Master Thesis

Data-Aware Compression for HPC using Machine Learning

vorgelegt von

Julius Plehn

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	M. Sc. Informatik
Matrikelnummer:	6535163
Erstgutachter:	Jun.-Prof. Dr. Michael Kuhn
Zweitgutachter:	Prof. Dr. Thomas Ludwig
Betreuer:	Jun.-Prof. Dr. Michael Kuhn, Anna Fuchs, Dr. Jakob Luettgau

Hamburg, 09. Mai 2022

Abstract

While compression can provide significant storage and cost savings, its use within HPC applications is often only of secondary concern. This is in part due to the inflexibility of existing approaches where a single compression algorithm has to be used throughout the whole application, but also because insights into the behavior of the algorithms within the context of individual applications are missing.

Several compression algorithms are available, with each one also having a unique set of options. These options have a direct influence on the achieved performance and compression results. Furthermore, the algorithms and options to use for a given dataset are highly dependent on the characteristics of said dataset.

This thesis explores how machine learning can help identify fitting compression algorithms with corresponding options based on actual data structure encountered during I/O. To do so, a data collection and training pipeline is introduced. Inferencing is performed during regular application runs and shows promising results. Moreover, it provides valuable insights into the benefits of using certain compression algorithms and options for specific data.

Contents

1 Introduction

High-performance computing (HPC) is used to distribute computing tasks that surpass the capabilities of a single computing node across a multitude of nodes. Similar to the considerations required for constructing a single node, those challenges are extended across a huge number of additional components. Besides the performance implications for an application when running in a non-distributed environment, the performance characteristics are influenced by shared and mutually used resources like the network and storage devices. Throughout the long history of vertical scaling architectures, many optimizations were introduced. Nowadays, researchers and system architects strive to develop new exascale supercomputers, aiming to solve new tasks faster and with greater accuracy. Advancements in this area are essential as HPC contributes significantly to scientific and industrial innovation. Among others, as noted in [1], famous use-cases include experiments at CERN and the Intergovernmental Panel on Climate Change (IPCC).

All those efforts have in common that HPC concepts are subject to ongoing changes and improvements. However, those essential building blocks of dependent components like CPUs, memory, networking, and storage are evolving at different rates. This is especially notable in storage related technology and concerns storage capacity, input/output operations per second and throughput. Furthermore, those components and requirements are strongly interleaved, resulting in advancements that must be propagated throughout the whole distributed stack to fully reach its potential. Recent examples include more practical use of machine learning approaches, which in many cases rely on random data access patterns and impose unique constraints on the storage architecture.

Besides these aspects, storage capacity is of importance. According to [2], only a small percentage of generated data is stored for post-processing. This is partly due to storage capacity but also influenced by the cost associated with capacity. To mitigate those issues, the amount of data that physically needs to be stored has to be reduced. Promising concepts include in-situ processing like evaluated in [3]. With this approach, the data is evaluated and processed as the computation occurs. Therefore only the final data of the simulation needs to be stored, and intermediate data is discarded. However, this requires thorough application changes and might not be applicable to all use-cases.

An alternative is to use data compression techniques. This approach reduces the data by exploiting redundancy to store a shorter representation of the initial file. As discussed in [4], this not only reduces the required provisioned storage capacity but can also result in higher throughput, lower latencies, and increased memory capacity.

1.1 Motivation

As indicated previously, the potential benefits of using compression in HPC are significant. However, a vast amount of compression algorithms are available, each comprising a multitude of options. Furthermore, depending on the use case, different characteristics of the available compression algorithms are important. The correct choice also depends on the requirements imposed by the application and the researcher. As argued in [5], this choice is a technical decision, and a compressor selected specifically for one dataset might not achieve appropriate results for a different dataset.

This manifests in the lack of use of compression at all or results in the integration of static compression variants which do not unleash the full possible potential. To advance the use of compression in general and to enable users to make well-informed decisions for proper usage, further developments are necessary. Ideally, this should be an automated process where the ideal compression algorithm and options are selected without further intervention by the user, depending on the user's imposed requirements. This motivates the contributions of this thesis, where a system is proposed which predicts the ideal compression algorithm and options in a data-aware manner.

1.2 Thesis goals

Motivated by the introduction above, the thesis goals are outlined in the following. First, further insights into the behavior of compression algorithms and options when applied to application data are needed. Depending on user-defined metrics like compression rate and throughput, the benefits and characteristics of compression techniques need to be highlighted and visualized. This is the basis and the fundamental building block of providing capabilities of identifying ideal compression variants for data encountered during I/O of HPC applications. These insights can then be used to automatically provide training data for a machine learning approach where the association between the data and ideal compression techniques are automatically explored without user intervention. The final goal is to use the learned characteristics of compressible data to predict the ideal compressor during runtime for the data encountered during I/O.

1.3 Outline

The remainder of the thesis is organized as follows:

- In the **Background** chapter, essential information about compression algorithms and machine learning is given. Furthermore, details regarding I/O in HPC and how applications interact with file systems are presented.
- The **Related Work** chapter provides insights into existing approaches that aim to optimize compression usage by either exploiting domain knowledge or relying on

strategies that use application-specific constraints. Additionally, further machine learning related techniques are introduced.

- In the **Architecture** chapter, the development-related contributions of the thesis are highlighted. This includes a library that intercepts I/O calls and can predict compressors for the intercepted data and tools that allow for the training of machine learning models.
- In the **Evaluation** chapter, an HPC application is used to train multiple neural networks. Those networks are then evaluated by testing them in several experiments.
- Afterward, ideas to improve the given architecture and prospects for **Future Work** are given.
- In the end, the overall achievements are highlighted in the **Conclusion**.

2 Background

In this chapter, background information on the involved components of this thesis is given. First of all, I/O in the context of HPC is introduced. This includes filesystems, I/O libraries, and layers which data is passed through. An introduction of compression algorithms follows this, and several implementations are shown in detail. Important distinctions are outlined as these are of importance for further data-awareness in the remainder. Finally, relevant topics regarding machine learning are presented.

2.1 I/O in HPC

The performance of I/O has a significant impact on the overall performance of HPC applications. Similar to the techniques implemented in applications, parallelization is a big concern. This requirement is reflected throughout all components involved and comes with its challenges. This is partly due to technical constraints of the underlying architecture but also includes the complexity associated with the implementation of parallel applications and their interaction with each other. The influence this has is shown in [6], where application runs were analyzed. The authors observed that the majority of jobs did not perform efficient I/O, and a significant time of the total run time was spent doing I/O. In the following, background about the involved components is given. Furthermore, libraries that are commonly used as an abstraction for I/O are introduced.

2.1.1 MPI-IO

MPI-IO is part of the Message Passing Interface (MPI) and relates to the possibility of extending the existing process communication interface to be able to perform I/O in parallel. Advanced interfaces are required for this use case as the Portable Operating System Interface (POSIX) imposes severe restrictions like files that are created as a contiguous stream of bytes. Instead, MPI-IO enables file access using natively provided MPI data types and derived types. Additionally, several consistency constraints are relaxed. Besides those aspects, parallelization techniques known from MPI are used. This includes non-blocking and blocking access, as well as collective operations.

While MPI-IO allows for low-level and individual interaction with files, a common approach is to create a logical view for each MPI rank. A view is created by assigning a data type, a file type, and a displacement. The data type can either represent a single

MPI type or be constructed from others. Furthermore, the file type represents the actual access pattern which is used afterward to interact with the physical view. For example, a continuous pattern can be chosen to write data according to the order of the MPI ranks.

The implementation of MPI-IO is in many cases provided by ROMIO[1]. Furthermore, an additional abstraction layer called ADIO is responsible for providing parallel I/O access to the individual file systems. More information can be found in [7], where the initial concept is proposed. Furthermore, thorough details about the Lustre[2] ADIO driver can be found in [8].

MPI-IO provides several possibilities to enable parallel I/O and uses an API familiar to MPI users. Furthermore, commonly used file systems are supported, and due to the popularity of MPI, this approach is used in various other libraries.

2.1.2 HDF5

HDF5[3] is the acronym of the Hierarchical Data Format in the fifth version. It is a file format that is widely used within scientific applications. Files created according to this format can be handled independently of the compute environment it was initially created in, as endianness and data types are handled accordingly. Another benefit of the use of this standardized file format is the support provided by external applications.

HDF5 provides an API that is used to set up the file structure according to the application's requirements and to perform I/O. Therefore, this API provides an abstraction to more complex interfaces like MPI-IO, which come with their own challenges. If the application implementing HDF5 already uses MPI for process communication, the adaption of parallel I/O is therefore straightforward.

Since version four of NetCDF (Network Common Data Form)[4], it is possible to use HDF5 as the underlying data format. Comparable to HDF5, NetCDF provides a self-describing data format and an API for various programming languages. It is maintained by the University Corporation for Atmospheric Research (UCAR), and it is mainly intended to be used for earth-related science. In order to promote a standard for this research field, NetCDF is endorsed by research groups like the NASA Earth Science Data Systems (ESDS) group [9]. Furthermore, the CF metadata conventions[5] have been created to provide a unique description for metadata across climate and forecast research.

[1] `https://wordpress.cels.anl.gov/romio`
[2] `https://www.lustre.org`
[3] `https://www.hdfgroup.org/solutions/hdf5`
[4] `https://www.unidata.ucar.edu/software/netcdf`
[5] `http://cfconventions.org`

2.2 Compression

This section gives an introduction to compression in general. In order to do so, some related concepts of information theory are provided. This is followed by several commonly used techniques and compression algorithms used in conjunction with more complex ones. Afterward, the main compression algorithms, which are used throughout this thesis, are introduced. Finally, metrics that are used to compare the performance of compression are defined.

2.2.1 Information theory

Information theory is a combination of statistics and probability studies with the purpose of providing a formal approach to information transmission. Important research subjects include the definition of what qualifies as information in the context of transmitting said content. Further topics include error correction and data compression. Those theories are the basic building blocks of compression algorithms used today, and in the following, entropy and entropy coding are introduced.

Entropy

Entropy denotes the weighted average amount of information content of a given variable and was introduced by Claude Shannon in [10]. The information content is defined as

$$I(x) = \log_b(\frac{1}{\mathrm{P}(x)}) = \log_b(1) - \log_b(\mathrm{P}(x)) = -\log_b(\mathrm{P}(x)) \tag{2.1}$$

where the logarithmic base b is the size of the alphabet and $\mathrm{P}(x)$ is the probability of x. In case of the use of bits, a single variable can carry the information about two states and therefore $b = 2$. Because of the logarithm, the information content is high when the probability of a variable is low. On the contrary, the information content is low when the probability is high. This follows the intuition that when the occurrence of information is very unlikely, it is more relevant than information that occurs regularly. Another property of this is that this specifies the minimal number of bits that are required to encode this information. This aspect can be seen in Figure 2.1. The highlighted point indicates a probability of 0.25 for a specific state. Therefore, 2 bits are required to transport this information.

The entropy is defined as:

$$H(X) = \sum_{i=1}^{n} \mathrm{P}(x_i)\mathrm{I}(x_i) = -\sum_{i=1}^{n} \mathrm{P}(x_i)\log_b \mathrm{P}(x_i) \tag{2.2}$$

The highest entropy is therefore achieved when the probability of every state is the same. This indicates the maximal number of bits required to transport this information. For example when $P_A, P_B, P_C = 0.\overline{3}$ then $H = -(3 \cdot (0.\overline{3} \cdot log_2(0.\overline{3})) = 1.58$. However, when

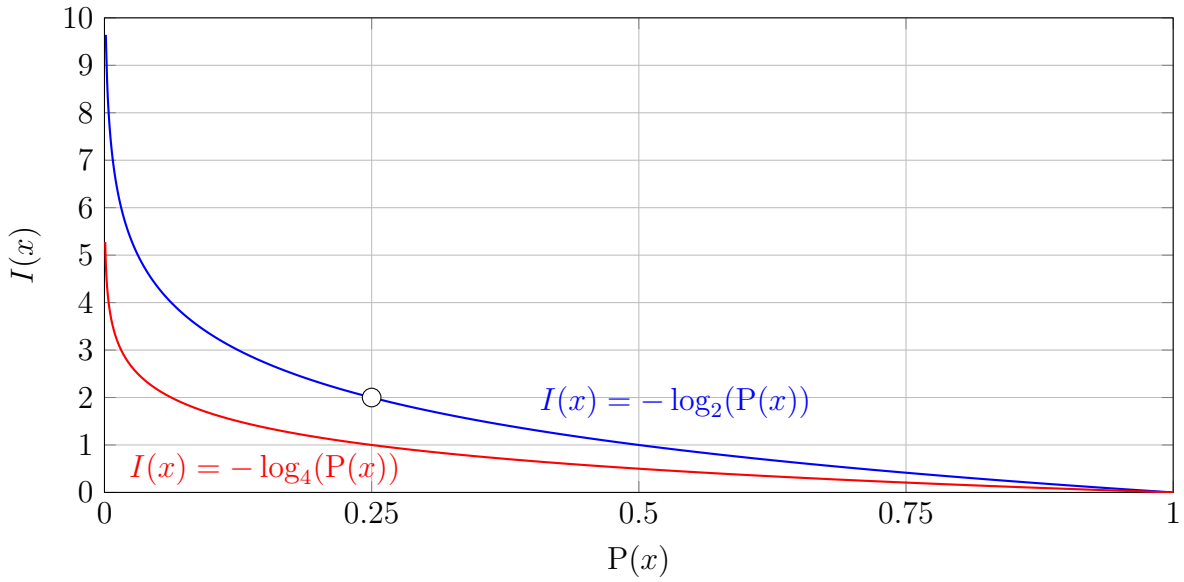


Figure 2.1: Comparison of the information content when using two different logarithmic bases.

the probability is unbalanced e.g. $P_A = 0.8, P_B, P_C = 0.1$ then $H = -(0.8 \cdot log_2(0.8) + 2 \cdot (0.1 \cdot log_2(0.1))) = 0.92$.

Entropy coding

Entropy coding encodes symbols that occur with high probability, with a shorter bit representation than rare symbols. Therefore this is a form of data compression. In addition, the entropy determines the theoretical optimal encoding of a symbol. As shown previously, there are cases where less than 1 bit is required for encoding according to the entropy. Depending on the use case, encoders are used to encode the representation according to the nearest bit, or they rely on approaches capable of achieving a denser representation. Common approaches include the use of Huffman coding and arithmetic coding, respectively. Those are introduced in the following.

Huffman coding The concept of Huffman coding was first described by David A. Huffman in 1952 in [11]. This coding variant closely follows the interpretation of entropy coding, and prominent symbols are encoded using a short representation. Ideally, the short representation follows the information content defined previously.

Huffman coding works by generating a binary tree. As initial input, the symbols to encode and their probabilities are required. They are all added as leaf nodes. The algorithm then takes the two symbols with the smallest probability, combines them by creating a new parent node, and assigns this new node their combined probability. This process is then repeated, and the nodes with the lowest probabilities are combined. To

retrieve the actual encoding, the tree can be traversed. Beginning with the root node, nodes to the left can be labeled as zero, and nodes to the right can be labeled as one. This results in a prefix-free code, in which no decoded symbol is the prefix of any other decoded symbol.

Arithmetic coding As previously described, arithmetic coding achieves compression rates closer to the limits imposed by entropy than the Huffman coding variant. The concept has first been outlined by Jorma Rissanen in [12]. This approach is using a rational number to represent information according to a specific interval. This number q is often chosen in an interval within $0.0 \leq q < 1.0$. Furthermore, a dedicated encoder and decoder function is used, and both have to use the same interval. Given this interval and input of symbols and probabilities, the compression and decompression process works as shown in the following.

The encoder uses the given interval and divides it into subintervals. Each subinterval belongs to a single type of symbol, and the size is chosen according to the known probability. Further subintervals are created for each symbol in the input sequence, and layers of those intervals are created in each step. It is important to note that the start and end of a new interval depend on the probabilities of the previous symbols. The final step is to store a single and short as possible real number from within the last subinterval.

The decoder then uses this number, and provided that the initial interval is the same, no additional data has to be transmitted. Another prerequisite is that knowledge about the available symbols and their probabilities has to be provided for both components. Decoding behaves very similarly to the initial encoding. Once again, subintervals are created. In each layer, the same transmitted real number is used to identify the subinterval it belongs to. The initial data is decoded as each subinterval stands for a specific symbol. This technique is repeated, and the identified subinterval is the starting point for the following subdivision. In [13] and [14], the authors provide much more detail and a reference implementation in C.

Comparison Huffman coding and arithmetic coding each have specific advantages and disadvantages. According to [15], arithmetic coding achieves greater compression rates at the cost of longer runtime, which is expected.

2.2.2 Techniques

In the following, two significant compression techniques are introduced. Both are thoroughly used within the compression algorithms evaluated later on.

Dictionary						Look a-head				Output
1	2	3	4	5	6	1	2	3	4	
						A	B	A	B	(0, 0, A)
					A	B	A	B	C	(0, 0, B)
				A	B	A	B	C	D	(5, 2, C)
	A	B	A	B	C	D				(0, 0, D)
A	B	A	B	C	D					

Table 2.1: LZ77 example showing the interaction with the sliding window when applying the message `ABABCD`. Based on [17]

LZ77

LZ77 has been created by Abraham Lempel and Jacob Ziv in 1977 [16]. In this approach, repetitive combinations of multiple symbols are compressed, and this algorithm is therefore not limited to the reduction of single symbols, as shown earlier. LZ77 maintains a so-called sliding window containing a dictionary of already processed data and a look-ahead buffer of data that still has to be compressed. In the beginning, the dictionary is empty, and no prior knowledge can be exploited to find redundancy in the first symbol. Then, as the dictionary fills with data from the look-ahead buffer, the biggest redundancy of data found in both of them is searched. After each iteration, the sliding window is shifted so that new data enters the look-ahead buffer. Additionally, the same amount of data leaves the dictionary. Each of those iterations results in a new triple. This contains the beginning of the match within the dictionary, the length of the match, and the first symbol which did not match from the look-ahead buffer.

An example is given in Table 2.1, where the message `ABABCD` is applied to LZ77. The dictionary size is set to six, and the size of the look-ahead buffer is set to four. In the first two iterations, no information from the dictionary contributed to compression. In the third iteration, however, the content within the dictionary is found in the first two symbols of the look look-ahead buffer. Therefore, the start index within the dictionary and the length of the match are stored. Additionally, `C` is noted as the symbol that follows. This symbol is therefore additionally added to the dictionary. Based on those triples, decompression works accordingly as they include all necessary information for reconstruction.

When relying on LZ77 as a component for data compression, several things have to be considered. For example, the sliding window is limited in size, influencing the compression performance and runtime. Furthermore, in the example given before, no prior dictionary was available. This has a noticeable impact on the efficiency of compressing small data.

