutional AE's output will have errors that are greater than the defined error bound. The simplest way to ensure that the error bound is not surpassed is by storing the residuals plus the latent representation. In such a way, the decoder could perfectly reconstruct the data by adding the residuals to the reconstructed data gotten with the latent representation. Nevertheless, the compression algorithm would be considered a lossless compression algorithm since no information is lost when reconstructing the data because we are considering the residuals in their entirety. Moreover, storing so much information would considerately affect the CF negatively. Luckily, setting an error bound lets us reduce the amount of information to be stored. Instead of storing all the residuals, we only store the residuals that are outside the error bound.

However, storing the residuals in such a way makes us lose information about their position in the 3-dimensional data array. As a consequence, we use another array that provides us that information.

We lossily code the residual arrays. The residual values are floating-point values which are generally more difficult to compress than integers [78]. Unlike fixed-length integers, which often have a fair number of leading zeros, floating-point values often use all of their available bits, making it harder to compress. For this reason, we decided to lossily compress the residual array by using quantization. Quantization maps a continuous set of values to a discrete set of values by breaking the input's interval into bins. All values that lie in a given bin are rounded to the reconstruction value associated with that bin [79]. Let $x \in \mathbb{R}$ be the input to the quantizer. The quantization function is defined as [79]:

$$Q(x) = round(\frac{x}{\Delta})$$

where $round(\cdot)$ is a function that rounds a value to its nearest integer value and Δ is the quantizer step size. By rounding, the errors gotten through quantization are zero-mean and bounded [79]:

$$-\frac{\Delta}{2} \leq \epsilon \leq \frac{\Delta}{2}$$

where ϵ represents the quantization error; having knowledge of the quantizer error bounds, we can keep controlling the error bound to set in the compression algorithm. For that, we define the quantizer step size to be two times the error bound to set in the compression algorithm. By means of that, the residual array values are now integers and the error bound is respected. Nevertheless, before quantizing, we take the absolute value of the residuals, so all values are positive. By making all the values positive, we reduce the values space, making it easier to code. Figure 4.6 provides a visual representation of the quantization operation.

To dequantize the values we multiply the quantized value with the quantizer step size:

$$D(Q(x)) = Q(x) \times \Delta$$

The position array does not need to be quantized. To store the positions, three integer values are defined: '0' for residuals within the error bound, '1' for positive residuals outside the error bound, and '2' for negative residuals that lie outside the

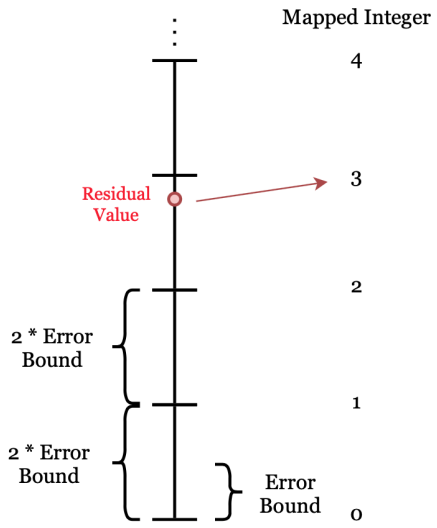

Figure 4.6: *Quantization.*

error bound.

Now we focus on the compression of the integer arrays. It is best if most integers are small since they can be represented more compactly by using shortcodes. A way to reduce the size of the integers is through differential coding. Differential coding, also known as delta coding, computes the difference between successive array elements. The value in the encoded array is the same as the first value in the input array. After that, the difference between the next element and the previous one is calculated and stored. Let $x \in \mathbb{N}^n$ be the elements of an integer array with size n, we calculate the differential coded value $\delta \in \mathbb{Z}^n$ as:

$$\delta_i = x_i - x_{i-1}$$

To decode, the inverse operation is done:

$$x_i = \delta_i + x_{i-1}$$

Calculating the difference between successive elements in an array means that the array has to be 1-dimensional. Unlike the quantized residual array, which is 1-dimensional, the position array is N-dimensional, according to the dimensions of the input data. Therefore, we flatten the array to be 1-dimensional. Figure 4.7 provides an example of differential coding.

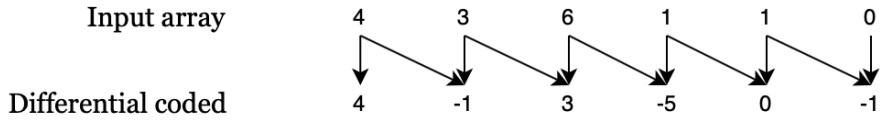

Figure 4.7: *Differential coding example.*

Now that the integers are smaller, we proceed to encode the differential coded array with bzip2. We choose bzip2 because it provided the highest compression factor among other compression modules like zlib [80] and lzma [81]. Algorithm 1 shows the pseudo-code of the residuals compression.

Algorithm 1 Residuals compression

Input Abs. error, Original Data, Reconstructed Data.

procedure CODERESIDUALS
  $difference \leftarrow original - reconstructed$
  $pos \leftarrow difference$
  $pos[(pos \leq threshold)$ `and` $(pos \geq \text{-}threshold)] = 0$
  $pos[(pos < 0] = 1$
  $pos[(pos > 0] = 2$
  $residuals \leftarrow$ `absolute`$(difference) > threshold$ ▷ Residuals outside abs. error
  $delta \leftarrow threshold \times 2$
  $residuals \leftarrow$ `round`$(\frac{residuals}{delta})$ ▷ Quantization
  $residuals \leftarrow$ `diffEncoder`$(residuals)$ ▷ Differential coding
  $pos \leftarrow$ `diffEncoder`(pos)
  $residuals \leftarrow$ `bzip2`$(residuals)$ ▷ Bzip2 coding
  $pos \leftarrow$ `bzip2`(pos)
end procedure

4.2 Implementation

The compression algorithm is implemented in Python 3.8.5[1]. We use Keras [82], an API that is included in Tensorflow 2.3.1 [83] for the implementation of the convolutional AE. Keras is a high-level deep learning API written in Python, running on top of the machine learning platform TensorFlow. Keras provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity, enabling fast experimentation.

For the lossless coding, we use the bzip2 [84] module and fpzip [18] python library. Bzip2 is a python module that provides a comprehensive interface for compressing and decompressing data using the bzip2 compression algorithm [16]. fpzip is a BSD-licensed open-source library for lossless or lossy compression of large multidimensional floating-point arrays [18].

The code of the implementation is available at `https://github.com/SilkeDH/lossy-ml`.

[1] `https://www.python.org/downloads/release/python-385/`

5 Results

This chapter presents the conducted experiments, discusses the performance of our compression algorithm under different setups and provides a comparison with state-of-the-art compression algorithms. First we provide details to the setup and preparation of the data and the training of the convolutional AE. Then, we compare different improvements to the base model. Next, we perform a hyperparameter tunning and analyze the CF gotten by the convolutional AE depending on the allowed absolute error. Lastly, we compare the proposed compression algorithm performance with state-of-the-art compression algorithms.

5.1 Experimental Setup

For the experiments, we use the global temperature values from 1979 and 1980 from the ERA5 hourly data on pressure levels [39]. We use the data from year 1979 for training and validation of the model and the data from 1980 for testing. One year of data is represented by a 4-dimensional 32-bit floating-point data array with the following dimensions $8784 \times 721 \times 1440 \times 37$ (time, latitude, longitude, level). One year of data is approximately 1.2 TiB in size.

Training the convolutional AE with such volume of data requires a considerable amount of time, therefore, a fixed amount of chunks of data are randomly sampled from the dataset for training and validation of the model. We decided to select 60k chunks of data for the training set and 6k for the validation set. The chunks of data have a size of $16 \times 48 \times 48$. We decided to keep the size of the chunks small so it will be easier to catch fine-grained data features [43]. Liu et al. [43] decided to test different block sizes for different datasets showing that for 3-dimensional floating-point datasets, smaller chunks lead to a better performance.

For the training of the different convolutional AEs we use the Adam optimization algorithm with an exponential decay rate of 0.9 for the moment estimates. The starting learning rate is set to 0.001 and decays from epoch 10 to epoch 100 in a gradual way to 0.0001. We train each model for 100 epochs with 4 GPUs and compute our evaluation metrics every 10 epochs. The batch size is 100, thus each GPU is working

with a batch size of 25. For the best performing models we increase the number of samples to 400k for the training dataset and 40k for the validation dataset. Different convolutional AEs are compared by computing the CF achieved by each of them.

The experiments were carried out on two clusters: the BwUniCluster 2.0 [85] and the HDFML System [86]. The bwUniCluster 2.0 is a parallel computer with distributed memory. Each node of the system consists of at least two Intel Xeon processors, local memory, disks, network adapters and optionally NVIDIA Tesla V100 accelerators. The HDFML System is provided by the Jülich Supercomputing Centre (JSC) [87] and is also a parallel computer with distributed memory. The nodes consists of Intel(R) Xeon(R) processors and NVIDIA Tesla V100 accelerators.

5.2 Number of Convolutional Layers

In order to find a suitable model for the problem, we decided to start by defining a convolutional AE with empirically chosen parameters to serve us as baseline. The convolutional AE architecture has 20 filters, each one with kernel size $3 \times 3 \times 3$ and stride $2 \times 2 \times 2$. The deconvolutional layers have the same structure as the convolutional layers. We use as activation function ReLU and the MSE as loss function. To compare the CF gotten by the convolutional AE depending on the error, we decided to bound the error by storing and further compressing the latent representation and the residuals with bzip2.

The number of convolutional layers with stride two define the amount of compression the convolutional AE can achieve. We decided to test three models with different estimated CF to define the number of convolutional layers. The first model has three convolutional layers, with a CF of 25.6, the second model, four convolutional layers with a CF of 204.8 and the third model, five convolutional layers with a CF of 1638.4. For the third model, we doubled the size of the chunk dimensions since adding a fifth convolutional layer halved the dimensions one more time, which was not possible for the initial size set.

For the calculation of the CFs, we decided to randomly select 50 data chunks of size $32 \times 721 \times 1440$. This will also be true for the following experiments unless otherwise stated. We chose32 time steps i.e., 32 hours to compress data containing day-night changes. Even though 12 hours would have been enough, the chunk size needs to be at least 16 time steps big. This is due to the amount of convolutional layers present in the model. Every layer halves each dimension, thus, four convolutional layers reduce the data 16 times. We decided to double the amount to 32 to compress more

data. The latitude and the longitude sizes cover all locations in the world. The mean CF is then calculated with each set absolute error. Figure 5.1 shows the different CF achieved by each of the models depending on the absolute error. The maximum absolute error set is 1 Kelvin (K). We decided to not take into account values higher than 1 K since we concentrate in small error bounds.