Deflate

The deflate compression algorithm was designed by Phil Katz and was formally specified in 1996 [18]. Internally it first uses LZ77 and then Huffman coding. The triples that LZ77 produces still contain redundant data. It is also noteworthy that multiple symbols have to be stored for a single symbol in the previous example. In those cases, the compressed data can even increase in size. However, such data is well-suited for Huffman coding, where symbols that occur many times are stored using less data.

2.2.3 LZ4

LZ4[6] is a compression algorithm created by Yann Collet. It is geared towards high compression and decompression speeds. There are two possible usage modes. First, a specific compression API provides the user with a high compression version, aiming to provide a good compression rate at the cost of higher CPU usage and runtime. The alternative is to use a variant that accelerates the compression speed at the cost of a lesser compression rate. The former provides 22 compression levels, where lower values indicate lesser required runtime. This is contrary to the second variant where, theoretically, up to 65 537 factors can be used. Internally, LZ4 relies on LZ77, which has been introduced previously.

The compressed data is stored in a fixed format, comprising of frames, which contain the actual data in so-called blocks. Both formats are documented in great detail in [19] and [20], respectively. This enables any developer to provide compatibility beyond the reference implementation written in C. Furthermore, LZ4 support is included within the Linux kernel, and in [21], it has been shown that the boot time can be decreased by compressing the kernel image using LZ4. For this use case, the faster loading of the image into the memory outweighs the overhead of decompressing the kernel on boot time.

2.2.4 Zstandard

Zstandard (ZSTD)[7] has also been developed by Yann Collet at Facebook. Contrary to the design goals of LZ4, high compression rates while maintaining reasonable speeds were important. Unlike LZ4, ZSTD combines an LZ77 algorithm with Huffman coding and Finite State Entropy (FSE). FSE is another entropy coding approach based on a CPU-efficient variant of asymmetric numeral systems (ANS). They try to combine the performance of Huffman coding with the compression rate of arithmetic coding and are introduced in [22]. Additionally, ZSTD is available in the Linux kernel, and similar to LZ4, it can be used natively in filesystems like ZFS [23].

[6] `https://github.com/lz4/lz4`
[7] `https://github.com/facebook/zstd`

2.2.5 ZLIB

ZLIB[8] was created by Jean-loup Gailly and Mark Adler in 1995. It is thereby the oldest compressor evaluated in this thesis. Both authors are well known for their effort in developing gzip, and Mark Adler is also the creator of the Adler-32 checksum algorithm.

Data compression is achieved by relying on the deflate algorithm. Therefore, as introduced earlier, this is a combination of LZ77 and Huffman coding. Great effort is put into ensuring that compression of incompressible blocks does not result in data expansion. If no considerable compression can be achieved, the uncompressed blocks are stored. As discussed in the technical details, this "overhead approaches the limiting value of 0.03 %." [24] Again, this characteristic is similar to the other evaluated compression algorithms, and the overhead consists of mandatory header data.

ZLIB is also available in the Linux kernel and is used in many applications, including filesystems, the PNG image format, and Git.

Reasons for this compression algorithm selection

As introduced previously, three compression algorithms are currently selected for further evaluation. This includes LZ4, ZSTD, and ZLIB. While many alternatives are available, this selection represents compressors that use similar techniques and differentiate only by some details and design goals. In general, they are not explicitly created for a limited use case, and it is expected that they achieve good results for HPC applications.

Another relevant aspect of this selection is their availability in the Linux kernel. For example, in the future, it might be helpful for similar approaches to use the prediction of an arbitrary decision component and to use compression within the kernel space.

2.2.6 Metrics

Each compression algorithm has its use case and surpasses the performance of another one in specific applications. The ability to compress some data with outstanding characteristics might not translate to any other data input and often comes at a cost. Therefore, it is essential to compare and evaluate compression algorithms by some type of measurement. This is done by relying on metrics, which capture the requirements imposed by users and applications accordingly. In the remainder of this thesis, the following metrics are used.

[8] `https://zlib.net`

Compression rate

The compression rate captures the relation of the compressed input to the input data. It is defined as $\text{CR} = \frac{\text{original size}}{\text{compressed size}}$. This metric is relevant when the total data size is important. For example, use cases might include data that is very unlikely to be reread or only on rare occasions.

Compression rate per time

This metric uses the time it takes for the compressor to run and relates it to the achieved compression rate. It attempts to highlight compressors that achieve exceptional compression rates while still maintaining a fast runtime.

Compression speed

This metric represents the throughput achieved by the compressor per time. This use case is ideal in scenarios where the application's runtime is of most importance. Using modern compressors like LZ4, it is still possible to achieve significant compression rates.

Decompression speed

Similar to the previous metric but for decompression. All introduced compression algorithms only provide configuration options for compression purposes. The compressed data is then decompressed by a deterministic algorithm which previously used options like compression levels should not influence. Nevertheless, as an experiment, the data that has been compressed using various configurations while measuring one of the previous metrics is decompressed for this purpose. The intention is to see how the structure of the compressed data, which depends on said options, influences decompression speeds.

2.3 Machine learning

Machine learning is a general term for algorithms that gain knowledge to fulfill some task by learning from experience. This generic description applies to an increasing number of use cases and conceptionally applies to many different approaches. A common approach for distinguishing techniques of machine learning is to categorize them into three categories:

- *Supervised learning*: In this category, pairs of input data and known labels are provided. The subject of this approach is to learn a function that is able to predict accurate labels for new input data. This category is commonly subdivided into other types like classification and regression. While classification algorithms assign a specific category to an input, regression algorithms estimate a numerical relationship between independent and dependent variables. Depending on the task,

various modeling approaches are used, including decision trees, linear regression, and neural networks.

- *Unsupervised learning*: Contrary to the input required when using supervised learning in this category, no labels are given. Instead, the primary goal is to find structure and discover the data's properties. A widely used technique is cluster analysis, with one of the most prominent algorithms being the k-means clustering algorithm.
- *Reinforcement learning*: In this category, an agent acts in an environment and is rewarded depending on the outcome of his actions. Unlike supervised learning, no exact labeled outcome for every input has to be given. Often, there is merely a limited set of actions. It is up to the agent to balance the amount of exploration and exploitation to achieve a goal imposed by an environment.

In this thesis, supervised learning is applied. Therefore this implies the requirement for an architecture capable of collecting training data and training a classifier based on that. To achieve this, neural networks are used. More detail on how they operate is given in the following.

2.3.1 Artificial neural networks

Artificial neural networks, in the literature often called neural networks, attempt to mimic the behavior of neurons in the brain. This is done by connecting nodes, representing neurons, with each other. Each node can receive input from other connected nodes and send output depending on a function f. This function is individually modeled by dependent parameters, commonly referred to as weights. This represents their capability to "modify their internal structure in relation to a function objective." [25]

Research began in 1943 where McCulloch and Pitts in [26] presented models of biology inspired neurons which are capable of evaluating logical expressions. Further milestones include the development of the perceptron algorithm and the relation to neurons in the brain in [27] and [28]. This allows for binary classification, whereas more complex relations can be trained, requiring the use of backpropagation and multi-layer networks as proposed in [29].

In the following, the focus lies on feedforward neural networks. This is a type of network where the input is passed throughout to the output in a single direction, resembling a directed acyclic graph (DAG) [30]. The most basic network type consists of a single layer of so-called perceptrons. More complex networks with greater capabilities consist of several layers, requiring more complex techniques to enable training. Those concepts are shown in the following.

Perceptrons

In its most basic variant, when using a single-layer network, a perceptron is a type of network capable of performing binary classification. Therefore it is possible to decide between two states for a given input. This mechanism is only applicable to linearly separable data, as otherwise, the algorithm would not converge [31]. A perceptron classifies inputs by applying them to a basic threshold function to generate an output. In the following, the perceptron algorithm is shown:

$$f(x) = \begin{cases} 1, & \sum_{i=1}^{m} w_{ij} x_i + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

Each perceptron i uses its inputs x_i and its learnable weights w_{ij}, as well as a bias, to perform classification. Due to the weights associated with the inputs, a trained perceptron and a neural network in general can classify deviating inputs to a certain degree.

In Figure 2.2a, a perceptron that implements the logical OR operation is shown. This is possible as this classification task is linearly separable, as shown in Figure 2.2b. The learning algorithm updates the weights according to whether the correct output is predicted. As discussed in more detail in [31, p.84-85], the weights are initialized randomly. Afterward, a random input vector is chosen, and the perceptron is evaluated. If the output is correct, no changes to the weights are performed. Otherwise, the weights increase or decrease depending on the outcome, respectively. This represents the perceptron learning rule, which is shown in the following. For the weight of a connection between neuron i and j the updated weight is defined as:

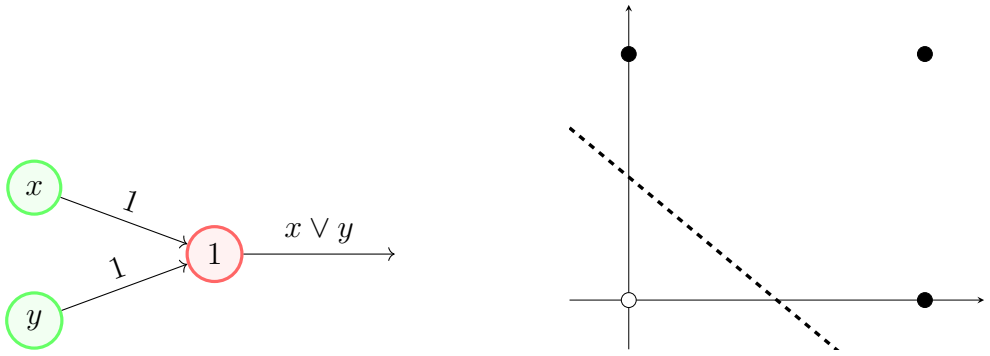
$$w_{ij} = w_{ij^{\text{old}}} + \alpha \cdot (t_j - o_j) \cdot x_i \tag{2.4}$$

In this equation, α represents a learning rate that influences the training speed. Furthermore, t_j denotes the intended output, whereas o_j represent the perceptron's output. If the data is linearly separable, this algorithm converges after several iterations.

Delta rule

The perceptron described previously can only classify input data that is linearly separable. To overcome this shortcoming, the delta rule can be used. It describes the single-layer version of an algorithm fundamental to every neural network optimization, namely the gradient descent. Instead of relying on a basic rule, a measurement of the error of the output is used. This algorithm then attempts to minimize the overall error of the network. The error is defined as shown in the following:

$$E = \sum_{j} \frac{1}{2} (t_j - y_j)^2 \tag{2.5}$$



(a) Perceptron (b) Linear separability of $x \vee y$

Figure 2.2: A perceptron which is capable of classifying the logical OR operation.

This incorporates the deviation of j observed outputs from the correct outputs t_j. The partial derivative of this error with respect to a weight results in the following derivative

$$\frac{\partial E}{\partial w_{ji}} \tag{2.6}$$

, which then can be transformed to the gradient descent update rule:

$$\Delta w_{ji} = \alpha(t_j - y_j)g'(h_j)x_i \tag{2.7}$$

Detailed insights regarding the derivation are available in various resources, e.g., in [32, 33]. This equations provides several valuable properties, as it contains the exchangeable activation function $g(h_j)$ and again the learning rate α. This allows the network to optimize towards a local minimum by following the steepest descent. The learning rate influences the significance of applied change and ideally results in faster model training. However, the gradient descent can overshoot the minimum if chosen too large, resulting in poor overall network accuracy.

2.3.2 Multi-layer perceptron

The multi-layer perceptron consists of multiple layers connected to each other. Similar to the single-layer perceptron, the output of the overall network is observed for a given input. The intermediate layers are known as hidden layers and, in many cases, do not represent a relation to a comprehensible meaning as the input and output layers do. This also requires a variant of the gradient descent used in the single-layer perceptron, which is capable of updating the weights of the hidden layers. The objective is the same as previously, and the error, as shown in Equation (2.5), is minimized.

Backpropagation

As the name indicates, given the current input, the error of the network is propagated back throughout the network. This avoids recomputation of intermediate values as each

weight depends on the part it contributes to the error of the neurons it is connected to. This is done by applying a generalized variant of the delta rule, which includes a distinction depending on whether the updated weight belongs to an output neuron or a hidden neuron.

2.3.3 Activation functions

Activation functions are used in neural networks to map the weighted input of the neuron to a new valuable value, which allows for meaningful decision-making throughout the network. As mentioned, nonlinear and differentiable activation functions are preferred instead of a threshold function like when introducing the perceptron. In Figure 2.3, a selection of current and historical functions is shown.

Sigmoid:

The sigmoid activation function is defined as $f(x) = \frac{1}{1+e^{-x}}$. However, this function is not commonly used, as it is prone to the vanishing gradient problem, as first explored in [34]. As the derivative is close to zero and due to the chain rule used during backpropagation, the updated gradient decreases as the propagation progresses. Depending on the number of hidden layers, no notable update of the weights at the beginning of the network might be noticeable.

Rectified Linear Unit (ReLU):

ReLU is defined as $f(x) = \max(0, x)$, where only the positive input of x is used. It is very popular as the derivative can be used efficiently, and it is not prone to the vanishing gradient problem. The network shown in Section 4.2.3 and evaluated in Chapter 5 uses this type of activation function.

Heaviside:

This is the activation function used in the single-layer perceptron. It can therefore be used for binary classification. However, it is not usable for backpropagation, as it is not differentiable. It is defined as:

$$f(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

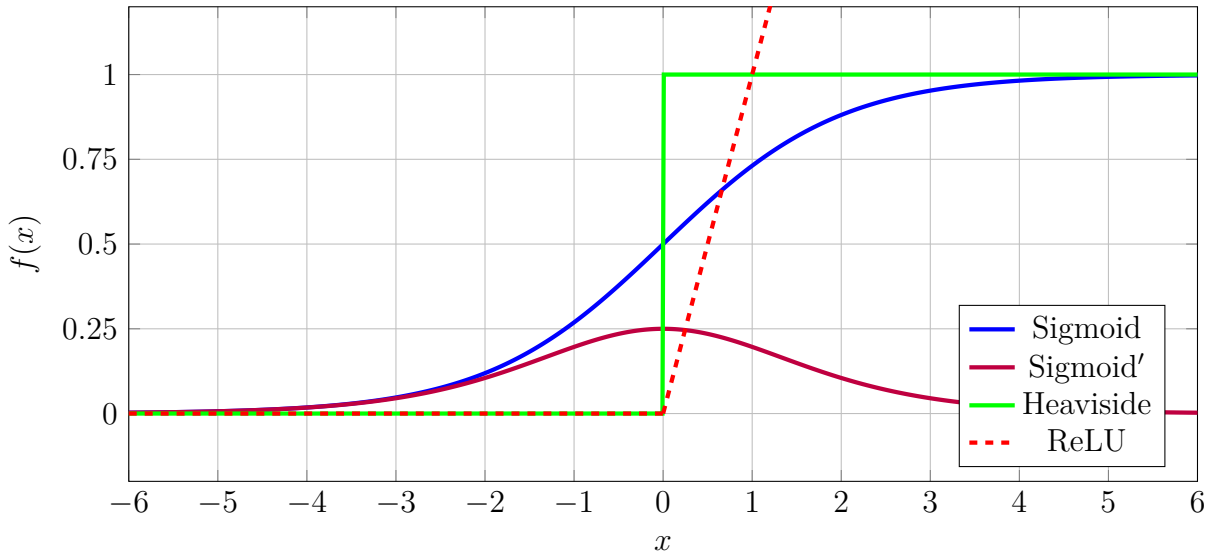


Figure 2.3: Visualization of multiple activation functions

3 Related work

This chapter gives an overview of existing approaches that achieve a similar goal of applying data compression in a data-aware manner. This can be because they rely on specific domain knowledge or by using machine learning approaches.

3.1 Data-specific compression

As the performance of compression algorithms varies with the data that is being compressed and, on some occasions, cannot be efficiently compressed at all, multiple alternatives have been proposed. For example, domain-specific compression algorithms exist, which exploit and implement the knowledge that is only relevant in this specific field. Additionally, on many occasions, not all of the provided data is required. Common examples include the use of compression mechanisms in visual and auditory applications. Both variants are relevant to this thesis as they highlight the importance of selecting the correct compression approach.

3.1.1 Domain-specific compression

A common example of domain-specific compression is the use within DNA related research. As discussed in [35], the amount of genomic data that is produced is ever increasing and the number of generated bases "doubles approximately every 18 months"[35, 36]. As DNA sequences only contain four symbols they can be encoded using 2 bit. However, existing algorithms fail to compress this data and even increase the size, as discussed in [37]. It is also noted that standard compression algorithms rely on local redundancy within a file. While this holds true for many text-related tasks, this does not apply to genomic data. What is proposed instead is the use of a combination of two compressors. For long and repetitive data, the LZ77 algorithm is used and short repeats are compressed using a Context Tree Weighting(CTW) method. This allows for sequences encoding to less than two bits per symbol.

The approach presented in [38] and further discussed in [39] exploits a law of how bases are distributed in bacterial DNA. Furthermore, in [40] the authors propose a dictionary, which is generated artificially.

3.1.2 Lossy compression

When using lossy compression, data is reduced by discarding and approximating data according to predefined error bounds. This compression variant is used throughout many applications, including image, video and audio compression. The approach here is to reduce the file size according to data quality, e.g., by discarding frequencies that the human does not perceive [41]. Another benefit of the use case of lossy compression in those specific fields is that, generally, an uncompressed or lossless variant is kept. This is useful for further processing and distributing updated versions.

In the context of HPC, further challenges arise. Lossy compression might be implemented directly when producing simulation data. Therefore, no additional data might be available afterwards. This imposes scientific requirements, which are discussed in [42]. First of all, lossy compression should not result in data that might create visual artifacts or influence the outcome of the evaluation of scientific experiments. Furthermore, this data might also be used as input for other models and therefore imposes significant accuracy constraints for the long run.

This compression technique provides great potential for future usage scenarios. Currently, there are several areas where research is conducted. First of all, researchers try to increase the performance of related compression algorithms, especially by keeping parallel processing and HPC in mind[43]. Additionally, an effort is put into providing scientists with thorough insights into the effects of lossy compression. In [44], the authors show the influence of the chosen compressors and data features on achieved compression performance. Furthermore, a system is shown which can help "domain scientists to make more informed data reduction decisions."

Implication

Both concepts show the significant impact of choosing compression algorithms according to their intended use case. Further domain knowledge is required in certain circumstances, and thorough considerations must be made to select the correct compression mechanism. This thesis intends to provide an automated machine learning approach to make that decision.

3.2 Approaches using machine learning

3.2.1 Autoencoders

Autoencoders consists of two parts. First, as shown in Figure 3.1, they use an encoder to reduce an input into a dense network representation. A decoder then uses this representation to attempt to recreate the initial input. Therefore, the goal of this architecture is to minimize the amount of information loss from the input to the output.

This artificially introduced bottleneck requires the input information to be compressed, and insignificant information might be lost. In terms of compression algorithms, autoencoders are a type of lossy compression. If the intermediate layer has at least the same size as the input layer, the network "could potentially just learn the identity function." [45]

The use-cases for such a network are manifold. Among others, autoencoders have been used for dimensionality reduction, anomaly detection, and image processing. In the latter, they show competitive results for lossy image compression compared to JPEG-2000 [46]. Furthermore, there are several scenarios for usage in HPC. In [47], autoencoders are proposed for data compression and anomaly detection purposes in the ATLAS experiment at CERN. Furthermore, in [48], the authors conduct a study which explores the use of autoencoders for compression of scientific data. This is done in comparison with lossy compression alternatives like SZ[1] and ZFP[2]. They conclude that autoencoders, when tuned correctly, can achieve superior compression rates. They also provide guidance for users to determine if their dataset is compressible using this network.

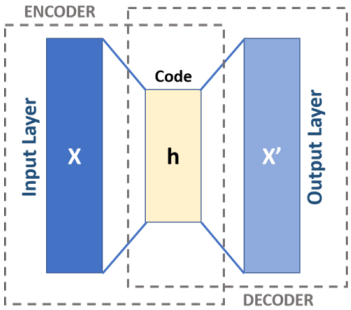


Figure 3.1: Autoencoder architecture [49]

3.2.2 Training on metadata

This thesis uses the application's actual data to train a neural network. The objective is to derive the ideal compressor from the encountered data during I/O. In earlier work, an attempt was made to use the available metadata within an HDF5 filter to make such a decision [50]. In this work, the goal was to recommend energy-efficient compression algorithms. Besides minimizing storage requirements also energy consumption is of importance for HPC sites. An additional design goal was to comprehend the predictions made by the decision component. For this, using a decision tree was ideal, and it is transparent to the user which compressor is chosen based on which metadata imposed rules.

[1] `https://szcompressor.org`
[2] `https://computing.llnl.gov/projects/zfp`

Implication

Those mentioned concepts already show the usability of machine learning approaches for data compression. It has also been shown that those techniques can be used for compression itself. However, this is currently limited to lossy data reduction. The approach of predicting a compression algorithm according to encountered metadata is similar to the concept discussed in this thesis. An advantage of the proposed approach in this work is the general availability of the data during I/O. The availability of metadata depends much more on the I/O path. However, a combination of both concepts could be worthwhile.

4 Architecture

In this chapter, the overall architecture is introduced. The outlined goals demand two main components developed specifically for this thesis. In the first part, a shared library is introduced. This library fulfills the need for intercepting I/O-related application calls and performing inferencing to predict compression algorithms for encountered data. The second component is used to train machine learning models, which the library then uses later.

The research purpose of this thesis imposes several requirements which have to be fulfilled by the underlying architecture. Those architecture-related requirements are outlined in the following:

- Predict the ideal compression algorithm during runtime based on I/O
- Perform predictions according to exchangeable metrics
- Gain insights into how compression algorithms, levels and metrics influence each other
- Integrate interchangeable machine learning models
- Easily usable within a multitude of scientific applications

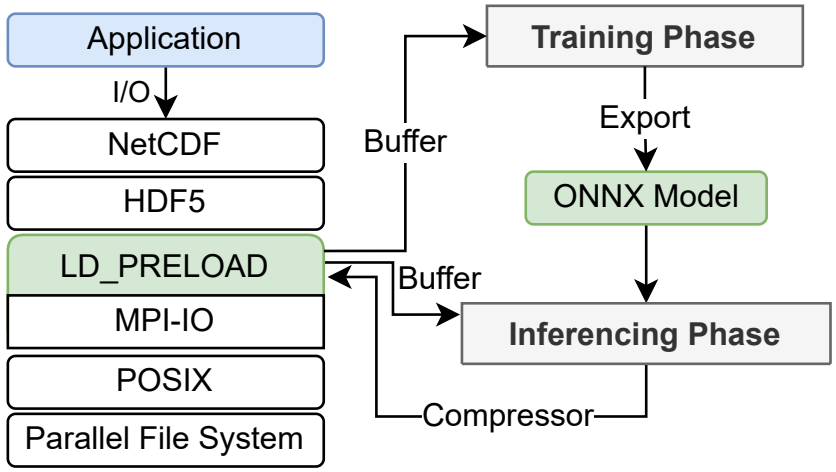
This results in an architecture that helps to automate three of the core challenges associated with implementing an adaptive data-dependent choice of compression algorithms:

1. *Sampling Phase*: Allowing for an automated sampling of input-target pairs for training without requiring application changes.
2. *Training Phase*: Leveraging state-of-the-art machine learning frameworks such as PyTorch[1] for training and hyperparameter search.
3. *Inferencing Phase*: Integration of in-band machine learning based decision components efficiently without the overhead of python runtime components by using the portable ONNX format and runtimes.

In Figure 4.1, a top view of the architecture is given. In order to intercept I/O easily from various applications, the MPI-IO layer is well-suited. As can be seen, this approach is based on preloading a custom library (using the `LD_PRELOAD` environment variable) and

[1] `https://pytorch.org`

intercepting the MPI-IO calls. For each I/O operation, certain metadata (size, datatype, etc.) is logged and the data buffer is compressed using a wide range of compression algorithms and settings. All metadata and resulting metrics, such as compression speed and compression rate, are stored in an HDF5 file for later analysis. This information is used as part of the training phase.



(a) HPC I/O Stack

(b) Sampling, training, and inferencing architecture for optimization of compressor selection.

Figure 4.1: Architecture overview showing a typical I/O path and points of instrumentation in relation to the training phase as well the inferencing phase. The training phase exports a machine learning model which is then used in an inferencing phase to predict a compressor based on data that is written by an application.

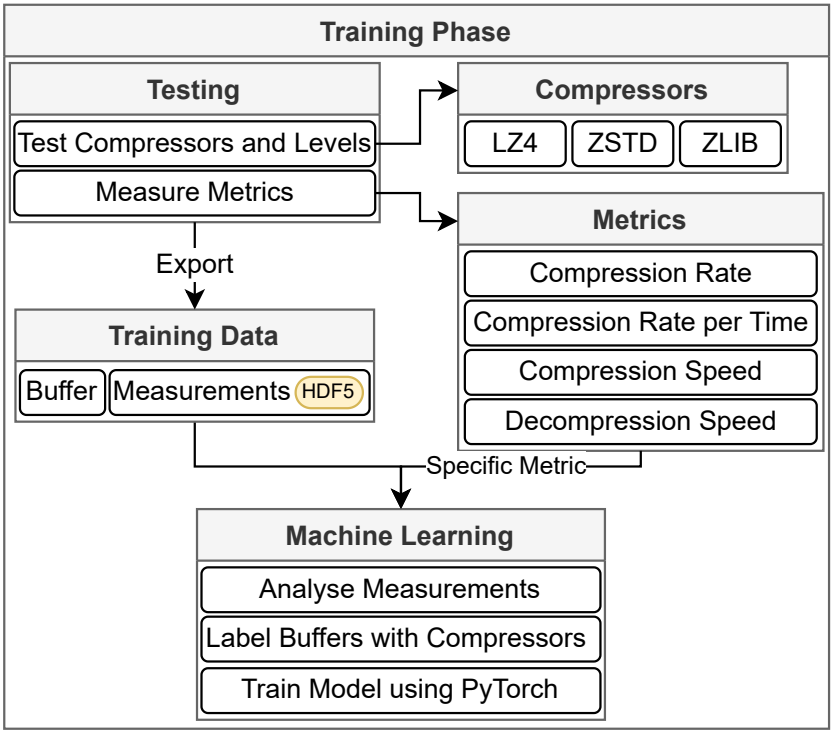


Figure 4.2: Training architecture

4.1 I/O interception library

The application that intercepts I/O related calls is implemented as a shared library. The order by which the Linux dynamic linker determines the required libraries can be influenced by using the `LD_PRELOAD` environment variable. Any library specified by this variable is loaded first. This makes it possible to create wrappers that encapsulate the original functionality of any other shared library.

Another benefit of this approach is that this is possible without changes to the users' application or recompilation. The only requirement is that the functionality that should be intercepted is available within a shared library and is not embedded via a static library. By being the library that is loaded first by the linker, it is possible to provide declarations that should be intercepted. During runtime, those are then used instead of the original ones.

As shown in Figure 4.1a, the functions which are of interest for this library are implemented by MPI. To intercept those, there exist two possibilities. The first approach is provided by the profiling interface defined by the MPI standard. This standard provides for every `MPI_` function a `PMPI_` equivalent. A regular application uses the `MPI_` variant. Internally, this function then uses its equivalent `PMPI_` to provide the required functionality. By following this practice, the new library implements the declarations of all MPI-IO write operations. Within those functions, the added functionality can be

```
1  int MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info,
2                    MPI_File *fh) {
3      int ret;
4      ret = PMPI_File_open(comm, filename, amode, info, fh);
5      if (tracing_stopped()) {
6          return ret;
7      }
8      if (!g_hash_table_contains(trackingDB_fh, fh)) {
9          IO_Object *object = g_new(IO_Object, 1);
10         object->fh = (void *)*fh;
11         object->filename = filename;
12         g_debug("filename: %s | handler: %p", filename, object->fh);
13         g_hash_table_insert(trackingDB_fh, object->fh, object);
14     }
15     return ret;
16 }
```

Listing 4.1: Example usage of the MPI profiling interface

provided. Finally, the equivalent `PMPI_` function is returned. This is shown in Listing 4.1, where `MPI_File_open` is extended by adding the functionality of storing attributes of the used file handler throughout the application run in a hash table. As highlighted in line 4 the associated `PMPI_File_open` function is used to provide the requested functionality. After the custom implementation is completed, the original return value is returned to the application. Further details can be found in *MPI: A Message-Passing Interface Standard* [51, p. 594].

The second approach is to use the corresponding functions by rewriting them directly. This is shown in Listing 4.2. In this example, the imminent `PMPI_Finalize` is changed by calling an additional function which handles the final metadata storage. Afterward, in line 9, the dynamic linker interface is used to find the address of the next occurrence of the respective symbol. This is then used to provide the intended functionality to the application.

While the first approach is preferred, both variants are used within the library. This is required when intercepting applications that use Fortran bindings of OpenMPI 2, as those perform calls to PMPI directly. However, this is only required for MPI-related functions which are used by the analyzed applications. Further down the I/O path, HDF5 uses MPI-IO methods according to the proposed standard.

4.1.1 Intercepted functions

All MPI-IO-related write operations are intercepted. Depending on whether collective or independent I/O is chosen, only two operations are relevant for HDF5. Additionally,

```c
int PMPI_Finalize() {
    int (*__real_PMPI_Finalize)(void) = NULL;
    int ret;
    if (!tracing_stopped() &&
        (opt_test_compression || opt_tracing || opt_inferencing)) {
        stop_tracing = TRUE;
        write_dataset();
    }
    __real_PMPI_Finalize = dlsym(RTLD_NEXT, "PMPI_Finalize");
    ret = __real_PMPI_Finalize();
    return ret;
}
```

Listing 4.2: Intercepting a call by reusing the declaration and loading the original symbol on demand.

as discussed above, the opening of files and the termination of the MPI processes are intercepted. The latter is used as a reliable trigger that stores all measurements in a state where the application can generate no additional metadata. All considered operations are listed below:

- `MPI_File_write`
- `MPI_File_write_all`
- `MPI_File_write_at`: Used by HDF5
- `MPI_File_write_at_all`: Used by HDF5
- `MPI_File_iwrite`
- `MPI_File_iwrite_all`
- `MPI_File_iwrite_at`
- `MPI_File_iwrite_at_all`
- `MPI_File_open`
- `MPI_Finalize`

4.1.2 Compression integration

An important aspect of the implementation of the various compression algorithms is the simplicity of adding new variants in the future. This is solved by defining a `struct`, which provides required definitions that have to be implemented by every compressor.

```c
typedef struct {
    size_t (*bound)(size_t length);
    size_t (*compress)(void *dst, size_t dstCapacity, const void *src,
                       size_t srcSize, int compressionLevel);
    size_t (*decompress)(const char *src, char *dst, size_t compressedSize,
                         size_t dstCapacity);
    int *levels;
    int levels_count;
    const char *name;
    CompressionAlgorithmID compression_id;
} CompressionAlgorithm;
```

Listing 4.3: Structure implemented by every compression algorithm

This interface is shown in Listing 4.3. A feature shared by all evaluated compression algorithms is the use of a function that determines the upper bound of the required size of the buffer which holds the compressed data. This is especially useful as the handling of associated buffers for compression and decompression is left to the caller. This is followed by the compression and decompression functions, which are similar, with the exception that the compression function also requires a compatible compression level. Further on, a selection of appropriate compression levels that are evaluated for this specific compressor is provided. Finally, a name and a unique identifier are chosen. Any compressor that is compatible with this structure can be easily integrated for further evaluation.

4.1.3 Usage modes

As shown previously in figure two, this library is used in either of two modes. Those are described in more detail in the following. Each mode also produces metadata that is used to either train or evaluate the performance of the machine learning models. This is done by writing to independent HDF5 datasets that are easily processed by other languages like Python. Incorporating those insights into machine learning frameworks like PyTorch is therefore eased.

Sampling mode

In this mode, all available compression algorithms and selected compression levels are applied to the data that is being written. For each of those combinations, measurements for the defined metrics are made. The corresponding HDF5 dataset is shown in Table 4.1.

Field	Comment
Operation name	e.g. `MPI_File_write_at`
Timestamp	Unix timestamp
Chunk Name	Used as input for training the machine learning model
Duration [µs]	Duration of compression in microseconds
MPI Datatype	Used to determine the size, e.g. `MPI_BYTE`
MPI Offset	Can be used to identify consecutive I/O
MPI Rank	
Variable Count	Number of written MPI Datatypes
Size	Variable Count $*$ sizeof(MPI Datatype)
Compressor Name	e.g ZSTD
Compressor Level	e.g. 22
Metric Name	e.g. Compression Rate
Metric Measurement	

Table 4.1: HDF5 dataset of the sampling mode

Inferencing mode

This mode is used to apply the trained models to incoming data and to evaluate the performance. This is done in three steps, which are highlighted below.

1. *Predict the compressor*: In this step, the intercepted data is applied to the machine learning model.
2. *Evaluate prediction*: The predicted compression algorithm and level are used to compress the data. Then, according to the metric, which is associated with the model, measurements are made.
3. *Determine ideal compressor*: Similar to the previous sampling mode, all available compression variants are tested for the current metric. For performance reasons, the predicted compressor is skipped, as those measurements were already made in step two.

The third step is only required for evaluation purposes. Those measurements would result in a significant slowdown of the overall application in a production environment. Furthermore, the metrics measurements in the second step might not be worthwhile during production runs. However, they might serve as an ongoing indication of the machine learning model's performance. In case a decline in prediction accuracy is detected, further adjustments could be made. For example, certain data blocks could be stored for further analysis or used for ongoing model adjustments.

Field	Comment
Timestamp	Unix timestamp
MPI Rank	
Size	Size of written data
Metric Name	e.g. Compression Rate
Predicted Compressor	e.g ZSTD
Predicted Level	e.g. 22
Predicted Compressor: Metric Measurement	
Predicted Compressor: Size	Compressed size
Ideal Compressor	e.g. ZLIB
Ideal Level	e.g. 9
Ideal Compressor: Metric Measurement	
Ideal Compressor: Size	

Table 4.2: HDF5 dataset of the inferencing mode

Configuration

The library is configured by setting the `IOA_OPTIONS` environment variable. Depending on the mode, several options can be selected. Those are shown in Table 4.3. Independent of the mode, a path for the HDF5 metadata has to be specified. Depending on the requirements, a minimal size of the analyzed write operations can be selected. Additionally, repeated measurements minimize possible inaccuracies due to other activities on the host system. In this case, the average of those measurements is stored.

Option	Description	Mode	
		Sampling	Inferencing
-m, --min-size=9	Min size of chunks to analyze in bytes	X	X
-r, --repeat=3	Number of times to repeat measurements	X	
-p, --meta-path=/tmp/meta.h5	Path for metadata storage	X	X
-t, --tracing	Activates tracing of MPI-Calls	X	
-s, --store-chunks	Activates chunk storage	X	
-c, --chunk-path=/tmp/chunks/	Storage path of chunks for later analysis	X	
-e, --test-compression	Activates compression tests according to metrics	X	
-x, --model-path	Path to exported ONNX model		X
-o, --settings-path	Path to exported ONNX settings		X
-i, --inferencing	Run inferencing		X
-d, --decompression	Measure decompression	X	

Table 4.3: Application options of the shared library

4.2 Machine learning process

In this component, the measurements in Table 4.1 are used to train the machine learning models. Common tasks like data preprocessing, model selection, hyperparameter tuning, and the final export of said model are described. The implementation is written using the Python programming language. This guarantees straightforward interoperability between the data generated in the previous step and common machine learning frameworks. Another benefit is the compatibility and ease of access to the sampling data provided by HDF5 and associated Python libraries like `h5py`.

4.2.1 Framework

For the training task, two major machine learning frameworks have been explored. At first, Tensorflow[2] was used. However, due to features related to the integration of the training data, PyTorch was eventually preferred. It is noteworthy that the selection of the framework is more due to personal preference. The data gathered during the sampling runs imposes no further restrictions on the machine learning components.

PyTorch is a machine learning framework developed by Facebook's AI Research lab. One of the initial significant differences in comparison to other popular frameworks is the use of an imperative approach to model creation. From the beginning, the developers' goal was to create a framework centered around the mechanics of Python and integrate it with existing developer tools. Those design goals and how to achieve them are highlighted in [52].

[2] `https://www.tensorflow.org`

4.2.2 Data preprocessing

The first step is to preprocess the data gathered during the sampling mode. Each measurement is associated with a unique chunk identifier. According to a selected metric, it is now possible to find the ideal combination of a compression algorithm and an associated level. This represents a label to data association, which is the input for all other machine learning-related tasks.

Internally, this is handled by implementing the `torch.utils.data.Dataset` class. The most important aspect here is to provide a function that returns the label and the chunk within a tuple. Additionally, the chunk is transformed into a `Tensor`, representing a multidimensional matrix. This matrix supports several data types, including single-precision floats with 32 bit, which are used throughout this model. Another property of this data type is the built-in support for CPUs and GPUs.

The data stored during the sampling phase contains the binary buffer received by the respective MPI-IO function. To be able to represent this data as a matrix with a continuous value, further preprocessing steps are required. Furthermore, depending on the architecture, a consistent matrix size has to be maintained when applying the data to the model. However, as writing patterns of scientific applications vary, this property is not a given. Therefore an input length is defined, and depending on whether the data is too long or too short, the matrix is resized to this specification.