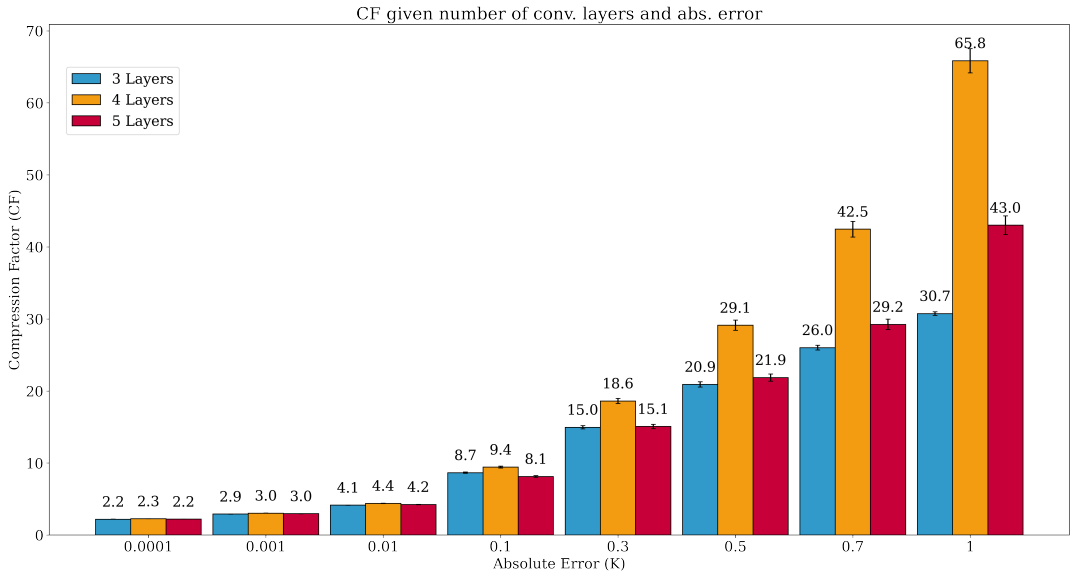

Figure 5.1: *CF achieved with different number of convolutional layers.*

For absolute errors below 0.1 K, the CF gotten by the models differ by at most 3%. It also shows that convolutional AEs are not that effective compressing values with very small error-bounds since smaller absolute errors leads to higher amount of residuals, thus decreasing the CF. This behaviour is also seen in Liu et al. [43], the authors use the Lorenzo predictor instead of the AE for such small errors since it achieves better CFs. As the error increases, we see that the model with four layers outperforms the other two. This lies in the fact that the convolutional AE with three convolutional layers already achieves almost its maximum CF when having an absolute error of 1 K. Note that the finally achieved CF is higher than the previously estimated through the convolutional AE. This is because the quantization and encoding of the latent representation and residuals further increase the CF. A similar behaviour is seen in convolutional AE with five convolutional layers. The CFs of the model increase slower than the model with four convolutional layers. This is due to higher errors in the predictions, thus, increasing the size of the error array and reducing the CF.

5.3 Adding Climate Information

Temperature changes faster depending on whether the surface is land or water. On land, the temperature warms and cools quicker, leading to strong temperature differences in regions like coasts where land and sea meet [88]. Moreover, the temperature varies depending on the altitude. In the lower atmosphere, the higher the altitude is, the colder it is in such region. In other words, temperature values that are located in mountain ranges like the Andes, present drastic changes throughout the region, increasing the complexity of the data. Providing location and surface information could help the model to learn about such complex areas and therefore achieve a better reconstruction.

The ERA5 dataset gives such formation by providing the latitude and the longitude of the temperature values on the grid. Moreover, there is another dataset called ERA5 hourly data on single levels from 1979 to present [89] that provides information about a large number of atmospheric, ocean-wave and land-surface quantities. The land-sea mask variable provides information about the proportion of land and water in a grid box. This parameter has values ranging between zero and one and is dimensionless. Values above 0.5 indicate that the surface is mostly land and below 0.5, mostly water. Since the temperature depends on whether the surface is land or water, we decided to include this parameter as an input. Figure 5.2 visualizes a time-step of the temperature and land-sea mask dataset.

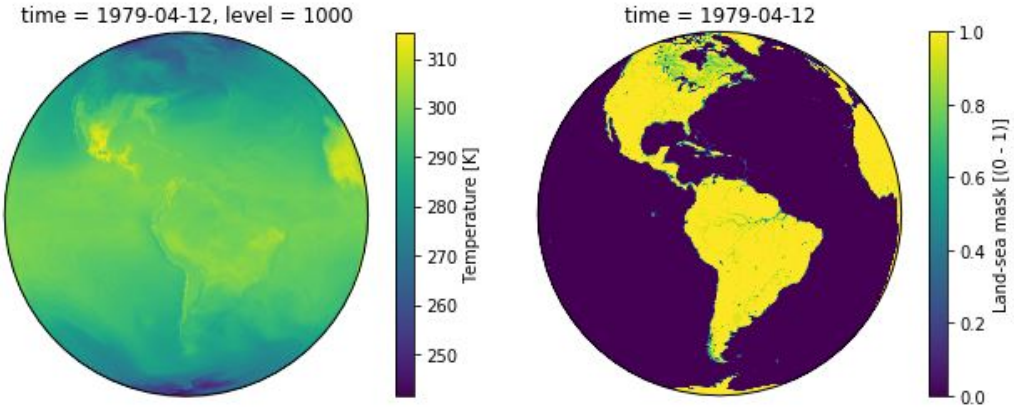

Figure 5.2: *Visualization of temperature and land-sea mask values.*

We decided to test if providing the model information about the location or type of surface of the chunk helps the model to improve the results. Three convolutional AEs with the same architecture but with different inputs were trained. All models have as base input the temperature values, the difference lies in the type of extra information each of one will get:

- Temperature (only)
- Temperature with land-sea mask
- Temperature with latitude and longitude position

The first model was trained only using temperature values, serving us as baseline to see if extra information improves the reconstruction. The second model has as extra information, the land-sea mask values, providing the model knowledge about the surface. The third model includes information about the latitude and longitude of the data. Before feeding the model the latitude and longitude values, we applied a transformation to such values. The latitude has a range of $[90, -90]$ degrees and the longitude of $[-180, 180]$ degrees. Since the earth is a sphere, we transformed the longitude values to indicate that -180 degrees and 180 degrees are the same point on earth. This was not necessary for the latitude since the limits of the range do not denote the same point. Moreover, to facilitate the training, we scaled to values to $[-1, 1]$. We code the latitude and longitude using two features, for the longitude:

$$\text{lon}_1 = \sin(2 \cdot \pi \cdot (\frac{\text{longitude} + 180}{180}))$$
$$\text{lon}_2 = \cos(2 \cdot \pi \cdot (\frac{\text{longitude} + 180}{180}))$$

and for latitude:

$$\text{lat}_1 = \sin(\pi \cdot (\frac{\text{latitude} + 90}{90}))$$
$$\text{lat}_2 = \cos(\pi \cdot (\frac{\text{latitude} + 90}{90})))$$

Note that the latitude was not multiplied by 2 like in the longitude since it only covers half the earth. We plotted the coded latitude and longitude in Figure 5.3.

The results are presented in Figure 5.4 and show that the performance of the model increases when adding information about the location and surface of the data. As expected, the model with the highest CF is the model that has as input the

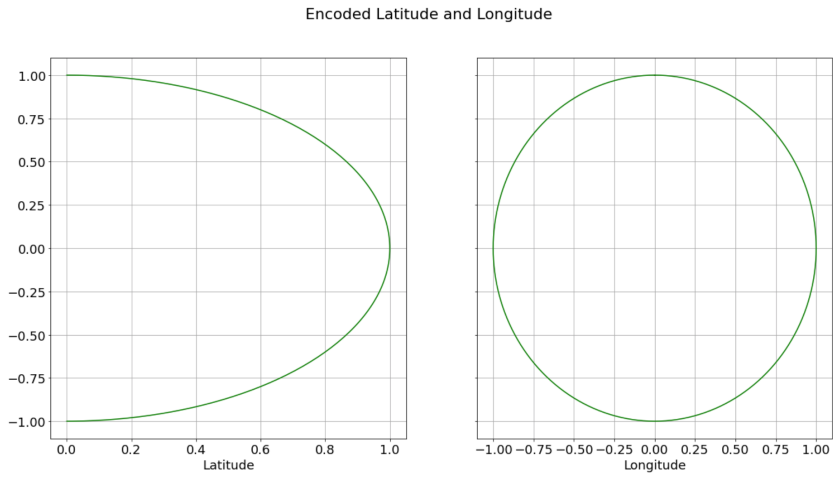

Figure 5.3: *Encoded latitude and longitude.*

temperature values with the land-sea mask, i.e. the second model. As mentioned before, the surface of the earth plays an important role in the behaviour of the temperature values. We also see that the third model which had information about the latitude and longitude achieves a higher CF than the model without any extra information.

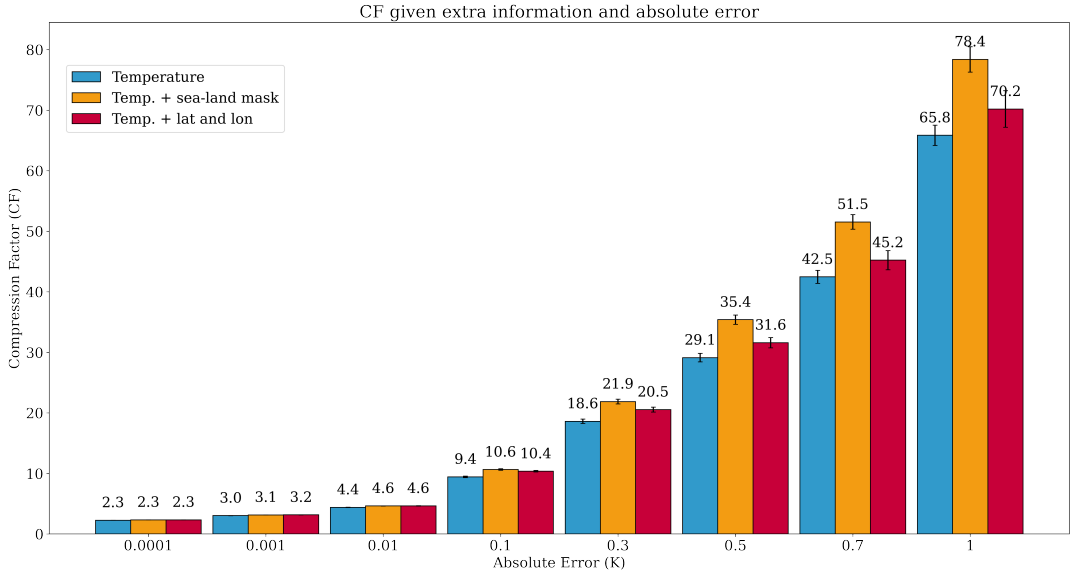

Figure 5.4: *Visualization of temperature and land-sea mask values.*

We also see in Figure 5.5 the training and validation losses of the three models. The first model starts learning faster than the other two, however, after a certain number of epochs, the second and third model start reaching smaller training and validation

losses, meaning that the models present smaller errors when reconstructing the data by taking into account the extra information about the location of the temperature values. For this reason, we decided to add as extra information the land-sea mask dataset to our final model in order to achieve higher CFs when compressing the temperature values.

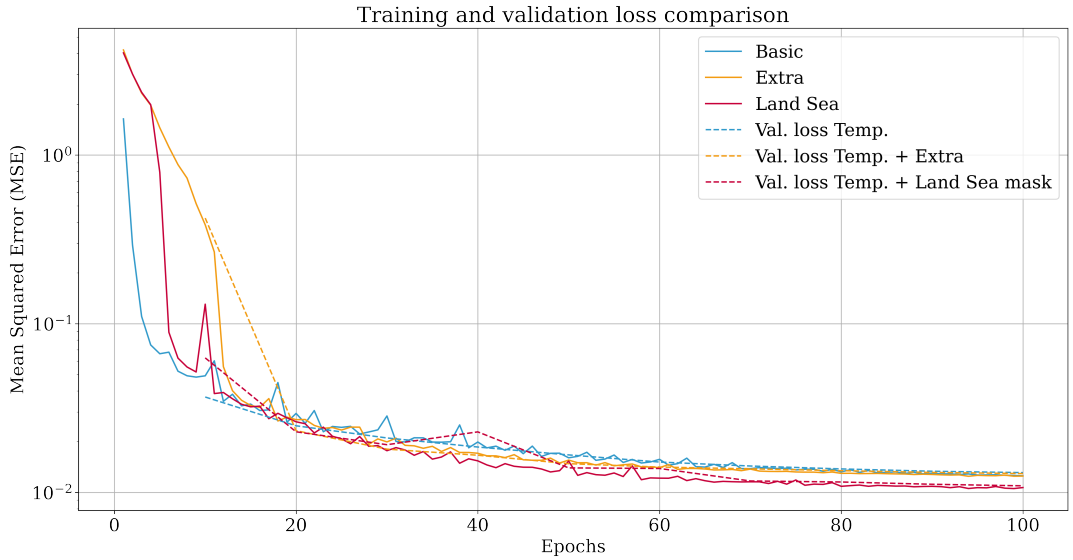

Figure 5.5: *Train and validation losses achieved by the models with different inputs.*

5.4 Residuals Encoding

After getting the reconstructed data by the convolutional AE, the residuals are computed, quantized and encoded. We tested two ways to quantize the residuals. The first method consists of storing two arrays, q_p and q_n. Array q_p stores the quantized positive residuals outside the absolute error and the second one, q_n, the quantized negative residuals outside the absolute error. These arrays keep the position information of the values by maintaining the shape of the residual array. The second method also stores two arrays, q_e and pos, but differently. Array q_e stores the residuals outside the absolute error and pos the positions of the residuals that are in q_e taking into account their sign (see Section 4.1.3). For the tests, the latent representation array *latent* was encoded with bzip2 and the rest encoded with differential coding and bzip2. Figure 5.6 shows the CF gotten with both methods. Note that the model used for testing is the one without extra information.