Another aspect is that depending on the bit-wise representation of the data, a matrix element might be interpreted as a "Not a Number" (NaN) value. For example, this is the case when every bit is set to one. The consequence is that those values have to be filtered and replaced, e.g., by zero. This step is required when training the initial model and during the inferencing phase in the library.

4.2.3 Neural network

Independent of the used framework, some common and recurring machine learning-related task have to be handled. In most cases, this includes the creation of a neural network based on preexisting standard components as well as the integration of forward and backpropagation.

Neural networks are implemented by creating a subclass of the `torch.nn.Module` class provided by PyTorch. Within this base class only the forward propagation has to be implemented. Due to a mechanism named "autograd" a directed acyclic graph (DAG) of the neural network is created automatically during forward propagation. This DAG is executed in reverse order during backpropagation to update the model by determining the gradients using the chain rule.

The network defined in Listing 4.4 is used throughout the evaluation in the next chapter. The constructor in line two is used to parameterize the network. This eases experiments using different configurations later on. Additionally, the modules provided by PyTorch

```
1  class NeuralNetwork(nn.Module):
2      def __init__(self, num_classes, l_features_in=4096, l_features=512):
3          super(NeuralNetwork, self).__init__()
4          self.l_features_in = l_features_in
5          self.flatten = nn.Flatten()
6          self.linear_relu_stack = nn.Sequential(
7              nn.Linear(1 * l_features_in, l_features),
8              nn.ReLU(),
9              nn.Linear(l_features, l_features),
10             nn.ReLU(),
11             nn.Linear(l_features, num_classes),
12         )
13
14     def forward(self, x):
15         x = F.normalize(x)
16         x = self.flatten(x)
17         pad = nn.ZeroPad2d((0, self.l_features_in - x.size()[1], 0, 0))
18         x = pad(x)
19         logits = self.linear_relu_stack(x)
20         return logits
```

Listing 4.4: Example of using the base class provided by PyTorch to implement custom neural networks

are initialized. This includes reducing the dimensions by flattening the input tensor and a sequential stack of modules that are used to provide the actual network. The final layer of the stack has as many output features as classes available according to the analysis of the gathered sampling data for a specific metric.

In line 14 the forward step is defined. The parameter `x` is a tensor containing data intercepted and collected previously. Besides normalizing the input, the tensor is padded so that each tensor has the same size. Finally, the result of the forward propagation is returned, which at this stage is a prediction. This is now evaluated in the following step.

4.2.4 Model training

The model is trained in this supervised learning task by providing the network with a dataset and a ground truth of associated labels. As noted previously, the parameters of the network are then updated according to the loss of the model encountered during the forward propagation. This process is repeated several times, and the ongoing performance of the model is tracked throughout the training.

At no point in time all the available data is used for training. Some data is kept out

of the training process and is used to evaluate the performance on data that is entirely new to the model. This validation data helps to train a model which also generalizes to unknown data, which is more realistic for real-world scenarios. In this case, 20 % of the data is only used for this purpose.

Additionally, the model training is influenced by two core challenges, namely the imbalance of the gathered training data and the search for the ideal model parameters.

Imbalanced data

Currently, the training data only contains compression algorithms that achieve the best performance according to a specific metric. This results in imbalanced data as, in some cases, a compressor might perform exceptionally well due to edge cases. However, those are especially interesting for evaluation and should not be ignored during training.

This is countered by assigning weights to the available compression algorithm or the used classes during training, respectively. To do so PyTorch provides this functionality in the `nn.CrossEntropyLoss` criterion. In this case, more emphasis is put on classes that are not encountered regularly. Therefore, the misclassification of those compression algorithms results in a greater loss. As this loss is minimized during training, the model should adapt and classify those rare classes as well.

Model parameters

As shown in Listing 4.4, the model itself has several parameters. Additionally, the actual training process and the gradient descent can also be configured. This results in several parameters that have a significant influence on the performance of the model, and this should be accounted for. This process is known as hyperparameter tuning. To avoid having to find those parameters manually by trial and error, automation is helpful.

In this case Ray Tune[3] is used. Using this library, it is possible to automatically try different parameters, which can also be tested in a distributed environment. This is done by providing a configuration that contains search spaces that are evaluated. Configurations that do not achieve promising results after a few epochs are automatically canceled, and only parameters that improve the model are further observed.

The following parameters are tested for every model:

- *Output features of linear transformation*: 64, 128, 256, 512, 1024, 2048
- *Learning rate*: Sampling uniformly in logarithmic scale between 1×10^{-4} and 1×10^{-1}
- *Momentum*: Sampling uniformly between 0.0 and 0.9
- *Batch size*: 16, 32, 64, 128

[3] `https://www.ray.io/ray-tune`

The learning rate relates to the α introduced in Equation (2.7). However, the other parameters are more relevant to the implementation details of typically used machine learning frameworks. The momentum relates to a variant of the gradient descent, namely the stochastic gradient descent. This variant uses a random subset of the available data within batches, which can accelerate training. By changing the momentum, the previous gradients are considered for the next update and amplify the previous direction. This can prevent being stuck in a local minimum. Furthermore, the batch size specifies how much data is used for training in each iteration. In [53], it has been shown that very large batches can result in poor generalization. However, small batches might not find the optimal global solution. Finally, the number of output features of the linear transformer has a significant influence on the number of trainable parameters and the total model size.

4.2.5 Model usage

After the model has been trained, it has to be made available to the library for inferencing. This also results in a change of the programming language from Python to C. PyTorch has inbuilt support for the Open Neural Network eXchange (ONNX)[4] format. Therefore, the model can be serialized into a vendor-neutral format which is supported by several runtimes. The runtime used within the library is provided by Microsoft and is called the ONNX Runtime[5]. It is designed to provide hardware acceleration automatically for various targets. Among other programming languages like Java, `C#` and Julia also C is supported.

The model itself is exported by creating a dummy input which is then traced throughout the model. All used operations are recorded and used to serialize to a ONNX compatible format. However, as shown in Section 4.2.2, the data during inferencing can have dynamic input sizes. The dummy input can not exactly represent the data encountered during inferencing. This is enabled by using dynamic axes. It is required to manually specify the possible input size to be unknown during export.

[4] `https://onnx.ai`
[5] `https://onnxruntime.ai`

5 Evaluation

In this chapter, insights into the application-specific compression behavior and the performance of the machine learning models are evaluated. This is done by introducing an HPC application and conducting experiments that test specific capabilities of the introduced architecture.

5.1 Test setup

The following benchmarks were executed on the Mistral supercomputer[1], provided by the German Climate Computing Center. All measurements were run on 10 nodes, each equipped with two 18-core Intel Xeon E5-2695 v4[2] processors and 64 GB of memory. The data is compressed in memory. Therefore, networking and storage do not influence the measurements.

In Table 5.1, the evaluated compression levels/factors can be seen. In general, a broad mix of options was tested. When using the LZ4-fast API, instead of relating to levels, factors are used. According to the documentation, each increment results in a speedup of about 3 %. However, this comes at the cost of a lower compression rate. Therefore, compression factors of 7 and 17 should result in increased compression throughput of ~21 % and ~51 %, respectively.

	Levels/Factors	
Compression Algorithm	Available	Tested
ZSTD	1-22	1, 3, 10, 22
ZLIB	1-9	1, 3, 6, 9
LZ4	1-12	1, 3, 6, 9, 12
LZ4-Fast	1-65537	1, 7, 17

Table 5.1: Evaluated compression configurations

[1] https://www.dkrz.de/en/systems/hpc/hlre-3-mistral

[2] https://ark.intel.com/content/www/de/de/ark/products/91316/intel-xeon-processor-e52695-v4-45m-cache-2-10-ghz.html

5.1.1 ICON

In this section, measurements are done using the ICON modeling framework [3], which is a project between the German Weather Service and the Max Planck Institute for Meteorology. ICON is used for weather prediction as well as climate modeling. The application itself is written in Fortran. However, NetCDF is used for I/O, which relies on HDF5, which then uses MPI-IO to read and write the data. Another feature of ICON is the possibility of using dedicated I/O processes. Those gather the data and write the results asynchronous to the file system. Such an architecture is expected to allow for valuable compression usage, as the CPU is not involved with the actual simulation.

As shown in Figure 4.1, the shared library is therefore capable of intercepting I/O-related calls without any changes to the application itself. Another aspect, which makes ICON ideal for evaluation, is that the amount of written data is configurable by choosing appropriate options for the simulation environment. This is done by either selecting certain variables of interest or by limiting or extending the time frame which should be simulated. For evaluation purposes, metrics have been measured during a four-month simulation period. The machine learning model is then evaluated for an additional eight months, which results in an observed period of twelve months.

5.2 Metrics

For each chunk of I/O that has been intercepted, all compressors and a selection of levels were tested. Therefore, when searching for the best fit for a specific metric, one specific compressor per chunk of I/O is selected for analysis. These perfect fits also serve as a label for the associated chunk when training the machine learning model further on. If the metric relies on exact time measurements, the measurements are repeated three times, and an average is stored.

Throughout a four-month simulation period within ICON, 7548 MPI-IO calls were traced, and 7064 were evaluated below. Some calls occasionally write 8 B of data that are not compressible and skipped during inferencing further on.

The following plots include the used compressors on one axis and the evaluated metric on the other. Additionally, the total and relative amount of associated compressors for the specific metric is shown at the top. The cumulative distribution along the respective metric can be seen on the right-hand side.

5.2.1 Compression rate

For visualization purposes, the range of shown compression rates in Figure 5.1 is limited on one occasion to below 3 and the other one to below 2000. This limits the number

[3] `https://code.mpimet.mpg.de/projects/iconpublic`

of shown measurements to 4838 and 6811, respectively. Those excluded measurements achieved exceptional rates because of the uniformity of the included data.

As can be seen in Figure 5.1a on the right, the majority of the measurements achieved a compression rate of below 1.5. However, there are notable exceptions, which reached a much greater rate. When comparing those to the measurements in Figure 5.1b, it can be seen that ZSTD-22 led to a vast range of compression rates. While the total number of those high rates seems neglectable compared to the total number of compression events, the implied potential storage savings are significant.

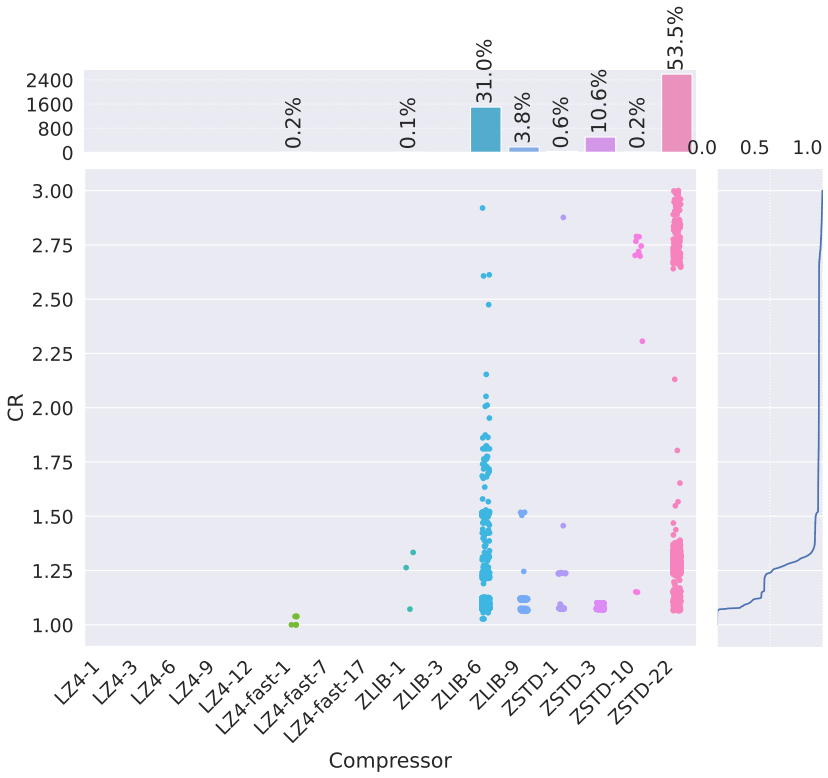
As expected and following the design goals outlined in Section 2.2, ZSTD achieves a much higher compression rate than LZ4. Another consistent choice is the use of ZLIB-6. While ZLIB-9 achieves in some cases a higher rate, it is important to note that for evaluation purposes, the compressor with the expected lowest overhead is used when both compressors achieve the same performance. Therefore, it is possible that certain compressor options, which conceptionally should result in a higher compression rate, do not necessarily achieve that performance in practice. These counterintuitive behaviors and the measurements of plateaus, meaning no observable changes during measurements with increased compression levels, are noticeable in other metrics as well.

5.2.2 Compression speed

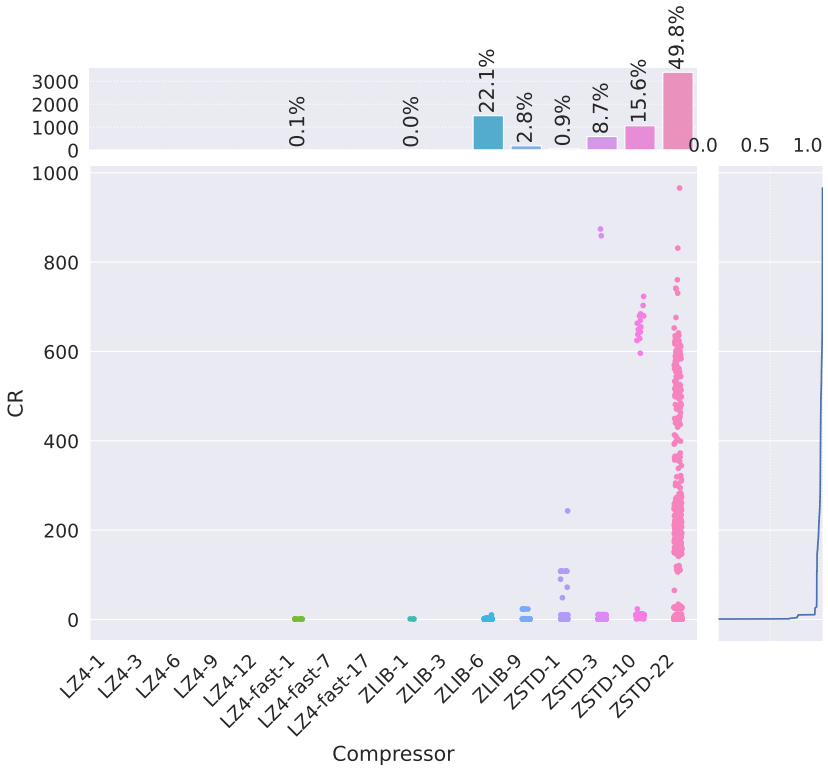
As shown in Figure 5.2, this metric is dominated by the fast compression mode of LZ4, and no other observed compressor matches this performance. Following the design goals of LZ4-fast, in 85.3 % of all cases, the compression level 17 achieves the highest compression speed. However, in 14.7 % of all measurements, a lower compression level reaches faster speeds. There are multiple possible explanations for this behavior. First of all, all metrics, except the compression rate, do depend on time measurements. Even with repeated measurements, as done in this evaluation, fluctuations can not be ruled out. Furthermore, when using LZ4-fast, data compressed using a smaller factor results in smaller files at the cost of taking more time during compression. As discussed in [54], compressed data can help with memory utilization, depending on the architecture. More efficiency in that regard could explain the behavior observed in these measurements.

5.2.3 Compression rate per time

This metric represents a relationship between the compression rate shown previously and the time it takes for the compressor to run. As shown in Figure 5.3, the results of the measurements are very imbalanced. In Section 5.2.1, it was shown that LZ4-fast achieves no significant compression rate in comparison to the other compression options. Nevertheless, its compression speed and consequently the time it takes for the compressor to run outweigh those shortcomings.



(a) Excerpt showing compression rate below 3



(b) Excerpt showing compression rate below 2000

Figure 5.1: Measurements showing compressors that were ideal for the compression rate metric.

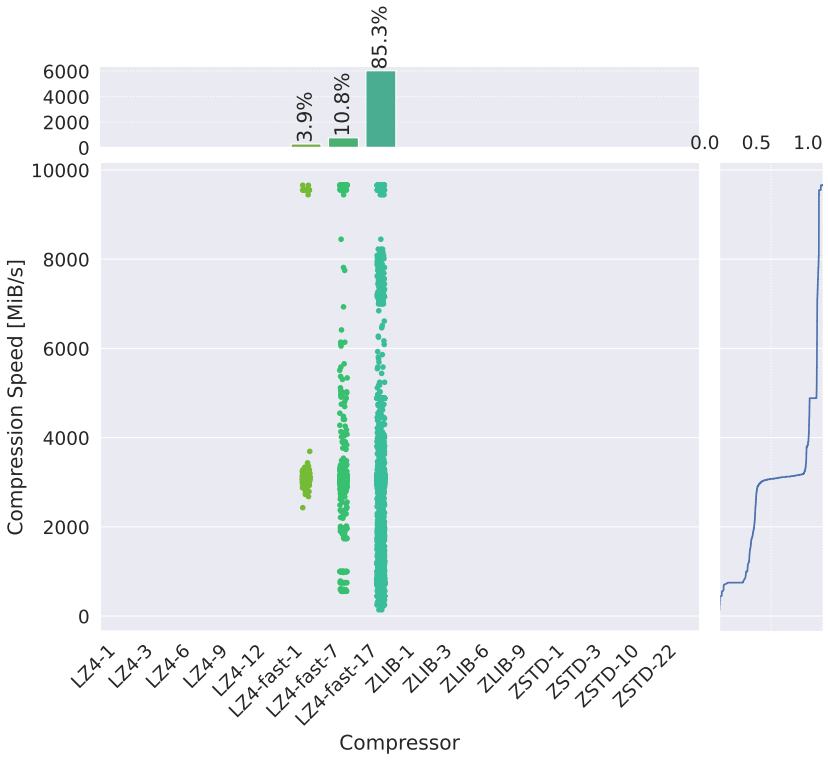


Figure 5.2: Compression speed metric

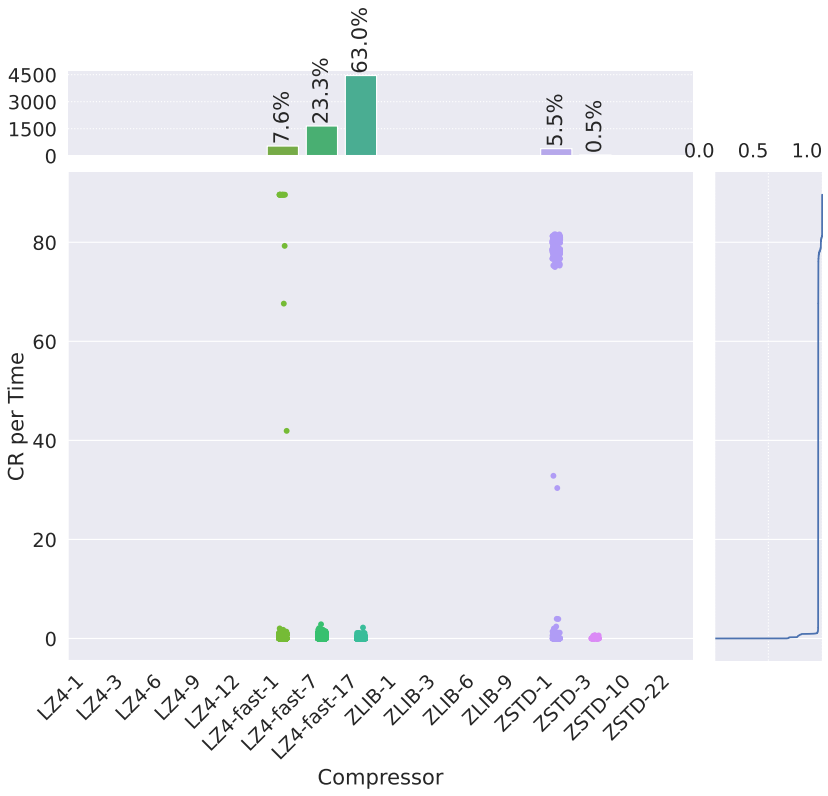


Figure 5.3: Compression rate per time metric

5.2.4 Decompression speed

As shown in Figure 5.4, in most cases, an LZ4 related compressor is chosen. As the decompressor works independent of the initial compressor, the intent for this metric and the use of different compression levels when compressing the data was to explore if the serialization of the underlying compression format impacts the decompression speed. There is a strong tendency to use LZ4 with compression level 1 when this metric is important. Until the strongest compression level of LZ4 is reached, the preference for using LZ4 in comparison to LZ4-1 is declining. However, LZ4-12 is capable of achieving the best decompression speed in 12.4 % of all cases.

Similar to the observation of LZ4 achieving high decompression speeds with low compression levels, LZ4-fast is doing the same with higher levels, as in this case, the compression speed is accelerated.

In an attempt to further assess those measurements, the relation between the observed decompression speed and the associated compression rate is shown in Figure 5.5. Similar to the measurements discussed before, the ideal decompressor is shown for every intercepted write request. Additionally, the associated compression rate of this compressor is marked. For visualization purposes, only compression rates greater than three and smaller than a hundred are shown. However, unlike previously, there are specific clusters visible. For example, LZ4-fast-17 is mainly shown in the top-right corner, while LZ4-12 is mainly in the center. Similarly, LZ-9 occurs for the most part in a compression rate of 80 to 100 while maintaining a decompression speed of around 10 000 MiB. This shows the expected correlation between high decompression speeds and high compression rates for the LZ4 compressors. In this case, the amount of highly compressed data outweighs negative implications of serialized dense data.

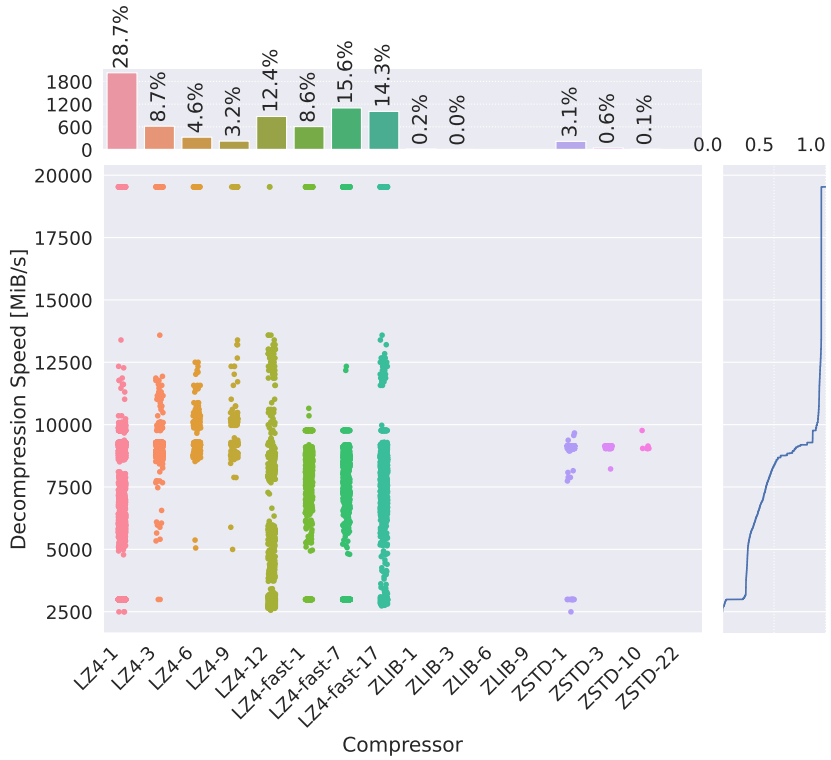


Figure 5.4: Decompression speed metric

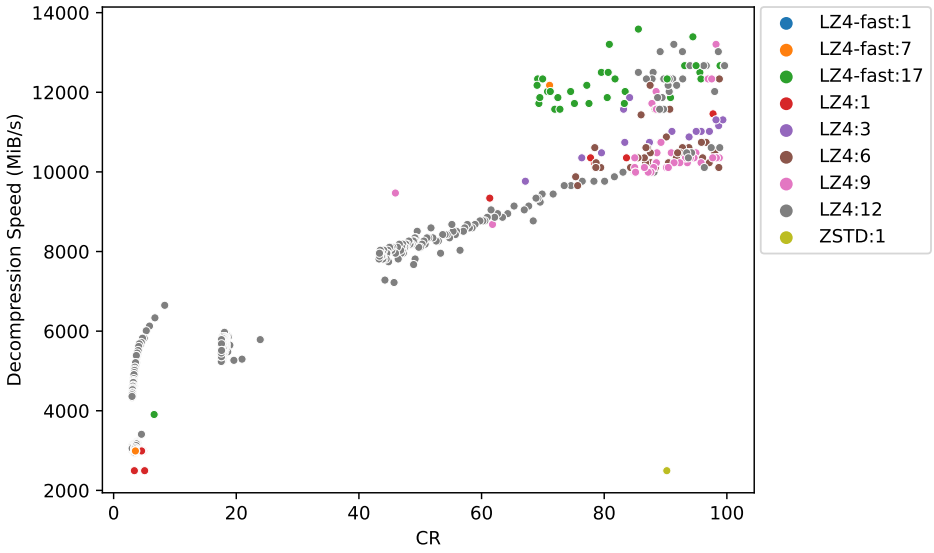


Figure 5.5: Relation between decompression metric and achieved compression rate. (Includes compression rates between three and a hundred)

5.3 Training

Based on the measurements shown previously, the machine learning models have been trained. As described previously, this supervised machine learning approach requires input data labeled accordingly. In this case, the data captured during I/O is used for this intent, and the ideal compressor variant serves as the label. The purpose of the neural network is to find features that help predict the ideal compression variant when presented with new data during application runs in the future.

The training data itself has various sizes. This depends on the defined data layout within NetCDF, which is then handled by HDF5 and finally written using MPI-IO. In this architecture, the data is transformed to a consistent size of 4096. The intent is to give the network a wide range of possibilities to adapt to the data. Further restrictions are enforced in Section 5.4.4, where the input size is intentionally limited.

In an attempt to optimize those models, hyperparameter tuning was applied. The parameters shown in Table 5.2 have been selected and are used to create the final models. A single model is trained for each metric, and the associated training and validation losses are shown in Figure 5.6. For better comparability, they share the same axes.

5.3.1 Compression rate

Figure 5.6a shows the training and validation loss of the compression rate metric. In total, 300 epochs were used. In the beginning, both loss metrics decline rapidly. However, while the training loss reduces further, the validation loss increases slightly and stabilizes after a while. An explanation for this is that the training data is imbalanced, and certain classes occur with a much higher probability. To counter that, the weights of sparse classes are weighted higher. This challenge exists with all introduced metrics, and further details are provided in Section 4.2.4.

5.3.2 Compression rate per time

The overall characteristics are similar when comparing the losses in Figure 5.6b with the previous measurements. However, the gap between the validation and training loss is closer to each other. Furthermore, some spikes in the validation losses are visible.

Metric	Output size	Learning rate	Momentum	Batch size
Compression Rate	64	0.0015078369731868298	0.7524708871836397	16
Compression Rate per Time	1024	0.00600528820268309	0.7598391737229157	128
Compression Speed	256	0.00019578516489309736	0.8856380172277318	64
Decompression Speed	1024	0.0014436667144263023	0.8123386279764642	64

Table 5.2: Parameters selected by hyperparameter tuning

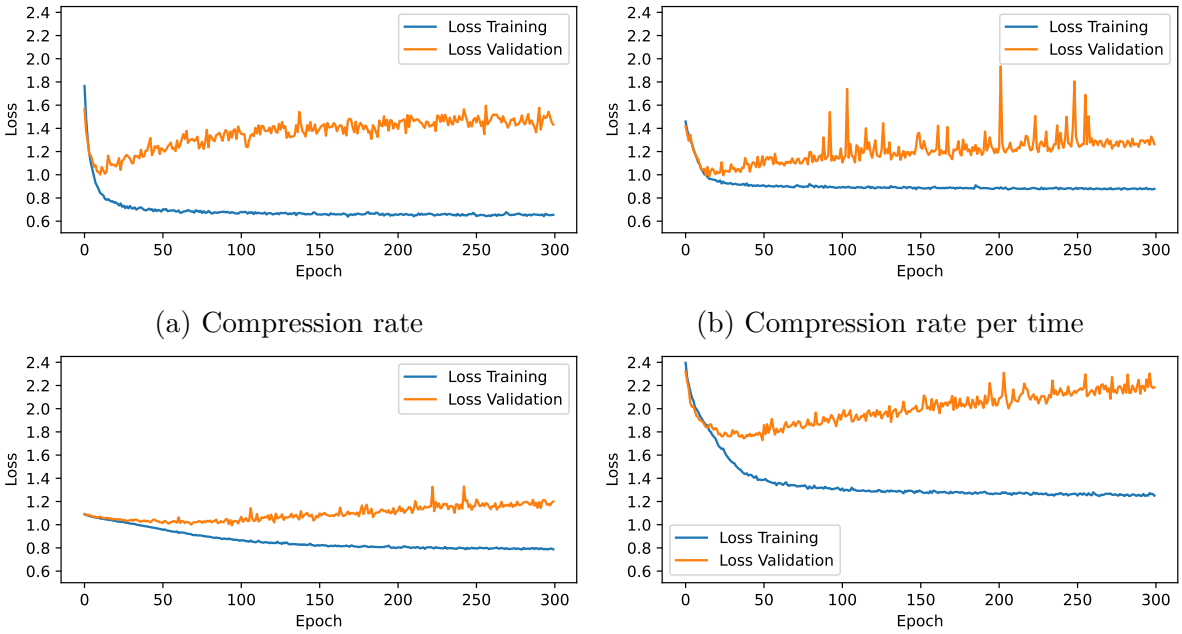
The reasons for this are manifold. One explanation for this is that the data is batched in different sizes, and the data within those batches differentiates. So, naturally, there might be outliers within those batches that do not conform to the model learned so far and result in more significant losses.

5.3.3 Compression speed

The losses shown in Figure 5.6c represent barely any learning progress. The training loss is not declining much further after ~150 epochs, and the validation loss increases slowly. This can also be attributed to the metric measurements shown in Figure 5.2. LZ4-fast with compression factor 17 is ideal in the overwhelming majority of measurements. Additionally, in the previous discussion in Section 5.2.1, the observation of plateaus was highlighted. Therefore, a lower factor might still be beneficial for some performance metrics. However, those relatively rare cases seemingly do not provide sufficient learnable data for this machine learning approach.

5.3.4 Decompression speed

In Figure 5.6d, the losses of the decompression speed metric are shown. A model that is capable of learning this metric has to be capable of differentiating the most labels compared to the metrics evaluated previously. The plot is similar to the behavior seen when training the compression rate model. However, this metric imposes the highest losses overall. This might correlate to the challenge of learning to predict a large number of compression algorithms and variants. Additionally, as discussed, the imbalance of the training data continues to be challenging for all approaches.



(a) Compression rate

(b) Compression rate per time

(c) Compression speed

(d) Decompression speed

Figure 5.6: Training and validation loss when creating the models according to specific metrics

5.4 Inferencing experiments

The trained model for the respective metric can now be evaluated by applying it to previously unseen ICON runs. This helps to evaluate how well the model generalizes to diverse write operations.

5.4.1 Extended simulation time steps

In this first test, the simulated time frame of ICON has been extended to twelve months. As shown in more detail in Section 5.2, the model was initially trained with four months of simulation data from ICON. For better insights into the model's performance with respect to the ideal case, confusion matrices are being used throughout the evaluation. They provide a good understanding of the performance on a per-class level.

Compression rate

In Figure 5.7, a selection of encountered compression rates in the range of two to eight is shown. These measurements are visualized as they change over the simulated time frame, respectively the analyzed I/O steps. Additionally, the data used during the training phase is highlighted. Finally, the correctness of the prediction performed by the machine learning model is shown as well.

The first thing to note is that the individual MPI ranks write data, which differentiates by the compression rates achieved. While ranks two and three behave similarly over time, the compression rate of rank one declines. A possible explanation for this behavior could be that data is refined as the simulation progresses, and an initial consistent state is updated. Another noteworthy observation is that overall, the accuracy of MPI rank two and especially rank three declines with new and unseen data. However, contrary to that, the performance of rank one and the performance of rank two in the compression rate rage of around four continues to be steady.

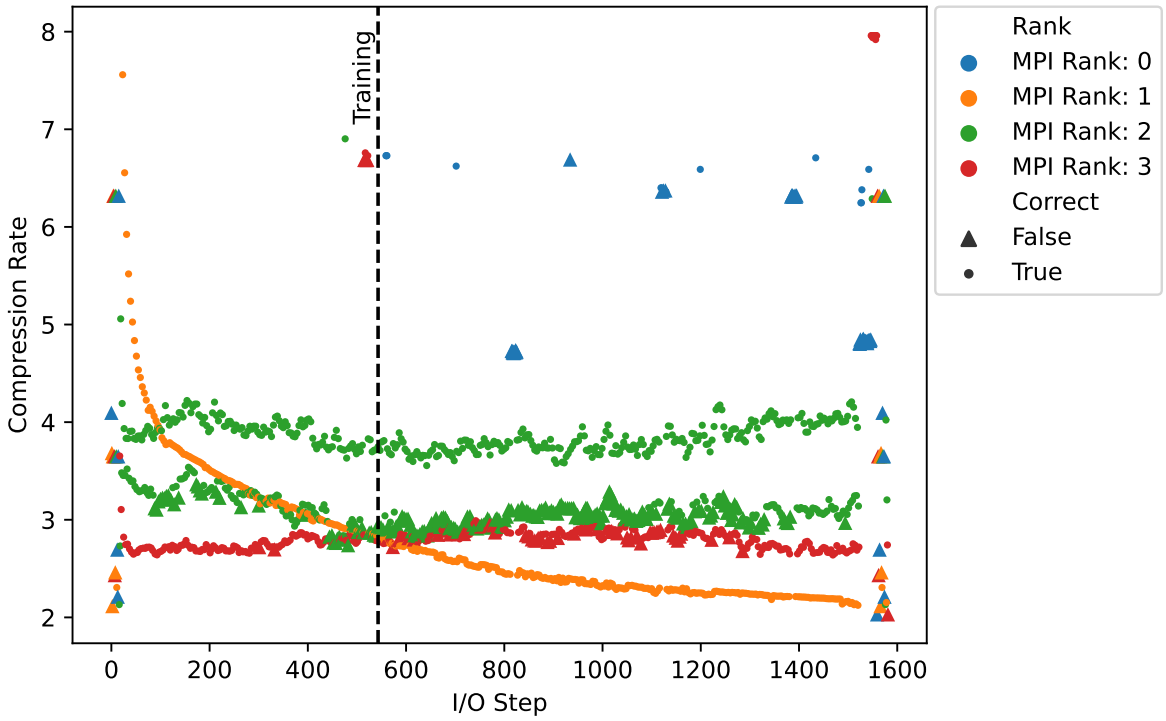


Figure 5.7: Classifications on an extended simulation time period using the compression rate model

Figure 5.8 shows a confusion matrix for the compression rate model. In most cases, ZSTD-22 is predicted correctly. ZLIB-6 and ZSTD-10 are also confidently chosen. However, in 2238 cases, ZSTD-22 is predicted instead of ZLIB-9. This shows once again the issue with underrepresented compression variants during the training phase. Furthermore, this also indicates changes of ideal compression algorithms as the ICON simulation progresses. For example, previously, in 2.9 % of all cases, ZLIB-9 was the ideal choice for the compression rate metric. However, within this extended simulation run, this increased to 14.24 % of all cases. This also indicates that the dynamics of the correct choice and a static selection are not sufficient for the long run.

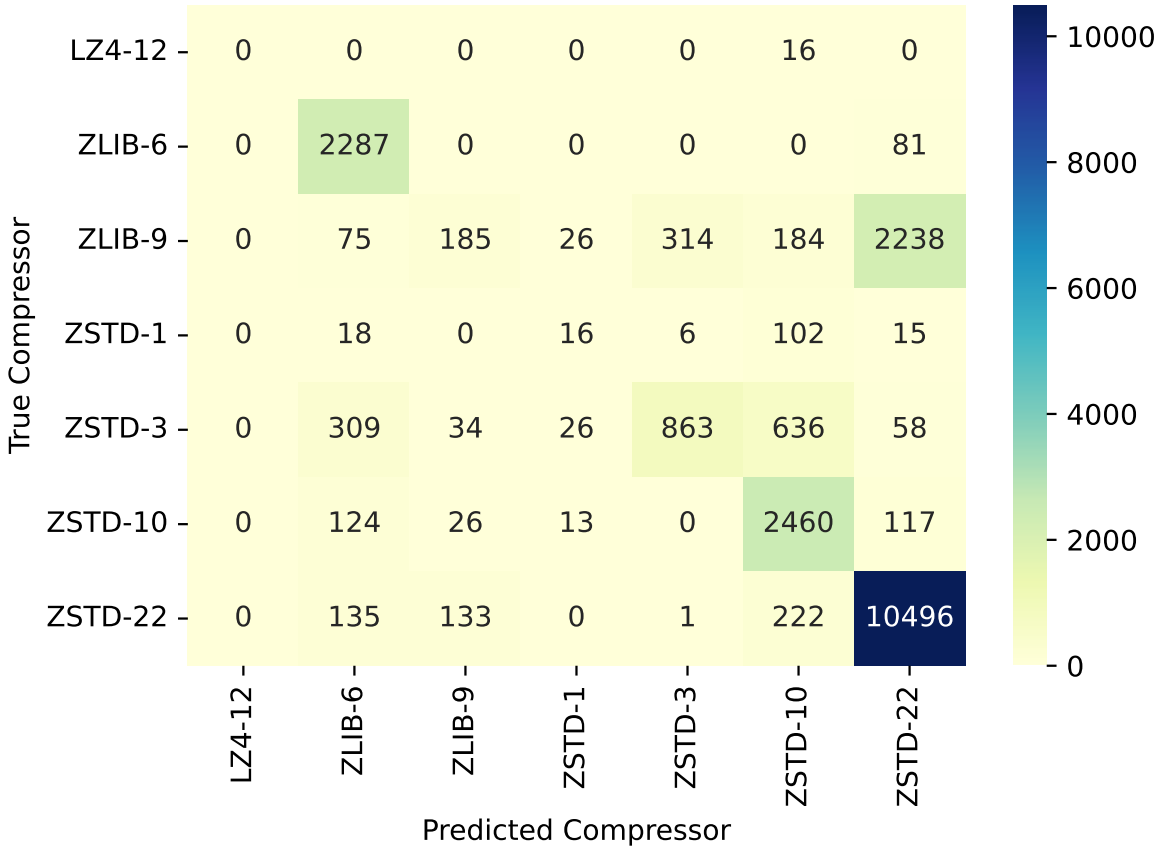


Figure 5.8: Confusion matrix when using the compression rate model

Compression speed

As discussed in Section 5.2.2 and Section 5.3.3, the challenges imposed by this metric are immediately noticeable in the confusion matrix shown in Figure 5.9. While this model shows some capability of predicting LZ4-fast with factor 17 correctly, it fails to achieve sufficient performance for any other compression algorithm. The current approach is not sufficient for this metric, at least not with the current data written by ICON.