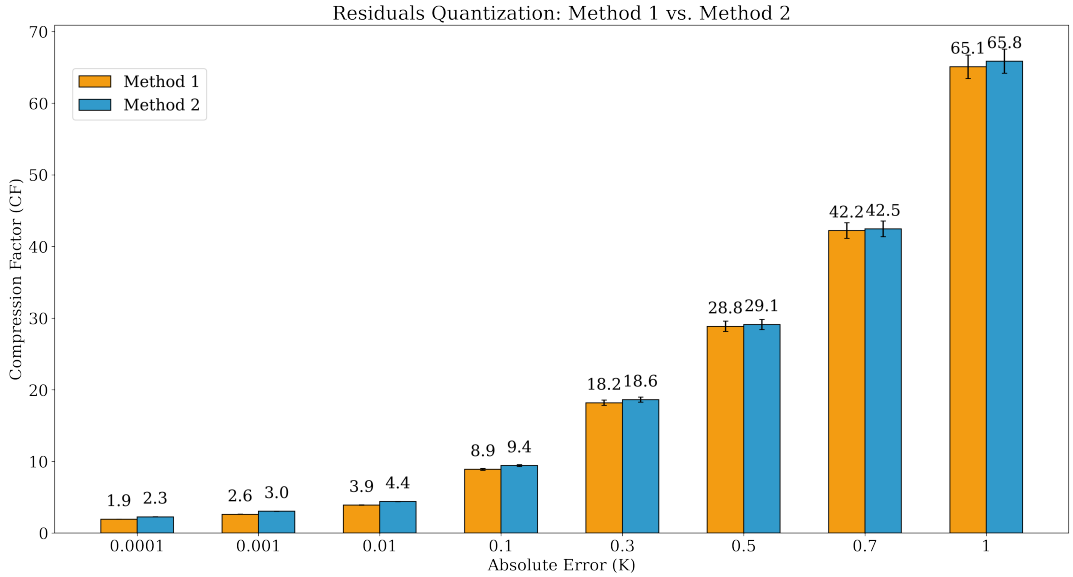

Figure 5.6: *CF gotten with different residual encoding methods.*

		Absolute Error (K)							
		0.0001	*0.001*	*0.01*	*0.1*	*0.3*	*0.5*	*0.7*	*1*
Method 1	$latent(\%)$	0.01	0.01	0.02	0.04	0.08	0.13	0.19	0.29
	$q_p(\%)$	0.57	0.57	0.56	0.55	0.53	0.51	0.47	0.40
	$q_n(\%)$	0.42	0.42	0.42	0.41	0.39	0.36	0.34	0.31
Method 2	$latent(\%)$	0.01	0.01	0.02	0.04	0.08	0.13	0.19	0.30
	$q_e(\%)$	0.40	0.91	0.86	0.60	0.27	0.15	0.10	0.06
	$pos(\%)$	0.05	0.07	0.12	0.36	0.65	0.72	0.71	0.64

Table 5.1: *Percentage of space consumption with different quantization methods.*

Figure 5.6 shows that the second method achieves higher CFs than the first method. Table 5.1 presents the percentage of space taken by the residuals and the latent representation by each method for each absolute error in the compressed output. Note that the header information included in the output file (mean, standard deviation and shape of data array) was not taken into account for the calculation of the space consumption.

The results show the behaviour of each method. We can see that the percentage of space taken by the latent representation in the compressed file with the first method is less than with the second method. This means that, with the first method, the residuals consume more space. The residuals are almost equally split into positive and negative values. In the case of the second method, all the residuals outside the

Method	Absolute Error (K)							
	0.0001	*0.001*	*0.01*	*0.1*	*0.3*	*0.5*	*0.7*	*1*
bzip2	**2.26**	**3.04**	4.40	**9.44**	**18.64**	**29.17**	**42.52**	**65.99**
lzma	1.66	2.23	3.23	6.93	14.62	23.12	33.71	52.30
zlib	**2.26**	3.0	**4.53**	9.15	17.89	27.45	39.0	59.56

Table 5.2: *CF with different lossless encoders.*

Method	Absolute Error (K)							
	0.0001	*0.001*	*0.01*	*0.1*	*0.3*	*0.5*	*0.7*	*1*
bzip2	2.26	3.03	4.40	9.42	18.52	28.70	41.71	64.57
fpzip	2.26	3.03	4.40	**9.46**	**18.67**	**29.07**	**42.50**	**66.47**

Table 5.3: *CF by coding the latent representation with bzip2 [84] and fpzip [18].*

absolute error are stored in q_e. The lower the absolute error, the higher the space consumption of the residuals since more residuals need to be stored. The second method achieving higher CFs than first method means that the residuals are better encoded when they are stored in the same array instead of having them separated by sign without a position array. Therefore, we decided to use the second method to quantize the residuals.

As mentioned before, the quantized residuals are differentially coded and then losslessly encoded. For the encoding, we tested three different lossless encoders that already come with Python to see which one achieves a higher CF: bzip2 [17], lzma [81] and zlib [80]. Table 5.2 shows the CFs gotten with each of the encoders. The results show that the highest CF is achieved by bzip2 [17]. Therefore, we decided to quantize the values with the second method and encode them with bzip2 [17]. We also decided to test fpzip [18] for the latent representation to see if we can further compress the data. The reason why we try fpzip is because it is designed to compress floating-point arrays like the latent representation. Table 5.3 shows that fpzip [18] works better for absolute errors above 0.1. For absolute errors below 0.1 the CFs are the same, this is due to the residuals playing a major role in the CF since they occupy more space as seen in Table 5.1. Thus, we decided to use fpzip [18] as lossless encoder for the latent representation.

5.5 Hyperparameter Optimization

We previously showed that the convolutional AE with four layers presented promising results and that the hyperparameters used for training the network were chosen empirically. In this section we perform a hyperparameter optimization to improve the model's predictions by using random search. Random search is a strategy used for hyperparameter optimization and works by randomly sampling a certain amount of values for each hyperparameter, comparing the performance with each set of hyperparameters and choosing the best one [90]. We decided to use random search since the amount of hyperparameters to be tuned and training time needed are high. We tuned five hyperaparameters of the convolutional AE with the following ranges:

- Number of filters $N \in \{10...40\}$.
- Kernel size $k \in \{3...7\}$.
- Number of residual blocks $res \in \{1...3\}$.
- Learning rate $lr \in [0.0001, 0.01]$.
- L2-loss λ coefficient $l2 \in [0.00005, 0.005]$.

44 models with a different set of hyperparameters were trained as indicated in Section 5.1, the training of the models was done without extra climate information (see Section 5.3). Figure 5.7 shows the validation loss (MSE) achieved by each of the models depending on the hyperparameter value. The first plot shows the number of filters and depicts that the minimum MSE values have been reached by setting 20 and 23 filters. However, we can see that other models that also had 20 and 23 filters achieved really high MSE values. This is due to other hyperparameters that affect the performance of the convolutional AE. For the kernel size, values between four and five present the best results. Adding residual blocks to the convolutional AE also helped reducing the MSE: Having one residual block leads to smaller MSE in comparison to none or more than one. For the learning rate, we can see that the higher the learning rate, the less the MSE. This is because the smaller the learning rate, the slower the convolutional AE learns, therefore, not being fast enough to achieve smaller errors since all models train with the same number of epochs. The opposite happens when the learning rate is set too high. When updating the weights, the network will most likely pass the minimum and not be able to achieve a smaller MSE. Instead of learning slowly, the model will not learn or even suffer of exploding gradients resulting in a very high MSE. There is not a clear tendency for the L2-regularization parameter. The highest values present a tendency to increase the MSE. This is because the L2 regularization starts dominating which makes optimizing the function by adjusting

the weights more difficult.

From the 44 models, we chose the two with the smallest validation error. Table 5.4 shows the parameters values of the best two models. Next, we calculate the CF reached by the two models and select the one with the highest CF as the final model.

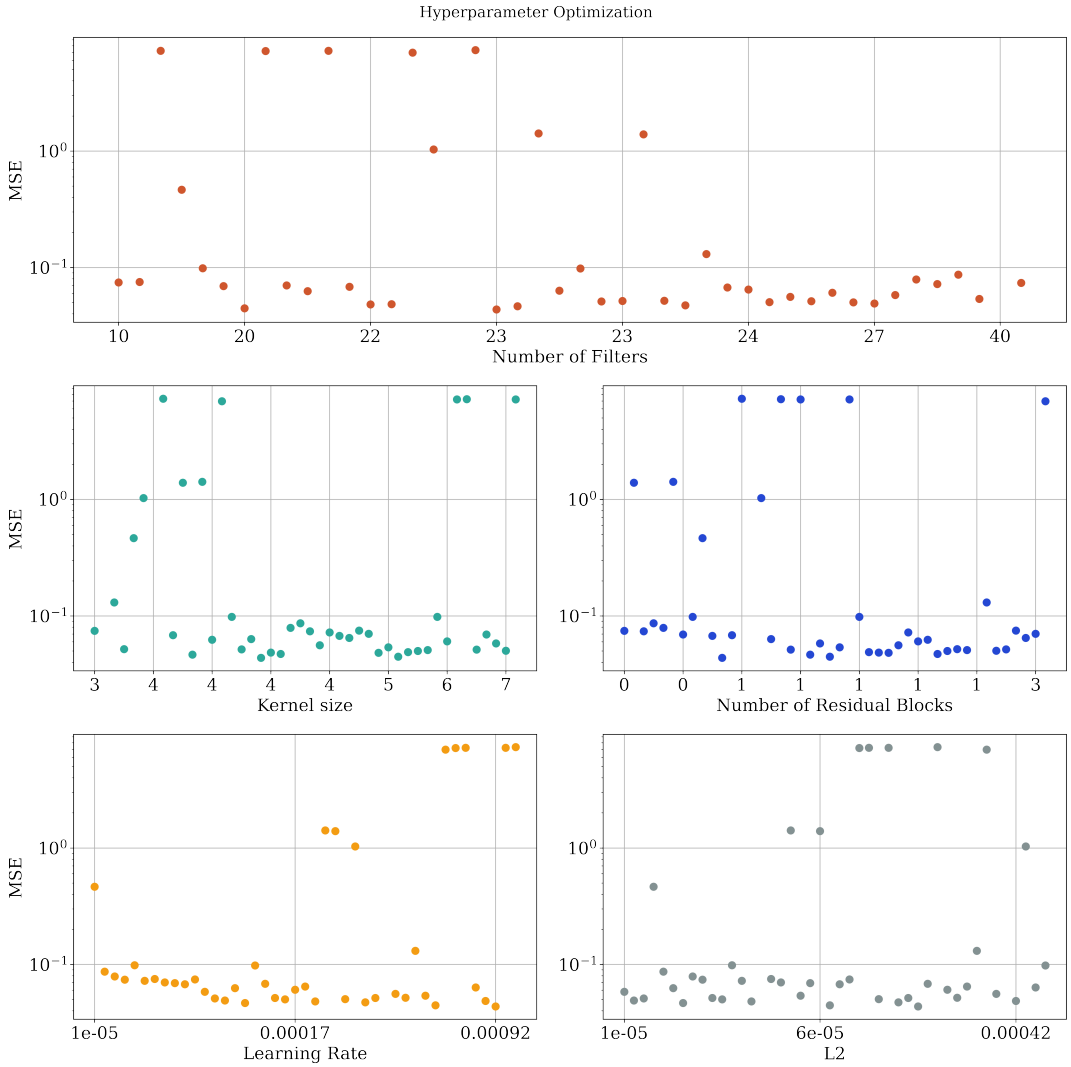

Figure 5.7: *MSE achieved by the models with different parameters.*

The best performing models resulting from random search were trained with a bigger training and validation set. For the training set, 400k chunks were randomly sampled from the dataset, and for the validation set, 40k. The models were trained for 130 epochs with the land-sea mask as extra information described in Section 5.3.