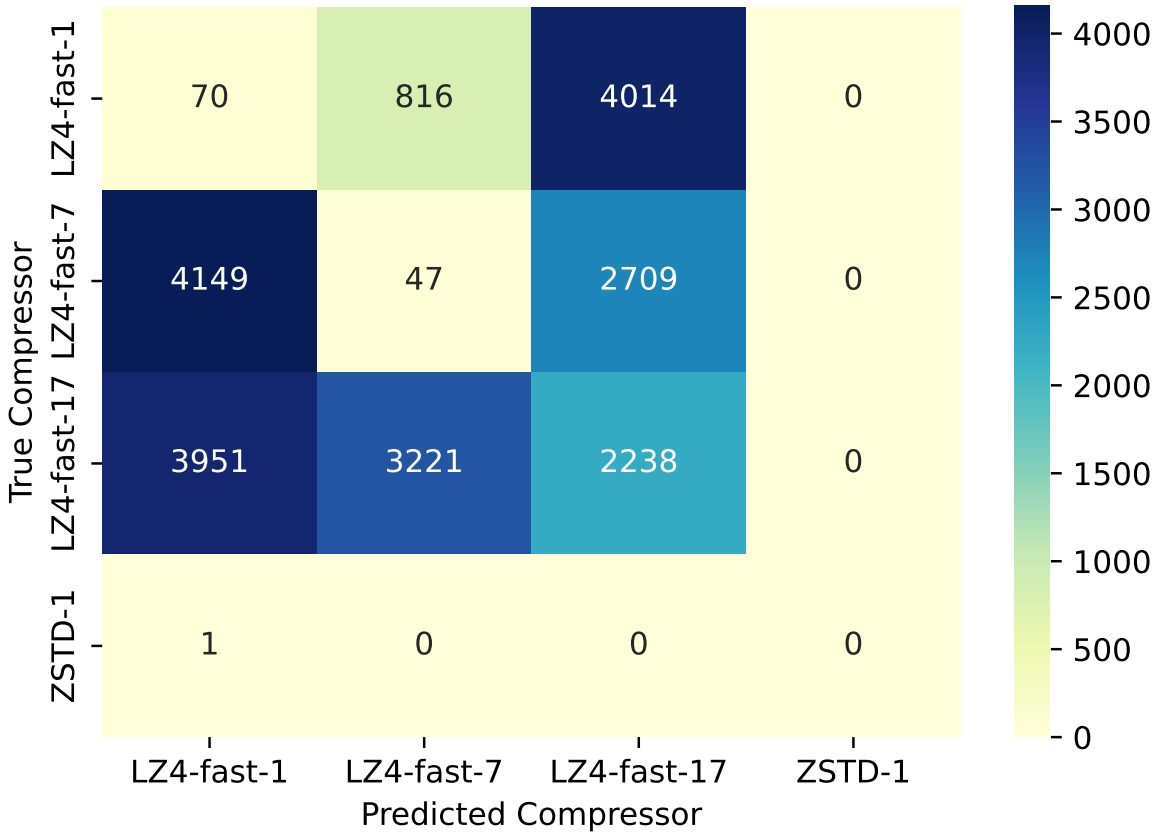


Figure 5.9: Confusion matrix when using the compression speed model

Compression rate per time

In the majority of all cases, LZ4-fast related compression algorithms are ideal for this metric. This is also reflected in Figure 5.10, where this also holds true for this extended simulation run. Somewhat surprisingly, the performance of the LZ4-fast compressor with factor 17 is not performing as well as the distribution among the initial metric measurements might suggest. For this variant, sufficient training data should be available for the model to adapt to. However, the current performance might also be due to the lack of learnable features within the data in relation to this compressor class. Similar to the behavior observed with the compression rate metric, changes in the use of compressors

over the long run are noticeable. For example, in this case, LZ4-fast with factor one is used in greater quantity than before.

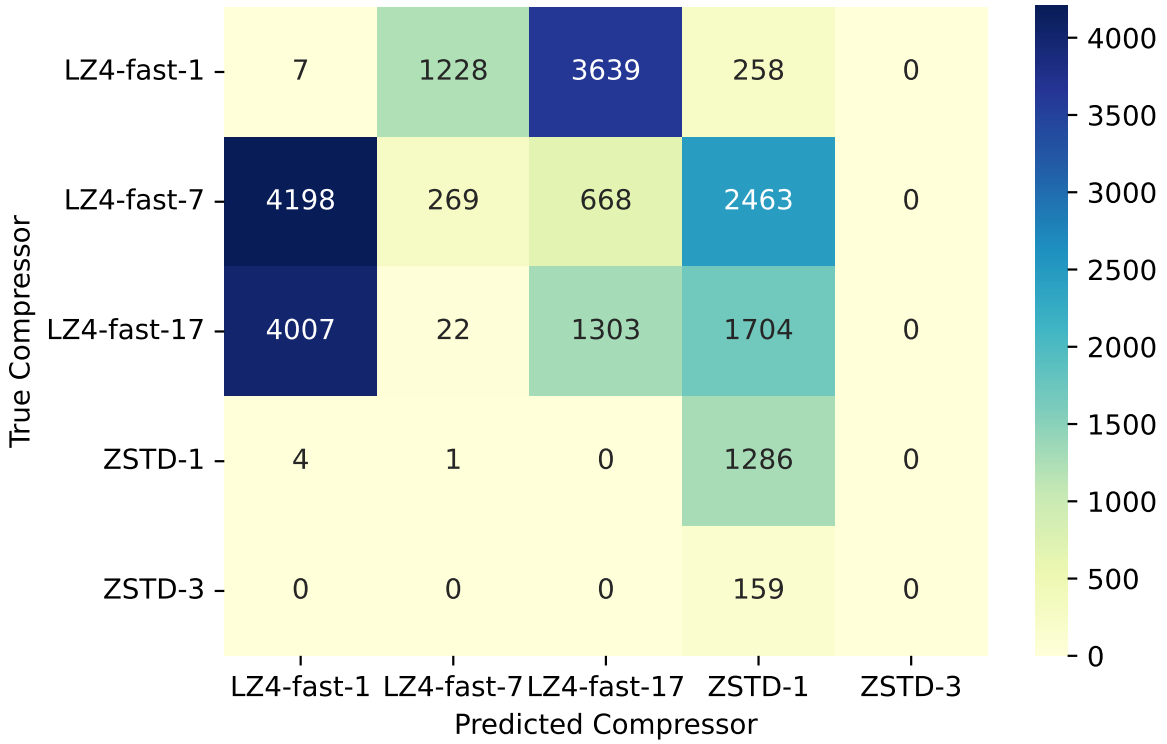


Figure 5.10: Confusion matrix when using the compression rate per time model

Decompression speed

As shown earlier, the decompression speed metric relies on the most compression algorithms compared to the other metrics. This is also shown in the confusion matrix in Figure 5.11. There are several things to note here. First of all, the prediction of LZ4-fast-1 related compression algorithms works very well. For this specific class, barely any misclassification occur. However, many other classes are wrongly classified as this class, especially other LZ4 variants. The bias towards this specific class is rather surprising as the use of LZ4-fast-1 is not that dominant when comparing these results with the measurements in Figure 5.4.

Besides the influence imposed by this class, LZ4-1 and LZ4-3 are performing well. This is expected especially for the first case, as this class has the most training data available.

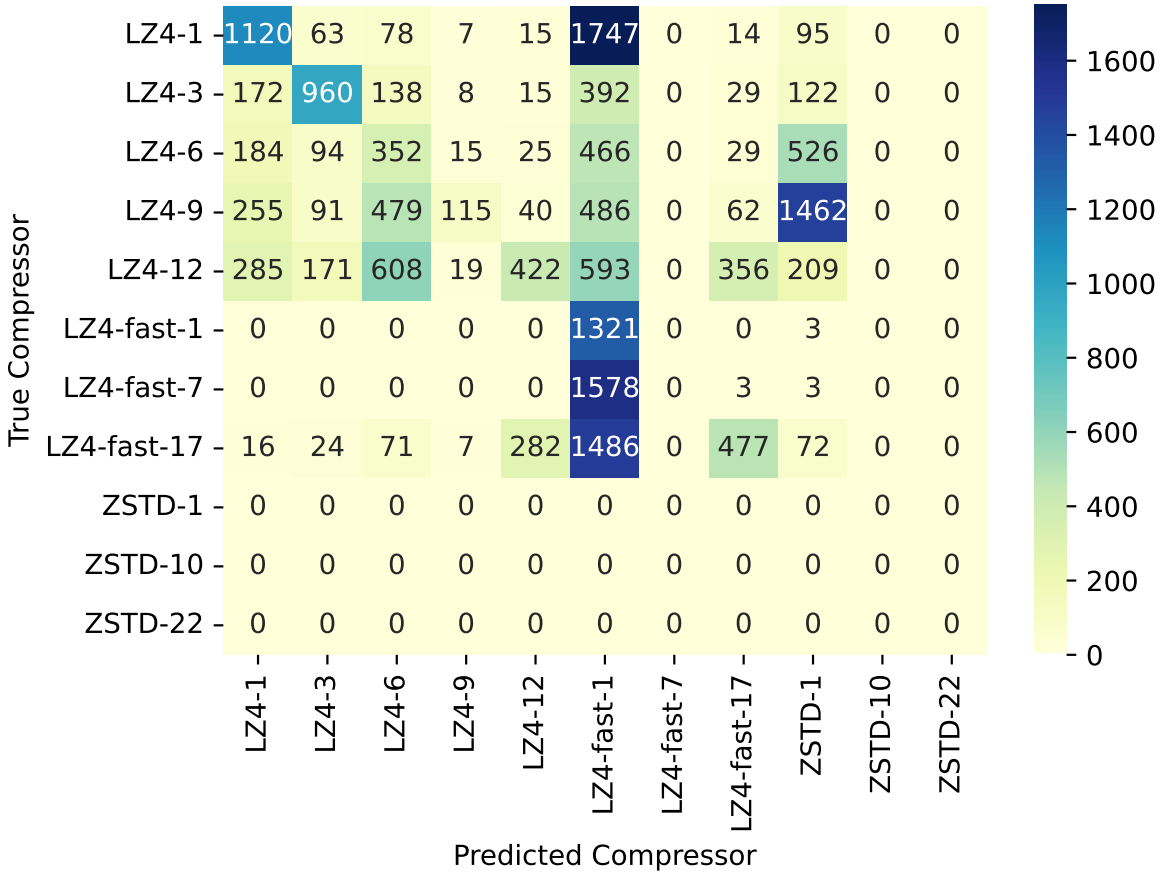


Figure 5.11: Confusion matrix when using the decompression speed model

5.4.2 Extended simulation output

The ICON application can be configured to only write specific variables to the HDF5 dataset. In the previous experiment, the physical output in the 3D environment is not passed to the I/O layers. Therefore this data has never been intercepted and analyzed by the architecture implemented in this thesis. Previously, the data that, till this point, is unknown to the neural network does stem from known variables but only changes due to progress in the simulation. In this new case, the data might be entirely out of context for the network, which can give more significant insights into how well the current approach generalizes to new scenarios. In the following, the compression rate metric is once again evaluated. Previously, this metric gave promising results, and measurements of this type are very accurate as they are not influenced by time measurements or any other side effects.

This influence can be seen in Figure 5.12. The increase in write operations is apparent when comparing this representation with the previous one in Figure 5.7. The additional data results in greater overall activity within the compression rate range between two and eight. Furthermore, increased activity is also related to specific MPI ranks. For further analysis, rank 3 is highlighted. Similarly, as before, the compression algorithms in the range of up to three are correctly predicted. However, the additional output in this specific range is not correctly predicted with acceptable accuracy. This suggests that the current approach is sensitive to unseen variables.

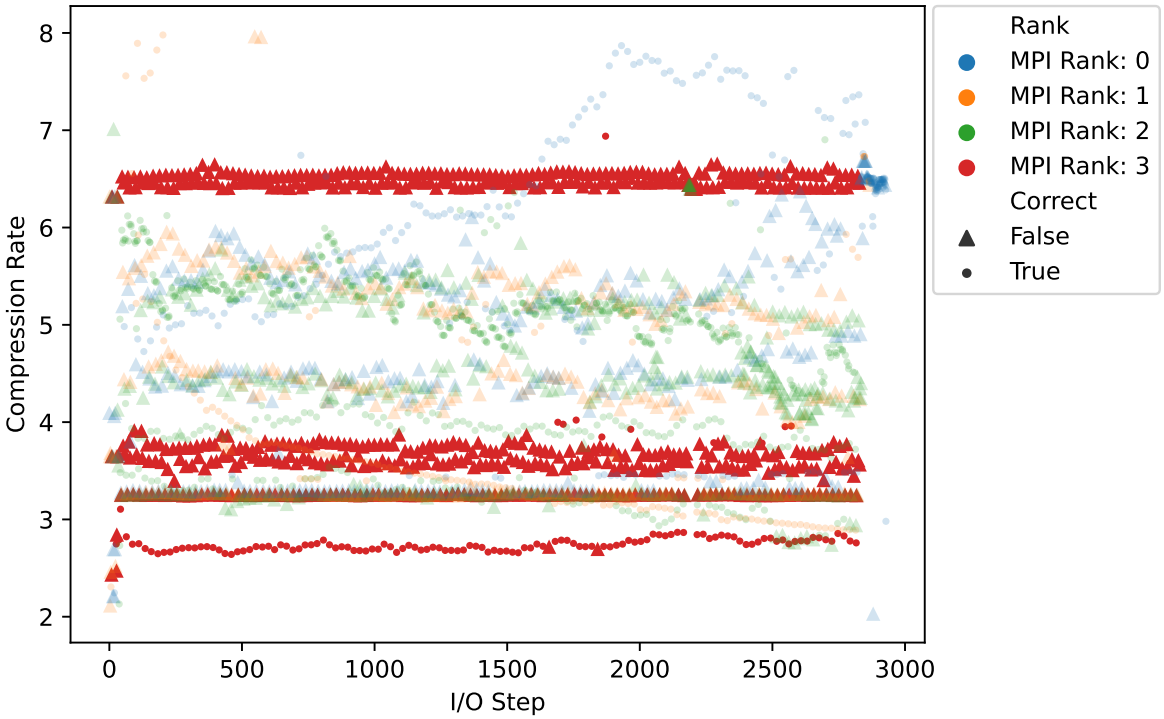


Figure 5.12: Classifications on an extended simulation output

Figure 5.13 shows the confusion matrix of all predictions during this experiment. For the most part, these measurements are similar to the ones done in Figure 5.8. The accuracy of predicting ZLIB-6, ZSTD-10 and ZSTD-22 is promising. In this case, the identification of ZLIB-9 compatible data is improved. However, data that ideally should be compressed using ZSTD-22 is wrongfully compressed using ZLIB-6 in 1220 cases.

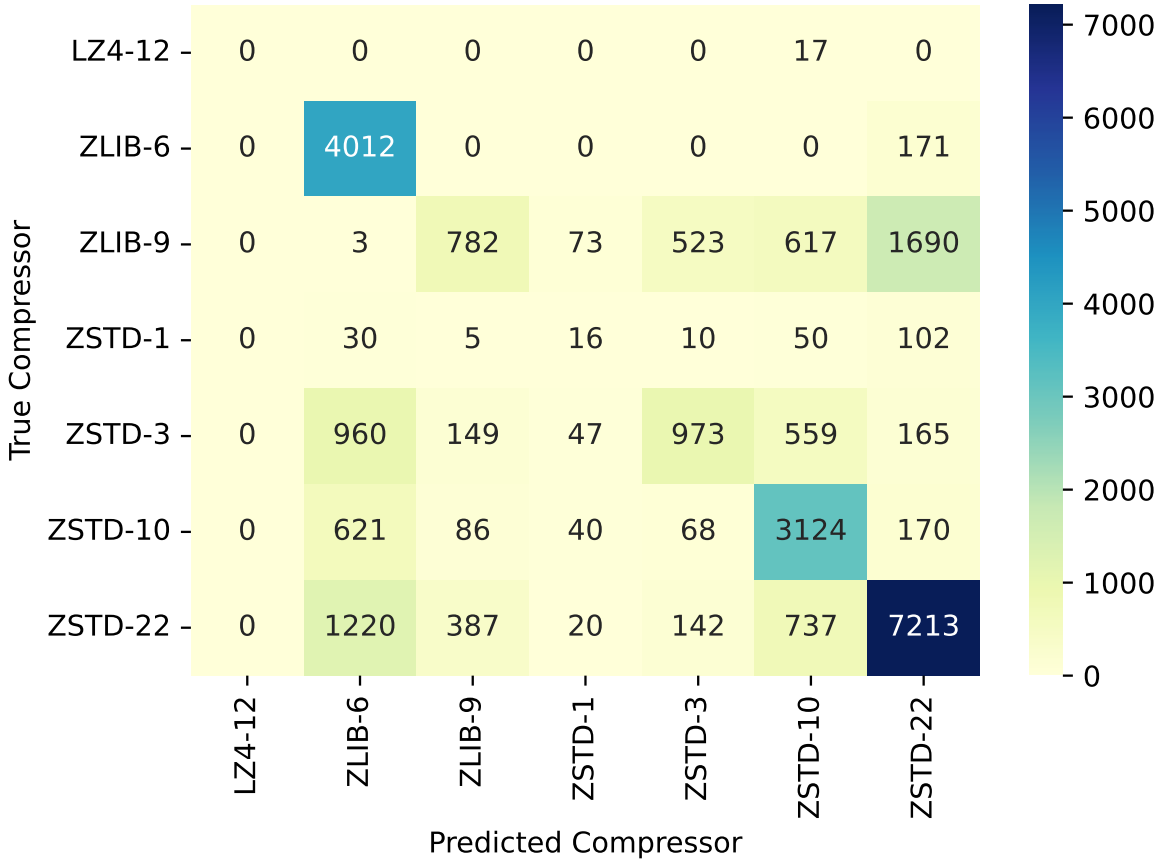


Figure 5.13: Confusion matrix when using the compression rate model on extended output

5.4.3 Reduced data collection phase

In this experiment, the initial data collection phase of ICON is shortened. This serves as a test on how much data is required to train a neural network with sufficient accuracy. The benefits of a shortened collection phase are significant:

- Reduced computation time of the application, which might only serve the purpose of training the initial neural network
- Faster network training
- Overall less cumbersome to use in HPC applications

Metric	Output size	Learning rate	Momentum	Batch size
Compression Rate	64	0.01884238882532786	0.2880154357402211	16

Table 5.3: Parameters selected by hyperparameter tuning when reducing the data collection phase

For this purpose, the compression rate metric is once again evaluated. However, only seven days are used instead of using the data collected over a four-month simulation period. During this period 40 000 write operations are collected, resulting in 625 samples for the training phase. In Table 5.3, the parameters found by hyperparameter tuning are shown. The training and validation losses seen in Figure 5.14 were achieved using these parameters. The most significant training progress is made up to ~75 epochs. Beyond that, a slight tendency to overfit can be observed.

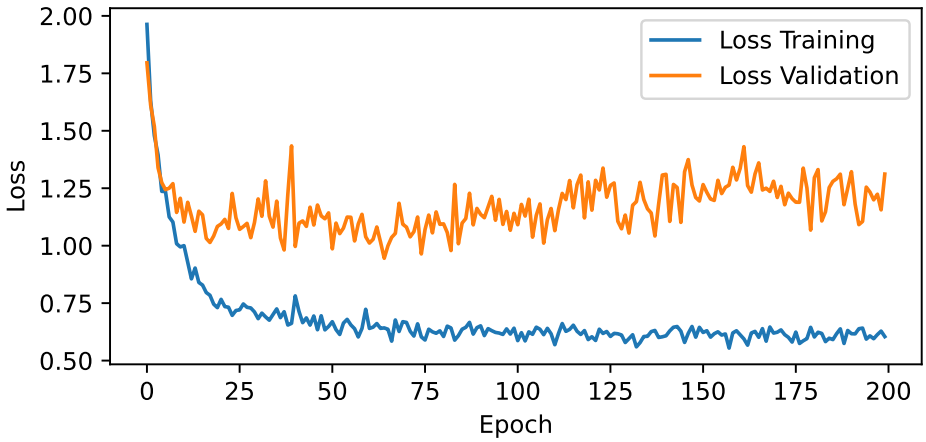


Figure 5.14: Training and validation loss when creating the model according to the compression rate metric. In this case only seven days of simulation data are used.

In Figure 5.15, once again, the amount of training data in relation to the overall inferencing phase is shown. It is important to remember that only an excerpt of the data within an observed compression rate between two and eight is shown. Of those 625 samples for the training phase, 50 samples achieve a compression rate within this range. This is of interest when comparing those measurements with the evaluation made earlier. In that case, four months of simulation data were used within the training phase. Previously, a total of 7548 samples were used for training. 534 samples were within this highlighted compression rate range. The overall accuracy is very different in comparison and highlights the challenges associated with training a sufficiently good performing neural network. Within this highlighted range, the accuracy of the model that has been trained with limited data is ~86.09 %. This is significantly higher than when training the network with more input data as done before. Previously, the accuracy was

~79.3 %. However, the overall accuracy of this limited model is only ~47.1 %, compared to 77.1 % measured before. One of the reasons for this observation could be that 8 % of the training data during this reduced data collection phase is within this limited range. Previously, only 7.07 % of the training data was within this range, which could limit the model's capabilities to achieve similar results. However, the model's overall performance is significantly increased when using more training data, which is preferable.

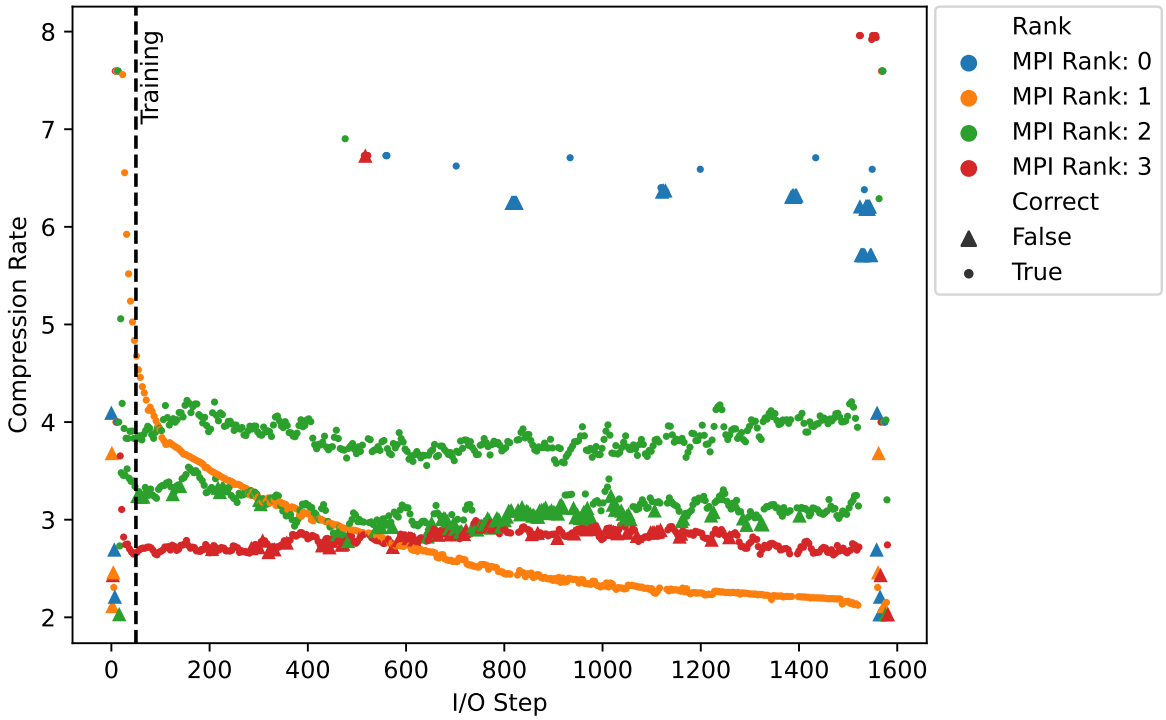


Figure 5.15: Classifications on a twelve month simulation period using only seven days of simulation as input.

Those differences also translate to the model's accuracy for specific compression algorithms. As shown in Figure 5.16, the accuracy of determining ZSTD-22 correctly declines. This algorithm is misclassified for ZLIB-9 in 6404 cases. Interestingly, the contrary also holds true, and ZLIB-9 is correctly predicted in an increased number of cases compared to the model trained with more input data.

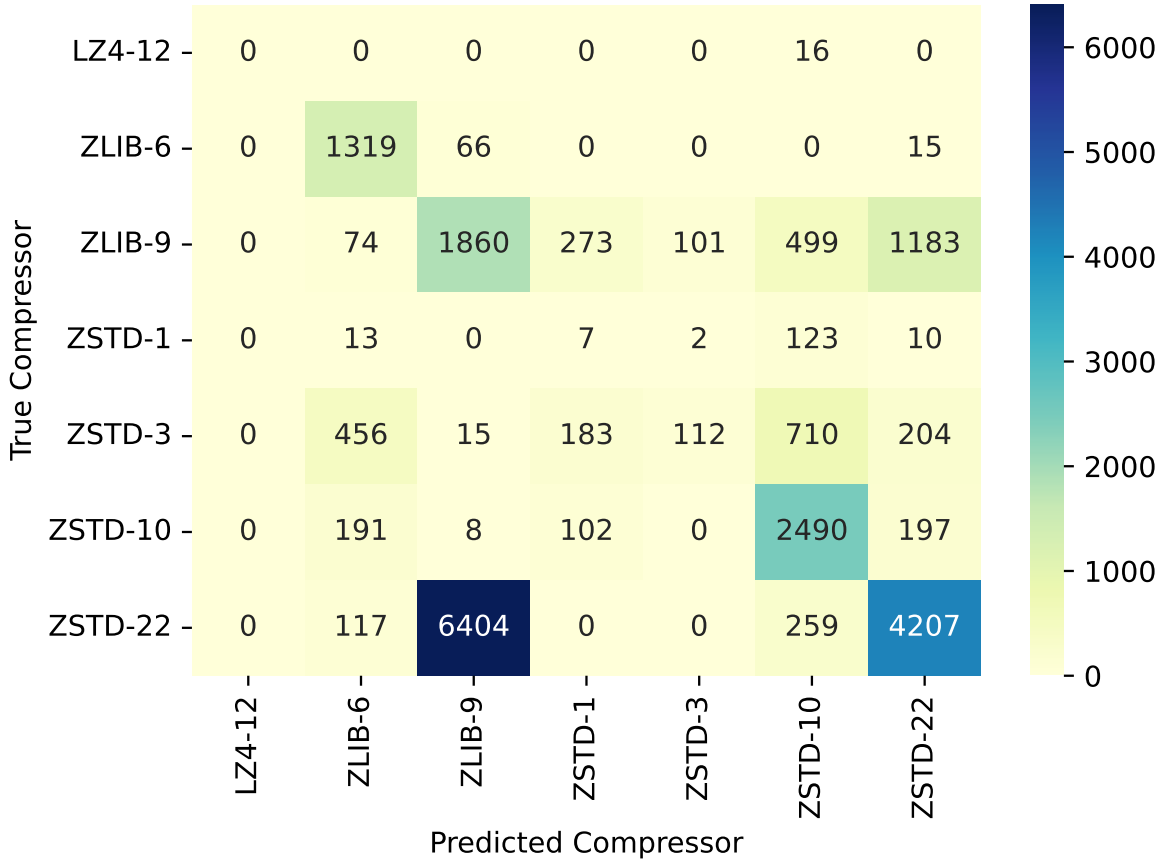


Figure 5.16: Confusion matrix when using the compression rate model with limited training data.

5.4.4 Reduced chunk size

In this experiment, further neural network architectures have been evaluated, which rely on fewer data as the input for the first layer of the network. For this purpose, input sizes of 8, 16, 32, and 64 have been tested. For each size, hyperparameter tuning was performed. The results are shown in Table 5.4. For further analysis, the highlighted input size is chosen. While the performance of all tested sizes is inferior to the previous experiment, an input size of 32 B performed the best. However, the performance did not change by a large margin. In comparison, even the performance of only using an input size of 8 B is surprisingly good. In part, this can be attributed to the used output size of the linear layer of the network. Theoretically, a single chunk could be stored within such a layer, as it is bigger than the input size. Those challenges are also related to the use of auto encoders, discussed in Section 3.2.1.

As shown in Figure 5.17, the performance by no means compares to the model using 4096 B of input. This is not limited to data which is completely new to the model but also includes the data that has been trained on. This is also obvious when evaluating the

Input size	Output size	Learning rate	Momentum	Batch size	Accuracy
8	16	0.002546558192892439	0.8790849793713035	128	0.55
16	16	0.001272695825250678	0.8614396430577418	128	0.56
32	32	0.00600528820268309	0.7598391737229157	128	0.6
64	64	0.0024588173734937996	0.24986583566525872	32	0.59

Table 5.4: Parameters selected by hyperparameter tuning for limited inputs

confusion matrix shown in Figure 5.18. Especially the accuracy of identifying ZSTD-22 has changed. In comparison with the measurements in Section 5.4.1, ZLIB-9 is predicted correctly in many cases. However, in 9825 cases, ZLIB-9 is predicted instead of ZSTD-22. This suggests that crucial data is missing from the input and the neural network is not capable of learning features that are relevant for separating those specific classes.

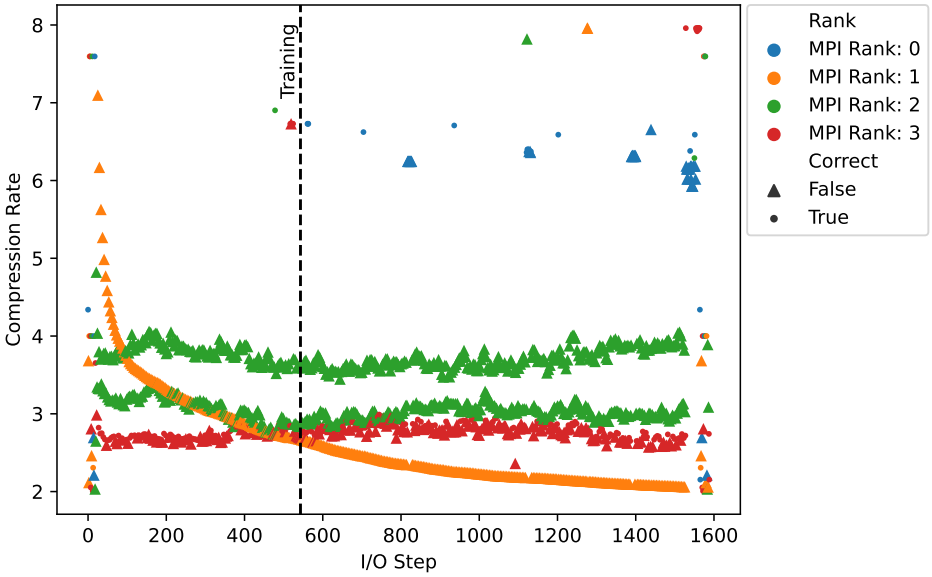


Figure 5.17: Classifications on an extended simulation time period using the compression rate model and using only 32 B of input data. The shown compression rate range is limited to two to eight.

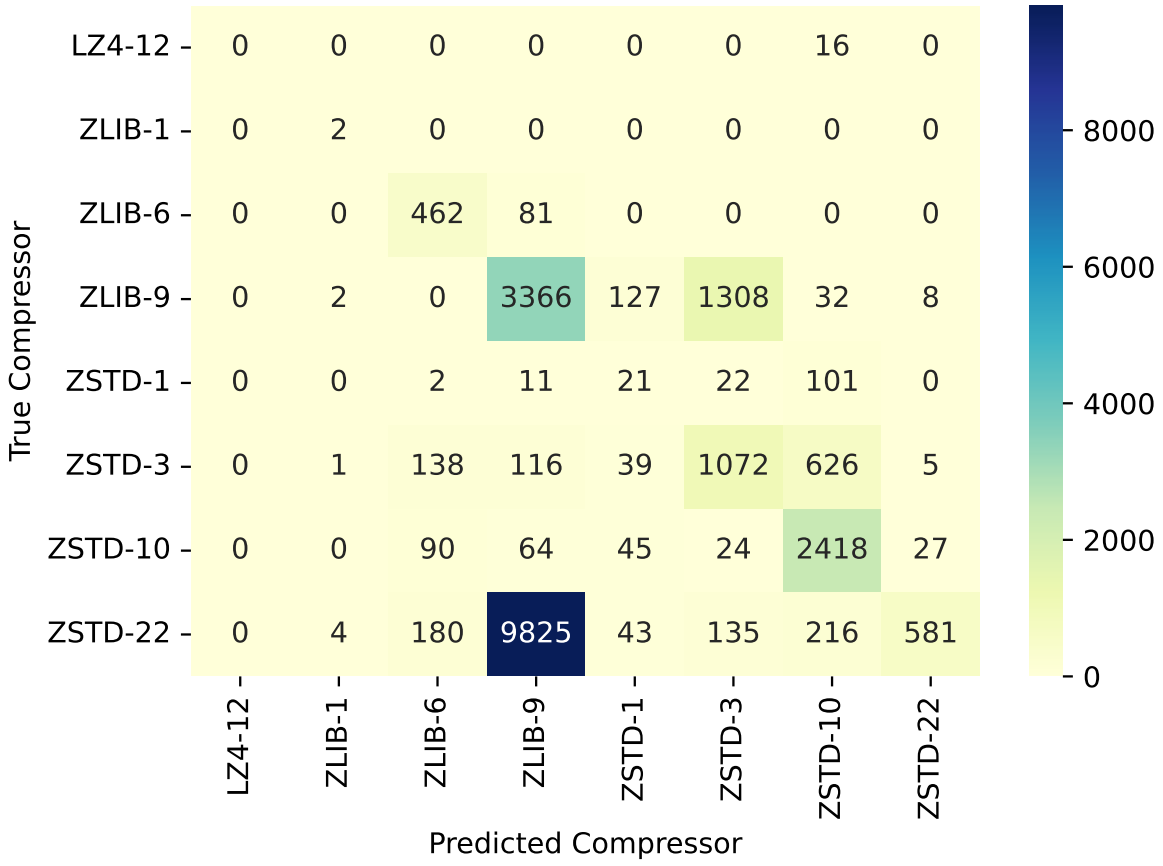


Figure 5.18: Confusion matrix of the model trained on limited input data

5.5 Findings

In this chapter, two main components of the suggested approach of predicting compression algorithms according to encountered data were evaluated. The first component includes the data collection and measuring architecture. In addition, the second component consists of the training of the machine learning model and the inferencing during runs of the ICON application.

As discussed, the essential information on which further insights are based on depends on the measured compression metrics. This includes the compression rate, compression rate per time, compression speed, and decompression speed. Those measurements were performed for the compressors LZ4, ZSTD, and ZLIB. Furthermore, a representative collection of compression levels/factors was chosen. Altogether this results in the following most significant findings:

- Depending on the data, significant fluctuations in the measurements are observed. This is especially noticeable for the compression rate metric.
- Not every metric needs to be evaluated dynamically. For example, the LZ4-fast-17 compressor outperforms any other compressor in the compression speed discipline.
- The compression rate metric can be predicted with the most accuracy in these experiments. Furthermore, certain distinctions within the encountered data and the great accuracy of the underlying measurements make this metric ideal for machine learning approaches.
- The number of observed simulation iterations and the length of the data used for training and inferencing greatly influence the overall prediction accuracy.

It is also important to note that those results relate to the application tested during evaluation. As discussed, the application's actual data significantly influences the ideal compression algorithm, and the gained insights do not necessarily apply to other applications.

6 Future work

In this chapter, details on how to improve the introduced architecture is given. This includes the data collection and inferencing library, as well as the currently used neural network techniques. Furthermore, ideas and concepts for further and continuous usage are reflected.