	Parameters					
	N	*k*	*lr*	*res*	*l2*	Val. Loss (MSE)
Model 1	20	5	1.148E-4	1	5.8989E-05	0.0446822
Model 2	23	4	1.204E-4	1	1.3702E-05	0.0465775

Table 5.4: *Parameters and validation loss of the two best models.*

We calculated the CF gotten by the two models and chose the best one. Figure 5.8 shows the compression factor reached by the two models.

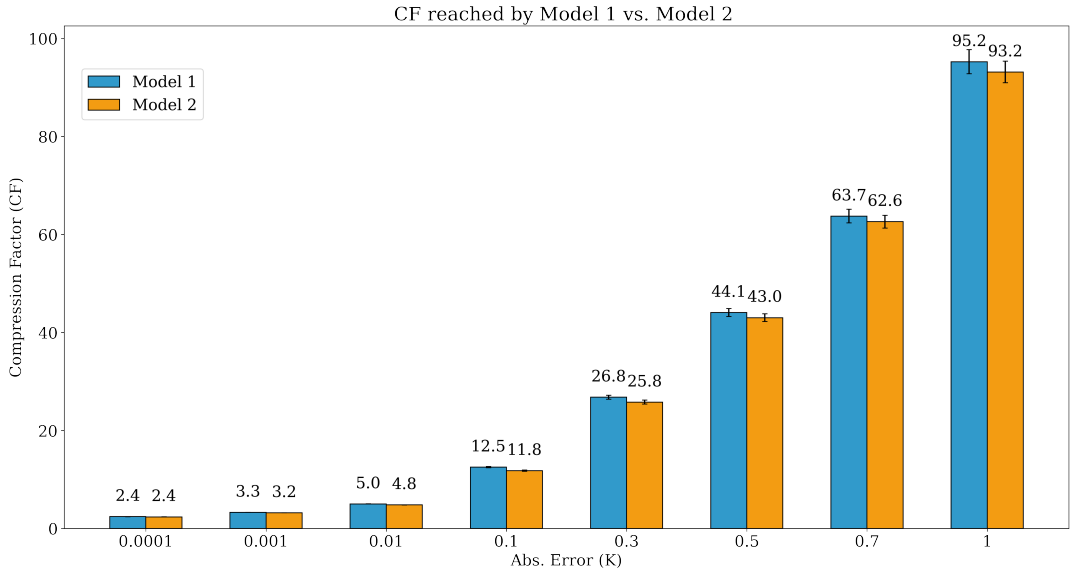

Figure 5.8: *CF achieved with the best two models gotten through hypeparameter tuning.*

The results show that the models achieve a similar CF but the first model is slightly better. Moreover, the CF increased from 78.4 to 96.2 in comparison to the CF reached by the model using the land-sea mask described in Section 5.3 without hyperparameter tunning and using less data for training. The maximum CF achieved is 95.2 for an absolute error of 1 K and a minimum of 2.4 for 0.0001 K. Since the best performing model is model 1, we present the final architecture of our convolutional AE in Table 5.5.

5.6 Compression Algorithm

In this section, we analyze our compression algorithm's behaviour. We randomly select a portion of data from the year 1980 and use it for the visualization of the

AE Architecture			
Layer	# Filters	Kernel size	Stride
Conv.	20	$5 \times 5 \times 5$	(1, 1, 1)
Res.Block	20	$3 \times 3 \times 3$	(1, 1, 1)
$4\times$ Conv.	20	$5 \times 5 \times 5$	(2, 2, 2)
$4\times$ Deconv.	20	$5 \times 5 \times 5$	(2, 2, 2)
Res.Block	20	$3 \times 3 \times 3$	(1, 1, 1)
Conv.	1	$5 \times 5 \times 5$	(1, 1, 1)

Table 5.5: *Convolutional AE Architecture.*

Absolute Error (K)	Count (>abs. error)	Percentage (%)
0.0001	1037963	99.97
0.001	1035491	99.73
0.01	1010825	97.36
0.1	769729	74.14
0.3	391564	37.71
0.5	209380	20.16
0.7	118343	11.40
1.0	56589	5.45
5.0	259	0.03
Total values = 1038240		

Table 5.6: *Percentage of residuals values above absolute error.*

results. We start by visualizing the reconstructed data from the convolutional AE in Figure 5.9. The figure shows a visual comparison between the original data and the reconstructed data together with the residuals. Since the higher frequencies of the data are filtered out by convolutional AE, we can see that the reconstructed data is a blurred version of the original data. In other words, the zones that present more temperature variations will tend to look more blurry in contrast to zones where the temperature is more stable, e.g., coast lines vs. ocean. We can also see that the residual values lie within the range of $[-12.17, 8.45]$ K, however, only a very small percentage of the data presents such extreme values as it can be seen in Table 5.6 where only 5.45% of the data presents an error greater than 1 K. Nevertheless, such huge errors are unacceptable for the analysis of scientific data.

Figure 5.10 visualizes the residuals gotten for different absolute errors with the same data used in Figure 5.9. For the smallest absolute errors (1E-4 to 1E-1), no pattern can be seen since almost every value lies outside the absolute error as seen

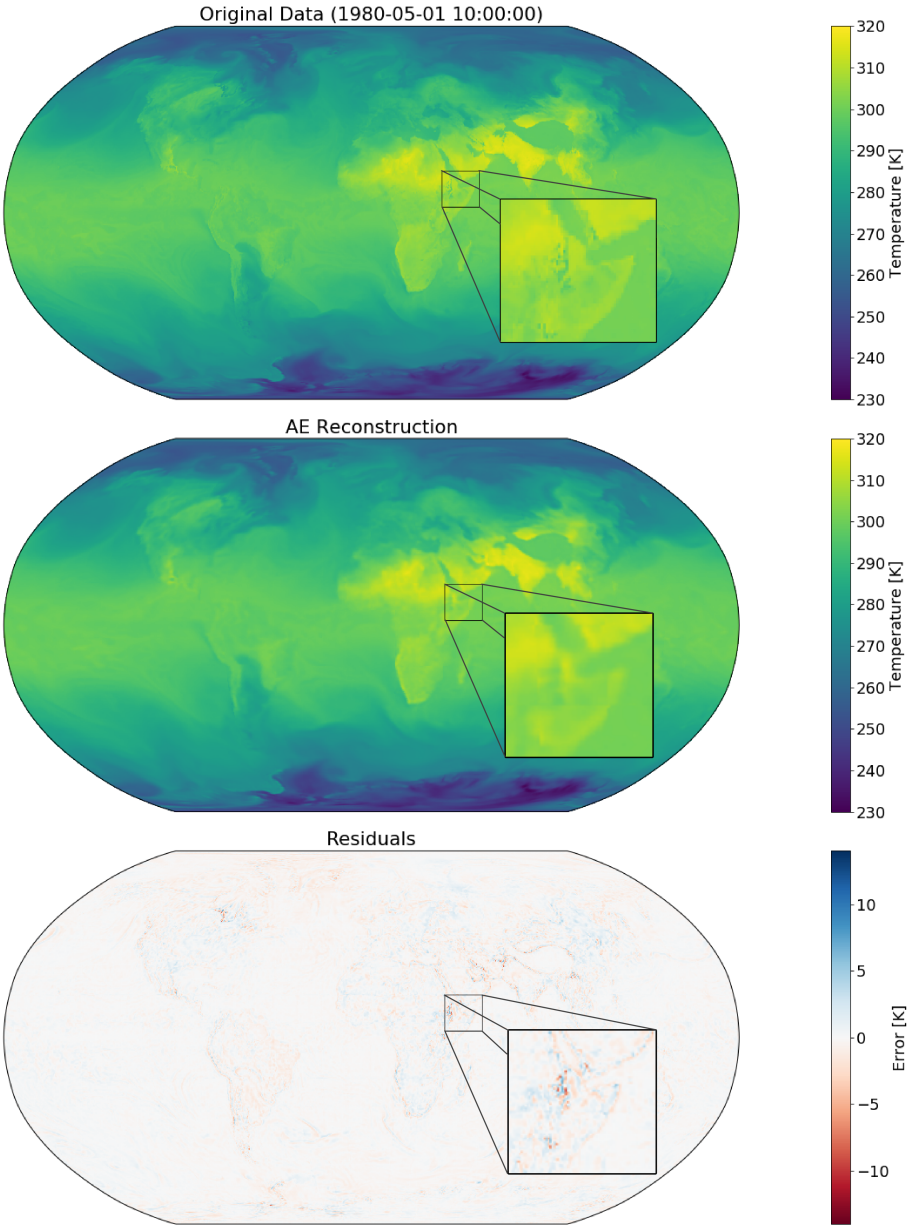

Figure 5.9: *Reconstruction of the original data with the convolutional AE.*

in Table 5.6. However, the larger the acceptable error, the better we see the zones on earth where the convolutional AE has trouble to reconstruct. To have a better understanding of the residuals, Figure 5.11 provides the respective histogram for each plot presented in Figure 5.10. We can see that the more we decrease the absolute error, the more uniform is the distribution of the residuals. Losslessly compressing values that follow a uniform distribution is very difficult since every value has equal probability to happen, thus, making them unpredictable and leaving no space for compression [91]. However, the values can be compressed if they are correlated. As we see in Table 5.6, using bzip2 for the residuals slightly increases the CF, showing that the residuals present some correlation.

We take a look to the encoder's output space consumption in Figure 5.12. We show the three main parts of the encoder output: the latent representation, the error array and the position array. Naturally, the greater the absolute error, the less space the error array takes. However, we can see that the position array takes more than half the space for storing the positions. Taking into account that nearly 5% of the data is outside the absolute error and only the position of these values are important to store, the position array is taking too much space. This is due to the storing of the position of all values, including the ones that are inside the absolute error. A different approach for cases where the amount of residuals are small should be taken e.g., storing the position index from the residuals outside the absolute error only.

After analysing the behaviour of our compression algorithm, we proceed to compare its performance with two of the current state-of-the art compression algorithms: SZ [20] and zfp [19].

5.7 Comparison with the State-of-the-Art

SZ [20] and zfp [19] are considered state-of-the-art compression algorithms for floating-point data. To prove the effectiveness of our compression algorithm, we perform three different evaluations. The first evaluation consists of comparing the CF achieved with different absolute errors by each of the compression algorithms. Second, we make a visual comparison with the same CF between the reconstructed values by each model. Moreover, we visually compare the results with the same absolute error and PSNR. The last evaluation consists of the compression and decompression speed achieved by each compression algorithm.

We present the CF results of all three lossy compression algorithms in Figure 5.13. The first experiment shows that our compression algorithm outperforms both of

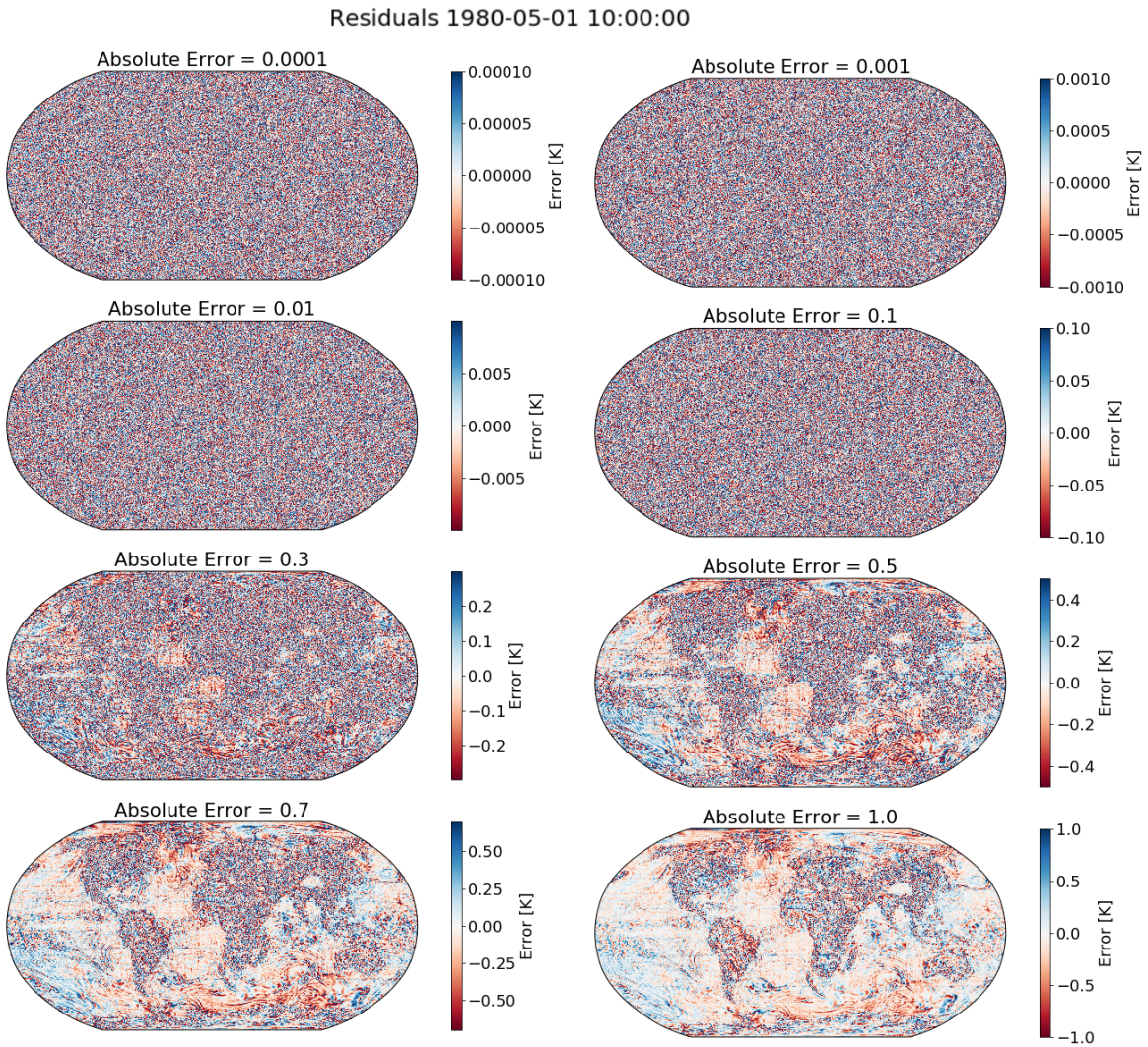

Figure 5.10: *Residuals gotten with different absolute errors.*

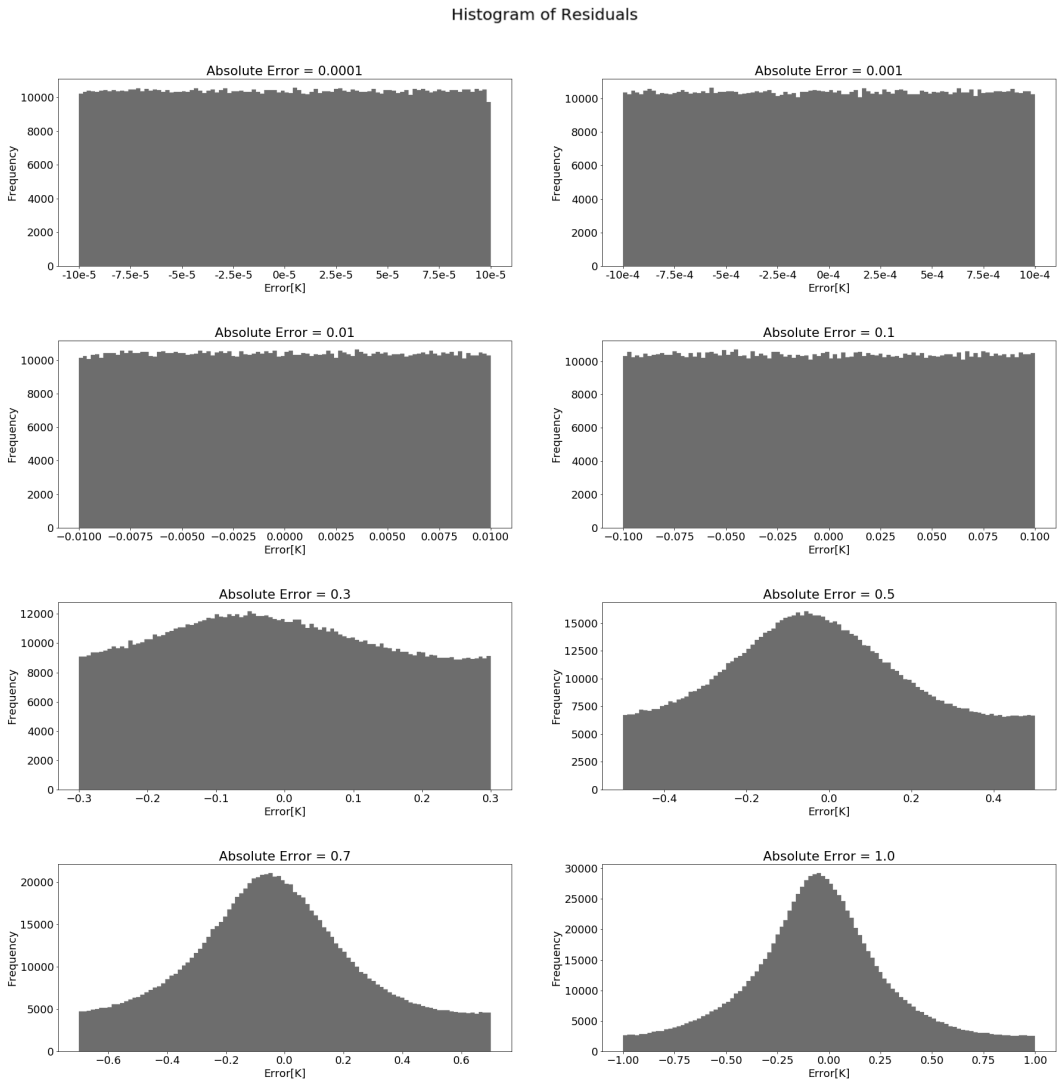

Figure 5.11: *Histogram of the residuals with different absolute errors.*

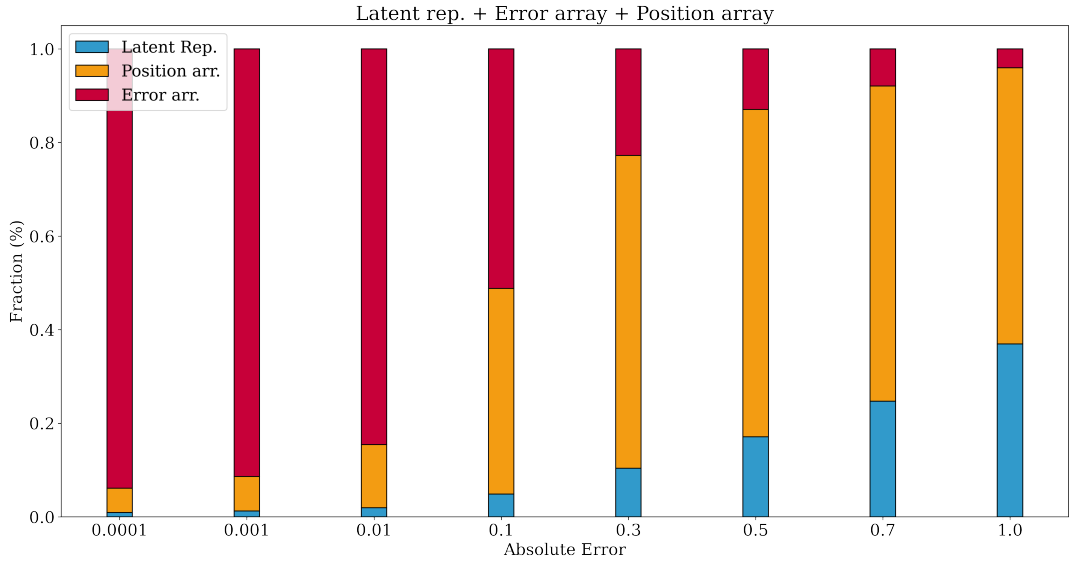

Figure 5.12: *Compressed output space consumption.*

the state-of-the-art compression algorithms. Specially for absolute errors above 0.3 where we obtain 130% $\sim$ 400% improvement in CF for the same absolute errors in comparison to SZ and zfp. We also see that the CF of our algorithm has higher variance than the state-of-the-art compression algorithms by a factor of four. This variance could be caused by the convolutional AE's reconstruction quality which might slightly differ depending on the complexity of the data to be compressed, thus increasing the amount of residuals outside the error threshold. We can also observe that our compression algorithm is the best even for very small absolute errors like 0.001 K which are promising results. Furthermore, we can observe that zfp performs the worst in all cases. This is due to zfp not using the full range of the absolute error, i.e., zfp presents in the decompressed data a much lower absolute error than the permissible one, leading to low CFs. Such behaviour is not seen in the proposed compression algorithm nor in SZ.

For the second evaluation, we compare the PSNR gotten by the three lossy compression algorithms. The PSNR (see Section 2.1.2) indicates the rate of distortion present in the reconstructed data. The higher the PSNR value is, the smaller the distortion in the decompressed data [43]. We present visualizations over a selected zone (South America) in Figure 5.14 with a fixed CF of 21.0. The results show that the reconstructed data of our compression algorithm has almost double PSNR than SZ with a smaller absolute error, completely outperforming SZ. In the case of zfp, we present two values for the absolute error, the first one is the calculated absolute error of the resulting reconstructed data and the second one the absolute error given

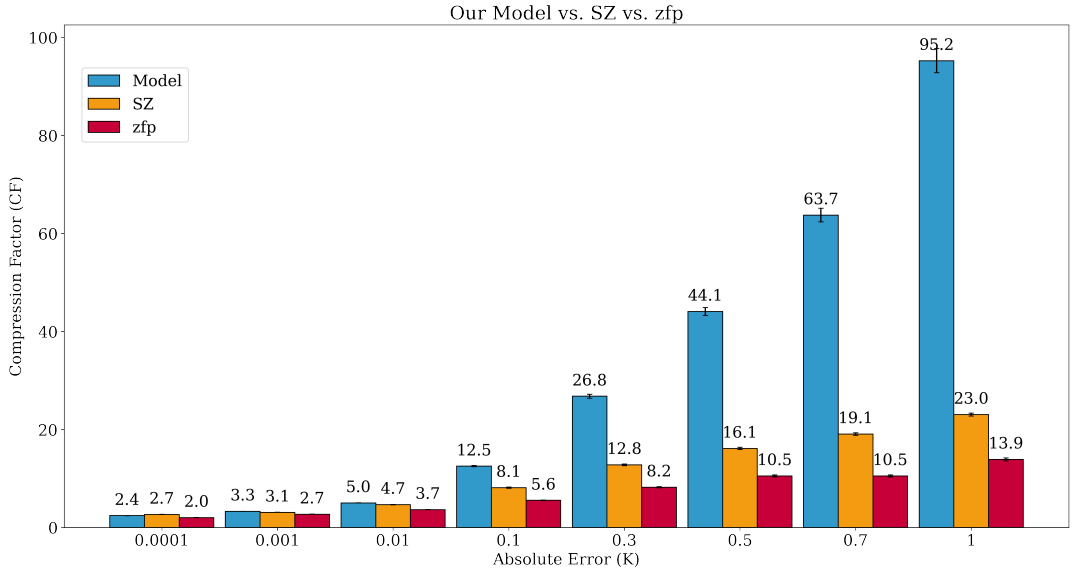

Figure 5.13: *CF comparison (Our model vs. SZ vs. zfp).*

as input in zfp. We can see that for an absolute error of 5 K, zfp reaches the same compression factor than our compression algorithm; however, an absolute error of 5 K is unacceptable for climate data analysis. Even though we set in zfp 5 K as absolute error, the actual absolute error present in the data is 0.8. In addition, prior works show that a PSNR in the range of $[30, 60]$ is good enough to have a high visual quality for different scientific applications [43].

Figure 5.15 presents the reconstructed data with each of the compression algorithms with a fixed PSNR of 38.0. The results show that for the same reconstruction quality, our compression algorithm achieves a three times higher CF than SZ while having a slightly higher absolute error. This means that our compression algorithm achieves the same reconstruction quality as SZ while admitting a higher error in the reconstruction. In the case of zpf, the CF achieved is higher than ours. However, the absolute error had to be set to 20 K in order achieve that. For higher absolute errors, our compression algorithm outperforms zfp in terms of CF but the visual quality decreases.

We present in Figure 5.16 the reconstructed data by the three compression algorithms with an absolute error of 0.01. Since the absolute error is small, a visual difference is difficult to identify. However, the results vary for every compression algorithm. We can see that our compression algorithm outperforms both state-of-the-art algorithms by reaching a CF of 52.34 while SZ and zfp achieve a CF of 7.51 and

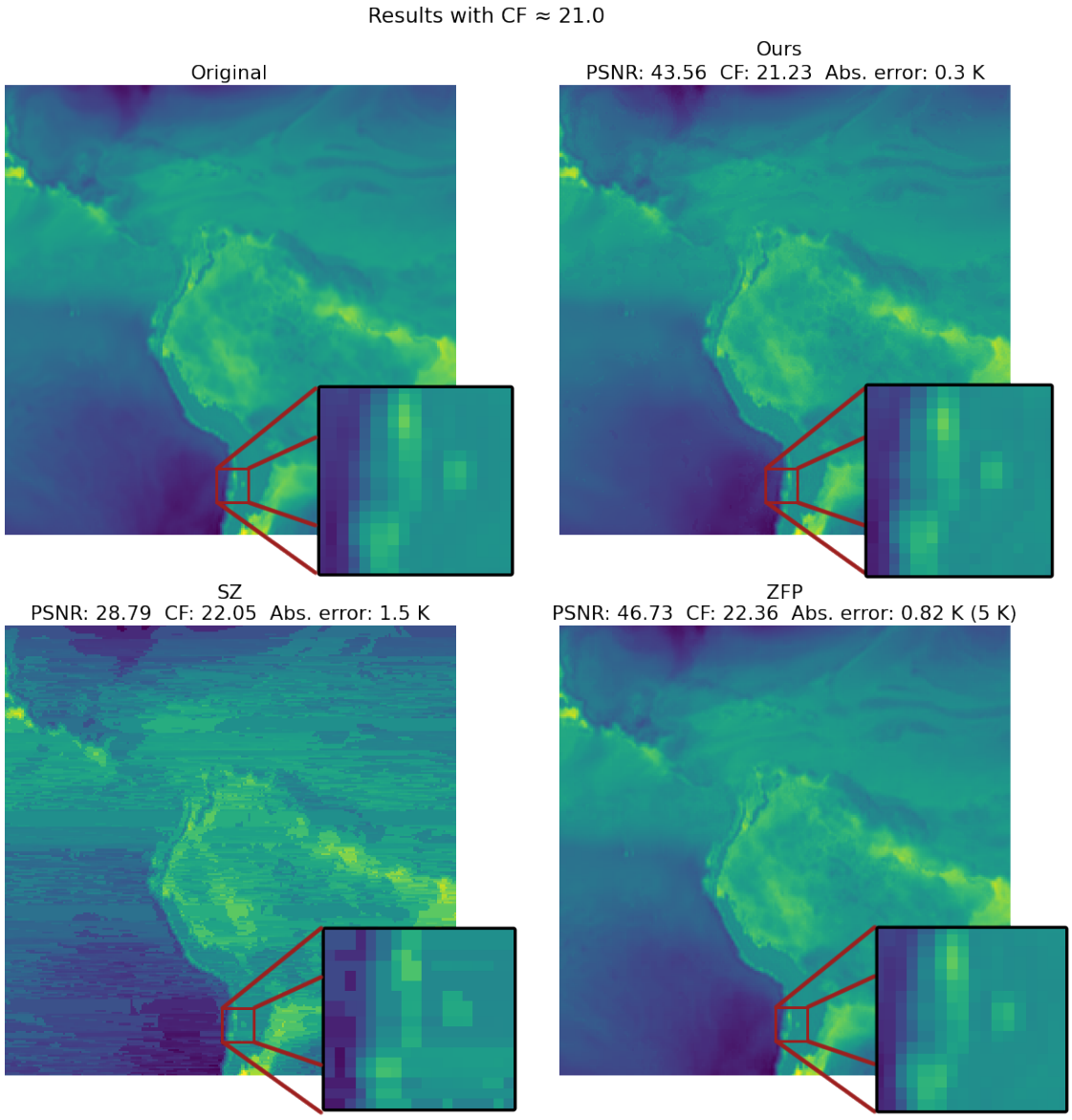

Figure 5.14: *Visualization of reconstructed data with CF ≈ 21.0.*

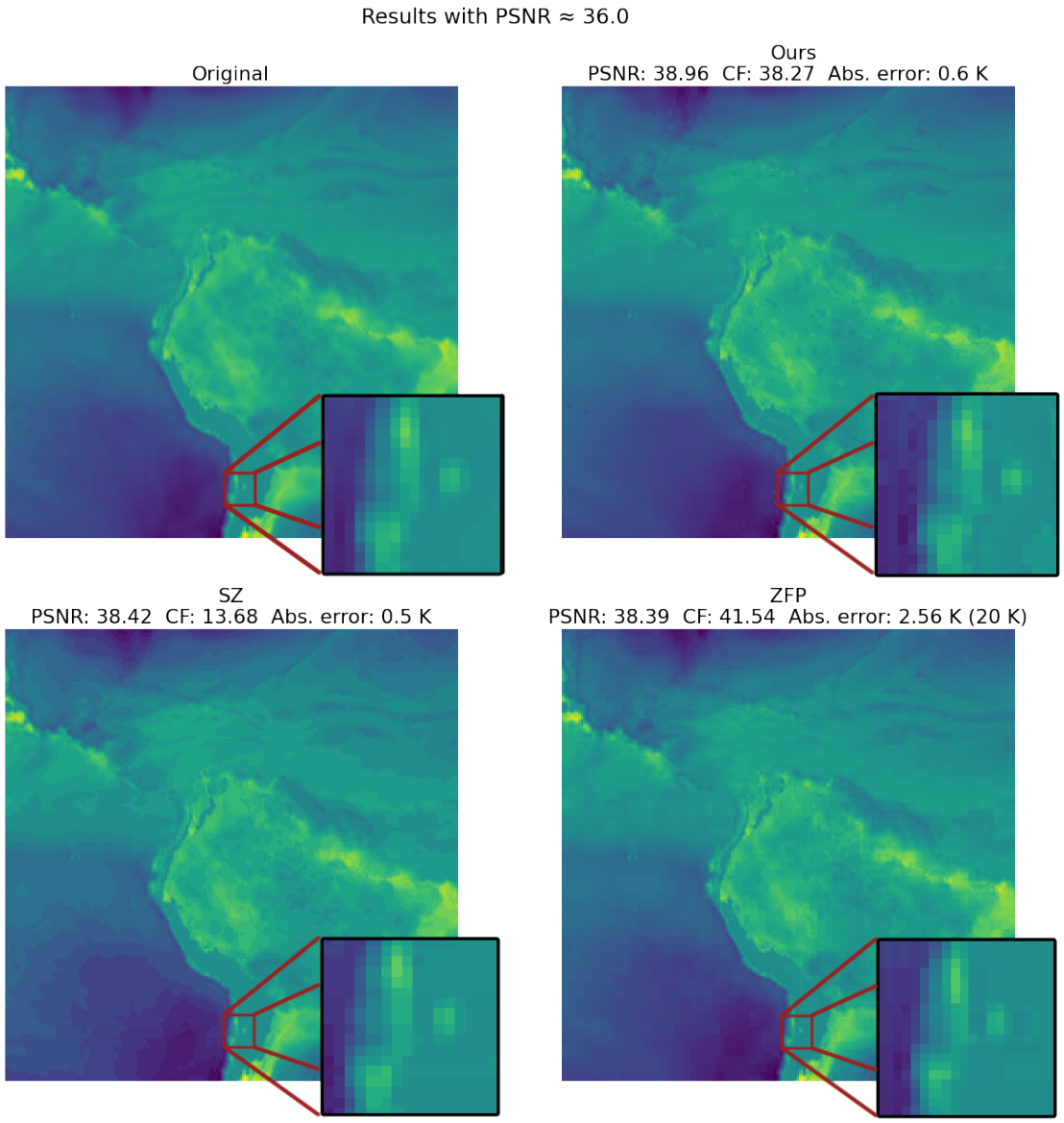

Figure 5.15: *Visualization of reconstructed data with PSNR ≈ 38.0.*

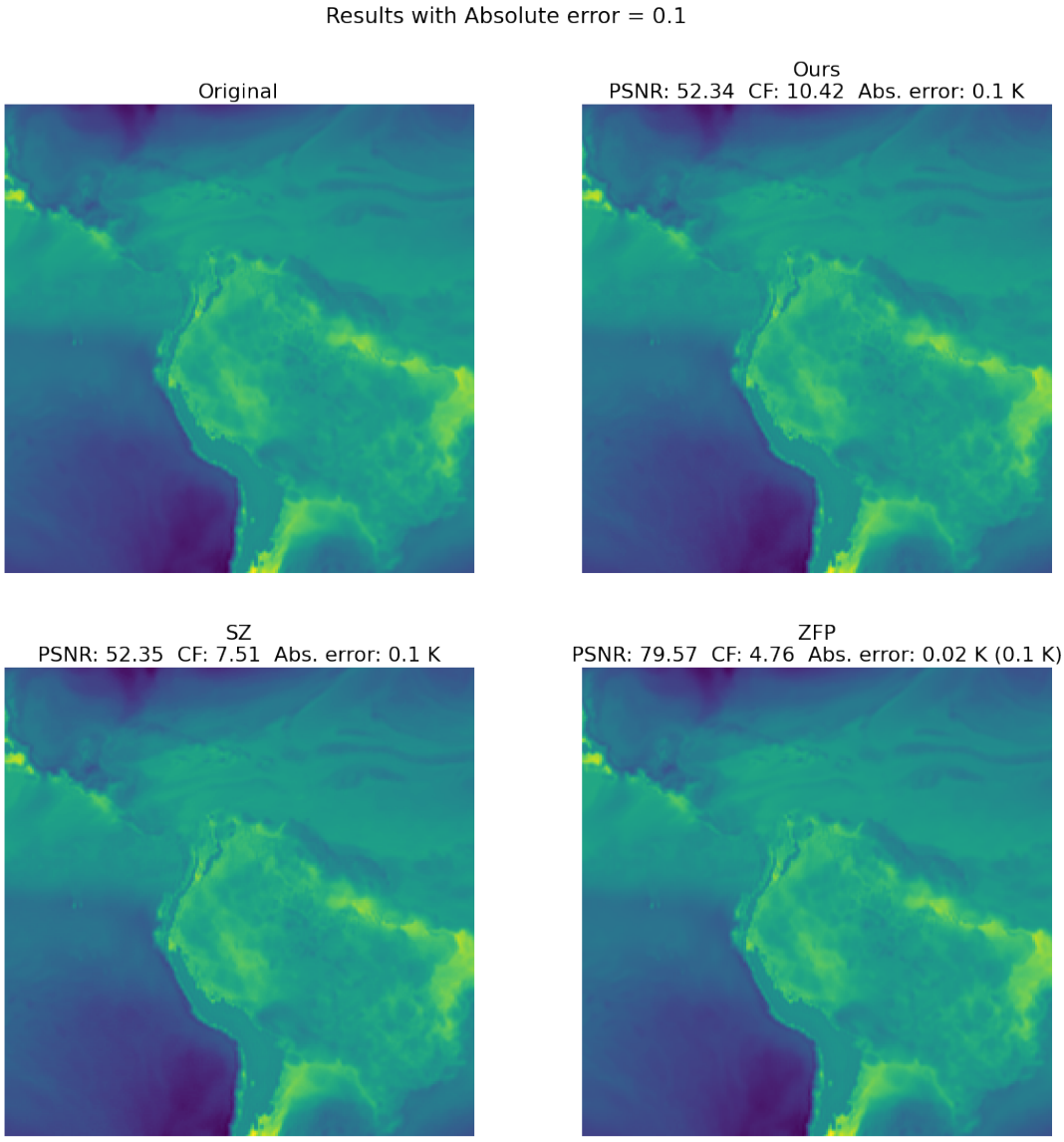

Figure 5.16: *Visualization of reconstructed data with absolute error* = 0.1

Absolute Error (K)	Ours	SZ [20]	ZFP [19]
0.0001	19.68	2.02	1.69
0.001	16.89	1.97	1.54
0.01	16.93	1.97	1.43
0.1	14.02	1.97	1.30
0.3	10.32	1.96	1.19
0.5	9.15	1.96	1.14
0.7	8.64	1.97	1.15
1.0	8.30	1.97	1.07

Table 5.7: *Compression + decompression speeds in seconds.*

4.76, respectively. The PSNR is the same in our compression algorithm and SZ's, i.e, our compression algorithm achievs higher CF while having the same recontruction quality as SZ. zfp reaches a higher PSNR since the actual absolute error is 0.02 and not 0.1 reducing the MSE, thus increasing the PSNR.

Next, we evaluated the execution time of the three compression algorithms. Table 5.7 shows the compression plus decompression time gotten by each of them. The results show that our compression algorithm can be $\sim 8\times$ slower than SZ and $\sim 19\times$ than ZFP. The operations of the convolutional AE are done in a GPU and consists of a fixed number of operations making the lossless encoders the reason why the compression algorithm takes such amounts of time for compressing and decompressing the values. The execution time of our algorithm varies according the absolute error. The higher the absolute error, the faster it compresses and decompresses. This is due to the error array depending on the amount of residuals outside the absolute error. The next chapter summarizes the achieved results and provides recommendations for improvement of our compression algorithm.

6 Conclusion and Recommendations

In this thesis, an error-bounded lossy compression algorithm for climate data with focus on high CFs has been designed, implemented and tested. The algorithm combines lossless residual compression with convolutional AEs to compress and decompress climate data. We studied the convolutional AE's behaviour by testing different CFs and providing extra information to the network such as the coordinates and the land-sea mask. We showed that the number of convolutional layers plays an important role in the performance of the compression algorithm and providing the surface type to the model improved the reconstruction of climate data. We tested different lossless encoders for the latent representation and the residuals, showing that bzip2 and fpzip reach the highest CF. Two different approaches for quantizing the residuals were also tested, demonstrating that a large part of the CF depends on the efficient compression of the residuals. Our compression algorithm was evaluated by comparing it with two state-of-the-art floating-point lossy compression algorithms, SZ and zfp. After parameter tunning, our compression algorithm achieved higher CFs than SZ and zfp do under the same error-bounds ([1E-3, 1]), showing an improvement by about 1.3 to 4 times higher compression factors, demonstrating a great potential for lossy compression with focus on high CFs. Our compression algorithm also achieves equal or a higher visual quality compared with SZ for the same absolute errors. Regarding compression/decompression speeds, our compression algorithm is $8\times \sim 19\times$ slower than SZ and zfp.

Our compression algorithm has difficulties compressing data with an absolute error less than 0.1 due to the performance of the convolutional AE. We suggest trying out a Wasserstein Autoencoder, presented in Liu et. al [43], to test if errors of low threshold can be reduced. Moreover, instead of using ReLU as activation function, generalized divisive normalization (GDN) could be used since they have been proven to help the model converge faster in comparison to ReLU [53] for compression purposes. Another approach to improve the convolutional AE's reconstruction is to use a pair of de-/convolutional layers for one upsampling/downsampling operation [32]. Instead of upsampling/downsampling the data consecutively, a de-/convolutional layer (with stride 1) is added before the upsampling/downsampling convolutional layer. To improve the processing of residuals other coders like Huffman coding [46] could be tested as well as storing the residuals differently for high absolute errors.

List of Figures

List of Tables

Bibliography

[1] P. Dueben and P. Bauer. "Challenges and design choices for global weather and climate models based on machine learning". In: *Geoscientific Model Development* 11 (Oct. 2018), pp. 3999–4009. DOI: https://doi.org/10.5194/gmd-11-3999-2018.

[2] N. Hübbe, A. Wegener, J. M. Kunkel, Y. Ling, and T. Ludwig. "Evaluating Lossy Compression on Climate Data". In: *Supercomputing*. Ed. by J. M. Kunkel, T. Ludwig, and H. W. Meuer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 343–356. ISBN: 978-3-642-38750-0.

[3] M. Asch, T. Moore, R. Badia, et al. "Big data and extreme-scale computing: Pathways to Convergence-Toward a shaping strategy for a future software and data ecosystem for scientific inquiry". In: *International Journal of High Performance Computing Applications* 32 (4 July 2018), pp. 435–479. DOI: https://doi.org/10.1177/1094342018778123.

[4] K. Sayood. *Introduction to data compression*. Elsevier, 2005. ISBN: 978-0-126-20862-7.

[5] P. Lindstrom and M. Isenburg. "Fast and Efficient Compression of Floating-Point Data". In: *IEEE transactions on visualization and computer graphics* 12 (Sept. 2006), pp. 1245–50. DOI: https://doi.org/10.1109/TVCG.2006.143.

[6] U. Jayasankar, V. Thirumal, and D. Ponnurangam. "A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications". In: *Journal of King Saud University - Computer and Information Sciences* 33.2 (2021), pp. 119–140. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2018.05.006.

[7] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, Cambridge, 2016. ISBN: 978-0-262-03561-3.

[8] P. Dueben et al. *Machine Learning at ECMWF: A roadmap for the next 10 years*. Jan. 2021. URL: https://www.ecmwf.int/sites/default/files/elibrary/2021/19877-machine-learning-ecmwf-roadmap-next-10-years.pdf.

[9] C. K. Sønderby, L. Espeholt, J. Heek, M. Dehghani, A. Oliver, T. Salimans, S. Agrawal, J. Hickey, and N. Kalchbrenner. *MetNet: A Neural Weather Model for Precipitation Forecasting*. 2020. arXiv: 2003.12140 [cs.LG].

[10] J. Hwang, P. Orenstein, J. Cohen, K. Pfeiffer, and L. Mackey. *Improving Subseasonal Forecasting in the Western U.S. with Machine Learning*. 2019. arXiv: 1809.07394 [stat.AP].

[11] F. Falasca, J. Crétat, P. Braconnot, and A. Bracco. "Spatiotemporal complexity and time-dependent networks in sea surface temperature from mid- to late Holocene". In: *The European Physical Journal Plus* 135 (5 May 2020), p. 392. DOI: https://doi.org/10.1140/epjp/s13360-020-00403-x.

[12] M. Taillardat and O. Mestre. "From research to applications – examples of operational ensemble post-processing in France using machine learning". In: *Nonlinear Processes in Geophysics* 27 (2 May 2020), pp. 329–347. DOI: https://doi.org/10.5194/npg-27-329-2020.

[13] *NetCDF*. URL: https://www.unidata.ucar.edu/software/netcdf/ (visited on 06/2021).

[14] U. Çayoğlu. "Compression Methods for Structured Floating-Point Data and their Application in Climate Research". dissertation. Karlsruhe Institute of Technology, 2019. URL: https://publikationen.bibliothek.kit.edu/1000105055.

[15] A. Glaws, R. King, and M. Sprague. "Deep learning for in situ data compression of large turbulent flow simulations". In: *Phys. Rev. Fluids* 5 (11 Nov. 2020), p. 114602. DOI: 10.1103/PhysRevFluids.5.114602. URL: https://link.aps.org/doi/10.1103/PhysRevFluids.5.114602.

[16] *bzip2*. Sept. 2018. URL: http://www.bzip.org.

[17] *bz2-Support for bzip2 compression*. May 2021. URL: https://docs.python.org/3/library/bz2.html.

[18] *fpzip: Compressed Large Multidimensional Floating-Point Arrays*. URL: https://computing.llnl.gov/projects/fpzip.

[19] *zfp 0.5.5 documentation*. URL: https://zfp.readthedocs.io/en/release0.5.5/introduction.html.

[20] *SZ Lossy Compression*. URL: https://szcompressor.org.

[21] T. Lu, Q. Liu, X. He, H. Luo, E. Suchyta, J. Choi, N. Podhorszki, S. Klasky, M. Wolf, T. Liu, and Z. Qiao. "Understanding and modeling lossy compression schemes on HPC scientific data". In: *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium, IPDPS 2018*. Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium, IPDPS 2018. United States: Institute of Electrical and Electronics Engineers Inc., Aug. 2018, pp. 348–357. ISBN: 9781538643686. DOI: 10.1109/IPDPS.2018.00044.

[22] G. Aurélien. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow : Concepts, Tools, and Techniques to Build Intelligent Systems.* Vol. Second edition. O'Reilly Media, 2019. ISBN: 9781492032649. URL: http://www.redi-bw.de/db/ebsco.php/search.ebscohost.com/login.aspx%3fdirect%3dtrue%26db%3dnlebk%26AN%3d2245240%26site%3dehost-live.

[23] K. He, X. Zhang, S. Ren, and J. Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[24] J. Vojt. "Deep neural networks and their implementation". MA thesis. Charles University in Prague, 2016.

[25] R. Reed and R. J. MarksII. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press, 1999.

[26] V. Nair and G. E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: ICML'10. Omnipress, 2010, pp. 807–814. ISBN: 9781605589077.

[27] X. Glorot, A. Bordes, and Y. Bengio. "Deep Sparse Rectifier Neural Networks." In: *AISTATS*. Ed. by G. J. Gordon, D. B. Dunson, and M. Dudík. Vol. 15. JMLR Proceedings. JMLR.org, 2011, pp. 315–323.

[28] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. *A Comprehensive Survey on Graph Neural Networks*. 2019. URL: http://arxiv.org/abs/1901.00596.

[29] H. Wang, A. Cruz-Roa, A. Basavanhally, H. Gilmore, N. Shih, M. Feldman, J. Tomaszewski, F. González, and A. Madabhushi. "Mitosis detection in breast cancer pathology images by combining handcrafted and convolutional neural network features". In: *Journal of Medical Imaging* 1 (Dec. 2014), pp. 1–8. DOI: 10.1117/1.JMI.1.3.034003.

[30] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi. "A Survey of the Recent Architectures of Deep Convolutional Neural Networks". In: *CoRR* abs/1901.06032 (2019). arXiv: 1901.06032. URL: http://arxiv.org/abs/1901.06032.

[31] S. University. *Convolutional Neural Networks (CNNs / ConvNets)*. URL: https://cs231n.github.io/convolutional-networks/.

[32] Z. Cheng, H. Sun, M. Takeuchi, and J. Katto. *Performance Comparison of Convolutional AutoEncoders, Generative Adversarial Networks and Super-Resolution for Image Compression*. 2018. arXiv: 1807.00270 [eess.IV].

[33] F. Chollet. *Building Autoencoders in Keras*. May 14, 2016. URL: https://blog.keras.io/building-autoencoders-in-keras.html.

[34] X. Guo, X. Liu, E. Zhu, and J. Yin. "Deep Clustering with Convolutional Autoencoders". In: Oct. 2017, pp. 373–382. ISBN: 978-3-319-70095-3. DOI: 10.1007/978-3-319-70096-0_39.

[35] N. Climate.gov. *Climate Models*. URL: https://www.climate.gov/maps-data/primer/climate-models.

[36] *Climate datasets*. URL: https://climate.copernicus.eu/climate-datasets.

[37] *European Centre for Medium-Range Weather Forecasts*. 2021. URL: https://www.ecmwf.int.

[38] *The family of ERA5 datasets*. https://confluence.ecmwf.int/display/CKB/The+family+of+ERA5+datasets. Accessed: 2021-05-21.

[39] *ERA5 hourly data on pressure levels from 1979 to present*. May 21, 2021. URL: https://cds.climate.copernicus.eu/cdsapp#!/dataset/reanalysis-era5-pressure-levels?tab=overview.

[40] K. Rogers. *Scientific modeling*. URL: https://www.britannica.com/science/scientific-modeling.

[41] A. Buetti-Dinh, V. Galli, S. Bellenberg, O. Ilie, M. Herold, S. Christel, M. Boretska, I. V. Pivkin, P. Wilmes, W. Sand, M. Vera, and M. Dopson. "Deep neural networks outperform human expert's capacity in characterizing bioleaching bacterial biofilm composition". In: *Biotechnology Reports* 22 (2019), e00321. ISSN: 2215-017X. DOI: https://doi.org/10.1016/j.btre.2019.e00321. URL: https://www.sciencedirect.com/science/article/pii/S2215017X18301954.

[42] T. Liu, J. Wang, Q. Liu, S. Alibhai, T. Lu, and X. He. "High-Ratio Lossy Compression: Exploring the Autoencoder to Compress Scientific Data". In: *IEEE Transactions on Big Data* (2021), pp. 1–1. DOI: 10.1109/TBDATA.2021.3066151.

[43] J. Liu, S. Di, K. Zhao, S. Jin, D. Tao, X. Liang, Z. Chen, and F. Cappello. *Exploring Autoencoder-Based Error-Bounded Compression for Scientific Data*. 2021. arXiv: 2105.11730 [cs.LG].

[44] S. Kolouri, P. E. Pope, C. E. Martin, and G. K. Rohde. *Sliced-Wasserstein Autoencoder: An Embarrassingly Simple Generative Model*. 2018. arXiv: 1804.01947 [cs.LG].

[45] D. Tao, S. Di, Z. Chen, and F. Cappello. "Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2017). DOI: 10.1109/ipdps.2017.115. URL: http://dx.doi.org/10.1109/IPDPS.2017.115.

[46] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[47] Y. Collet. *Zstandard*. 2016. URL: https://facebook.github.io/zstd/#other-languages (visited on 06/2021).

[48] Y. Pan, F. Zhu, T. Gao, and H. Yu. "Adaptive Deep Learning based Time-Varying Volume Compression". In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 1187–1194. DOI: 10.1109/BigData47090.2019.9006146.

[49] B. C. Mummadisetty, A. Puri, E. Sharifahmadian, and S. Latifi. "Lossless Compression of Climate Data". In: *Progress in Systems Engineering*. Ed. by H. Selvaraj, D. Zydek, and G. Chmaj. Cham: Springer International Publishing, 2015, pp. 391–400. ISBN: 978-3-319-08422-0.

[50] J. A. Saenz, N. Lubbers, and N. M. Urban. "Dimensionality-Reduction of Climate Data using Deep Autoencoders". In: *arXiv e-prints*, arXiv:1809.00027 (Aug. 2018), arXiv:1809.00027.

[51] J. Choi, M. Churchill, Q. Gong, S.-H. Ku, J. Lee, A. Rangarajan, S. Ranka, D. Pugmire, C. Chang, and S. Klasky. "Neural data compression for physics plasma simulation". In: *Neural Compression: From Information Theory to Applications – Workshop @ ICLR 2021*. 2021. URL: https://openreview.net/forum?id=eEp5uad8bM.

[52] *GNU GZIP*. Aug. 9, 2020. URL: https://www.gnu.org/software/gzip/.

[53] J. Ballé, V. Laparra, and E. P. Simoncelli. "End-to-end Optimized Image Compression". In: *CoRR* abs/1611.01704 (2016). arXiv: 1611.01704. URL: http://arxiv.org/abs/1611.01704.

[54] L. Theis, W. Shi, A. Cunningham, and F. Huszár. *Lossy Image Compression with Compressive Autoencoders*. 2017. arXiv: 1703.00395 [stat.ML].

[55] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang. *Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network*. 2016. arXiv: 1609.05158 [cs.CV].

[56] M. Li, W. Zuo, S. Gu, J. You, and D. Zhang. *Learning Content-Weighted Deep Image Compression*. 2019. arXiv: 1904.00664 [cs.CV].

[57] F. Mentzer, E. Agustsson, M. Tschannen, R. Timofte, and L. V. Gool. *Conditional Probability Models for Deep Image Compression*. 2019. arXiv: 1801.04260 [cs.CV].

[58] *JPEG 1*. URL: https://jpeg.org/jpeg/index.html (visited on 06/2021).

[59] *JPEG 2000*. URL: https://jpeg.org/jpeg2000/index.html (visited on 06/2021).

[60] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[61] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Representations by Back-propagating Errors". In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: http://www.nature.com/articles/323533a0.

[62] E. Agustsson, M. Tschannen, F. Mentzer, R. Timofte, and L. V. Gool. *Generative Adversarial Networks for Extreme Learned Image Compression*. 2019. arXiv: 1804.02958 [cs.CV].

[63] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa. *DeepZip: Lossless Data Compression using Recurrent Neural Networks*. 2018. arXiv: 1811.08162 [cs.CL].

[64] B. F. *Lossless Data Compression with Neural Networks Long Short-Term Memory Model*. 2019. URL: https://bellard.org/nncp/nncp.pdf.

[65] A. H. Baker, D. M. Hammerling, S. A. Mickelson, H. Xu, M. B. Stolpe, P. Naveau, B. Sanderson, I. Ebert-Uphoff, S. Samarasinghe, F. De Simone, F. Carbone, C. N. Gencarelli, J. M. Dennis, J. E. Kay, and P. Lindstrom. "Evaluating lossy data compression on climate simulation data within a large ensemble". In: *Geoscientific Model Development* 9.12 (2016), pp. 4381–4403. DOI: 10.5194/gmd-9-4381-2016. URL: https://gmd.copernicus.org/articles/9/4381/2016/.

[66] D. Hammerling, A. Baker, A. Pinard, and P. Lindstrom. "A Collaborative Effort to Improve Lossy Compression Methods for Climate Data". In: Nov. 2019. DOI: 10.1109/DRBSD-549595.2019.00008.

[67] X. Huang, Y. Ni, D. Chen, S. Liu, H. Fu, and G. Yang. "Czip: A Fast Lossless Compression Algorithm for Climate Data". In: *Int. J. Parallel Program.* 44.6 (2016), pp. 1248–1267. DOI: 10.1007/s10766-016-0403-z. URL: https://doi.org/10.1007/s10766-016-0403-z.

[68] U. Cayoglu, F. Tristram, J. Meyer, T. Kerzenmacher, P. Braesicke, and A. Streit. "Concept and Analysis of Information Spaces to improve Prediction-Based Compression". In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 3392–3401. DOI: 10.1109/BigData.2018.8622313.

[69] U. Cayoglu, F. Tristram, J. Meyer, J. Schröter, T. Kerzenmacher, P. Braesicke, and A. Streit. "Data Encoding in Lossless Prediction-Based Compression Algorithms". In: *2019 15th International Conference on eScience (eScience)*. 2019, pp. 226–234. DOI: 10.1109/eScience.2019.00032.

[70] U. Cayoglu, J. Schröter, J. Meyer, A. Streit, and P. Braesicke. "A Modular Software Framework for Compression of Structured Climate Data". In: *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPATIAL '18. Seattle, Washington: Association for Computing Machinery, 2018, pp. 556–559. ISBN: 9781450358897. DOI: 10.1145/3274895.3274897. URL: https://doi.org/10.1145/3274895.3274897.

[71] S. Di and F. Cappello. "Fast Error-Bounded Lossy HPC Data Compression with SZ". In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016, pp. 730–739. DOI: 10.1109/IPDPS.2016.11.

[72] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello. "Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets". In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 438–447. DOI: 10.1109/BigData.2018.8622520.

[73] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak. "Out-of-core compression and decompression of large n-dimensional scalar fields". In: *Computer Graphics Forum* (2003). ISSN: 1467-8659. DOI: 10.1111/1467-8659.00681.

[74] P. Lindstrom. "Fixed-Rate Compressed Floating-Point Arrays". In: *IEEE Transactions on Visualization and Computer Graphics* 20 (Aug. 2014). DOI: 10.1109/TVCG.2014.2346458.

[75] S. Bhatnagar, Y. Afshar, S. Pan, K. Duraisamy, and S. Kaushik. "Prediction of aerodynamic flow fields using convolutional neural networks". In: *Computational Mechanics* 64.2 (June 2019), pp. 525–545. ISSN: 1432-0924. DOI: 10.1007/s00466-019-01740-0. URL: http://dx.doi.org/10.1007/s00466-019-01740-0.

[76] E. Kreyszig, H. Kreyszig, and E. J. Norminton. *Advanced Engineering Mathematics*. Tenth. Hoboken, NJ: Wiley, 2011. ISBN: 0470458364.

[77] M. Kloewer, M. Razinger, J. Dominguez, et al. *Compressing atmospheric data into its real information*. June 2021. DOI: https://doi.org/10.21203/rs.3.rs-590601/v1.

[78] J. Lockerman and A. Kulkarni. *Time-series compression algorithms, explained*. Apr. 2020. URL: https://blog.timescale.com/blog/time-series-compression-algorithms-explained/.

[79] T. Duy Trác. *Sparse Signal Processing*. URL: https://slidetodoc.com/sparse-signal-processing-trn-duy-trc-ece-department/f.

[80] *zlib — Compression compatible with gzip*. July 2021. URL: https://docs.python.org/3/library/zlib.html.

[81] *lzma*. URL: https://docs.python.org/3.6/library/lzma.html.

[82] *Keras*. URL: https://keras.io/.

[83] *TensorFlow*. URL: https://www.tensorflow.org/.

[84] *bz2 — Support for bzip2 compression*. URL: https://docs.python.org/3/library/bz2.html.

[85] *BwUniCluster 2.0*. June 2021. URL: https://wiki.bwhpc.de/e/Category:BwUniCluster_2.0.

[86] M. Götz. *HDFML Execution Environment Reproducibility Report*. URL: https://b2share.eudat.eu/records/cb02356f7dc5445cbf0952324f23d90c.

[87] *Forschungszentrum Jülich*. URL: https://www.fz-juelich.de/portal/EN/Home/home_node.html.

[88] S. Trojniak. *Sea-, Lake-, and Land- Breezes: How the Temperature Difference Between Water and Land Influences Weather Along the Coastline*. Aug. 2018. URL: https://www.globalweatherclimatecenter.com/weather-education/sea-lake-and-land-breezes-how-the-temperature-difference-between-water-and-land-influences-weather-along-the-coastline-photo-credit-north-carolina-climate-office.

[89] *ERA5 hourly data on single levels from 1979 to present*. July 14, 2021. URL: https://cds.climate.copernicus.eu/cdsapp#!/dataset/reanalysis-era5-single-levels?tab=overview.

[90] J. Bergstra and Y. Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305. URL: http://jmlr.org/papers/v13/bergstra12a.html.

[91] A. Pak. *Image Data Compression - Introduction to Coding*. URL: https://ies.anthropomatik.kit.edu/ies/download/lehre/idc/IDC-02-Coding-Intro.pdf.