Integrating additional compression algorithms

The architecture introduced in Chapter 4 has been shown to provide a flexible and easy-to-use interface for integrating additional compression algorithms. The current interface did not limit the usage of any algorithms and new variants should be integrated. Currently, the available algorithms include LZ4, ZSTD and ZLIB, which are also available in the Linux kernel. This is geared towards a possible use of a decision component within parallel file systems like Lustre. Further benefits of such an approach are highlighted in [4], where performance implications are shown depending on whether the compression is applied on the client or on the storage node. For this potential use case, the current selection provides valuable insights into the behavior of data-aware compression.

Additional compression algorithms might include:

- *Brotli*[1]: Based on LZ77 and provides a dictionary of common web-related texts. While probably being too specific for HPC-related applications, the optimization for low-resource devices might be promising for busy HPC environments. At least the fundamental concepts should be evaluated.
- *Zopfli*[2]: Preferably used for data that is compressed only once and read multiple times. Achieves a higher compression rate than ZLIB, but at the expense of a longer runtime.
- *Blosc*[3]: Designed to be faster than `memcpy()` and to perform well on binary data. Uses a blocking technique which divides the input into blocks that fit into the cache.

This is not an exclusive list. Any compression algorithm that might be of interest should

[1] `https://github.com/google/brotli`
[2] `https://github.com/google/zopfli`
[3] `https://www.blosc.org`

be straightforward to evaluate. The lzbench[4] benchmark provides a promising basis for further compressors that might be worth analyzing. Furthermore, algorithms that were created for a specific task could be added. This has proven to be beneficial for specific use cases, as shown in Section 3.1.1.

Evaluate additional applications

The current approach is only evaluated by performing experiments on a single application. Among others, a promising observation has been to see the change of a written variable over time. Those behaviors should be examined with other applications, too. Furthermore, additional intercepted I/O could help to train a more balanced network that generalizes to a more application-independent model.

Integrate decision component into I/O

The current architecture is limited to evaluating the concept of training a neural network based on data encountered during I/O and making predictions based on this data. Intercepting MPI-IO-related calls is not feasible for real-world usage as when the data is compressed at this stage, the exact alignment during all further data usage has to be maintained. To mitigate this, the current architecture could be implemented within an I/O library like HDF5. Furthermore, the prediction of which compression algorithm should be used could be passed to the file system where compression could take place on a block level, independent of the access pattern.

Test additional machine learning techniques

While PyTorch is used to train a neural network in the current architecture, this is not a requirement for further training, as the model is imported for inferencing using the vendor-neutral ONNX format. Therefore, it is possible to use arbitrary components, either by continuing using PyTorch or by using any other library supporting this format.

Currently, a neural network is used. However, machine learning provides a vast number of alternative approaches. While also more complex networks can be evaluated, even simpler methods might also be worthwhile. For example, as discussed in [55], random forests are “not sensitive to over-fitting”.

[4] `https://github.com/inikep/lzbench`

7 Conclusion

In this chapter, the achievements of the thesis are recapped and evaluated against the initial goals outlined in the beginning.

This thesis aimed to provide an approach to data-aware compression for HPC using machine learning. The main contribution to this goal was to develop an architecture capable of relating the overall content of the data written during HPC application runs with a fitting compression algorithm. In addition, it was outlined that further insights into the behavior of the chosen compression algorithms are needed. These insights were also required to train a supervised machine learning model. Using this model, it was possible to predict ideal compression algorithms and associated levels using only the data available during I/O. The accuracy of said approach varies depending on the intended performance characteristics required. The current approach works especially well when the compression rate is of concern. Furthermore, a significant challenge was to collect appropriate training data due to imbalances encountered during the training phase.

Nevertheless, the introduced architecture has been flexible enough to accommodate various compression algorithms easily. Furthermore, the machine learning pipeline can be easily adapted to alternative machine learning approaches. The current abstraction between the training in modern and widely used data science frameworks and the integration into the shared library during inferencing is capable of executing a variety of alternative model architectures. The source code, as well as the entire training data, is available at [56]. This is intended to encourage further experiments and to improve the given architecture.

Finally, an earlier version of these contributions was already published as part of the CHEOPS'22 workshop [57]. This shows how relevant research in compression-related disciplines is, and new approaches are welcomed.

Bibliography

[1] ETP4HPC. *Strategic Research Agenda 3 for High Performance Computing in Europe*. en. 2017. URL: https://www.etp4hpc.eu/pujades/files/SRA%203.pdf (visited on 04/30/2022).

[2] Michael Malms, Marcin Ostasz, Maike Gilliot, Pascale Bernier-Bruna, Laurent Cargemel, Estela Suarez, Herbert Cornelius, Marc Duranton, Benny Koren, Pascale Rosse-Laurent, María S. Pérez-Hernández, Manolis Marazakis, Guy Lonsdale, Paul Carpenter, Gabriel Antoniu, Sai Narasimhamurthy, André Brinkman, Dirk Pleiter, Adrian Tate, Jens Krueger, Hans-Christian Hoppe, Erwin Laure, and Andreas Wierse. *ETP4HPC's Strategic Research Agenda for High- Performance Computing in Europe 4*. Mar. 2020. DOI: 10.5281/zenodo.4605344. URL: https://doi.org/10.5281/zenodo.4605344.

[3] Janine C. Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, Philippe Pebay, David Thompson, Hongfeng Yu, Fan Zhang, and Jacqueline Chen. "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ISSN: 2167-4337. Nov. 2012, pp. 1–9. DOI: 10.1109/SC.2012.31.

[4] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. "Data Compression for Climate Data". In: *Supercomputing Frontiers and Innovations*. Volume 3, Number 1 (June 2016). Ed. by Jack Dongarra and Vladimir Voevodin, pp. 75–94. DOI: https://doi.org/10.14529/jsfi160105. URL: http://superfri.org/superfri/article/view/101.

[5] Yevhen Alforov, Thomas Ludwig, Anastasiia Novikova, Michael Kuhn, and Julian Kunkel. "Towards Green Scientific Data Compression Through High-Level I/O Interfaces". en. In: *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Lyon, France: IEEE, Sept. 2018, pp. 209–216. ISBN: 978-1-5386-7769-8. DOI: 10.1109/CAHPC.2018.8645921. URL: https://ieeexplore.ieee.org/document/8645921/ (visited on 04/30/2022).

[6] Arnab K. Paul, Olaf Faaland, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Ali R. Butt. "Understanding HPC Application I/O Behavior Using System Level Statistics". In: *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. ISSN: 2640-0316. Dec. 2020, pp. 202–211. DOI: 10.1109/HiPC50609.2020.00034.

[7] R. Thakur, W. Gropp, and E. Lusk. "An abstract-device interface for implementing portable parallel-I/O interfaces". In: *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*. ISSN: 1088-4955. Oct. 1996, pp. 180–187. DOI: 10.1109/FMPC.1996.558080.

[8] LiuYing. "Lustre Technical White Paper - Lustre ADIO collective write driver". en. In: (2008), p. 16. URL: https://wiki.lustre.org/images/6/65/Lustre_ADIO_Driver_Whitepaper_0926.pdf.

[9] *NetCDF Classic and 64-bit Offset File Formats | Earthdata*. en. URL: https://earthdata.nasa.gov/esdis/esco/standards-and-references/netcdf-classic/ (visited on 05/06/2022).

[10] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (July 1948), pp. 379–423. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[11] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (Sept. 1952). Conference Name: Proceedings of the IRE, pp. 1098–1101. ISSN: 2162-6634. DOI: 10.1109/JRPROC.1952.273898.

[12] J. J. Rissanen. "Generalized Kraft Inequality and Arithmetic Coding". In: *IBM Journal of Research and Development* 20.3 (May 1976). Conference Name: IBM Journal of Research and Development, pp. 198–203. ISSN: 0018-8646. DOI: 10.1147/rd.203.0198.

[13] Paul G. Howard and Jeffrey Scott Vitter. "Practical Implementations of Arithmetic Coding". en. In: *Image and Text Compression*. Ed. by James A. Storer. Vol. 176. Series Title: The Kluwer International Series in Engineering and Computer Science. Boston, MA: Springer US, 1992, pp. 85–112. ISBN: 978-1-4613-6598-3 978-1-4615-3596-6. DOI: 10.1007/978-1-4615-3596-6_4. URL: http://link.springer.com/10.1007/978-1-4615-3596-6_4 (visited on 05/03/2022).

[14] Ian H Witten, Radford M Neal, and John G Cleary. "ARITHMETIC CODING FOR DATA COMPRESSION". en. In: *Communications of the ACM* 30.6 (1987), p. 21. URL: https://web.stanford.edu/class/ee398a/handouts/papers/WittenACM87ArithmCoding.pdf (visited on 05/03/2022).

[15] Asadollah Shahbahrami, Ramin Bahrampour, Mobin Sabbaghi Rostami, and Mostafa Ayoubi Mobarhan. "Evaluation of Huffman and Arithmetic Algorithms for Multimedia Compression Standards". In: *arXiv:1109.0216 [cs, math]* (Aug. 2011). arXiv: 1109.0216. URL: http://arxiv.org/abs/1109.0216 (visited on 05/03/2022).

[16] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (May 1977). Conference Name: IEEE Transactions on Information Theory, pp. 337–343. ISSN: 1557-9654. DOI: 10.1109/TIT.1977.1055714.

[17] Heiko Schwarz. "Dictionary-based Coding". en. In: (), p. 37. URL: http://iphome.hhi.de/schwarz/assets/dc/07-DictionaryCoding.pdf.

[18] P. Deutsch. *RFC1951: DEFLATE Compressed Data Format Specification Version 1.3*. USA, 1996.

[19] Yann Collet. *LZ4 - Frame format*. original-date: 2014-03-25T15:52:21Z. May 2022. URL: https://github.com/lz4/lz4/blob/ce8ee024b240befac4ca8ab12c6cd812f4a7e38b/doc/lz4_Frame_format.md (visited on 05/02/2022).

[20] Yann Collet. *LZ4 - Block format*. original-date: 2014-03-25T15:52:21Z. May 2022. URL: https://github.com/lz4/lz4/blob/ce8ee024b240befac4ca8ab12c6cd812f4a7e38b/doc/lz4_Block_format.md (visited on 05/02/2022).

[21] Kyungsik Lee and LG Electronics. *LZ4 Compression and Improving Boot Time*. en. URL: https://events.static.linuxfound.org/sites/events/files/lcjpcojp13_klee.pdf (visited on 05/02/2022).

[22] Jarek Duda. "Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding". In: *arXiv:1311.2540 [cs, math]* (Jan. 2014). arXiv: 1311.2540. URL: http://arxiv.org/abs/1311.2540 (visited on 05/06/2022).

[23] Allan Jude. "Zstandard Compression in OpenZFS". en. In: (), p. 6. URL: https://freebsdfoundation.org/wp-content/uploads/2021/05/Zstandard-Compression-in-OpenZFS.pdf.

[24] Jean-loup Gailly and Mark Adler. *zlib Technical Details*. URL: https://zlib.net/zlib_tech.html (visited on 05/03/2022).

[25] Enzo Grossi and Massimo Buscema. "Introduction to artificial neural networks". In: *European journal of gastroenterology & hepatology* 19 (Jan. 2008), pp. 1046–54. DOI: 10.1097/MEG.0b013e3282f198a0.

[26] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". en. In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259 (visited on 03/30/2022).

[27] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65 6 (1958), pp. 386–408.

[28] F. Rosenblatt. *The perceptron - A perceiving and recognizing automaton*. Tech. rep. 85-460-1. Ithaca, New York: Cornell Aeronautical Laboratory, Jan. 1957.

[29] Paul Werbos and Paul John. "Beyond regression : new tools for prediction and analysis in the behavioral sciences /". In: (Jan. 1974).

[30] Carter Chiu and Justin Zhan. "An Evolutionary Approach to Compact DAG Neural Network Optimization". In: *IEEE Access* 7 (2019), pp. 178331–178341. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2954795.

[31] Raúl Rojas. "Perceptron Learning". In: *Neural Networks: A Systematic Introduction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 77–98. ISBN: 978-3-642-61068-4. DOI: 10.1007/978-3-642-61068-4_4. URL: https://doi.org/10.1007/978-3-642-61068-4_4.

[32] MARIAN P. Kazmierkowski. "CHAPTER 10 - Neural Networks and Fuzzy Logic Control in Power Electronics". en. In: *Control in Power Electronics*. Ed. by MARIAN P. Kazmierkowski, R. Krishnan, and FREDE Blaabjerg. Academic Press Series in Engineering. Burlington: Academic Press, Jan. 2002, pp. 351–418. ISBN: 978-0-12-402772-5. DOI: 10.1016/B978-012402772-5/50011-9. URL: https://www.sciencedirect.com/science/article/pii/B9780124027725500119 (visited on 05/07/2022).

[33] D. Veit. "2 - Neural networks and their application to textile technology". en. In: *Simulation in Textile Technology*. Ed. by D. Veit. Woodhead Publishing Series in Textiles. Woodhead Publishing, Jan. 2012, pp. 9–71. ISBN: 978-0-85709-029-4. DOI: 10.1533/9780857097088.9. URL: https://www.sciencedirect.com/science/article/pii/B978085709029450002X (visited on 05/07/2022).

[34] Sepp Hochreiter. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6 (Apr. 1998), pp. 107–116. DOI: 10.1142/S0218488598000094.

[35] Deloula Mansouri, Xiaohui Yuan, and Abdeldjalil Saidani. "A New Lossless DNA Compression Algorithm Based on A Single-Block Encoding Scheme". In: *Algorithms* 13.4 (2020). ISSN: 1999-4893. DOI: 10.3390/a13040099. URL: https://www.mdpi.com/1999-4893/13/4/99.

[36] *GenBank and WGS Statistics*. URL: https://www.ncbi.nlm.nih.gov/genbank/statistics/ (visited on 04/11/2022).

[37] T Matsumoto, K Sadakane, and H Imai. "Biological sequence compression algorithms". en. In: *Genome Inform Ser Workshop Genome Inform* 11 (2000), pp. 43–52.

[38] Nour S. Bakr and Amr A. Sharawi. "Improve the compression of bacterial DNA sequence". In: *2017 13th International Computer Engineering Conference (ICENCO)*. 2017, pp. 286–290. DOI: 10.1109/ICENCO.2017.8289802.

[39] Shengwang Du, Junyi Li, and Naizheng Bian. "A compression method for DNA". In: *PLOS ONE* 15.11 (Nov. 2020), pp. 1–8. DOI: 10.1371/journal.pone.0238220. URL: https://doi.org/10.1371/journal.pone.0238220.

[40] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. "Reference Sequence Construction for Relative Compression of Genomes". In: *String Processing and Information Retrieval*. Ed. by Roberto Grossi, Fabrizio Sebastiani, and Fabrizio Silvestri. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 420–425. ISBN: 978-3-642-24583-1.

[41] Nguyen Anh, Hoang Dung, and Nguyen Nhung. "Adaptive Cross-Correlation Compression Method in Lossless Audio Streaming Compression". In: *International Journal of Engineering and Applied Sciences (IJEAS)* 6 (Mar. 2019). DOI: 10.31873/IJEAS/6.3.2019.30.

[42] Nathanael Hübbe, Al Wegener, Julian Martin Kunkel, Yi Ling, and Thomas Ludwig. "Evaluating Lossy Compression on Climate Data". In: *Supercomputing*. Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 343–356. ISBN: 978-3-642-38750-0.

[43] Xiangyu Zou, Tao Lu, Sheng Di, Dingwen Tao, Wen Xia, Xuan Wang, Weizhe Zhang, and Qing Liao. "Accelerating Lossy Compression on HPC Datasets via Partitioning Computation for Parallel Processing". In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2019, pp. 1791–1797. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00246.

[44] Tao Lu, Qing Liu, Xubin He, Huizhang Luo, Eric Suchyta, Jong Choi, Norbert Podhorszki, Scott Klasky, Mathew Wolf, Tong Liu, and Zhenbo Qiao. "Understanding and Modeling Lossy Compression Schemes on HPC Scientific Data". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 348–357. DOI: 10.1109/IPDPS.2018.00044.

[45] Yoshua Bengio. "Learning Deep Architectures for AI". In: *Foundations and Trends® in Machine Learning* 2.1 (2009), pp. 1–127. ISSN: 1935-8237. DOI: 10.1561/2200000006. URL: http://dx.doi.org/10.1561/2200000006.

[46] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. "Lossy Image Compression with Compressive Autoencoders". In: *arXiv:1703.00395 [cs, stat]* (Mar. 2017). arXiv: 1703.00395. URL: http://arxiv.org/abs/1703.00395 (visited on 04/20/2022).

[47] *Deep autoencoders for ATLAS data compression*. URL: https://hepsoftwarefoundation.org/gsoc/2020/proposal_ATLASMLcompression.html (visited on 04/20/2022).

[48] Tong Liu, Jinzhen Wang, Qing Liu, Shakeel Alibhai, Tao Lu, and Xubin 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.

[49] Michela Massi. *Autoencoder schema*. 2019. URL: https://commons.wikimedia.org/wiki/File:Autoencoder_schema.png (visited on 04/20/2022).

[50] Michael Kuhn, Julius Plehn, Yevhen Alforov, and Thomas Ludwig. "Improving Energy Efficiency of Scientific Data Compression with Decision Trees". In: *ENERGY 2020: The Tenth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*. Lisbon, Portugal: IARIA XPS Press, Sept. 2020, pp. 17–23. ISBN: 978-1-61208-788-7. URL: https://www.thinkmind.org/index.php?view=article&articleid=energy_2020_1_40_30038.

[51] *MPI: A Message-Passing Interface Standard*. en. June 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[52] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[53] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *arXiv:1609.04836 [cs, math]* (Feb. 2017). arXiv: 1609.04836. URL: http://arxiv.org/abs/1609.04836 (visited on 05/08/2022).

[54] Francesc Alted. *Breaking Down Memory Walls*. en. URL: https://www.blosc.org/docs/Breaking-Down-Memory-Walls.pdf (visited on 04/29/2022).

[55] Ned Horning. "Random Forests : An algorithm for image classification and generation of continuous fields data sets". In: 2010.

[56] Julius Plehn. *Data-Aware Compression for HPC using Machine Learning - Training data and tools*. Version 1.0.0. May 2022. DOI: 10.5281/zenodo.6506394. URL: https://doi.org/10.5281/zenodo.6506394.

[57] Julius Plehn, Anna Fuchs, Michael Kuhn, Jakob Lüttgau, and Thomas Ludwig. "Data-Aware Compression for HPC Using Machine Learning". In: *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. CHEOPS '22. Rennes, France: Association for Computing Machinery, 2022, pp. 8–15. ISBN: 9781450392099. DOI: 10.1145/3503646.3524294. URL: https://doi.org/10.1145/3503646.3524294.

Appendices

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang M. Sc. Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift