TU WIEN Informatics

Versionierung der Modellzustandsentwicklung von Online Machine-Learning Modellen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christoph Eitenberger, BSc
Matrikelnummer 01248272

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.univ.Prof. Dr. Andreas Rauber

Wien, 7. Dezember 2023

Christoph Eitenberger　　　Andreas Rauber

TU WIEN Informatics

Versioning of Model State Evolution of Machine Learning Models in Online Learning Settings

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Christoph Eitenberger, BSc
Registration Number 01248272

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.univ.Prof. Dr. Andreas Rauber

Vienna, 7th December, 2023

Christoph Eitenberger — Andreas Rauber

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Christoph Eitenberger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. Dezember 2023

Christoph Eitenberger

Acknowledgements

Firstly, I would like to thank my advisor Andreas Rauber for his time, persistent interest, very extensive feedback, ideas and long counseling meetings, which I have learnt much from. Secondly, I would like to thank Elisabeth Arnold for embellishing the explanatory graphics, her design and editing advice and ongoing support. Finally, I would like to thank the TU Vienna and its staff for being outstanding, inspiring and helpful - from which I have gained a lot.

Abstract

Deep neural networks are being increasingly utilized for making significant decisions. Understanding these decisions can be required by the GDPR's "right to explain" or other legal claims. It may be necessary to restore previously used versions as these neural networks commonly require updating over time, e.g., due to data evolution. Tracking versions in an online learning setting can be challenging as newer versions must be frequently saved while minimizing their impact on the learning process. The optimization and measurement of a version system for an online setting have yet to be explored.

This work proposes a novel Versioning System for Online Learning systems (VSOL) that can be easily integrated into existing machine learning workflows without requiring a modification of the learning process. The VSOL is integrated as a framework specific callback for the Keras Machine Learning Framework, showcasing the effortless integration into a ML Framework. Several compression approaches suitable for fast execution and requiring minimal storage space were designed and contextualized, bringing together different research fields addressing compression. The VSOL was tested under constant virtual data drift, simulated through introducing an unseen label from a static, not online specific data set. A convolutional as well as a long short-term memory neural network were evaluated where their drift was aligned by using specific parameters and a novel measurement unit.

While executing faster than the learning time of one data batch and without affecting the accuracy, the resulting compression ratio lies between 12.5 to 30.0. Accepting a slower execution and an accuracy decrease of 1% lead to a compression ratio between 52.2 and 129.7. This thesis shows that a versioning system can be easily integrated and achieve a competitive compression ratio while satisfying the constraints of an online learning setting.

Keywords Deep Learning · Model Versioning · Online Learning · Model Compression · Drift Simulation · Traceability in Deep Learning · Model Deployment

Kurzfassung

Mehrschichtige neuronale Netze werden zunehmend für wichtige Entscheidungen eingesetzt. Die Nachvollziehbarkeit dieser Entscheidungen kann aufgrund des in der Datenschutz-Grundverordnung verankerten "Anspruchs auf Erläuterung von Entscheidungen" oder anderer Rechtsansprüche erforderlich sein. Es kann sich die Notwendigkeit ergeben, zuvor verwendete Versionen wiederherzustellen, da diese neuronalen Netze in der Regel im Laufe der Zeit aktualisiert werden müssen, z.B. aufgrund einer Datenveränderung. Die Versionsnachverfolgung in einer Online-Lernumgebung kann eine Herausforderung darstellen, da neuere Versionen häufig gespeichert, ihre Auswirkungen auf den Lernprozess jedoch minimiert werden müssen. Die Optimierung und Messung eines Versionssystems für eine Online-Umgebung ist bisher unerforscht.

In dieser Diplomarbeit wird ein neuartiges 'Versionierungssystem für Online-Lernsysteme' (VSOL) vorgeschlagen, das leicht in bestehende Workflows für maschinelles Lernen integriert werden kann, ohne, dass eine Änderung des Lernprozesses erforderlich ist. Das VSOL wird als frameworkspezifischer Callback für das Keras Machine Learning Framework integriert, um die einfache Integration in ein ML Framework zu demonstrieren. Verschiedene Kompressionsansätze, die für eine schnelle Ausführung geeignet sind und nur minimalen Speicherplatz benötigen, wurden entworfen und angepasst, wobei verschiedene Forschungsbereiche, die sich mit Kompression befassen, zusammengebracht wurden. Das VSOL wurde unter konstantem virtuellen Datendrift getestet, der durch die Einführung eines unbekannten Labels aus einem statischen, nicht online-spezifischen Datensatz simuliert wurde. Es wurden sowohl ein faltendes neuronales Netzwerk als auch ein neuronales Netzwerk mit langem Kurzzeitgedächtnis evaluiert, wobei deren Drift durch die Verwendung spezifischer Parameter und einer neuartigen Messeinheit aneinander angeglichen wurde.

Bei einer Ausführungszeit, die unter der Lernzeit eines Batches liegt, und ohne die Genauigkeit zu beeinträchtigen, erzielt das VSOL eine Kompressionsrate zwischen 12,5 und 30,0. Wird eine langsamere Ausführung und eine Verringerung der Genauigkeit um 1% in Kauf genommen, ergibt sich eine Kompressionsrate zwischen 52,2 und 129,7. Diese Diplomarbeit zeigt, dass ein Versionierungssystem leicht in Online-Lernumgebungen integriert und dabei eine kompetitive Kompressionsrate erreicht werden kann, während es gleichzeitig deren Einschränkungen berücksichtigt.

Schlagworte Mehrschichtiges Lernen · Modellversionierung · Online-Lernen · Modellkompression · Driftsimulation · Nachvollziehbarkeit bei mehrschichtigen Lernen · Modellbereitstellung

Contents

CHAPTER 1

Introduction

1.1 Motivation & Problem Statement

As models in an online machine learning setting evolve continuously over time, it is important to be able to restore earlier deployed versions. If a model degrades or errors are detected, a rollback to an older version can be the only fast solution. This can be caused by data meant to poison the model. In this case, such older saved versions can even help to identify the bad data and exclude it for further training. Also, for compliance it is important to be able to find out why a model came to a certain decision at a certain point in time. Such compliance requirements can arise from GDPR's "right to explain" or industries where products have to be traceable and follow strict transparency guidelines judged by government institutions such as in the medical sector. Older model versions can give an insight into how and to what degree which data impacted the model to what degree. Such version control similar systems are therefore relevant for data scientists creating the models as well as operations staff responsible for deploying such models.

All of the mentioned aspects gain in importance in an online learning setting. Firstly, the new data is less controllable due to the fact that no human is manually evaluating the used data in contrast to a pre-trained model setting where the training data is usually carefully evaluated prior to use. Therefore, backtracking model issues becomes more important. Additionally, the amount and frequency of new data and the resulting new deployed model versions can be harder to manage then in an experimental environment where the used data is fixed.

Current model saving systems described in the scientific literature (see Section 2.1), to our knowledge, mostly describe their applicability before the deployment phase. Such systems include documenting the experiment phase for finding a fitting model set-up or archiving models to be redistributed over a website for further use. Hence, important properties for an online learning setting are often disregarded, such as the execution speed, the easy of integration and the memory impact.

The system proposed by this diploma thesis should be able to save all versions deployed by an online learning system, if feasible. The most important aspect will be to impact the learning process itself as little as possible when saving the individual versions, i.e. states of the model. Otherwise, it affects its main purpose in the first place. This includes not altering the model training process, which also reduces the effort for integration. To measure these performance requirements and to demonstrate the ease of integration into an actual learning flow, it should be integrated in an existing machine learning framework and thus be implemented as a framework extension. However, the resulting findings should be applicable for other frameworks. To be able to analyze a former decision, it must be possible to reconstruct the used version at that point in time. From a user's perspective, recreating the version should be as easy as providing that timestamp. Online learning systems can be heterogeneous e.g. having a changing model architecture. Different to other systems meant to capture such diverse model layouts, this system would only save the weights of an established and fixed model layout. Distributing a saved version is a related topic when talking about versioning which will only be discussed in theory and not implemented or evaluated. The source code of both the resulting Versioning System for Online Learning systems (VSOL) and its evaluation is available on GitHub[1].

In summary, the main contributions of this thesis are the following:

- Designing and contextualizing different appropriate neural network compression algorithms
- Implementing the designed compression algorithms
- Evaluating and combining implemented algorithms, resulting in fitting and pre-configured combinations for different requirements
- Designing and implementing an automatic online learning drift simulation system and a matching evaluation system
- Integrating the Versioning System for Online Learning systems and drift simulation into the machine learning framework Keras to demonstrate the ease of integrated
- Bringing together several separate research areas in the context of online learning

1.2 Research Question

Main Research Question *What is an appropriate way to store the sequence of states of an evolving deep learning model in an online learning setting?*
Saving the model state makes past model decisions traceable and old states can be reused for other models. Current system such as [MLDD17a] or [CCD$^+$20] focus on documenting

[1]https://github.com/christopheitenberger/VSOL

the model creation process during the initial model experimentation phase. Therefore, they are not optimized or tested to be applied for or integrated into an online learning process.

Finding an 'appropriate way to store ... the model in an online learning setting', as mentioned in the main research question, requires that characteristics specific to online learning compared to offline learning are present in during evaluation of this 'way to store the model'. As prerequisite, the online specific characteristics during the evaluation are not treated as a separate research question, as the RQs only address the 'way to store the model' itself, but still will be addressed in this thesis.

Using such a State Saving Solution, as referred to in this section, is only feasible if the learning process and therefore the main benefit of an online model is only mildly impacted. 'Appropriate', in summary, means having minimal to no impact on the model's error rate (such as accuracy), on the duration of the training process and, on the storage overhead, while requiring low effort for reconstructing a specific state. The following sub questions will highlight these aspects and how they will be evaluated:

RQ1: Impact Reduction on Online Learning *To what extent can the performance impact of saving intermediate model states during online learning be minimized?*
Saving the state is only feasible if the main benefit of having a model that can react quickly to new data is not affected by the saving process. As mentioned above, most systems focus on saving the state of models during the exploratory phase of creating a model and are not deeply integrated in a machine learning framework. It remains unclear wether they are suitable for such use.
To evaluate this question, the processing time for saving the models with the State Saving Solution after each online learning iteration will be measured. An increased processing time delays the training process which increases the reaction time to new data.
When loading each saved model through the State Saving Solution in a second evaluation phase, which simulates the retrieval of the weights, after they have been stored in the first phase, the error rate and decompression time will be measured. The results will be compared against different baselines. Deviating from the baseline error rate would be the most undesirable outcome. But to evaluate the full potential of the State Saving Solution, different thresholds besides zero are evaluated. Although the decompression time is mentioned in the description of the measurement above, it is captured by 'RQ4: Reduce Retrieval Time' (Section 1.2) since it is not as important as the error rate deviation or processing time.

RQ2: Reduce Required Storage *To what extent can the storage amount of the saved model states be reduced?*
Considering that a version of an already large model can be created every second, the resulting storage costs can be significant and influence the decision on using such a tool. [MLDD17a] already implements a difference-based saving system but it is not lossless

and can be further improved to use even less storage. Other float compression algorithms exist [CAB18] but besides the implementation in the former mentioned paper, none were tested or deployed in this setting to our knowledge. Although this is not the first listed research question, most of the effort will be invested in optimizing it while still fulfilling 'RQ1: Impact Reduction on Online Learning' (Section 1.2).
This question is evaluated by measuring the size of the saved models through the State Saving Solution, resulting from the evaluation process described in 'RQ1: Impact Reduction on Online Learning' (Section 1.2). Different types of machine learning applications will be evaluated to ensure that the State Saving Solution works for not just a specific application, losing its general applicability. This compression ratio will be compared against compressing the saved model from the framework with a lossless compression algorithm like Zstandard and the results of [MLDD17a].

RQ3: Reduce Integration Effort *To what extent can the integration process of such a model state saving system for a concrete machine learning framework be simplified for a data scientist?*
To keep the difficulty threshold of integrating the State Saving Solution as low as possible for the end user like a data scientist or an application developer, it should be easy to add it to an existing project. Other systems mentioned in 'RQ1: Impact Reduction on Online Learning' (Section 1.2) need an extensive setup procedure such as managing an associated database, or excluding the integration into the machine learning framework. An additional database can also impact the performance of the learning process. The learning process should also not be changed since this would require a more extensive integration process and potential adaptions to the learning process itself from a data science to handle the caused deviations. [PDMM21] for example changes the training weights of each batch, as described for the specific compression approach 'Top-K' (Section 2.2.2). To the our knowledge, research about such kind extension system with low setup effort does not yet exist besides the conventional snapshot system of the machine learning framework. Also, the other systems can only be executed manually while the State Saving Solution has to execute automatically.
The ease of use will be self assessed by commenting on a tutorial code section which shows how to integrate the State Saving Solution into a machine learning pipeline. Besides highlighting the benefits and drawbacks, the quantitative measures of additional parameters and lines of code will be analyzed. Also, the requirements of not changing the learning process and not requiring external resources, as listed in the paragraph above, must be fulfilled.

RQ4: Reduce Retrieval Time *To what extent can the execution time of the retrieval process of a former saved model state be reduced?*
Data scientists or other actors who need to retrieve a saved version, want this process to happen in a timely manner since the State Saving Solution otherwise becomes unusable.

This question is in direct conflict to 'RQ2: Reduce Required Storage' (Section 1.2) since a higher compression increases the retrieval time. RQ2 remains more important but this measurement should not get out of hand and therefore also be evaluated. The papers cited in RQ2 already highlight this tradeoff. [MLDD17a] mentions this tradeoff to be a main concern. But since RQ2 will lean to a more compression emphasized implementation and the mentioned paper did not explore this tradeoff combination, this question will be new in the area of model compression.
The retrieval will be evaluated by measuring the time until the model is loaded from storage and ready to be used, assessed for all saved models. This decompression measurement will be compared to a fix maximum loading time.

1.3 Methodology

Literature Research A systematic literature research is conducted, as described below. In the first step of the literature research, we acquire a broad overview to understand the presented topics and to find a first set of appropriate words for a search query. This is accomplished by reading overview papers and popular papers of the given fields and some of their references. The resulting search queries of the first step are then used to find new and more specific papers. They can the help to refine the given search queries. This process is repeated until a saturation of high quality and recent set of papers is found.

Gaining an overview over current model versioning systems is important to assess how they fulfill the research questions 1-4. This can help to improve on them or even show how the proposed system is novel in this field.

Assessing the specifics of online learning and how online learning is accomplished is also required to create a realistic evaluation conditions and an appropriate neural network learning method. Hence, it is also required for all research questions since it is the basis for any measurement and the proper integration into a machine learning system.

Finding existing methods to compress floats and especially neural networks in a continuous setting helps to answer RQ 1, 2 and 4. Different to the previous research topics, this should not be evaluated to the full extend since this is a very broad and extensive field and the Versioning System for Online Learning systems should only include several approaches to show how they could interact instead of featuring only the most cutting edge approaches of all fields.

Since the topics 'online learning' and 'compression of neural network' are very broad, the search queries should be broad and versatile to find the most promising papers of the field instead of going in depth for a specific topic. The opposite is applied for the research field 'model versioning systems'

Developing a Software Artifact One result of this thesis should be the Versioning System for Online Learning systems, already described in more detail in 'Motivation & Problem Statement' (Section 1.1). In order to measure the degree to which it fulfills the requirements described in the research questions (see Section 1.2), it has to be materialized as a software artifact.

Extending a Framework The software artifact should not impact the performance of the learning process (RQ1) and should be easy to use (RQ3). Therefore, the software artifact should be an extension of a framework.

Prototyping Different performance requirements (RQ 1,2,4) call for applicability of a range of ideas, e.g., different compression algorithms mentioned in 'Compressing Model Versions' (Section 2.2). To evaluate the applicability of the ideas, prototyping will be used during the software development process. This methodology strongly relates to benchmarking since the performance of a prototype can only be evaluated through it.

Evaluation of the Framework Extension

The following methodologies show to which degree the resulting framework extension fulfills the research questions.

Benchmarks Benchmarks are used to measure the degree of the fulfillment of several research questions (RQ 1,2,4). Most of the documentation regarding the research questions deals with the benchmarking details. For this thesis, it also enables the verification of the prototype methodology.

Baselines for Benchmarking Baselines for benchmarking help to assess the impact of an improvement (RQ 1,2,4). A baseline becomes especially important if there are no other measurements from other scientific papers to compare it to (RQ1).

Simulation Simulation is used to mimic the process of online learning and ensures that the benchmarks are comparable to a real-world setting (RQ 1,2,4). In an online setting, the new data can diverge from the previous data. This is important to simulate since more diverse data can change the model more strongly and lead to uneven changes per batch which therefore also affects the compression (RQ2).

CHAPTER 2

Related Work

2.1 Model Recreation

The model recreation systems discussed in this section cover different use cases and highlights how VSOL differs from them.

Metadata System A 'Metadata System' tries to assist data scientists during the initial experiment phase when trying to find the right model with the right hyper parameters [MD18, SBK$^+$17, VSL$^+$16]. They have a variety of features such as storing the different setups, models script, results and other settings and assist in finding old experiments. The main difference to the proposed Versioning System for Online Learning systems (VSOL) is that they are design to be used before deployment and support a variety of setups. The VSOL should be used during the production phase, specialized for an online setting and therefore, consider the performance impact on the training process. Since recreating the model during the experiment phase is not time critical, this was no concern in the existing systems and not treated in their design but is a relevant and therefore considered metric for the proposed system. Furthermore, they are not easily integrable and often require an extensive setup e.g. running a database or a running service. The VSOL should be easily integrable into an existing machine learning framework with minimal effort and therefore should improve the usability considerably.

Retraining for Model Recreation System A Retraining for Model Recreation System (RMRS) combines a 'Metadata System' with storing all the used training data and any further information required to full reproduce the executed learning process. To recreate a previous version, the training process between a saved version, referred to as 'reset point', and the desired previous version is re-executed. Reset points are required to cap the required training steps to the desired version and is similar to the 'Delta Reset Point Creator' (Section 3.2.8) of the VSOL. This most obvious alternative approach to the proposed VSOL has, to our knowledge, not been previously explored. Being able to reproduce the learning process is not required by the VSOL since recreating a former version with the used settings and data would require retraining the model for all steps up to the version, which is time and energy-consuming, affecting 'RQ4: Reduce Retrieval Time' (Section 1.2). Since this would contain a 'Metadata System' it additionally has it's drawbacks affecting 'RQ3: Reduce Integration Effort' (Section 1.2). Further drawbacks of this approach, according to the findings of this thesis, are discussed in 'Drawbacks of Retraining for Model Recreation System' (Section 4.10).

Storing Model Versions [MLDD17b] is the only paper that covers versioning with compression of machine learning models. The used compression details will be depicted in 'Compressing Model Versions' (Section 2.2). It is similar to the proposed work of this thesis in that it also versions different machine learning models and tries to compress them. The paper also introduces a querying mechanism, enabling a SQL-like search for metadata. This has an impact on the compression ratio which then can be improved. Additionally, the paper focuses on lossy versions with a strong decline in the error rate. The VSOL can be configured to be lossless and any lossy configurations should only have a low impact on the error rate. The processing time of [MLDD17a] also remains unclear for an online learning setting. An additional requirement of the paper was to support changing model graphs. This is not in the scope of this thesis, focusing on the same network structure could lead to further improvements.

[DMRM19] focuses on the deployment of the model and discusses state of the art systems for model deployment. The VSOL does not implement or test the deployment of the model but discusses how the deployment could be integrated in 'Weight Deployment Scenarios' (Section 3.3).

[CCD+20] discusses newly added features to the machine learning lifecycle platform 'MLflow' such as a model registry. MLflow features a fairly easy integration to save model versions but requires a running service, has an unclear impact on the runtime, does not compress the versions and is not specialized for an online setting. Additionally, it does not directly feature the possibility to load models through a timestamp.

[CLC+19] developed a model repository which focuses on discoverability, evaluation and deployment. The authors of the paper acknowledge that such a system has to execute fast to be sensible in a real time setting and therefore measures the deployment and evaluation time of the models. To our knowledge, the performance impact of any of the other versioning or metadata systems was not measured. The processing time when saving a model was not measured and the integration process is more complex than for example using 'MLflow'. Also, the models are not compressed. According to the vague description, it should be possible to load models by the timestamp of the model.

2.2 Compressing Model Versions

To fulfill 'RQ2: Reduce Required Storage' (Section 1.2) a 2.1 would be fitting but since it affects the other RQs, as already discussed, only compressing the model is feasible to reduce the size of the high number of versions in an online learning setting.

The following section discusses several different research fields which can be used for compressing the model versions. Several of them will used and combined for the Versioning System for Online Learning systems. They are selected and discussed by their effect on 'RQ1: Impact Reduction on Online Learning' (Section 1.2) and 'RQ4: Reduce Retrieval Time' (Section 1.2).

2.2.1 General compression

To utilize the reduced entropy caused by some of the compression techniques, a general compression scheme is required to reduce the storage amount. [DKS19, p.84] compared the compression ratio and processing times of several popular compression libraries on different scientific data types, including floats. [MLDD17b] only uses zLib without considering its impact or other options.

2.2.2 Top-K

Top-K or gradient sparsification reduces the number of parameter changes of the weights to only the top absolute k percentage. Section 3.2.2 describes Top-k in more detail. Originally, it is used to reduce network traffic for training neural network in a distributed training setting. [LHM+20] summarizes and combines four methods to reduce the accuracy loss while using Top-k. [PDMM21] applied Top-k in an online learning setting to distribute the weights to remote machines only responsible for classification.

[CCB+18] uses a dynamic weight selection technique to raise the selected number of values depending on the Weight Change Degree and the former value on a per layer basis.

2.2.3 Float Compression

Quantization [AGL+17, SCG+22, PPA18, BNS19] describe the usage of quantization to reduce network traffic when distributing weight updates to remote machines. Quantization can increase the compression ratio drastically while still maintaining a high accuracy. Besides Top-k, quantization is a widely discussed method to reduce the data amount required to transmit a model.

Sparse Array Representation [SH15] developed a sparse array representation which only requires two arrays to represent the index of the non zero decimal values. Since Top-k deltas mostly consist of zeros, this algorithm can reduce the required memory considerably.

General Float Compression [KTF21, Lin14] and many others specialize on compression of various scientific float data sets. To the best of our knowledge, only some methods mentioned in these papers are combined with and adapted for neural networks.

Reduced Decimal Storage Formats [MLDD17b] compares several decimal storage formats such as 16-bit precision floats and quantization not specialized for neural network. Most of these formats have a high accuracy loss and are therefore unfitting due to 'RQ1: Impact Reduction on Online Learning' (Section 1.2). Fixed point encoding uses one exponent per matrix while storing sign and mantissas separately per value. The test results show only little impact on the accuracy while reducing the storage amount by about 25%.

Bytewise Float Segmentation [MLDD17b] uses the lossy "Bytewise Segmentation for Float Matrices" compression algorithm which reorders segments of the float to group lower entropy section together (see Section 3.2.6). This also enables offloading the byte section featuring more precision while the first byte section is saved locally to be queried with less precision.

2.2.4 Delta Encoding and Composition

Encoding the weight changes as a difference or delta reduces size of the decimal numbers and hence the entropy. [Sik97] describes a delta encoding scheme for video frame with different types of frames fulfilling different roles and requiring different amounts of storage.

[MLDD17b] compares different types of encoding a delta and introduces a novel delta storage graphs creation algorithm between the matrices of the weights focusing on compression and the recreation speed. The compression time is theoretically quadratic and was not measured. [BCH+15] compares several delta storage graph algorithms for data set versioning. All considered graph algorithms have at least a linear runtime, are complex, require additional memory and also add a stronger dependency between the deltas. Especially considering the additional compression time, a complex delta storage graph algorithm will not be considered since other compression algorithms execute quicker.

2.2.5 Matrix Compression

There exists a variety of papers which specialize on compressing matrixes which mostly use matrix factorization [IKJS17, TMK16]. Since machine learning models are built up by several matrixes, this approach can reduce the size of the saved models. The matrix compression introduces by the mentioned papers only makes sense when values are not removed from the matrices, as with 'Top-K' (Section 2.2.2). Since Top-k features more promising compression ratios and these algorithms must be further adapted to fit with the other used compression algorithm while being similar to 'Quantization' (Section 2.2.3), it will not be used.

2.3 Online Learning Setting

In online learning, a machine learning algorithm tries to adapt to newly incoming data. While the characteristics of this data differ among publications, its size is commonly large, it arrives continuously and its source concepts drifts over time. [PFG18, p. 282]

In most common settings, labels for data can arrive after the data itsself. According to [PFG18, p. 294] it is still unclear how to handle such cases. Since this is not the main topic for this thesis, it is assumed that the new data includes labels.

2.3.1 Neural Network Adaption

Adapting a model to new data can be mainly divided into two approaches. Changing the models weights or parameters or changing the model network itself by removing or adding nodes. [PFG18, pp. 283-284].

The proposed system concentrates on a fixed network architecture and tries to compress the weights of the model and therefore, can only handle online learning systems of the first kind with changing weights.

2.3.2 Data Instance Selection

Neural networks require a certain amount of data called 'batch' for each learning iteration. As new data instances arrive one after the other, a sampling method for selecting data instances for one batch must be used. A selected batch of data in an online setting is called 'window' since the selection mechanism considers only a section of the infinite data stream. According to [GŽB+14, GBEB18, pp. 11, pp. 21] there exists a wide range of window types which vary in complexity of implementation and configuration. This can include reusing older data samples, as also used in [PDMM21], having a trigger, such as a certain amount of elapsed time, from which on old samples are discard or giving newer data instances more importance when calculating the change of the weights.

2.3.3 Concept Drift

'Concept drift' describes that data can change over time. 'Drift' implies the change while 'concept' reflects that an underlying concept of the source of the data has changed. Adapting to the 'concept drift' is one of the main concerns to address in an online learning setting since the model should adapt to those changes to accurately evaluate newer data which also have the new concepts embedded. Distinguishing noise from a real drifting concept is one of the challenges of online learning [PFG18]. [GBEB18, pp. 233-234] describes several drifts, such as a changing distribution of the labels.

Drift Detection Drift detection mainly relies on either the statistical analysis of the changing label distribution of incoming data or the error rate of an evaluating machine learning system, according to the reviews [GŽB+14, BS18, LLD+19]. [LLD+19] concludes that integrating drift detection into machine learning techniques required further research.

[BM08] uses two naive bayes learners to detect a drift if their error rates deviate from each other, exceeding a certain threshold. The 'recent' learner was only trained on the most recent data while the 'stable' learner uses all data. When the recent learner outperforms the stable learner it is replaced by the recent learner. Replacing the older learner completely is uncommon for a drift detection algorithm.

2.3.4 Deployment

Important for this work is the frequency of deployment since this indicates the frequency of saving the model as any deployed model must be saved to trace back past decisions. More frequent deployments show a requirement for a faster saving method if the saving method blocks the subsequent deployment. To our knowledge, none of the papers deal with the concrete deployment frequency.

Some data selection mechanism such as a sliding window for data indirectly hint at an update and therefore deployment [PFG18]. Since the sliding window mechanisms try to somehow split the data into batches and then execute training, we assume that the models should be deployed after each batch. In conclusion, the speed and amount of data and window size impacts the time span between deployments. In [PDMM21, p. 201], a scenario is calculated where a special distribution algorithm could deploy a new model every 800ms. Although this should only show the capabilities of the algorithm, it can still provide a rough idea which deployment scenarios the authors expect.

[DMRM19] have developed a continuous deployment platform, focusing on data management and proactive training. Their proposed training and deployment schedular can either be static, triggered after a set time interval or dynamic, depending on the classification time of the incoming unlabeled data and its prediction latency.

2.3.5 Performance Measurement

[PDMM21, GŽB+14, BS18, LLD+19, HSLZ21] agree that prequential measurements, but especially prequential accuracy are mostly used for measuring the error rate and the adaptability of an online learning algorithm. 'Prequential Accuracy' (Section 4.3.1) uses each data instance first for evaluation and then for training. [HSLZ21] summarizes alternative measurements to prequential accuracy which also consider the cost of a false classification. [LLD+19] comprehensively discusses alternative measurements addressing class imbalance. None of the alternative measurements to 'Prequential Accuracy' are applicable for this thesis and hence, are excluded.

2.4 Traceability for Compliance

[DLF20] explains the setting and the legality of GDPR's "Right to Explain". It highlights the approaches and shortcomings of different companies in the insurance sector. The proposed VSOL can help to accelerate and increase the accuracy of GDPR inquiries.
[RGZ+21] describes guidelines for implementing traceability in the realm of data, which

can be translated and therefore followed by the VSOL. Re-executing the query or classification to evaluate the fault of the used version during the query time requires the timestamps of the query and of all versions to identify the timestamp of the used version. Besides (a) the query timestamp and (b) the timestamps of all versions, (c) the used version, (d) the data of the query, and (e) the model in which the version is loaded, are required. The used version is then loaded into the model and the query is then classified, comprehending the querying result transparently. This process can be divided into three distinct parts, using the terminology of the paper:

1. The VSOL covers saving the versions and the deployment timestamp of the versions, covering the described concepts of 'Data Versioning' and 'Timestamping'.
2. 'Query Store Facilities', which save all required information of the query to re-execute it, are responsible for saving the query and the timestamp of the query.
3. Storing the model itself is referred to the 'Data Infrastructure'.

All three parts are required to comprehend a specific decision, but the VSOL covers, fulfills, and measures only the first part since the RQs only addresses the impact on the learning process which is tightly coupled to the first part while 'Query Store Facilities' and 'Data Infrastructure' are separate concerns that can be addressed independently from the learning process. Only storing the timestamp in the 'Query Store Facilities' without the query can reduce the storage and complexity of a it while a claim could still evaluate the version to show that the version misclassifies specific queries in general.

2.5 Summary

The chapter has shown (a) which existing and possible model recreation systems there are and how they are not sufficient for all RQs, (b) what compression methods exists and how they are applicable for the RQs and how not, (c) what the characteristics of online learning are to address them in the VSOL and its evaluation and (d) why traceability matters, how it should be conducted and what parts it involves. The upcoming chapter uses the gained knowledge and shows how the VSOL is built up, how the compression algorithms are designed or contextualized, how it addresses online learning and highlights its limitations.

CHAPTER 3

Designing an Online Learning DNN Versioning System

The following section will give a theoretical overview of the Versioning System for Online Learning systems (VSOL). This includes an detailed explaination of the used compression algorithms, a theoretical discussion of different deployment scenarios and the limitations regarding measurement results and production usage including its consequential potential.

3.1 Overview of VSOL

Term Definitions for Neural Networks 'Weights' of a neural network are essentially a set of decimal numbers, represented as 32-bit floats in this thesis. The weights are subdivided into layers, which correspond to the layers of the neural network. The floats of each layer can be represented as a matrix. It is not possible to represent all the layers as a single uniform matrix as each layer matrix has a different dimensions. Therefore, the weights are represented as a list of float matrices. Each matrix is implementation as a NumPy array. NumPy is a Python framework which is used for handling and computing matrices. A 'Set of Weights' refers to a specific version of a weight set with specified values while 'Weights' refers to the general concept. 'Parameters' refers to the number of floats in a set of weights or in one of its layers.

'Model Architecture' refers to the buildup of the neural network model. This includes the types of transformation functions and their in- and output dimensions per layer. 'Model' or 'Neural Network' refers to the combination of a 'Model Architecture' and a specific 'Set of Weights'. Only a full model is able to classify data since only the weights miss the transforming functions to use the weights while the architecture lacks the learned weights holding the information required for the classification.

VSOL Saving Process In a simple online machine learning setup, the neural network receives a new batch of data and uses it to train on it. The result is a new set of weights which are then deployed to production to evaluate newly incoming data. When a new training data batch arrives, the just deployed weights are then overwritten by a new set of weights and the old weights are lost. Hence, it is not possible to restore the weights to check why the neural network came to the conclusion of the evaluation of any newly incoming data.

The main task of the proposed VSOL is to save the deployed weights so that this set of weights can be recovered. To achieve this, it saves the weights it receives from the neural network and the deployment timestamp. Figure 3.1 shows an overview of this process. On the left, the resulting 'training weights' from the training model are handed over to the VSOL to be compressed and stored with any additional information required for restoring the weights. On the right, the saved weights are then deployed to a production neural network. If the VSOL is configured to be lossless, iteration numbers and weights of training and production would be the same. The training weights during processing by the VSOL are referred to as 'processing weights'. The VSOL keeps the production weights which were saved last in memory, referred to as 'previous weights'. Summarized in terms of terminology, 'training weights' of the training model become 'processing weights' during the processing by the VSOL. If the processing weights are saved, they overwrite the 'previous weights' in memory and are deployed and saved as 'production weights'. The time at which the processing of the 'processing weights' by the VSOL is finished and are then referred to as 'production weights' is the 'deployment timestamp'.

But to minimize the required storage even further, the VSOL can be configured to be lossy, which leads to an alteration of the weights or skipping the deployment of a new set of weights. In Figure 3.1, the different colors of the weights indicate the potential change between the training and production weights. The dotted arrow shows which training version resulted in which production version. Not all trained model weights have to lead to a new deployment and thus, the saving of a new model version. This is visible in the example where the second training weights never lead to production weights. Since in this case the weights of the production model now differ from the training model, the training model cannot also be used as a production model which halts the training process to evaluate new data. Refraining from any changes to the training process is required for an easy integration and for the training process to be unaffected by the VSOL.

There exist two iteration numbers, n for addressing the data batch B_n for the training weights TW_n and x for addressing the changed weights, if the VSOL is configured to be lossy, processing weights PcW_x, previous weights PvW_x and production weights PW_x. The processing weights PcW_x change during the processing of the VSOL and hence, are not a fixed and concrete like the other described weights.

To fulfill the property of full traceability, the saved version should be the same as the deployed version. Therefore, any deployment or save can only occur after any lossy alterations in the VSOL are executed. Hence, any mentions in the text of saving the model weights also entails their deployment to the production model and vice versa.

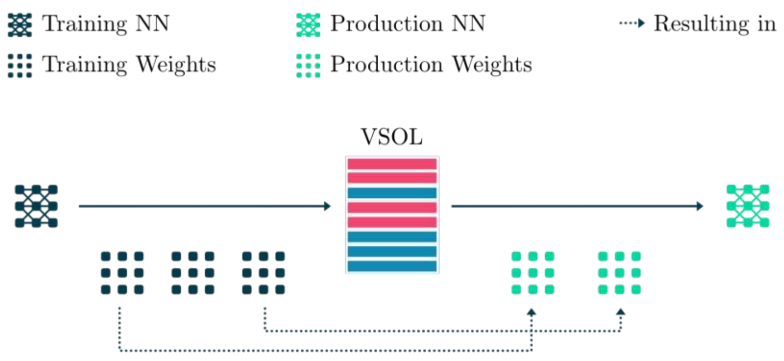

Figure 3.1: Illustrating how training weights from the training model are processed by the VSOL and deployed to the production model. The second training weight set does not result in a production weight set.

Composition Overview of VSOL The composition of the VSOL responsible for processing the weights is configured to be composed of a subset of a pool of different algorithms or processing step, both referred to as 'Processing Step'. The Processing Step can be grouped by their intention or output into Stages. Each Processing Step or Stage can require a certain type of Processing Step or Stage to be its predecessor. E.g., 'General Compression' has to be applied before the compressed weights are saved since otherwise the compression is not applied to the final saved weight file.

VSOL in Figure 3.1 depicts these Stages in order of execution. Figure 3.2 shows the zoomed in version of the pipeline with the name of each proposed Stage.

The Processing Steps of each Stage are applied from top to bottom to compress the given weights, as shown by the full arrow. Each Processing Step passes the changed or compressed weights on to the next algorithm. Hence, the order of the Processing Steps must be upheld since certain Processing Steps expect a certain change or format from the former algorithm. Applying the Processing Steps in reverse order decompresses the weights, as shown by the dashed line. Lossy Processing Steps are only executed during saving since they only reduce the amount of saved information. The only exception would be if a lossy Processing Step changes the saved format of the floats.

The idea is to show which fields of study can be applied in general and how they can interact. The focus is not on finding the absolute optimal solution of each field for this task. For some scenarios it could make sense to not use a certain Stage or algorithm. 'Evaluation' (Chapter 4) evaluates which Stages/Processing Steps should be used for which evaluation metric requirements.

'Processing Pipeline of the VSOL' (Section 3.2) will explain each Stage in detail with its containing algorithms.

A selected subset of all Processing Steps with specific 'Processing Step Parameter' values is revered to as a 'configuration' of the VSOL. Omitting or chosing a Processing Step for a configuration depends on their properties and the resulting properties of the configuration. E.g. if the weights should be deployed lossless, only lossless Processing Steps are chosen for the configuration. The properties are the evaluation metrics, as discussed and listed in 'Evaluation Metrics' (Section 4.3), affecting the training processing unit, as discussed in 'Classifying Data Impacting Training' (Section 3.4.2) and being lossy. A configuration can have several or zero Processing Steps of each Stage.

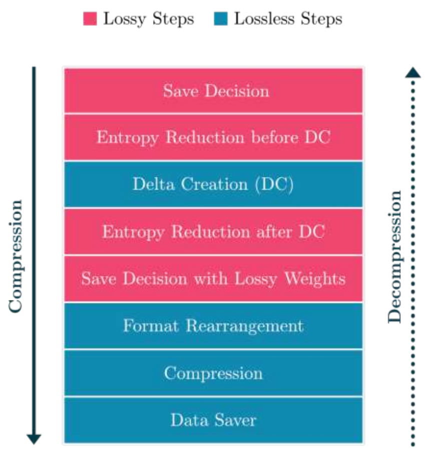

Figure 3.2: Illustrating the full Versioning System for Online Learning systems with all possible Stages. The arrows depict the order for compression and decompression.

Entropy Reduction for General Compression Some of the Processing Steps in the Stages above the compression Stage in Figure 3.2 do not have a direct compression effect but reduce the entropy, whereby the compression Stage (see Section 3.2.7) then can achieve a higher compression and execute faster. 'Entropy Reduction' can either entail the lossy change of values to reduce their precision and hence their complexity or the context sensitive rearrangement of memory segments so that similar segments are closer to each other in memory which makes it easier for the 'General Compression' Processing Step to recognize and compress them. E.g., 'Top-K' (Section 3.2.2) sets many floats of the weights to zero. Since zeros are now predominant, the compression Stage can then encode it by a shorter symbol through variable length encoding. Top-K only replaced the values, reducing the entropy of the weights, which can then be utilized by the compression Stage. But without the compression Stage, each zero would still require the storage amount of any other float.

3.2 Processing Pipeline of the VSOL

The following section will describe the idea and purpose of each Stage and which Processing Steps exist in each Stage from top to bottom, as shown in Figure 3.2. The given order should be upheld since some Processing Steps reorder the layout of the received weights and others expect to receive a certain format. Not all suggested Processing Steps are implemented or evaluated and are listed to show further options of each Stage.

3.2.1 Save Decision

This Stage can decide if the new weight set TW_n of the training model should lead to a save or not and hence, is lossy. Being able to avoid saving a set of weights all together requires no additional memory. The basis for this decision can be the training data B_n, the new training model TW_n or just the iteration number n. In this Stage, none of the Processing Steps are implemented and hence, evaluated. But since drift detection is an important field in online learning, the process would be incomplete without this Stage and hence, was added.

Iteration based

Skipping saving every y-th iteration can help to reduce the memory impact by $1/y$. This approach is easy to execute. Also, it can be used to simulate the impact of not saving every incoming batch. After an evaluation not provided in this thesis this approach was discarded because it lead to a high error rate increase.

Drift Detection based

'Drift Detection' (Section 2.3.3) mechanisms could evaluate through the new data if a drift has occurred and therefore the new batch must have had a strong impact on the model and should therefore be saved. This is not implemented in this thesis and only shows another mechanism for the save decision Stage. Section 6.3.1 describes the benefits of such an algorithm.

3.2.2 Entropy Reduction before Delta Creation

This Stage is lossy and applies 'Entropy Reduction' Section 3.1 to the weights. A Processing Step in this Stage receives the processing weights as whole number, different to the 'Entropy Reduction after Delta Creator' were the weights are encoded as a xor delta.

Top-K

'Top-K' selects the k percent largest floats changes which should be kept and sets the rest to zero. The idea of the 'Top-K' approach is that only the higher weight changes mainly impact the decision process and the rest can therefore be neglected. The resulting weight changes now mostly consisting of zeros have a significantly lower entropy. 'Top-K' only works together with the 'Delta Creator' (Section 3.2.3) since zero values changes are only zeros as a delta and otherwise a full float. The Processing Step Parameter representing the k value is referred to as 'Top-k-Percentage' (TkP).

'Top-K' implemented by the papers discussed in Section 2.2.2 uses the changed weights for the next training step and hence, directly influence the training process. Since the proposed system should not modify the training process, 'Top-K' is applied only on the processing weights and does not change the training weights.

The following process is executed per layer of the weights, as in [PDMM21], and broken down into the following steps:

1. The absolute difference between the previous weights and processing weights is calculated.
2. The 'Top k Absolute number of Parameters for Layer' ($TkAbPl$) is calculated from the total number of parameters for the layer and the given 'Top-k-Percentage' (TkP), rounded up.
3. The $TkAbPl$ indices of the top values are selected through Numpy's 'argpartition' function.
4. The layer of the previous weights is copied and the parameters of the selected indices are overwritten by the currently processed training weights. When the resulting layer is processed by the 'Delta Creator', the delta of the selected indices is zero since they now have the same value as the previous weight. The original changes which are now overwritten are preserved in the training model and could be chosen during the next 'Top-K' iteration.

Minimum per Layer As some of the test models have very few parameters for some layers, we were concerned that too much information would be lost for them without gaining a great amount of compression. Therefore, a second parameter 'Minimum per Layer Percentage' ($minLP$) was used to guarantee a minimum parameter use per layer, dependent on the 'number of Parameters of layer with the most parameters' ($Plmax$). Since most models have great parameter count differences per layer, those large layers use up most of the storage and using a smaller percentage than its parameter count for the other smaller layers should not affect the memory as much. This is an additional functionality which can be omitted by setting the Processing Step Parameter to 'None'.

Equation 3.1 shows how the 'Top k Absolute number of Parameters for Layer Minimum bounded' ($TkAbMin$) is calculated from the Processing Step Parameter 'Minimum per Layer Percentage' ($minLP$), which is broken down into the following steps:

1. The 'number of Parameters of layer with the most parameters' ($Plmax$) is selected from all 'number of Parameters per Layer' (Pl)
2. 'Minimum number of Parameters per Layer' ($minPl$) is calculated by multiplying 'number of Parameters of layer with the most parameters' ($Plmax$) with $minPl$, which creates a dependence on the model size.

3. For each layer, 'Top k Absolute number of Parameters for Layer' ($TkAbPl$) is replaced by $minPl$ if 'Top k Absolute number of Parameters for Layer' ($TkAbPl$) is smaller then $minPl$.

$minPl$ must be significantly lower then 'Top-k-Percentage' (TkP) since otherwise it will significantly increase the required storage amount when each layer uses the same amount of parameters as the largest layer.

$$\begin{aligned} max(Pl_n) &= Plmax \\ Plmax * minLP &= minPl \\ min\{minPl, TkAbPl_n\} &= TkAbMin \\ minPl &\ll TkP \end{aligned} \tag{3.1}$$

The following example shows how this Processing Step Parameter affects the number of selected parameters for a smaller layer: Let M be a Model with $layer_{max}$ and $layer_{min}$ having 500.000 and 1.000 parameters each and $kPercentage$ being 1% while $minPerLayerPercentage$ being 0.1%. Without $minLP$, 10 parameters are selected for $layer_{min}$. Using $minLP$ results in a $minPl$ of 500, calculated by multiplying 500.000 with 0.1%. Hence, 500 parameters are selected for $layer_{min}$ instead of 10.

All Layer Top-k For 'Top-K' (Section 3.2.2), the 'Top-k-Percentage' (TkP) values are selected for each layer. Hence, the relative number of selected parameters per layer is about the same, excluding rounding. 'All Layer Top-k' selects the TkP values over all layers, leading to different relative numbers of selected parameters per layer and hence, is an alternative implementation to 'Top-K'. The hypothesis is that this should work better then 'Top-K' since it can select higher and hence more important values from layers which feature more of them.

'All Layer Top-k' uses the same steps as 'Top-K' but executing the steps for the full weight set as if it was a layer. Since 'Minimum per Layer' (Section 3.2.2) requires each layer to be processes separately it is executed with the algorithm of 'Top-K'. The indices of 'All Layer Top-k' and 'Minimum per Layer' are then combined after they are retrieved in Step 3 of 'All Layer Top-k'.

Loss Adaptive TkP When 'Top-K' is applied for a scenario were a drift occurs, a mechanism for dealing with a higher 'learning effect' could be required. The Top-k-Percentage (TkP) of Top-k would then adapt to change in the network which is reflected by the loss value. I.e., the loss value is inserted into a function, mapping it to a now fluctuating TkP. The 'Loss' (Lo) value is the numerical difference between the classification result from the model and the actual labels of the training data. It is used to determine the degree of change for a neural network and hence, can be used as an indicator of the amount of change in the network. Using the 'Loss' (Lo) for this measure is computationally cheap since the value has to be calculated for the model training anyways. A higher change of the weights is referred to as a higher 'Weight Change Degree'. The 'Loss Upper Bound' ($LoUB$) is a constant which bounds the used loss value and is set once by examining the resulting loss values during the evaluation to see which loss values are outliers and which are not. The value for $LoUB$ is mentioned in the evaluation section.

Equation 3.2 shows how 'Top k-Percentage loss adapted' ($TkPLoA$) is calculated from the 'Loss' (Lo) of the current training weights in the following steps:

1. Bound the loss to the 'Loss Upper Bound' ($LoUB$) to ensure that the range of the loss is not distorted by one outlier and to normalize the resulting value to a 0 to 1 scale, resulting in 'Loss Bound and Normalized' ($LoBN$).

2. Apply the 'Loss Transformation Function' ($f_{LoTrans}$) to in- or decrease the resulting 'Top k-Percentage loss adapted' ($TkPLoA$) for middle loss ranges, resulting in 'loss bound normalized and transformed' ($LoBNT$). The hypothesis is that the loss value of a mid-range Weight Change Degree can require a different 'Top-k-Percentage' (TkP) value then when the change is at its maximum or minimum. Several functions will be evaluated, including linear and several grades of convex and concave curves, depicted in Figure 4.6.

   Equation 3.3 shows how $f_{LoTrans}$ is applied to the Processing Step Parameter 'l'. 'Move to Axis' (MtA) is an 'l' dependent value to move the curve to the value +1 of the x and y axis. Different functions are applied if 'l' is zero or above, being convex, and otherwise being concave.

3. Map the $LoBNT$ to the range between $minPl$ as minimum and 'Top-k-Percentage' (TkP) as maximum to get the resulting 'Top k-Percentage loss adapted' ($TkPLoA$), replacing the TkP used in Top-k. I.e., the new $TkPLoA$, replacing the value of TkP for the selection of weights, now lies between the values $minPl$ and TkP. $minPl$ and TkP are reused to reduce the number of Processing Step Parameters. Since $minPl$ is used, 'Minimum per Layer' must be used when using 'Loss Adaptive TkP'. Using separate parameters was evaluated briefly and omitted due increasing the evaluation space significantly while not featuring any significant improvements.

$$\begin{aligned} \|min\{Lo, LoUB\}\|_0^{LoUB\to 1} &= LoBN \\ f_{LoTrans}(LoBN) &= LoBNT \\ LoBNT * (TkP - minPl) + minPl &= TkPLoA \end{aligned} \tag{3.2}$$

$$\begin{aligned} (1/2) * ((1/\sqrt{(|l|/(|l|+4))}) - 1) &= MtA \\ l < 0 \to 1 - (1/(|l| * (x + MtA))) - MtA \wedge \\ l \geq 0 \to (1/(|1 - l| * (x + MtA))) - MtA \end{aligned} \tag{3.3}$$

3.2.3 Delta Creator

The number of weights changed in a batch training and thus differing between the processing weights and the previous weights should be significantly smaller than any standalone set of weights, referred to as ’materialization’. This change set is referred to as ’delta’, while combining deltas to a materialization is referred to as ’materializing deltas’. As [MLDD17b] describes, a delta should have less entropy then a materialization per step. To their measurements, this should lead to a 5% reduction in memory.

Additionally, delta creation mimics the update process of a remote delta update for models, i.e., when only the delta of a model training iteration is broadcast to other deployed models over a network. Therefore, further compression steps relying on a delta can only be leveraged if only deltas are saved. This is especially important since one might argue that a 5% reduction might not justify the additional processing time and complexity of computing the delta and keeping the last weights in memory during compression and loading several deltas to materialize them during decompression.

The ’Delta Creator’ internally has a save of the last deployed version PvW_x of the last weights, already defined as ’previous weights’. Combined with the processing weights PcW_n, a delta $D_{n,x}$ can be created and is handed over to the next Processing Stepbelow. This is depicted in Figure 3.3 in the ’Delta Creator’ Stage where PcW_4 and PvW_2 from memory are combined to $D_{4,2}$.

In a lossless configuration, all training, processing and production weights are equal, and hence, the materialized processing weights the ’Delta Creator’ receives replaces the previous weights instantly. This is shown in Figure 3.3, blending out the lossy layers, TW_4 could be saved directly as previous weights after the delta $D_{4,2}$ is created. Since in a lossless configuration all training and production weights are equal, and hence, the same applies for n and x, the delta above should not have the missing iteration number 3 and the delta would be $D_{4,3}$.

Replacing the previous weights with the materialized processing weights the ’Delta Creator’ receives is not possible when the processing weights can be further altered by the ’Entropy Reduction after Delta Creator’ or ’Delta Creator’ decides to discard the processing weights and to rollback to the previous weights. The previous weights are a materialization while the processing weights are a delta after the ’Delta Creator’ Stage. Hence, the materialized processing weights are saved in memory when receiving them together with the previous weights until the materialized processing weights can replace the previous weights. This is depicted in Figure 3.3, where PvW_2, the previous weights, and PvW_3, the materialized processing weights, are kept in memory until it becomes clear that PvW_3 is used. At that point, PvW_2 can be deleted. Otherwise, if PvW_3 is rejected, PvW_2 is kept in memory while PvW_3 is deleted. Therefore, two materializations are kept in memory for a short time.

In a lossy configuration using ’Entropy Reduction after Delta Creator’, changes to the processing weights in delta format should be passed on to the materialized processing weights. This is done by combining the previous weights with the delta processing weights to replace the materialized processing weights after the last Processing Step of 'Entropy Reduction after Delta Creator' is finished. Since only the 'Delta Creator' has the ability to execute this delta operation, a callback is triggered together with the delta processing weights to execute this delta operation by the 'Delta Creator' after the last 'Entropy Reduction after Delta Creator' Processing Step has finished. This is depicted by Figure 3.3, where the 'Entropy Reduction after Delta Creator' Stage altered the delta processing weights $D_{4,2}$ and they are combined with the previous weights PvW_2 to replace the materialized processing wights PvW_3. When the VSOL is finished and the 'Save Decision through New Model' decides to deploy the processing weights, PvW_3 replaces PvW_2. The VSOL is now ready for the next weight set PcW_5.

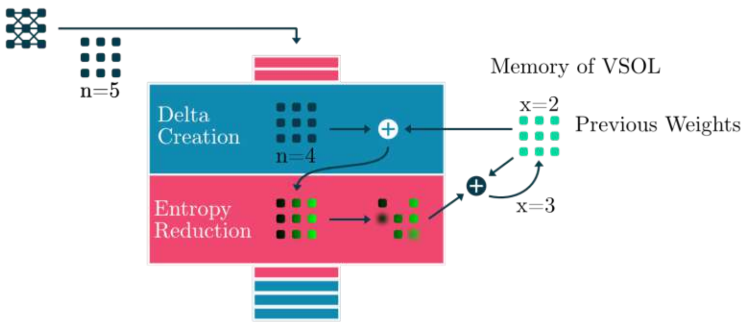

Figure 3.3: Illustrating how processing weights are processed in the 'Delta Creator' and 'Entropy Reduction after Delta Creator' Stages and how they interact with the previous weights and materialized processing weights in memory.

Delta Calculation: Byte-wise xor

For the delta value creation, a byte-wise xor calculation is used. As discussed in [MLDD17b], a byte-wise xor approach not only has the highest entropy reduction but is also lossless.

3.2.4 Entropy Reduction after Delta Creator

This Stage is lossy and applies 'Entropy Reduction' Section 3.1 to the weights.

Entropy reduction Processing Step below the delta creator can change the final values which are going to be saved and can directly evaluate through the xor format (see Section 3.2.3) which parts of the float will change.

Reset Least Significant Bits when Most Significant Bits set

A float consists of a sign, an exponent and a fraction/significant. When the Most Significant Bits of Fraction (MSBF) of the floats of the delta are set, we assumed that the exact change of the Least Significant Bits of Fraction (LSBF) would not have a great effect and could therefore be set to zero. Setting the LSBF to zero yields equally long low entropy zero blocks which then can be easily encoded by 'General Compression' (Section 3.2.7). 'Bytewise Segmentation for Float Matrices' (Section 3.2.6) partially lifts out the zero blocks out of the full float to be processed in succession by 'General Compression'. Consequently, 'General Compression' should be able to more easily recognize the pattern and require less memory to encode it.

This Processing Step uses the Processing Step Parameters Most Significant Bits of Fraction to Assess (MSBFA) and Least Significant Bits of Fraction to Overwrite with zeros (LSBFO) and is broken down into the following steps, which are executed per layer:

1. The indices where at least of the MSBF is set to true must be selected. Floats where this is the case will have a value above zero while all others are set to zero. Filtering out which floats are above zero and hence, identifying the indices, is done in the next step. Most Significant Bits of Fraction to Assess (MSBFA) is the exact number of MSBF that will be checked if they contain at least one true bit and is passed as an integer parameter to the algorithm. The implementation details of this process are described as follows: The processing weights are converted to a byte format since Numpy can only execute bit operations on a byte format. A bit mask, where the MSBFA are set to true, is combined with the processing weights through a logical and-operation, setting any non-MSBFA to false. The resulting floats either consist of only zeros if none of the MSBFA were true or have at least one true bit.

2. It is evaluated if the resulting floats are above zero and its indices are selected for the next step. Recall, only floats were one of the MSBFA were true are above zero. This is done by the comparison operation '> 0' of Numpy. The selected indices are stored as a Numpy boolean matrix with the dimensions of the weights, were selected indices are represented as true and the non-selected as false and is referred to as 'indices mask'.

3. For the selected floats, the Least Significant Bits of Fraction to Overwrite with zeros (LSBFO) are then set to false. Least Significant Bits of Fraction to Overwrite with zeros (LSBFO) is the exact number of LSBF that will be set to false and is passed as an integer parameter to the algorithm. This is done by a bit mask where all bits are set to true except the LSBFO, called 'bit selection mask'. The bit selection mask is then combined with the processing weights selected by the indices mask, representing the selected float values, through a logical and-operation. Since the bit selection mask is only applied on the selected indices, the other values, where the MSBFA were false, remain unchanged.

Hence, two bit operations and one value operation are required.

Reset Least Significant Bits Another variant is to overwrite the 'Least Significant Bits of Fraction' (LSBF) with zeros without checking the 'Most Significant Bits of

Fraction' (MSBF). This can reduce the processing time since only one masking operation is required. It is implemented in the same way as Step 3 of 'Reset Least Significant Bits when Most Significant Bits set' (Section 3.2.4) but instead selects all floats to be combined with the bit selection mask. Hence, 'Reset Least Significant Bits' has only one Processing Step Parameter, which is the Least Significant Bits of Fraction to Overwrite with zeros (LSBFO).

To be noted, it would have been an option to reduce the size of the floats directly by reducing the number of bits of the fraction, e.g. the resulting float would only have 24 bits instead of 32 if the lowest 8 bits were removed. It was decided against this change in format since any Processing Steps below would have to be adapted to the new format. Also, the change in format would have required additional processing time and the zero blocks are also singled out, further reducing the required space, as discussed above.

Comparison to Quantization Quantization is widely used with neural networks, as discussed in 'Float Compression' (Section 2.2.3). It has a runtime complexity of at least $\mathcal{O}(n)$ since every value must be considered individually to which value it will be mapped, impacting the compression time and hence, 'RQ1: Impact Reduction on Online Learning' (Section 1.2). Lower Bits Reduction (see Section 3.2.4) only requires one bit operation which is considerably faster. Additionally, the thesis tries to explore different ideas. Hence, the Processing Steps above were used instead of quantization.

The future work Section 6.3.1 will compare both approaches after the Processing Steps mentioned in this section were evaluated.

3.2.5 Save Decision through New Model

Executing this step after the lossy steps has the advantage that the lossy processing weights are available and can be used to come to a decision. Besides the lossy processing weights, any information mentioned in 'Save Decision' (Section 3.2.1) can also be used.

As described in 'Delta Creator' (Section 3.2.3), the materialized processing weights must be already restored to be used in this step. Since evaluating the model's error rate through classification is executed on the same processing unit as the normal batch learning is executed, this is the only step that can affect the training process, as mentioned in 'VSOL Execution Blocks Training' (Section 3.4.1).

Decide Save by Error rate Decay

Decide Save by Error rate Decay (DSED) bases the decision to save and deploy or to discard the processing weights on the error rate measurement difference between the model using the previous/deployed weights and processing weights. DSED can evaluate if the error rate increase of not rolling out the processing weights would be acceptable. In detail, models with the processing weights and the previous weights both evaluate the data since the last save. If the error rate difference does not surpass the Processing Step

Parameter 'Error Rate Bound', the processing weights will not be saved and discarded. In further steps, the previous performance measurement of the previous/deployed weights can be reused and only the new data batch is then evaluated. This saves memory since previous data does not have to be stored in memory. The same error rate is used as the evaluation error rate, as described in 'Prequential Accuracy' (Section 4.3.1), although they do not have the same validity since future data cannot be used.

A bool Processing Step Parameter can be set to only use the new incoming data instead of using all the data since the last saved model for evaluation.

Since this involves in the worst case loading two sets of weights and evaluating data on two different models, this could have a great impact on the compression time. The processing weights have to be recreated in memory for the evaluation while the previous weights already are in memory since 'Delta Creator' (Section 3.2.3) requires them. The old and new weights are then loaded into the compiled model architecture one after the other.

3.2.6 Format Rearrangement

Format rearrangement converts one format into another format. A 'format' is the bit representation of a specific type of information, in this case the weights. E.g., the initial weight format is an array of 32-bit Numpy arrays, as described in Section 3.1.

Changing the format can reduce the required storage. This is achieved by directly reducing the required storage, e.g. encoding reoccurring values through a special and shorter representation. Alternatively, this is achieved by grouping similar information in memory closer to each other in a context sensitive way which the (see Section 3.2.7) Stage might not anticipate since it is not specialized on a specific format or context.

Bytewise Segmentation for Float Matrices

The weights of the neural network are floats which have a very high entropy. Some segments of a float are more similar to each other then other segments, e.g., the exponent bits since specific exponent ranges of the weights are used more frequently. 'Bytewise Segmentation for Float Matrices' rearranges these segments in memory to be sequential. This allows the 'General Compression' (Section 3.2.7) to identify similarities among byte segments in corresponding positions, resulting in an increased compression ratio. [MLDD17b, p. 581] has approximately measured a 10% compression ratio improvement when using this compression approach.

Roughly speaking, before applying this approach, all of the floats are stored in memory sequentially. As a first step, each float is split into byte long segments. These segments are then reordered in memory to be in sequence. I.e., the first bytes of all of the floats are now stored in sequence, followed by the second bytes of all off the floats and so forth.

This operation can potentially be slow since all weights have to be transformed. Therefore, we implemented three different variants to reduce the potential impact:

1. 'SplitFloatAndStack' is executed per layer and the results are kept separately as entries in an array. For each layer, the floats are splits into byte long segments and reordered in memory to be in sequence, referred to as 'Bytewise Segmentation'. The sequences of each byte segments are then also ordered in memory to be sequential to form one value. During decompression, for each layer, the singular value is split into the sequential byte segments. Each individual byte segment is the merged with the correspondent other byte segments to again form a float.

2. 'SplitFloatAndStackByByteSegments' executes 'Bytewise Segmentation' like 'SplitFloatAndStack' but instead of combining the byte segment sequences per layer, the same of a kind byte segment sequences are combined over all layers before ordering them in memory to be sequential to form one value. Having all same of a kind byte segments in one sequence can help 'General Compression' to detect similar sections more easily also across layers but could required more compression time to combine the layer. The resulting value is then split back into layers during decompression before the 'Bytewise Segmentation' is reversed per layer.

3. 'SplitFloatAndStackByByteSegmentsSplitLater' works like 'SplitFloatAndStack' but instead of executing the 'Bytewise Segmentation' per layer, all layers are treaded as one. This yields the same result as 'SplitFloatAndStackByByteSegments'. Hence, 'Bytewise Segmentation' is executed on all layers at once, resulting in one value instead of an array. During decompression, the same is done in reverse, hence, the merged layers must be separated to an array again.

GCXS

GCRS/GCCS [SH15], also referred to as GCXS, is a sparse array storage format which can save memory by only storing non-zero values and their indices instead of each zero value using the full float storage amount in memory. Sparse arrays arise when using 'Top-K' (Section 3.2.2). Although 'General Compression' (see Section 3.2.7) should be able to encode zero values efficiently due to their low entropy, GCXS will be evaluated.

The python package 'sparse' (see [spa21], version 0.13.0) implements GCXS and is used for compressing and decompressing the Numpy array to the GCXS format. The GCXS format in code is a class containing the values and indices in separate arrays as class attributes.

Four combinable optimizations are implemented and applied after the library applied GCXS which will be evaluated.

Combine Layers Since each layer of the neural network is encoded as a separate Numpy array, one GCXS object is created per layer. Combining the layers to one Numpy array before applying GCXS results in only one GCXS object were all of the values and indices are stored closer to each other. Centralizing all similar value assists the general compression in reducing the required memory since detected patterns can be applied centrally to all values.

Since the layers have different sizes, they first have to be flattened to a 1-dimensional array to be concatenated since a Numpy array must have a uniform shape. The dimensions of the layers are stored and the reapplied when reversing the combination step.

Index Differential Encoding The index values may reach high value ranges due to the large number of weights. Hence, the increments between the indices ought to be smaller than the index itself. Additionally, some increment intervals between indices may be repeated, resulting in lower entropy. Thus, the indices will be differentially encoded by subtracting the previous index. This compression algorithm, named 'index differential encoding', is implemented by using Numpys 'diff' function for compression and 'cumsum' function for decompression.

Bytewise Segmentation GCXS changes the weight representation to a class. Hence, to use 'Bytewise Segmentation for Float Matrices' on the attributes of the GCXS object, it must be integrated directly into this implementation, wrapping GCXS. Besides applying 'Bytewise Segmentation for Float Matrices' to the values array, it will be separately tested if it is advantageous to also apply it to indices array which uses a 64-bit long, resulting in two boolean Processing Step Parameter. The used variant of 'Bytewise Segmentation for Float Matrices' will be chosen in the evaluation section.

3.2.7 Compression

This lossless compression Stage consists of general or float specific compression algorithm which uses entropy reduction (see Section 3.1) or similar approaches to reduce the files size. This Stage is essential since most other steps do not reduce the file size directly but only prepare the data for this Stage.

General Compression

General compression refers to a preexisting compression algorithm which is unspecific regarding the type of data it is optimized for. Since the format of the weights passed to this algorithm can vary depending on the Processing Steps used above it, a general compression algorithm can help to iterate on different combinations more easily since no further adaptions are required to the incoming data format due to its universality.

Most general compression algorithms have a 'compression level' integer parameter which can in- or decreases the required storage of the compressed data. The increased compression is traded off against a higher compression and decompression time. Since this tradeoff inversely affects RQ1, RQ2 and RQ3, it will be evaluated extensively. The possible value range of this parameter varies per algorithm.

Section 2.2.1 discusses papers which evaluated different general compression algorithms and their performance on float data. BZ2 (Python v3.10.11), ZLib (v1.2.13), Zstandard (v1.5.2.6), Brotli (v1.0.9) and LZMA (Python v3.10.11) were selected for further evaluation from these papers due to their high compression and speed on float data.

Some general compression algorithms like Zstandard require the data to be in byte format before compressing it. Therefore, any data handed over to one of the general compression algorithms will be first converted to byte format. The methods 'tobytes' and 'frombuffer' from Numpy are used for byte format conversion for compression and decompression, respectively.

3.2.8 Data Saver

This Stage saves any data required to restore the weights from a certain timestamp to disk. Hence, during decompression, it is responsible for loading the files required for a specific weight set. In the current implementation, the required metadata is only stored in memory, hence lacking in 'Fail Safety' (Section 3.4.4).

Enumerated File Saver

The 'Enumerated File Saver' saves the compressed weights received from the Processing Step above to storage as an individual file with the deployment timestamp. The files are enumerated and retrieved by their sequence number.

When the weight set of a specific timestamp is requested, the given timestamp must be mapped to the right sequence number. This is done by an array of deployment timestamps where the position in the array represents the sequence number of the file. The timestamp in the array before the specific timestamp is searched by binary search.

The data structure for this task only requires an insert at the end, since the timestamps by design are already in order during insertion, and a search operation. An array is already optimal since the insert at the end is executed in $\mathcal{O}(1)$ while the binary search on the ordered array is executed in $\mathcal{O}(n * log(n))$.

Delta Reset Point Creator

The 'Delta Creator' (Section 3.2.3) processes the processing weights to be encoded as the difference to the previous weights, referred to as 'delta'. Hence, to recreate or materialize a set of weights from a specific point in time, all proceeding deltas have to be reapplied, linearly increasing the decompression time for each additional saved set of weights. To cap the decompression time for any set of weight, the weights are saved with the actual materialized values periodically, referred to as 'reset point', additionally to the deltas. This is required to bound the decompression time, as required by 'RQ4: Reduce Retrieval Time' (Section 1.2).

To guarantee that the decompression time stays below the Processing Step Parameter 'Maximum Decompression Time Threshold' ($MDTT$), a fitting 'Reset Point Interval' (RPI) is calculated through Equation 3.4 in the following steps:

1. In the first execution, the initial offline trained weights are saved as a reset point. That guarantees that the production weights before the first regular reset point have a prior reset point.

2. The decompression time for one delta is estimated by adding the measured compression time of all lossless processing steps executed before the 'Delta Reset Point Creator'. This value is referred to as 'Maximum Compression Time measured' ($max(t_c)$) and overwrites the currently saved $max(t_c)$ if it is higher. The execution time of the lossy Processing Steps are ignored since all implemented lossy Processing Steps are not executed for decompression as they do not change the format.

3. It is calculated if the Processing Step Parameter 'Maximum Decompression Time Threshold' ($MDTT$) is exceeded during the processing of the upcoming processing weights, as shown in the Equation 3.4. The $max(t_c)$ is multiplied with the highest number of deltas that are required for the materialization that is the farthest away from a reset point. This value is calculated by adding the current count of previously saved deltas, referred to as RPI, and one to account for the upcoming execution. The result is then divided by two since reset points can be used bidirectionally and is rounded up to account for an uneven number of intervals.
   The bidirectional delta materialization is explained by the following example: Let RP_0 and RP_5 be two reset points saved at batch numbers 0 and 5 and let D_x be a delta with the interval number x while ws_3 is the weight set with the batch number 3 that should be recreated. Materializing ws_3 from RP_0 requires decompressing and combining RP_0, D_0, d_1, and D_2. Materializing ws_3 from RP_5 requires decompressing and combining RP_5, D_4 and D_3, which is one delta less and hence favorable.

4. If the $MDTT$ is exceeded in the upcoming run, calculated in the last step, a reset point has to be created to prevent this. In this case the 'Reset Point Interval' (RPI) is fixed and the current and previous step is skipped in upcoming executions.

5. If the RPI is fixed and the current interval number divided by the RPI results in an even number, a reset point is created. Keeping a fixed interval length helps to reduce the complexity of which full weight save and which deltas have to be loaded to recreate the model weights of a point in time. Creating a reset point requires executing all lossless steps for the previous weights to also compress them. The VSOL executes slower when a reset point is created due to the compression of the reset point.

$$max(t_c) * \lceil ((RPI + 1)/2) \rceil \leq MDTT \tag{3.4}$$

Since each evaluation run can have different run times, the iteration number after which a reset point is created is fixed to be comparable across different runs for the evaluation section. Hence, for the evaluation, only 1 and 5 are executed. The predetermined interval number is a Processing Step Parameter which replaces the Processing Step Parameter 'Maximum Decompression Time Threshold' ($MDTT$).

This idea is similar to video encoding [Sik97] where the I-Frames are equivalent to the delta reset points, which do not require additional information to be loaded, and the deltas are equivalent to a B-Frame since they can be used bidirectionally. But removing a reset point would not render all subsequent deltas useless since it does not contain any additional information and could be recreated by applying all deltas up to this reset point from another reset point. Hence, removing or adding reset points can be done afterwards, depending on memory or decompression time requirements of past data. Dealing with missing reset points is not implemented in the loading mechanism since it expects a reset point every x iterations. Different consequent optimizations are discussed in 'Optimizing Storage of saved Weight Sets' (Section 6.3.2).

3.3 Weight Deployment Scenarios

Processing Time affecting Error Rate In the VSOL, the weights are deployed after the VSOL has been compressed and saved, hence fully processed, the new training weight set.

A deployment should lead to a lower error rate since fresher data is incorporated into the model. This is especially time critical when the underlying concepts of the data changes, described in 'Concept Drift' (Section 2.3.3), and the currently deployed model version would misclassify any new data with this changed concept.

Lets consider the error rate of an example concept drift scenario for a 10 second window and the effect of a delayed deployment of an adapted model. In second 0, a new labeled training data batch arrives, containing data with a new concept. If the old model in second 0 is trained on the new training data batch, it can correctly classify any data with the new concept, otherwise it will misclassify it. As of second 0, all of the arriving data which should be classified arrives with the new concept. Optimally, the model would be trained quickly and be deployed instantly after the training in second 1 and correctly classifies all arriving data after second 1, leading to a error rate of 10% in the 10 second window. Now lets consider a slow deployment process which requires 3 seconds to deploy and correctly classifies all arriving data after second 4, leading to an error rate of 40% in the 10 second window. The slow deployment process increased the error rate four fold for the given time window.

It is unclear if this effect on the error rate is noteworthy in practice and depends on the concept change rate, the percentage of affected data and the data arrival rate. The delay to the deployment of the VSOL is indirectly measured as the compression time metric (see Section 4.3.3).

Since this effect is not discussed in the literature, to our knowledge, and the thesis' topic is not to experiment with this effect, it will not be considered during the evaluation and its subsequent decisions.

'Measuring Error Rate Impact by Delayed Deployment' (Section 6.3.3) describes how an evaluation of this effect could be conducted.

Proposed Deployment Scenarios Reducing the time of deployment can decrease the error rate, as described in Section 3.3. Hence, deploying the weights before the VSOL finished processing the training weights, is desirable. The deployment of the production weights is not implemented in the VSOL since this thesis focuses on impact of the VSOL on the storage and learning process. The following paragraph will describe possible deployment approaches for different circumstances. These approaches build upon the Stagestructure of the Processing Steps of Figure 3.2

Figure 3.4 shows two deployment approaches, the first one without lossy Processing Steps and the second one with lossy algorithms. They differ since the training weights are changed in a lossy approach, hence any changes to the training weights have to be applied before deployment to guarantee that the deployed and saved version of the weights are the same.

Deployment can be handled asynchronously but should have a queue for the resulting weights if 'Delta Creator' or any other lossy Processing Steps are used, since the weights only result in a materialization if all deltas are used in order. In other words, a new training weight set has to wait in the queue until the previous training weights are fully processed before it can be processed by the VSOL.

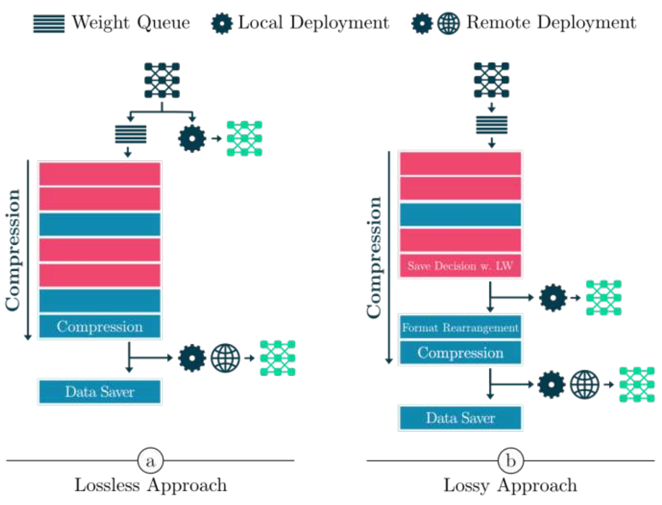

Figure 3.4: Points after which Stages the weights can be deployed. Each approach shows the point for a local and remote deployment. Figure a shows the deployment of the lossless approach, Figure b the lossy approach

For a deployment over the network, bandwidth is the main limitation of the deployment speed. Hence, in a remote deployment setting, it makes sense to execute all Processing Steps to reduce the size of the weights to the maximum before sending it over the network. The additional runtime of the Processing Steps is less than the additional time required by the uncompressed weights to be transferred over the network compared to the compressed weights, as discussed in [PDMM21]. The remote deployment point between Stages is the same for both lossy and lossless versioning systems since both have to finish all compression algorithm, as discussed above. This is depicted by the remote deployment between the compression and 'Data Saver' Stage on the bottom of the VSOL. The remote weights receiver would then require the same pipeline setup to be able to decompress the weights. 'Local deployment' refers to a scenario in which the weights do not have to be transferred over the network to be deployed.

In the lossless versioning approach, the weights can be deployed locally before the versioning system is triggered since they will not be changed in the versioning system. This is depicted at the top of figure in which the deployment executes while the weights are added to the processing queue.

In the lossy versioning approach, the lossy steps have to be executed before the deployment since the training weights are changed to the lossy production weights and only the production weights should be deployed. Therefore, the local deployment is then executed after the last lossy Stage in the middle of the versioning system. Hence, all lossy Processing Steps have to be applied before the local deployment can be executed.

Deploying requires tracking the remote deployment timestamps, as discussed in 'Timestamp Inconsistency for Remote Deployment' (Section 3.4.3).

3.4 Limitations of Implementation

Since the main focus of this work is the identification and evaluation of Processing Steps that minimize the impact the Versioning System for Online Learning systems (VSOL) has on the storage and learning process, the resulting code is not production ready and some aspects are excluded. The following section mentions the most important shortcomings which are relevant for a real-world application.

3.4.1 VSOL Execution Blocks Training

The implemented VSOL blocks the training process, i.e. the training process is halted until the VSOL has finished processing the new training weights. A delay of the training process should be minimized, as declared by 'RQ1: Impact Reduction on Online Learning' (Section 1.2). Section 3.3 describes how the delay can impact the error rate. This VSOL processing time is measured as the compression time, as mentioned in Section 4.3.3, and hence, measures the extend of the impact not executing VSOL asynchronously.

Executing the VSOL synchronously was chosen to reduce the implementation complexity since the asynchronous implementation would have required a queue and further unforeseeable adaptions for the remaining code to be asynchronously executable. An asynchronous execution could have also slowed down the training when run in parallel which is also undesirable according to 'RQ1: Impact Reduction on Online Learning' (Section 1.2). The queue would have also required significantly more memory, depending on how much slower the VSOL is compared to the training time of one batch. The execution time deviation could also have been increased when both processes block each other in an unfavorable way for some evaluation runs.

The 'Weight Deployment Scenarios' (Section 3.3) describes what the same VSOL requires to work asynchronously and how the deployment could also be executed asynchronously.

3.4.2 Classifying Data Impacting Training

Considering that the model training is executed on another more specific processing unit like a GPU or TPU, the VSOL should have only minimal impact on the training process. This does not hold for VSOL Processing Steps which require the model to classify data, as described in Section 3.2.1 and Section 3.2.5, which also requires access to the neural network processing unit. Executing the training and those Processing Steps in parallel can impact the training time.

'VSOL Execution Blocks Training' (Section 3.4.1) describes why a parallel execution was not conducted and hence, the direct impact on the training process was not evaluated. The effect will be indirectly measured by the additional compression time it takes to load the model and the data for classification.

Since this effect would be hard to measure and should be not present due to the blocking nature of the VSOL (see Section 3.4.1) it will be disregarded.

3.4.3 Timestamp Inconsistency for Remote Deployment

In a remote deployment, a set of production weights are distributed to remote computation instance which only classifies newly incoming data, referred to as 'compute node'. The time of the deployment for different compute nodes can vary. Hence, when a data instance is classified by a compute node after a new set of weights is deployed but before it has arrived at the compute node, it is classified by an outdated DNN model which does not correspond to the timestamp saved at the deploying instance. In such a case, the arrival timestamp of the data point and the timestamp of the deployment cannot be directly mapped and a procedure has to implemented to deal with this timestamp inconsistency. The compute node therefore also has to track the local deployment timestamps.

Since deployment is not a direct concern of this thesis, a solution for this issue will not be considered. 'Weight Deployment Scenarios' (Section 3.3) discusses how the VSOL could be changed to support deployment.

3.4.4 Fail Safety

Parameters of the VSOL that are set during the processing of a new weight set are not saved when the versioning system is shut down, e.g. if an outage occurs. Correspondingly, the versioning system has to be kept in memory to be functional. Although the deployed weights are saved, the versioning system still requires those parameters to retrieve previous weights from storage. Examples of such parameters are listed in the 'Memory Impact' (Section 3.4.5) as storage requirements and most importantly includes the array dimensions per layer of the weights.

Fail safety is not implemented since is not required to measure the effect of the VSOL which is the focus of this thesis.

The VSOL also does not save the model architecture or current training state. It only provides the set of weights for the model.

3.4.5 Memory Impact

The Processing Steps are chosen by their speed and not by their small memory impact since a smaller execution time leads to less blocking or interference with the training process which, regarding to 'RQ1: Impact Reduction on Online Learning' (Section 1.2), is the main focus. This section will discuss the memory impact of the VSOL and its individual compression algorithms.

Definition Memory Quantities and Allocation Timespans While the VSOL has several values in memory which only have a small footprint, such as the iteration count, there are two components which require the majority of the memory. These are the sizes of one set of weights, referred to as 'weight set size' and the size of one batch of the training data, referred to as 'data batch size'. There are three different types of time periods these memory sizes can be allocated. The first period is the 'VSOL running time' which stretches from the initialization to the shutdown of the VSOL. The second is the 'VSOL execution time' which stretches from the start of the invocation of the VSOL for a new set of weights and ends when the set of weights is either saved to storage or is finished with the processing of the set of weights since it did not lead to a deployment. The third is the 'compression algorithm execution time' which is the stretches from the start of a specific compression algorithm during the execution of the VSOL and ends with the start of the subsequent Processing Step or the end of the VSOL execution.

Memory Requirements When the VSOL processes the given training weight set, their size in memory is only reduced by the lower Stages 'Format Rearrangement' and 'Compression' and hence one 'weight set size' is present for the 'VSOL execution time'.

When using 'Delta Creator' (Section 3.2.3), one 'weight set size' is required to store the last the previous for executing the diff for the 'VSOL running time'. An additional 'weight set size' is required during the 'VSOL exection time' for the materialized processing weights, which keeps the current processing weights in memory, e.g., for replacing the previous weights after the VSOL is finished. Since 'Delta Reset Point Creator' (Section 3.2.8) is required for 'Delta Creator' to bound the decompression time, the memory requirement of it has to be add to the 'Delta Creator'. When a reset point is created, during the 'execution time' of 'Delta Reset Point Creator', the memory requirements for the 'execution time' of all other used lossless Processing Steps is required since they are executed again in sequence for the reset point.

During the 'execution time' of 'Top-K' (Section 3.2.2), one 'weights set size' is required to first calculate the diff between the last saved weight set and afterwards to retain the indices which have to be reset to zero in memory.

During the 'execution time of 'Reset Least Significant Bits when Most Significant Bits set' (Section 3.2.4), up to one 'weight set size' is required to save the indices which have to be set to zero. This is not required if Reset Least Significant Bits is executed since all indices are selected.

During the 'execution time' of 'Decide Save by Error rate Decay' (Section 3.2.5), an additional 'weight set size' is required for the training weights that are currently being processed. This is necessary since after the 3.2.3, the format is changed and the weights have to therefore be recreated. The last saved weights are already present to be used by the 'Delta Creator'. One 'batch data size' is required per data batch that is kept in memory to be evaluated by the new set of weights. If the flag for evaluating all data between the saved and the current weights is set, the number of required data batches equals the number of weights that were not saved. This parameter is generally undesirable from a memory perspective. Evaluating only the current data batch requires only one batch in memory.

'GCXS' (Section 3.2.6) and 'General Compression' (Section 3.2.7) are not discussed in theory since they are not implemented by us and a statement would require a code analysis of the used packages.

If the asynchronous processing queue is implemented (see Section 3.3), each unprocessed set of weights is also in memory.

Decision against Memory Measurement The memory usage is complex since some allocations are only present during the execution of one Processing Step or during the processing time of the VSOL while other allocations are present as long as the VSOL is running. Consequentially, measuring the memory impact would be elaborate since several distinct measurements in time would be necessary to reflect the variable allocation periods described above while also excluding the memory usage of the machine learning framework and the testing environment. Then again, the memory impact can be easily estimated per Processing Step since the relative sizes such as 'weight set size' and 'data batch size' are coupled and not dependent on other algorithms. In conclusion, since the memory impact is not the main focus of the thesis and a measurement reflecting the true complexity is elaborate while it would only reflect the described relative sizes, a memory measurement during the execution of the experiment is not conducted.

3.4.6 Retrieving Multiple Weights Sets

The VSOL is designed and optimized to retrieve the set of weights for one point in time for one evaluation since understanding past classifications is the main focus, as declared in 'Motivation & Problem Statement' (Section 1.1).

Querying scenarios like finding out when a certain data point changes the evaluated label for a certain time span would also be possible through the given retrieving system but is time consuming since each set of weights is loaded individually. Such a system is described in [MLDD17b].

3.4.7 Versioning opposes Privacy Concerns of Data Source

Neural Networks changes resulting from a data batch somewhat incorporate the used data. In an online setting where previous versions are not versioned, the changes from a certain batch would become diluted over time. Whereas when versioning the systems, the exact changes of one batch are specifically recorded, indirectly and vaguely recording the used data. This adds an additional privacy concern.

This can be counteracted by using lossy Processing Steps in the VSOL which dilute the exact training weights not only for the saved versions but also by the deployed version by design. The VSOL can also help to retrieve and remove several versions after data was used which the data source requests to be removed if the data source has the timestamp of the respective data. The next undeleted version incorporates the changes resulting from the concerned data batch but is further anonymized since each additional deleted version dilutes the changes of the respective batch by the changes of the deleted batch.

Since privacy is not considered as a research question, this topic will not be further discussed.

3.4.8 Portability to other Frameworks

This section describes in detail what a machine learning framework has to provide for VSOL to be integrable. 'Keras Usage and Integration' (Section 5.2) describes in detail how the VSOL is integrated into Keras architecturally and therefore acts as a showcase for any other integrations.

The format of the model weights is expected to be a list of NumPy arrays where each element in the list represents one layer. The data type of the NumPy arrays are expected to be a float32/single precision float. The framework has to feature the possibility to access the current set of weights, the used training data and the loss value and execute the VSOL, all after each batch. With the three given parameters, the VSOL can then process the weights. Requiring only weights and excluding the models architecture increases the ease of integration since the described weight format only requires one data type. Metadata systems which focus on versioning other settings such as the model architecture are described in 'Model Recreation' (Section 2.1).

Additionally, there must exist an interface were the VSOL provides a set of weights and data and receives the classification results for the given set of weights. This is currently not decoupled through an interface in the VSOL and therefore the used Keras code snippit executing this task has to be replaced. This is required to assess the quality of the last deployed weights and the current weights which decides if the current weights should be deployed, as described in 'Save Decision through New Model' (Section 3.2.5).

The VSOL does not provide 'Fail Safety' (Section 3.4.4) and hence, important parameters like the weight dimensions are lost when shutting down the VSOL. To be failsafe, a file saving these parameters is required to be restored the state of the VSOL, which is not implemented.

CHAPTER 4

Evaluation

4.1 Online Learning Virtual Drift Simulation

The key aspect of any online learning system is to adapt to a change in the data distribution as described in [PFG18, p. 285]. The versioning system should perform well under the most extreme conditions, which is an occurring data drift. This is required to fulfill the 'Main Research Question' (Section 1.2) which states that the model which should be versioned should be 'evolving'.

4.1.1 Virtual Drift

Drift can be categorized into two main types, a real concept drift where decision boundaries can change, and virtual drift, where the distribution of the data changes but not the decision boundaries. According to [HPC12, pp. 92], in practice, it mostly does not matter for the mechanisms dealing with the drift if it is real or virtual. Since changing the data distribution is easier then changing an underlying and unknown concept in the data, the versioning system will be tested under a virtual drift. Drift can have several forms such as a linear increase of a label, a sudden increase or several other forms as mentioned in [GŽB+14, pp. 5]. Since the drift should affect every batch as an extreme measurement and to capture if the versioning system can save the newly learned concept, a linear and increasing percentage of the former underrepresented label guarantees a constant change over time.

The code responsible for splitting the data and creating the appropriate batches can be applied to any data set as long as the data has distinct labels.

4.1.2 Off- and Online Data Split

For a model to be trained offline and then experience a drift during online training, an offline and online data set must exist. Figure 4.1 shows the offline data set on the left and the online dataset on the right. The x-axis represents batches, the y-axis the percentage of the underrepresented label per batch.

Offline Split

Percentage Offline data (POff) represents the percentage of data used for the offline training. This value should be at least 50% to ensure the offline model is properly trained. Underrepresented label Percentage Offline data (UPOff) describes which percentage of the underrepresented label data is used in the offline dataset. It should be very low to ensure that the model has to learn the concept during drift but not 0 since the label was anticipated during development of the model. Both percentages relate to the total number of available data.

Online Split

The online training data set has the two parameters, Total Batches Online data (TBOn) and Underrepresented label Percentage Maximum Online data (UPMOn), which are the sides of the imaginary triangle of the underrepresented label linear increase. Since for some cases the underrepresented label data is limited, for some dataset UPMOn cannot reach 100%. Optimally, all data is used but if the underrepresented label data has a high percentage, it is possible that not all of it can be used to uphold the two mentioned parameters.

The underrepresented label percentage of batch n is calculated through evaluating the area until n under the linear function minus the previous used values and then rounding the value down. This guarantees that missing values from a previous batch are included in the next. This is reflected in the small oscillating steps in the Figure 4.1

4.1.3 Optimal Split Settings

Optimally, the online split settings would be equal for all data sets while upholding the restrictions described in the offline split settings. Since some compression techniques differ for different number of batches, TBOn must be the same for all models to be comparable. The prediction performance curve, which reflects the adaption to the drift, should be similar for all used models. Similar curves should ensure a more homogeneous drift behavior over all models. Since the maximum accuracy per model is different, only the curves shape should be compared, not the direct absolute difference between the curves. This is accomplished by using min-max normalization for accuracy, referred as accuracy normalized from here on out.

Since the curve shape is different for each label, the curve of each label is compared to the curve of each other label. When comparing two curves, the absolute difference of accuracy normalized for each batch number pair is computed. The mean value of all differences for all label pairs is then computed for comparison. A lower value should represent more equal curves. This metric is references as mean curve difference. While

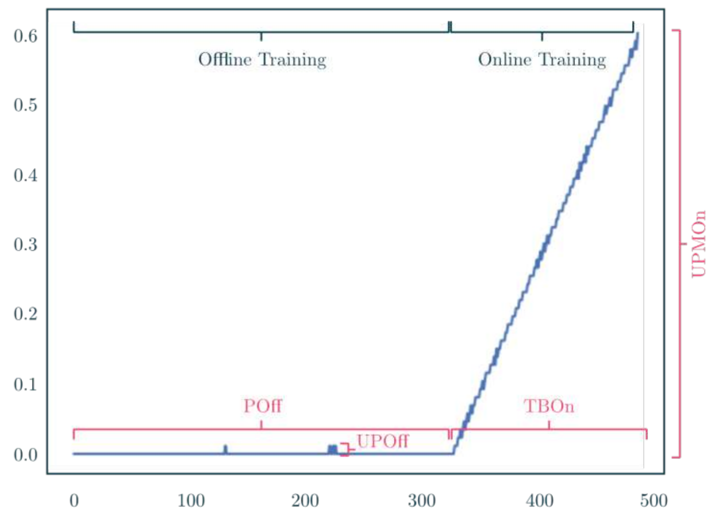

Figure 4.1: Illustrating percentage of underrepresented label data per batch in percent for off- and online datasets and the setting parameters.

this evaluation has a runtime of n^2, this will not be relevant for this evaluation since only two models with less then ten labels each are used.

Last Accuracy is chosen for the evaluation performance curve since it shows clearly how the model adapts to new label and when a plateau is reached. The drawback is that Last Accuracy is less stable the Next Accuracy since it only evaluates against one batch. For this metric, the values of the curves are normalized to their minimum and maximum. This yields a better comparability in figures since they then better align in hight and in values since the value differences are smaller and are then also more comparable for two different data sets. The resulting metrics is referred to as Normalized Last Accuracy (NLAcc). The difference between the curves, calculated as the mean of the NLAcc per batch is referred to as Normalized Last Accuracy mean Difference (NLAccDiff).

Additionally, the model should reach a evaluation performance plateau the earliest in the last third of the drift to ensure that mostly the evaluation is executed under a drift. After discussing the selected data and evaluation metrics, 'Determine Data Split Parameter Settings for Virtual Drift Simulation' (Section 4.5) will discuss the used settings per model.

Also, due to the limitations in the data amount, not all parameter combinations can be fulfilled. The automatic process will raise an error if a certain setting is not possible or undesirable. Since the label proportion in the used datasets are equal, these settings can remain the same for any other chosen label. Since the main focus of the split settings is the online drift simulation and it is mostly influenced by the underrepresented label data, all parameters regulating it are key. Hence, UPMOn, TBOn and UPOff must be selected carefully while POff can be dynamically set to the maximum possible value and use any remaining non underrepresented label data for offline training. Some underrepresented label data will surely be left over and hence, unused since its remaining data cannot be used during offline training to retain the drift effect.

4.1.4 Comparison to other Drift Simulations

Other Drift Simulations

The most popular online drift data sets can be categorized into two distinct groups:

Data Set Generators Data set generators [BS18, LLD$^+$19, pp. 352, pp. 12] mostly generates real value data which it generates from an underlying reoccurring simulation such as a math function. Due to the simulated source of the data it is easy to simulate drift and creates an abundance of data but only features easy reoccurring concepts. Then again, the resulting learning task does not require a large neural network for an accurate prediction. A more complex task is to categorize the rotation of a rotating chess board captured from above, which represents an image classification task.

Real-World Data Sets Real-world data sets [LLD$^+$19, pp. 12] also feature complex text classification or image classifications. The number of instances and the difficulty of the classification tasks, varies strongly. The summary table of [LLD$^+$19, p. 14] lists a number of popular datasets for which the number of instances ranges from 1500 to 2219803 and the types of data are text (7), regression (6), images (1). The start and end of the underlying drift in the data is not precisely known and it often features a variety of drift types.

Advantages of Implemented Drift System

A larger network is required for testing the VSOL since smaller network pose less of a storage problem which is part of the 'RQ2: Reduce Required Storage' (Section 1.2). Since any data with class labels can be used by the Implemented Drift System (IDS) (see Section 4.1.2), a data set can be chosen which requires a large neural network to be solve sufficiently and has a large number of instances. The mentioned 'Data Set Generators' (Section 4.1.4) are insufficient since the tasks are to easy. The mention 'Real-World Data Sets' (Section 4.1.4) are only partially sufficient due to their variability of the size and the difficulty of the task.

When only training on arriving online data without incorporating offline or past data, the model prediction quality deteriorates and requires retraining, as measure by [PDMM21]. Since the used models will not be tuned for online learning, as mentioned in 'Model Selection Process and Limitations' (Section 4.4), only a small number can be handled before the model quality deteriorates and the IDS only features a limited number of batches, which is determined in 'Determine TBOn Setting' (Section 4.5.1).

Additionally, the IDS specifies which data can be used for offline training. A neural network requires more data and iterations to learn the original concepts before the online learning phase than other machine learning techniques and drift should not occur during the offline training to keep the original concepts clear. Selecting the offline data becomes more elaborate since finding a subset of a real-world data set with stable concepts is hard since drift in the data sets is not clearly defined. Using an additional dataset without drift would require to find such a dataset with similar properties. Also, due to the artificial separation, one dataset can be easily divided several times and be randomized to achieve more reliable results, as mentioned in 'Data Splitting with K-Label Cross-Validation' (Section 4.2.1). While the start and end of the drift in 'Real-World Data Sets' (Section 4.1.4) is unclear, the drift of IDS is very precise since it is controlled artificially.

[PDMM21] for example solves those issues by splitting the dataset randomly without any criteria for the offline and online set. This approach is similar to the IDS and solves all of the issues mentioned above except that no clear drift arises during the online learning phase which is required. In summary, due the requirement to have a dataset with a hard task, a defined drift and sufficient data, only the IDS was sufficient and therefore required.

Limitations of Implemented Drift System

Since the Implemented Drift System (IDS) only provides one hard drift period, the VSOL is simulated exclusively under very hard conditions. Hence, it is unclear if the compression ratio (see Section 4.3.2) would be significantly lower in a full simulation were the weights change less since the underlying concepts of the drift are easier to learn. Due to using only one type of drift, it remains unclear if VSOL only can handle the tested type of drift and how the error rate would be affected by a different type of drift. Since the VSOL is only tested under drift it remains unclear how the evaluation metrics would change for a non drift period.

The drift of the IDS is limited by the instance number of the label with least instances since the online simulation requires many instances. Such a limit could be either due to too many labels in one data set or a limited number of instances in the data set. E.g. a drift lasting 150 batches (Total Batches Online data (TBOn)) with 80 instances per batch and a peak of 60% underrepresented labels in the last batch (Underrepresented label Percentage Maximum Online data (UPMOn)) would require at least 3600 instances per label, as shown by the Equation 4.1. A data set with 10 label would then require 36000 instances.

$$\begin{aligned}
\#batches * \text{TBOn} &= \#instancesOnline \\
&= 150 * 80 = 12000 \\
\text{UPMOn} * 0.5(slopeOfLabelIncrease) &= \%Underrepr.LabelOnline \\
&= 0.6 * 0.5 = 0.3 \\
\#instancesOnline * \%Underrepr.LabelOnline &= \#minInstancesPerLabel \\
&= 12000 * 0.3 = \underline{3600}
\end{aligned} \tag{4.1}$$

4.2 Evaluation Process

The evaluation process consists of splitting the data sets into an offline and online part, as described in the previous 'Online Learning Virtual Drift Simulation' (Section 4.1), training the offline model, simulating the online learning with the VSOL and then loading the saved weights for evaluating the evaluation performance. The following section will describe each step in detail and the limitations of this approach. Before each step is executed, the seed for all random functions involved is set. The seed creation is described in the next section.

4.2.1 Data Splitting with K-Label Cross-Validation

Other machine learning evaluations use K-Fold Cross-Validation to ensure that the result uphold under different validation splits. In this virtual drift section, the selected label could also have an unaccounted affect. Therefore, using different labels for different validation data sets of cross validation should be a good equivalent. To also apply a different seed per cross validation and keeping one stable seed, the label number and the seed are added to result in a different seed per label split. E.g. a dataset with ten labels will result in ten different data splits with their own seed each.

If the number of k-labels is not high enough for the targeted number of cross validations sets it would be possible to explicitly use the data from the offline split in the online split and vice versa. Since the online split of the data functions also as a validation set, this should have a similar affect. However, this approach was not implemented and therefore not explained in detail.

Hence, the process described in the next sections is repeated for each label of the dataset and the mean value of them is calculated.

4.2.2 Offline Training

The data for the offline training is chosen by the selected label, the described off- and online parameters (see Section 4.1) and a seed parameter. The seed is also set for any other random functions involved in the training process before this and any other step. Through theses given parameters, the resulting model is reproducible. The offline model is trained with the offline split of the data for several epochs.

The resulting model weights are saved so that all further tests on the VSOL do not have to retrain the offline model.

4.2.3 Online Training with VSOL

Before the online training is executed, the offline model is trained or loaded, as described in the section before. Then the online learning is executed with the online data set with increasing drift. The last batch of the dataset is omitted since it is only used for the evaluation. After each batch, the VSOL is executed synchronously, therefore blocking the next training step, with the newly trained weights TW_n. After the VSOL was executed, the deployment timestamp is saved to simulate an evaluation on the saved weights for TW_n. If the VSOL saved a new version, the timestamp is also saved in a separate list for saved runs. When loading a saved run, the index of saved run list shows which run number should be loaded.

4.2.4 Online Training Data Selection

In an online setting new training data with labels arrives and a system is then chosen when and how the new data is used to train the online model. Several different data chosing principles are explored, as mentioned in Section 2.3.2.

Since the envisioned online learning system should be easy to use and therefore only require little configuration and the main focus lies in the Processing Steps rather than a specific online learning adaption and the learning process such as the change calculation of the weights should not be changed, the simplest windowing technique is selected. This is a 'fixed landmark window' (described in [GBEB18, p. 23]) which collects a fixed number of new data instances. When this fixed number is reached, the data window is used as one batch for a learning iteration. After the bach was processed, the data batch is then discarded and the training process continues with a new batch when the fixed number of data samples is collected. This window technique also used by [PDMM21] as a baseline for online learning.

For simplification, it is assumed that the next batch can be executed immediately after the previous one since a queue of new data is always filled and the training process cannot keep up with the arriving data. Simulating time gaps between each batch has no advantages for the measured evaluation metrics and only increases the required time for evaluation.

Any further optimizations that leads to a better evaluation metrics can be integrated in the given version saving system as long as the network structure is not changed. E.g. chosing a different windowing technique for data selection can be used as long as it upholds the static batch size. 'Keras Usage and Integration' (Section 5.2) describes how such a different technique can be integrated into the existing pipeline.

4.2.5 Evaluation of Saved Versions

For the evaluation run the first batch is skipped, as described in 'Prequential Accuracy' (Section 4.3.1) in more detail. Then the simulation timestamp list is traversed and the saved version for the given timestamp is loaded. Since not every training step leads to a save, it is possible that the previous version is loaded again. Each version is loaded separately to measure the decompression time. Both evaluation measurements mentioned in Section 4.3 are evaluated while each version was loaded. Freshly loading the weights from the saved files guarantees that only the saved weights are used to evaluate the performance.

4.3 Evaluation Metrics

According to the set research questions, upholding the evaluation performance, a low compression time and a storage reduction is priority while the decompression time should be acceptable. In the following section each metric will be discussed. For any of the evaluation metrics, K-Label Cross-Validation is used. The results of each run are summarized via mean.The mean of all results from all of the runs are used for comparison. Also, mean is used to summarize the values of each individual run.

4.3.1 Prequential Accuracy

The standard error rate metric of an online learning algorithm is the prequential accuracy (see Section 2.3.5). This involves using each online sample first to evaluate the current model and then using it as the next training sample, i.e. after the model was trained with batch 1, the resulting model evaluates the data of batch 2 and the resulting accuracy is the computed. In a drifting scenario this helps to evaluate if the model is able to adapt to the occurring drift which is in this case present in the current and upcoming batch while also using data the model has not seen before. This is also very data saving since any data batch can be used for training as well as evaluation. Only two batches can be used for either evaluation and training. The first batch can only be applied for training since the model first has to train on at least one batch with drift so that its adaptability can be tested. One batch at the end can only be used for evaluation since using it also as a training batch would then result in having no fresh data batch for the model trained with it to be tested against. Hence, when n number of batches are available for online learning, only $n-1$ training batches are executed.

This metric reflects the 'RQ1: Impact Reduction on Online Learning' (Section 1.2) since a changing accuracy has a great impact on the model over all, as discussed.

Baseline The only baseline is using the weights of any system that preserves the weights from the training process. Depending on the comparison to other metrics, this can be just saving the weights directly or with any lossless Versioning System for Online Learning systems configuration. Depending on the use case, the VSOL should not fall too much below this metric.

Any accuracy measurement are given as a deviation from the lossless case since the deviation shows how much accuracy reduction has to be accepted to achieve another evaluation metric. This deviation is mentioned in several tables as 'Mean Diff to Lossless Last Accuracy' and in the text often abbreviated as "accuracy" since it is mostly the only accuracy measurement that was used. If a different accuracy measurement is used, it is explicitly mentioned. Positive values represent an improvement while negative values a loss in accuracy.

Drift Target Accuracy To clearly evaluate if the VSOL is adapting properly to the drift, the last online batch with the highest percentage of former underrepresented label of the online run is used to retrace that the accuracy is rising steadily. It works like prequential accuracy but instead of using the upcoming batch as validation data, the last batch is used. This measure is very unstable since its size is only one batch but can be used as a debugging measurement for lossy Processing Steps to see if this value steadily rises, as it should, which is not always clear with prequential accuracy. This debugging measurement is named 'Drift Target Accuracy' by us.

4.3.2 Compression Ratio

All saved versions are saved in one folder. This includes Delta Reset Points (see Section 3.2.8) which are used to save the start weights to include in the since there are mandatory for further deltas besides any other reset points. When the size of a run is evaluated, the folder size is measured. The in-memory data of the compression pipeline is not included since this should not carry weight when the VSOL already save many deltas. The compression ratio is the percentage of disk space the tested pipeline requires, compared to the main baseline. Hence, a higher value is better and refers to a lower file size. This measure directly corresponds to the 'RQ2: Reduce Required Storage' (Section 1.2).

Baselines To show the compression ratio improvement an additional Processing Step has on a configuration, a fitting baseline must be used.

1. The main baseline is using the **'General Compression'** to compress every production weight set individually. It will be used for any lossless configurations and for the final lossy configurations. Comparing other baselines to this one shows which online learning specific measures had an additional effect to a trivial approach.

2. Since any lossy VSOL configuration builds upon the best lossless configuration and the additional gains over only using the lossy VSOL should be shown, the **best lossless VSOL configuration** is used as a baseline for any lossy evaluation runs.

3. This baseline for evaluating the best 'General Compression' is saving each weight **without compression**. This has about the same size as you would have if you would use the checkpoint system of Keras. Since the checkpoint system of Keras is not thought out for a timestamp approach it was easier to save the weights directly instead of creating a wrapper code for it.

4.3.3 Compression Time

The compression time is measured during the online training run. Each step in the compression pipeline is measured on its own and then added. Since steps of the pipeline can be skipped, as described in 'Save Decision' (Section 3.2.1), the compression time can vary strongly from run to run. Therefore, the maximum compression time is also evaluated.

Baselines A upper bound baseline is the 'Batch Learning Time Baseline'. If the compression time exceeds the 'Batch Learning Time Baseline', the weights to deploy become outdated before they can even be rolled out, as discussed in 'Processing Time affecting Error Rate' (Section 3.3). This value will not be listed in any table since it does not have a running time. Some results will be close to this baseline or even above and will be proclaimed as such.

As additional baselines, the same baselines are used as for the 'Compression Ratio' (Section 4.3.2) to see how it increases compared to configurations with less steps.

4.3.4 Decompression Time

The decompression time is measured during the evaluation run where each saved set of weights is loaded. Since all lossy steps are already executed, the decompression time should be significantly lower then the compression time, excluding the load time if several deltas have to be loaded, see 'Delta Creator' (Section 3.2.3). Since the decompression time can vary strongly, as with the compression time, the mean max value of all the runs is reported additionally.

Baselines do not always make sense because the decompression time from some of the Processing Steps of the other baselines are very low due to not using delta encoding. As additional baselines, the same baselines are used as for the 'Compression Ratio' (Section 4.3.2) to see how it increases compared to configurations with less steps.

4.4 Selected Models and Data Sets

Requirements for Chosen Models The chosen models for the evaluation run must be implemented in Keras since it was preferred due do to the standardized interfaces and training process. 'Keras Usage and Integration' (Section 5.2) describes the reasons for chosing Keras in detail. The chosen Keras model should be not too large so that different ideas could be executed quickly on a consumer grade computer. They have very different layers and architectures to guarantee that the VSOL works well for different model and tasks. Both models have to be classifiers so that the chosen drift simulation can be executed (see Section 5.2).

Model Selection Process and Limitations Since the main focus of this thesis was not to have the best performing models but rather to keep up their performance, the used models were obtained from Kaggle since most models from papers are implemented in PyTorch and a translation to Keras would have been too elaborate and out of scope of this thesis. The models were then slightly adapted and simplified. To ensure that the chosen models are relevant, their size and accuracy are compared to other neural networks from leading papers in the sections bellow. The neural networks to compare to were selected from the website 'paperswithcode.com' [pap23b, pap23a] which rank high in accuracy and have a comparable architecture.

Furthermore, no online specific configurations were added to the selected models. Since the VSOL is only executed for a small number of batches, this should not be an issue, as argued in 'Advantages of Implemented Drift System' (Section 4.1.4). 'Neural Network Adaption' (Section 2.3.1) mentions several adaptions an online neural network could have. Since the VSOL is capable of handling any model configuration as long as the model architecture does not change, the chosen models can represent an online learning specific model since they both are made up of floats which change over time. It remains unclear how certain online learning specific configuration affects the degree and type of change in the floats and hence, the 'Compression Ratio' (Section 4.3.2).

Since only two types of neural networks are used for evaluation, it remains unclear if other neural network types would significantly change the metrics when using VSOL. Also, both networks have a smaller number of parameters compared to other models, as shown in Table 4.1 and Table 4.2, to reduce the required time for evaluation. Hence, the effect on the evaluation metrics of a model with significantly more parameters remains unclear.

Accuracy of Models The accuracy of the used models when trained with training data set is used to compare it state of the art models. The models were trained for 10 epochs with the full training data set. The Keras function 'Early Stopping' saved the weights and accuracy of the model with the best accuracy for the test data set from all epochs. This differs from the offline model training for the online validation run.

4.4.1 Image Recognition

For Image Recognition, the Dataset MNIST was chosen which features 60,000 hand drawn single digit numbers with a test set consisting of 10,000 pictures. The used convolutional neural network [kag17], referenced as Conv in this thesis, has an Error rate of 0.51 and 887,530 parameters. It consists of two convolutional layers followed by a max pooling and dropout layer. This sequence is repeated twice. The fully connected layer consists of 803,072 parameters which is over 90% of the overall number of parameters.

Table 4.1 shows the error rate and number of parameters for recent comparable neural networks. Both neural networks besides CapNet feature similar architectures to the chosen neural network Conv. While Conv's error rate is notably higher, the difference is not considerable. Additionally, the other training processes included additional steps which were not applied to Conv, such as the augmentation of the training data like in [Ass19]. Furthermore, the training of Conv did not include a learning rate annealer since this would be an additional parameter which must have been adapted for the online learning run. The number of parameters of Conv seems to be in the middle between the smaller and larger models which were chosen for this data set. Hence, Convs Error Rate, architecture and number of parameters seems to be close enough to represent other convolutional models for the online evaluation run.

	Error Rate	Number of Parameters
CapNet [BKD21]	**0.13**	1,514,187
SOPCNN [Ass19]	0.17	1,400,000
SimpleNet [HRFS16]	0.25	**300,000**
Conv (chosen for evaluation)	0.51	887,530

Table 4.1: Depicts error rate and number of parameters of current top image classification neural networks and model used in the evaluation process. Best values are highlighted.

4.4.2 Text classification

For text classification, the dataset AG News was chosen. It has 120000 training and 7600 test text snippets with four different text type labels. The used LSTM neural network [kag21], referenced as LSTM in this thesis, has an Error rate of 8.64 and 1,905,924 parameters. The embedding layer consists of 1,280,000 parameters, where 10,000 tokens each consist of 128 dimensions. Two bidirectional LSTM layers are used in sequence.

Table 4.2 shows the error rate and the number of parameters for recent comparable neural networks. LSTM has a large gap of 4.19 in the error rate compared to the best methods but since it does not use a pretrained set of word embedding, this difference should be explained by this and hence be tolerable. A pretrained embedding was omitted to keep the setup of the model low. As already mentioned, additional measures like using a learning rate annealer was omitted due to the online learning set. Since L MIXED has a similar architecture and size to LSTM, excluding the embedding size, it should be representative for other current neural networks tasked for text classification which are not based on a order of magnitude larger model size like XLNet.

	Error Rate	# Parameters
XLNet [YDY$^+$19]	**4.45**	~300,000,00 *
L MIXED [SZS19]	4.95	slightly more then LSTM **
LSTM (chosen for evaluation)	8.64	**1,905,924**

Table 4.2: Depicts error rate and number of parameters of current top text classification neural networks and model used in the evaluation process. Best values are highlighted. Obtaining the number of parameters for language models is not directly mentioned and hence harder to obtain. All models use word embeddings which can make up a large portion of all parameters.

* Describe in experiment as having a similar number of parameters as Bert Large which has about the shown number of parameters

** Not explicitly mentioned, excluding the very large embedding size due to similar architecture should be close to LSTM

4.5 Determine Data Split Parameter Settings for Virtual Drift Simulation

In this section, the parameter settings for the off- and online data split (see Section 4.1.2) will be determined through several sequential experiments evaluating different values per parameter. The data split parameters are chosen so that the Normalized Last Accuracy mean Difference (NLAccDiff) is minimized which ensures more comparability between the drift of the two neural networks. The figures in this section show the Normalized Last Accuracy (NLAcc) value which is closely related to NLAcc, as discussed in 'Optimal Split Settings' (Section 4.1.3) The chosen set of parameters will be used by all further experiments.

The NLAcc figures in the following sections show the curves for the third label. It was chosen to be representable for all other labels after looking at curves for labels. Selecting only one label reduces the number of curves that have to be shown per parameter and therefore reduce the complexity of the figures. The parameter POff is not mentioned in the following section since it is set automatically depending on the other values. The range of values for UPMOn and especially UPMOn are limited by the available data for one label. This limitation partially explains the chosen value ranges.

4.5.1 Determine TBOn Setting

The total number of batches of the online run should tend to be higher so that the model has more data to adapt to the drift. Processing Steps preventing saves, as described in 'Save Decision' (Section 3.2.1), require more batches since they can skip several saves which would distort the results if only a limited number of saves are executed since the number of batches is also low. TBOn should be the same for both models since otherwise compression techniques dependent on the number of saves lead to incomparable results. Hence, the resulting table only has one setting for both models.

Table 4.3 shows the NLAccDiff for different TBOn settings. The other parameters are set to fixed untested values and will be evaluated in the upcoming steps. UPMOn is set to 0.5, UPOff is set to 0 and POff is dynamic and set to the remaining data. 150 batches seems to be slightly better then 160 batches and clearly outperforming all other values.

TBonline	Mean Curve Diff
100	0.176
125	0.145
150	**0.122**
160	0.122

Table 4.3: Depicting the Normalized Last Accuracy mean Difference between the models Conv and LTSM for different **TBOn** settings.

Figure 4.2 shows why the TBOn setting 100 has a significantly worse NLAccDiff then 150. This is mainly due to the fact that LSTM reaches its optimum not at the last batch but in the last quarter and hence the normalized accuracy lines are more equivalent.

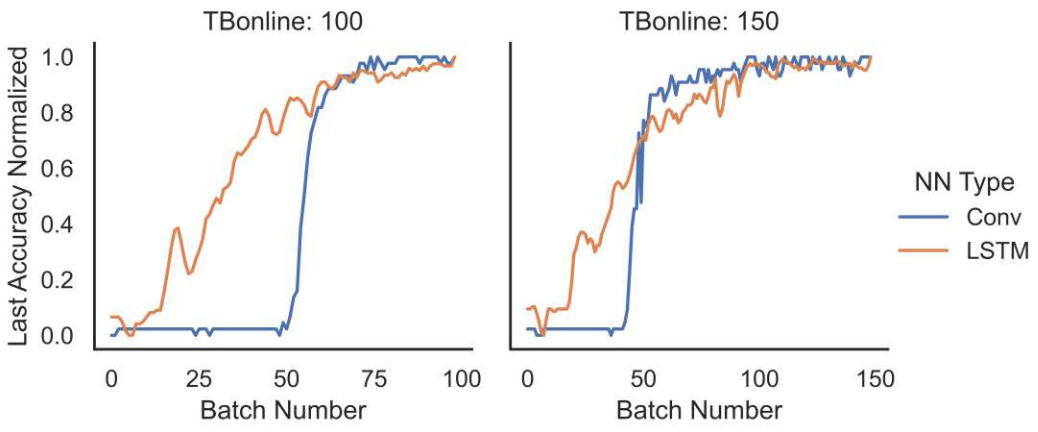

Figure 4.2: Depicts NLAcc during online training for the two used neural networks for different TBOn settings for specific label. X-Axis different since number of online batches change.

4.5.2 Determine UPOff Setting

UPOff is the percentage of the underrepresented label in the offline dataset. Having more data of the underrepresented label in the offline data set should lead to a significantly lower drift.

Table 4.4 shows the NLAccDiff for different UPOff percentage settings. TBOn is set to 150 from the last run and UPMOn is set to 0.5. Conv having a higher UPOff seems to result a worse difference while the opposite seems to be true for LSTM. Hence, the best UPOff settings seem to be 0.01% for Conv and 0.1% for LSTM.

Conv\LSTM	0.01	0.1
0.01	0.165	**0.115**
0.1	0.239	0.177

Table 4.4: Depicting the NLAccDiff between the models Conv and LTSM for different **UPOff** settings.

Figure 4.3 shows that a higher UPOff value leads to an early increase of NLAcc. This is expectable since the model was exposed to more data of the underrepresented label and hence, has to adapt less to it during the online run. Since Conv reaches its accuracy peak early it makes sense that the UPOff value is higher for LSTM then Conv. Both Conv and LSTM have an accuracy drop after an short increase after about 10 batches for a UPOff value of 0.001, which is not present with a value of 0.0001. This could be because the neural networks are over-fitted for the very small sample of the underrepresented label and first have to escape a local optimum before correctly classifying the underrepresented label data of the online data set. Hence, although a higher UPOff value can help to increase the curvature, as discussed above, this accuracy drop has to be considered.

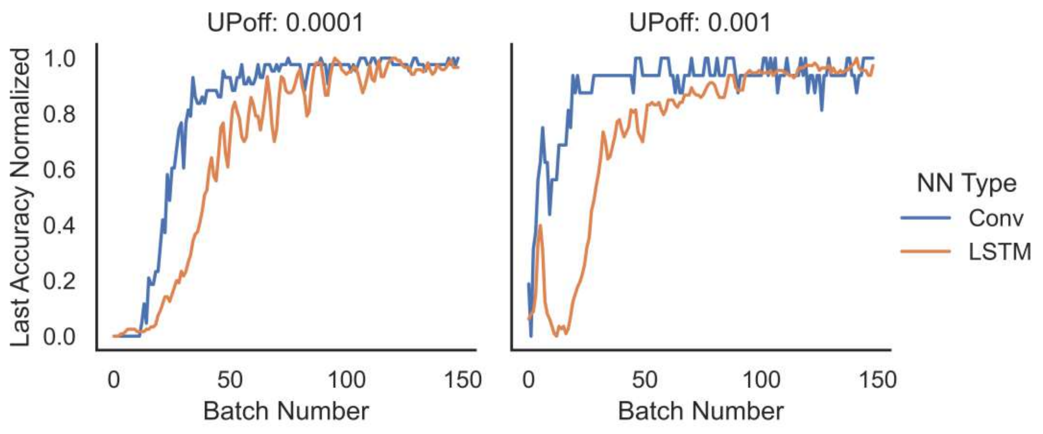

Figure 4.3: Depicts Normalized Last Accuracy during online training for the two used neural networks for different UPOff settings for specific label.

4.5.3 Determine UPMOn Setting

UPMOn is the maximum percentage of the underrepresented label in the last online batch after the linear increase from 0% at the beginning of the online data split. As with TBOn, it would be desirable that both Neural Networks share the same UPMOn value since otherwise each batch of data is not comparable and different accuracy values could result from different underrepresented label percentages. But since it does not affect any Processing Step directly, the comparison table will also feature combinations where the value is different per neural network.

Table 4.5 shows the NLAccDiff for UPMOn settings. According the table, a higher UPMOn for Conv seems to enlarge the difference between the NN while the opposite is true for LSTM except for the LSTM 50% column. Compared to the other metrics, the range of differences are not as great. Hence, having different UPMOn values per neural network to reduce the NLAccDiff does not outweigh the drawback mentioned above. Although having a UPMOn of 60% for both networks has the lowest NLAccDiff, 50% was chosen since the upcoming experiments where already executed before this table was created and since the difference is minimal, those experiments were not rerun with 60% instead of 50%. Since 60% is lower then 50% it is possible that a value above 60% would yield an even lower NLAccDiff value but the data amount for the LSTM is too low to test this hypothesis.

Conv\LSTM	40.0%	50.0%	60.0%
40.0%	0.107	0.115	**0.095**
50.0%	0.110	0.115	0.097
60.0%	0.119	0.126	0.102

Table 4.5: Depicting the NLAccDiff between the models Conv and LSTM for different **UPMOn** settings.

Figure 4.3 shows that the UPMOn value does not seem to have a clearly visible effect on the course of the curve. This could be explained by the fact that although the model now has more data of the underrepresented label during online learning, the last accuracy target batch has also a higher underrepresented label percentage and hence, the model must have also adapted more strongly to the additional label to reach the higher accuracy value.

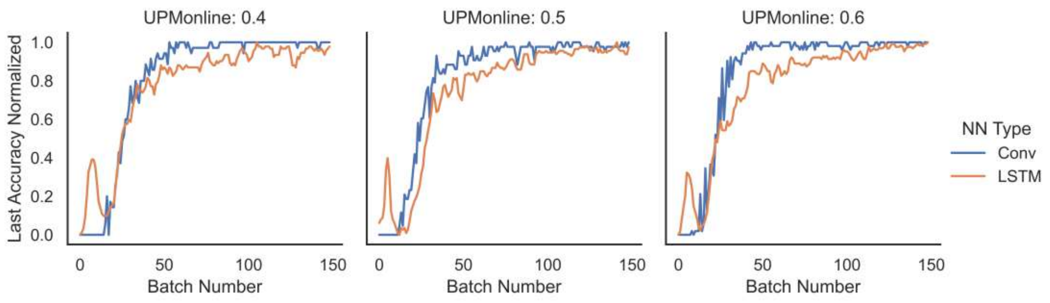

Figure 4.4: Depicts Normalized Last Accuracy during online training for the two used neural networks for different UPMOn settings for specific label.

4.5.4 Effectiveness of Virtual Drift Conversion

Considering the difference in data and architecture of the two neural networks and the amount of data available, the accuracy curves were fitted very closely. The Normalized Last Accuracy mean Difference (NLAccDiff) from all runs was improved to 0.115 from the maximum value of 0.239 which is a decrease of 0.124 or 208%. The best value is 0.095 (252%) but was not used and could have even further been improved with more data, as both discussed in 'Determine UPMOn Setting' (Section 4.5.3). Figure 4.4 shows the resulting NLAcc curves which are very close and also share a very similar curve shape.

Table 4.6 compares the improvement per data split parameter of the explored values. It shows that UPOff is the most influential by fare, being 115% away from the next best improvement, while TBOn and UPMOn only differ by 35%.

Parameter Name	Percentage improvement in %	Minimum	Maximum
UPOff	208%	0.115	0.239
TBOn	144%	0.176	0.122
UPMOn	117%	0.119	0.102

Table 4.6: Depicts improvement per data split parameter for virtual drift simulation in descending order. The improvement percentage is calculated between the minimum and maximum Normalized Last Accuracy of all explored parameter values.

4.6 Evaluation Concepts and Environment for Configuration Scenarios

Description of Procedure of the Evaluation and Underlying Thoughts Finding several usable and sensible configurations with the right settings should be the outcome of this evaluation. This ensures that a user can quickly identify which preset combination will fit their need. These scenarios will have different tradeoffs between the main metrics, prequential accuracy, compression percentage, compression time and decompression time, as discussed in 'Evaluation Metrics' (Section 4.3).

Since one full evaluation of a configuration can take up to 25 minutes on the test machine, testing all possible combinations is infeasible. Therefore, smaller portions of each Processing Step or configuration will be tested in isolation. Firstly, a small set of parameters featuring a good tradeoff must be found for each Processing Step to reduce the overall search space. After testing each Processing Step in isolation, they will be tested together and potentially adapted to work better together.

The main tradeoff is the prequential accuracy since it can affect the utility of the model itself. Hence, the user first would have to decide if the combination should be lossless or lossy. Since the lossy Processing Steps are added on top of the lossless algorithms, the combinations of lossless Processing Steps will be evaluated first. When useful lossless configurations are established, they can then be combined and tested in combination with lossy algorithms. This should significantly reduce the search space when adding even more algorithms.

Running the different configurations for two different neural networks with two different tasks should show that the Processing Steps not just apply for a specific case and are not tuned to be too specific. It is important that the evaluation metrics are stable for both neural networks and hence, a more stable result from an Processing Step is preferred over a great result for one neural network but performing poorly on the other.

Evaluation Metrics Calculation and Presentation For summarizing the results from each run per label, the mean is calculated. The compression ratio is calculated from the baseline of each table. Accuracies are given as deviation from the baseline of the table. A positive deviation represents an improvement over the baseline and vice versa.

The standard deviation of the accuracy per run was analyzed to ensure that none of the Processing Steps have strong deviations which would result in an unremarkable mean. Since none of the Processing Steps and their combination had a high standard deviation, the value was omitted from all tables to reduce their size. Often the Processing Step standard deviation was even smaller than the baseline.

Table row names contain an abbreviation of the used Processing Steps and Processing Step Parameters which are described close to the table. Each Processing Step name starts with a capital letter and followed by a number or capital letter for the set parameters. E.g., **Zstwm1c** refers to the **Zst**andard compression algorithm, using the **c**ompression rate of **1**. To shorten the row names of the configuration, Processing Steps which are used in each row are omitted and mentioned in the caption.

Undeterministic Learning Process Accuracies per run can vary since each run is not absolutely deterministic although a seed was set and the offline model is reloaded per executing. Tensorflow has a feature to only use deterministic algorithms, as mentioned in the documentation [ten22]. But since it is still experimental, very slow and potentially does not work for all of the used neural network layers, it will not be used and a small amount of variability will be accepted. Without any exact measurements but from observation the accuracy from the same run seems to deviate up to 0.04%, but this can strongly vary when the accuracy drops in general.

Selection Accuracy Bounds Since there are no baselines for accuracy, meaningful artificial boundaries must be chosen. Chosing boundaries significantly reduces the number of configurations that are evaluated in the final run. This is due to the reduction in the number of permutations when only a maximum of three parameters are selected per algorithm. The most extreme boundary would be a deviation around 0% accuracy of even a positive value. The next boundary was inspired by [LDG$^+$17], which states that a 0.1% decrease of model accuracy can lead to signification loss in revenue. Hence, staying above or close to 0.1% is the second boundary. To show the full potential of a lossy configuration, the very high bound of 1% is used which is probably too high for most use cases but does not reflect a complete degradation of the model. In summary, the accuracy bounds are 0%, 0.1%, 1%

Batch Learning Time Baseline Table 4.7 shows the per batch learning time. This is one of the baselines and will be used as a comparison to the compression time. It is referred to as 'Batch Learning Time Baseline'. Since these values are static and do not make sense in any of the tables, it will only be listed here once.

4.6.1 Evaluation Machine

The machine used for evaluation has a NVIDIA GeForce GTX 970 graphics card, an Intel i7-4790K processor, 32 GB of memory, a hard disk with a writing speed of 1000 MB/s

	'Batch Learning Time Baseline' (in sec.)	'Batch Learning Time Max Baseline' (in sec.)
Conv	0.033	0.074
LSTM	0.094	0.134

Table 4.7: Depicts the learning time of one batch for the selected neural networks in seconds.

and Windows 10 as operating system. The versions of the most important packages are 11.3 for CUDA and 2.10.1 for Tensorflow

The evaluations were executed in an Jupyter Notebook environment to keep values in memory such as the training data while experimenting with different settings for each algorithm.

4.7 Evaluation Lossless Configurations

The following section will evaluate all lossless Processing Steps in isolation and finally evaluate their combinations. Since the training weights are not changed by any of the lossless configurations evaluated in this section, which was verified for all of them, the accuracies are not impacted by the configurations and hence, will not be shown or evaluated. A small accuracy deviation is discussed in 'Undeterministic Learning Process' (Section 4.6) and hence, the Weight Change Degree can very slightly differ per evaluation, changing the compression circumstances. If only one number is given in the text for an evaluation metric, the rounded average between the values of the two neural networks are used. Two values starting with 'C' and 'L' in braces refer to the the values of Conv and LSTM each. 'Enumerated File Saver' (Section 3.2.8) is used in any configuration since saving the production weights to disk is required to archive them.

Although 'GCXS' (Section 3.2.6) is lossless, it will not be evaluated since it only reduces the compression ratio when the weights have a higher sparsity (>75%), which is only the case for 'Top-K' (Section 3.2.2), which resets several delta changes to zero. Otherwise it just increases the compression and decompression time and even decrease the compression ratio due to the additional indices array.

4.7.1 General Compression for Further Testing

In this section, one fast general compression algorithm should be chosen for any further individual Processing Step evaluations to reduce the time of the evaluation. Additional slower Processing Steps can be selected if they result in a significantly lower compression ratio. They will be reevaluated in the 'Final Lossless Configurations with Different Reset Point Interval Settings' section. 'General Compression' (Section 3.2.7) explains how the selection of general compression algorithms were chosen. In the following text, a number after the compression algorithm name refers to its compression level parameter setting.

Table 4.8 shows the main evaluation metrics of all chosen general compression algorithms with the highest and lowest compression setting, if available. If the compression setting revealed to have a drastic effect of some of the performance criteria, a fitting setting between the extremes was added, e.g. 9 for Brotli between 11 and 1.

The evaluation metrics between the two neural networks are comparable and none of the chosen Processing Steps perform vastly better on only one of the neural networks. Compared to [DKS19] the compression ratios are very low, e.g. Brotli has a compression ratio of 14.88 on a generic float data set while only having 1.093 on the convolutional neural network. Hence, the entropy of the neural network weights seems to be extensively higher than a normal float dataset.

The compression ratios of Brotli 11 (1.09) and LZMA (1.086) are higher then Zstandard 5 (1.081), which is algorithm with the next best compression ratio, while their compression time is at least 23 times higher and therefore unacceptable for the selection of the fast general compression algorithm. Zstandard 1 always has the lowest compression time (C: 0.017 | L: 0.024) while its compression ratio is maximally 0.001 less the any other Zstandard compression level which have the highest compression ratio after Brotli 11 and LZMA. **Zstandard 1** seems to have the right tradeoff between compression ration and time and will therefore be used as general compression algorithm for any individual compression algorithm evaluation.

Short Algorithm Name Differences	Compression Ratio		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)		Decompression Time Max (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-NoCompression	1.000	1.000	0.004	0.007	0.021	0.017	0.002	0.004	0.003	0.011
Browm11c	**1.093**	**1.091**	7.209	16.362	9.109	20.165	0.038	0.081	0.048	0.101
Browm10c	1.093	1.091	3.731	8.614	4.675	10.960	0.040	0.080	0.058	0.094
Lzmwm	1.086	1.085	0.753	1.906	0.841	2.062	0.160	0.332	0.187	0.366
Zstwm5c	1.080	1.082	0.033	0.058	0.050	0.118	0.012	0.023	0.023	0.037
Zstwm15c	1.080	1.082	0.110	0.197	0.163	0.235	0.012	**0.021**	0.027	0.038
Zstwm10c	1.080	1.082	0.115	0.212	0.164	0.245	**0.011**	0.022	**0.020**	0.042
Zstwm1c	1.080	1.081	**0.017**	**0.024**	0.034	0.051	0.014	0.023	0.039	0.048
Browm9c	1.080	1.081	0.130	0.306	0.163	0.415	0.026	0.052	0.036	0.063
Browm1c	1.080	1.081	0.019	0.036	**0.030**	**0.049**	0.027	0.052	0.040	0.061
Zstwm22c	1.079	1.081	0.560	1.075	0.976	1.175	0.012	0.021	0.026	**0.032**
Zliwm9c	1.078	1.079	0.142	0.292	0.166	0.337	0.017	0.038	0.028	0.060
Zliwm1c	1.075	1.076	0.128	0.264	0.146	0.298	0.018	0.039	0.030	0.063
Bz2wm9c	1.053	1.053	0.360	0.734	0.430	0.795	0.191	0.374	0.242	0.426
Bz2wm1c	1.037	1.037	0.344	0.718	0.386	0.771	0.159	0.337	0.191	0.395

Table 4.8: Evaluation Metrics of General Purpose Algorithms. A higher compression level of an algorithm leads to a higher compression ratio and compression time.

Row Names, Short Algorithm Names: Bz2wm = BZ2 + Zliwm = ZLib + Zst = ZStandard + Browm = Brotli, Numer + 'c' is compression level parameter, Lzmwm = LZMA , BL-NoCompression = No Compression Baseline - saving weights without any compression

4.7.2 Reset Point Frequency for Delta Creator

When using 'Delta Creator' (Section 3.2.3), reset points must also be used to bound the decompression time, as argued in 'Delta Reset Point Creator' (Section 3.2.8). Reset points therefore guarantee an upper limit of decompression time. When a reset point has to be saved, the full lossless compression pipeline has to be executed a second time for that save, increasing the compression time for that execution. Also, the compression ratio is affected not only by the additional reset points but also by the larger size of the weight matrix to be saved since they contain the materialization and not only a delta. The range of the 'Reset Point Interval' (RPI) parameter to test will span from 2 to 149. 2 is the lower limit since saving a reset point every save would exceed the memory the delta compression saved. Using the Total Batches Online data (TBOn) as an upper limit guarantees having one reset point at the first and last production weight set. This would be the minimal numbers of saves compared to only one save at the beginning without any additional reset points. Therefore, the upper limit is 148. The numbers in between were chosen to be denser in more relevant regions and have no jumps.

Figure 4.5 shows the compression ratio and the maximum decompression time for different RPI and the two used neural networks. The maximum time is used since the number of deltas between the weight to be decompressed to the reset point can vary drastically and therefore the worst case scenario should be evaluated. When the RPI decreases, the disk space slowly decreases close to not having a reset point while the decompression time increases linearly. The difference between the highest and lowest compression ratio for Conv is ≈93% while it is only ≈37.5% for LSTM. The curve shape of both neural networks seems to be similar. Since the maximum decompression time is still acceptable for 'RQ4: Reduce Retrieval Time' (Section 1.2), one setting would be to have a reset point at the beginning and end which would be the setting 148. 28 seems to be a good tradeoff since the decompression time is less then half of 148 but has a comparable compression ratio. Since a higher RPI leads to a higher execution time during execution, **28** is used for all further tests while **148** is only used for the final test.

Table 4.9 shows the chosen RPI 28 and 148. The compression times are still close but higher then the baseline compression time (C: 0.005 | L: 0.008). The compression ratios differ only slightly between the settings (C: ≈0.34%pt. | L: ≈0.05%pt.). This difference can increase when the deltas are more compressed and the reset points then have a higher impact. Although Delta Compression will enable more compression during the lossy experiments, the compression ratio increases are so signification that is also necessary for the lossless compression (C: 2.82 | L: 0.34). The compression ratios are vastly different for the two chosen neural networks (28: 2.19 | 148: 2.48).

4.7.3 Different Bytewise Segmentation for Float Matrices Algorithms

Table 4.10 shows three different 'Bytewise Segmentation for Float Matrices' (Section 3.2.6) implementations, which move similar byte segments closer to each in memory, making it easier for the 'General Compression' to identify these similarities and hence, reduce

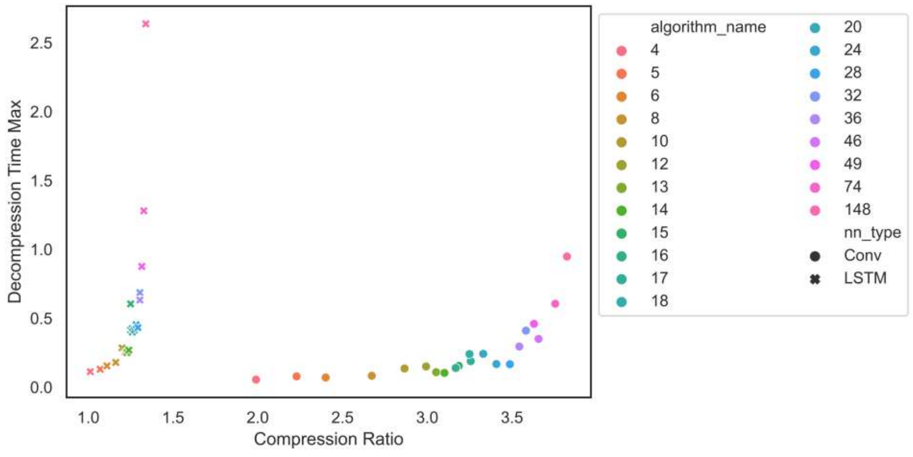

Figure 4.5: Depicts comparison of Decompression Time Max and Compression Ratio of different Reset Point Interval for both evaluated Neural Networks.

Short Algorithm Name Differences	Compression Ratio		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)		Decompression Time Max (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-GeneralCompr	1.000	1.000	0.014	0.023	0.031	0.041	0.013	0.022	0.031	0.052
Difrs148f	**3.824**	**1.340**	0.019	0.031	0.040	0.060	0.446	1.089	0.948	2.637
Difrs28f	3.487	1.293	**0.017**	**0.027**	**0.029**	**0.053**	**0.085**	**0.214**	**0.167**	**0.432**

Table 4.9: Evaluation Metrics of chosen Reset Point Interval.
Row Names, Short Algorithm Names: Difrs = DifResetSaver ('Delta Reset Point Creator') - Number + 'f' is Reset Point Interval , BL-GenComp = General Compression Baseline - Zstandard with compression level 1

the compression ratio. SplitFloatAndStackByByteSegmentsSplitLater leads to the same output as SplitFloatAndStackByByteSegments will having a slightly higher decompression time (C: 0.002 | L: 0.004) and is therefore excluded. SplitFloatAndStack has a lower decompression time (C: -0.012 | L: -0.029) and maximum compression time (C: -0.003 | L: -0.008) compared to the next lowest Processing Step SplitFloatAndStackByByteSegments. SplitFloatAndStackByByteSegments has a higher compression ratio compared to the next highest Processing Step SplitFloatAndStack (C: 0.011 (≈7%pt), L: 0.020 (≈6%pt)). Since both **SplitFloatAndStack** and **SplitFloatAndStackByByteSegments** have advantages and could interact differently with different general compression algorithms, they both will be used in the final lossless run.

4.7.4 Combining Selected General Compression and Bytewise Segmentation for Float Matrices Processing Steps for Full Run

To reduce the number of runs for the final run or rather execute the final run partially, the used Processing Steps will be reevaluated together with the delta compressor and the float split algorithms. This experiment should evaluate if maybe a former underperforming

Short Algorithm Name Differences	Compression Ratio		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)		Decompression Time Max (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-GeneralCompr	1.000	1.000	0.012	0.025	0.032	0.048	0.012	0.024	0.033	0.045
Splsl	**1.095**	**1.097**	0.017	0.029	0.033	0.048	0.023	0.051	0.038	0.068
Splbs	1.095	1.097	0.016	**0.027**	0.032	0.048	0.021	0.047	0.034	0.076
Splas	1.084	1.077	**0.016**	0.028	**0.029**	**0.040**	**0.009**	**0.018**	**0.019**	**0.032**

Table 4.10: Performance of different 'Bytewise Segmentation for Float Matrices' implementations. All runs also use Zstandard with compression level 1.
Row Names, Short Algorithm Names: Splas = SplitFloatAndStack, Splbs = SplitFloatAndStackByByteSegments, Splsl = SplitFloatAndStackByByteSegmentsSplitLater, BL-GenComp = General Compression Baseline - Zstandard with compression level 1

general compression algorithm can perform better together with 'Bytewise Segmentation for Float Matrices' and 'Delta Creator' Processing Steps then in the general float compression scenario.

Some of the fourteen general compression algorithms can be excluded for this run when reevaluating the float compression comparison. Table 4.8 shows than BZ2 1 features a considerably worse compression ratio (≈2%pt.) while being only slightly faster then BZ2 9 (-0.016). Hence, only BZ2 with the compression parameter of 9 will be considered. The same applies for Zlib, although the compression difference is only minimal (≈0.0035%pt.). Brotli with a compression parameter of 9 will be also removed since parameter 1 achieves the same compression while being up to 7 fold faster and the higher compression parameter of 10 will still be evaluated. Since 11 almost required double the compression time while having the same compression ratio, 11 will also be omitted. The Zstandard parameters 1, 5, 10 and 15 had similar compression ratios while their compression times differed according to their compression level. Therefore, only the extreme parameters 1 and 15 where used. Since Zstandard 22 is significantly slower compared to Zst 15 (+ C: 0.45 | L: 0.878) while having a similar performance as the rest of the Zstandard parameters and the high compression level of 15 is still used, it will also be omitted.

In conclusion, 7 compression algorithms were remove while 7 will still be used for the run.

The Reset Point Interval of the 'Delta Reset Point Creator' (Section 3.2.8) will be set to 28 to execute the runs faster. To see if some general compression algorithms deliver vastly different result depending on other used algorithms, the selected implementations 'SplitFloatAndStackByByteSegments' and 'SplitFloatAndStack' from the Processing Step 'Bytewise Segmentation for Float Matrices' (Section 3.2.6) and omitting this Processing Step will be evaluated in combination with the other selected general compression algorithms.

Table 4.11 shows the chosen general compression algorithms with the 'Bytewise Segmentation for Float Matrices' Processing Steps chosen in Section 4.7.3. SplitFloatAndStackByByteSegmentsSplitLater seems to always lead to better compression ratios regardless of the general compression algorithm although the total difference is now insignificant. The compression ratio order of the table is the same as in Table 4.8 and hence, none of the general compression algorithms could benefit from the output of any Processing Step combination significantly, except Zstandard 1 which is now slightly better than Zstandard 15 (≈0.01). Hence, the hypothesis of this test can be rejected.

Brotli 10 and LZMA have the highest compression ratio, being both close to 3.9 for Conv and 1.545 fro LSTM while the next best compression ratios of Zstandard 1 are close to 3.7 and 1.50, but as before, LZMA only requires a third of the compression time while having about double the decompression time and hence will be used in all other final runs. Although SplitFloatAndStackByByteSegments has a very slight better compression ratio (+ C: 0.011 | L: 0.003), SplitFloatAndStack will be used for LZMA since it is always significantly faster, especially for the decompression time (- C: 0.1 | L: 0.08). Zstandard 1 outperforms Zstandard 15 in any measurement except for the decompression time of LSTM, hence Zstandard 1 will be used. SplitFloatAndStackByByteSegments for Zstandard 1 seems to significantly outperform SplitFloatAndStack in all measurements except the decompression time for LSTM, hence it will be used for the final run. All other general compression algorithms seem to be too slow while also having a worse compression ratio and are therefore excluded for the final run.

Resulting from this experiment, **LZMA with SplitFloatAndStack** and **Zstandard 1 with SplitFloatAndStackByByteSegments** will be used as a slower and faster configurations for the final run.

4.7.5 Final Evaluation Lossless Configurations

For the final run, the slow LZMA and fast Zstandard combinations, selected from the last section, will be combined with the two selected Reset Point Interval 28 and 148 from 'Reset Point Frequency for Delta Creator' (Section 4.7.2):

Table 4.12 shows the results of the final run configurations. Any Zstandard confiugration was able to stay well below the baseline of the 'Batch Learning Time Baseline' (C: 0.033 | L: 0.094) while still having a compression ratio close to LZMA (max. - C: 0.24 | L: 0.03). Hence, any LZMA configuration is excluded from being a final recommended configuration. The compression times are twice as much as the general compression baseline although this only applies for LSTM while Conv is very close to the baseline (-0.005). The compression ratio of 1.19 from [MLDD17b] for using delta compression with 'Bytewise Segmentation for Float Matrices' was vastly exceeded although the test settings are very different and hence, the evaluation of the paper could have had more different weights then in this online learning setting.

SplbsZstwm1cDifrs148f will be selected as the only preset lossless configuration for the VSOL. The Reset Point Interval is not mentioned in the configuration name since it will be selected during the runtime, based on the fixed decompression time threshold, as described in 'Delta Reset Point Creator' (Section 3.2.8). Since all other configurations

Short Algorithm Name Differences	Compression Ratio		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)		Decompression Time Max (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-GeneralCompr	1.000	1.000	0.017	0.024	0.039	0.043	0.014	0.024	0.032	0.043
SplbsBrowm10c	**3.918**	1.554	1.806	6.087	3.933	11.729	0.339	0.952	0.697	1.926
SplasBrowm10c	3.912	**1.555**	1.830	6.027	3.989	12.043	0.234	0.778	0.480	1.582
SplbsLzmwm	3.896	1.535	0.633	1.995	1.538	4.453	0.545	1.283	1.142	2.989
SplasLzmwm	3.885	1.532	0.629	1.986	1.533	4.254	0.446	1.205	0.961	2.733
SplbsZstwm1c	3.702	1.508	**0.030**	**0.048**	**0.062**	**0.089**	0.258	0.475	0.524	0.942
SplbsZstwm15c	3.680	1.504	0.224	0.658	0.481	1.333	0.257	0.713	0.518	1.465
SplasZstwm1c	3.668	1.495	0.070	0.050	0.129	0.093	0.371	**0.299**	0.729	**0.612**
SplasZstwm15c	3.646	1.497	0.225	0.649	0.600	1.402	**0.154**	0.544	**0.326**	1.103
SplbsBrowm1c	3.609	1.494	0.033	0.099	0.066	0.161	0.339	0.861	0.674	1.763
SplasBrowm1c	3.588	1.484	0.031	0.100	0.065	0.180	0.234	0.729	0.475	1.457
SplbsZliwm9c	3.571	1.497	0.784	1.735	1.638	3.687	0.322	0.830	0.648	1.660
SplasZliwm9c	3.550	1.490	0.781	1.659	1.596	3.465	0.216	0.679	0.435	1.380
SplbsBz2wm9c	3.466	1.470	0.148	0.603	0.513	1.343	0.803	2.651	1.552	5.279
SplasBz2wm9c	3.450	1.464	0.147	0.603	0.507	1.363	0.706	2.480	1.362	4.752

Table 4.11: Performance of different combinations of 'Bytewise Segmentation for Float Matrices' implementations and different general compression algorithms. All runs also use Zstandard with compression level 1 and 'Delta Creator' with a Reset Point Interval of 28.

Row Names, Short Algorithm Names: Splas = SplitFloatAndStack, Splbs = SplitFloatAndStackByByte-Segments, Splsl = SplitFloatAndStackByByteSegmentsSplitLater, Bz2wm = BZ2 + Zliwm = ZLib + Zst = ZStandard + Browm = Brotli, Numer + 'c' is compression level parameter, Lzmwm = LZMA , BL-GenComp = General Compression Baseline - Zstandard with compression level 1

have signification drawbacks as described above, it will be the only selected lossless configuration.

Short Algorithm Name Differences	Compression Ratio		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)		Decompression Time Max (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-GeneralCompr	1.000	1.000	0.022	0.025	0.040	0.048	0.016	0.024	0.039	0.045
SplasLzmwmDifrs148f	**4.200**	**1.591**	0.582	1.907	1.591	4.046	1.985	6.065	4.089	12.529
SplbsZstwm1cDifrs148f	3.962	1.565	0.028	**0.050**	0.059	**0.086**	1.014	2.306	2.040	4.536
SplasLzmwmDifrs28f	3.796	1.531	0.617	1.990	1.515	4.238	0.404	1.198	0.866	2.732
SplbsZstwm1cDifrs28f	3.631	1.507	**0.027**	0.056	**0.053**	0.091	**0.221**	**0.496**	**0.601**	**0.992**

Table 4.12: Performance of final lossless run. Selected Processing Steps with settings are the result of the past evaluations.

Row Names, Short Algorithm Names: Splas = SplitFloatAndStack, Splbs = SplitFloatAndStackByByteSeg-ments, Splsl = SplitFloatAndStackByByteSegmentsSplitLater, Zst = ZStandard - Numer + 'c' is compression level parameter Lzmwm = LZMA , Difrs = DifResetSaver ('Delta Reset Point Creator') - Number + 'f' is Reset Point Interval , BL-GenComp = General Compression Baseline - Zstandard with compression level 1

4.7.6 Summary

For the chosen lossless configuration 'SplbsZstwm1cDifrs148f', we can expect a compression time between 0.27 and 0.56, which is below the 'Batch Learning Time Baseline' (C: 0.033 | L: 0.094), with a maximum of 0.091. The compression ratio is unstable and ranges between 1.59 and 4.20. The decompression time ranges between 0.221 and 1.014 with a maximum of 4.536. This configuration consists of SplitFloatAndStackByByteSegments, Delta Creator and Zstandard 1. It will be used for all further lossy evlautions in the upcoming section.

4.8 Lossy Algorithms

The baseline of all lossy evaluations is the chosen lossless configuration 'SplbsZstwm1cDifrs148f', mentioned in 'Final Lossless Configurations with Different Reset Point Interval Settings' (Section 4.8.5). This helps to identify differences which only result from the additional lossy algorithms. Table 4.12 shows that LSTM has a lower compression ratio compared to Conv for the lossless baseline. Hence, LSTM requires ≈2.5 times higher compression ratio then Conv in any of the tables to achieve an equal compression ratio compared to the general compression baseline. This difference should be considered when LSTM has a significantly higher value then Conv in the upcoming tables. 'Selection Accuracy Bounds' (Section 4.6) argues and declares the accuracy bounds of 0%, 0.1% and 1%.

4.8.1 Top-K

'Top-K' (Section 3.2.2) requires a fitting Top-k-Percentage (TkP) Processing Step Parameter. Any result which deviates by more then 1% accuracy exceed the defined bounds. But since it is expected that the next evaluation of 'Minimum per Layer' (Section 3.2.2) will increase the accuracy for any TkP, the cutoff value is doubled to 2% during this evaluation. The selected TkP to test includes the values 1%, 0.1% and 0.01% of [PDMM21], although the application of the method differs and should therefore lead to different results, together with additional complementary values. After testing the three values from the paper, the TkP values were expanded up to 25% to achieve acceptable accuracy results. The tested TkP values are 0.01%, 0.1%, 0.25%, 0.5%, 1%, 1.75%, 2.5%, 5%, 7.5%, 10%, 15% and 25%, expanding the values between 0.1% and 1% by two values and the range between 1% and 25% by six values.

A small evaluation not depicted in this thesis showed that the accuracy is higher when the selection of Top-K is applied on the real and not the xor difference (see Section 3.2.3). It is faster since the changes to the xor difference values have to be reapplied to the previous weights.

Table 4.13 shows the evaluation metrics for the selected TkP Processing Step Parameters. The compression ratio of LSTM is always significantly higher then Conv. The accuracy of Conv is always higher then for LSTM except for a TkP of 0.01%. A lower TkP seems to have a slight positive effect on the compression and decompression time. E.g., the compression time difference between *TkP* of 25% and 0.01% is (C: -0.023(≈33%)| L: -0.072(≈%0.25)) while the decompression time difference is (C: -0.041(≈0.13%)| L: -0.098(≈0.12)). All compression times exceed the 'Batch Learning Time Baseline' (C: 0.033 | L: 0.094) and hence, the lossless compression baseline, especially LSTM. E.g., *TkP* of 5% exceeds the training baseline by (C: -0.026 | L: -0.158).

The results show that a post training Top-k requires a higher *TkP* than described in [PDMM21] where a k of 0.01% still leads to almost the same accuracy. For the post training Top-k, *TkP* below 10% already lead to a signification accuracy drop, being -0.1 for LSTM. Any *TkP* below 1.75 is below the set 2% accuracy, hence these *TkP* will not be evaluated in further evaluations. Also, 25% seems to be too high since 15% has almost the same accuracy and hence, can be excluded. This also applies to 10% being similar to %7.5. Hence, the *TkP* values 15%, 7.5%, 5%, 2.5% and 1.75% will be used in further evaluations.

Short Algorithm Name Differences	Compression Ratio		Mean Diff to Lossless Last Accuracy		Compression Time (sec.)		Compression Time (sec.) Max		Decompression Time (sec.)		Decompression Time (sec.) Max	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.000	0.000	0.030	0.037	0.063	0.079	0.262	0.434	0.530	0.875
Topk25.0p	1.147	2.001	0.210	**0.010**	0.070	0.287	0.117	0.373	0.306	1.175	0.614	2.413
Topk15.0p	1.355	2.658	**0.220**	-0.010	0.066	0.274	0.115	0.360	0.302	1.169	0.603	2.377
Topk10.0p	1.626	3.348	0.170	-0.050	0.063	0.263	0.096	0.349	0.296	1.156	0.607	2.349
Topk7.5p	1.855	3.949	0.160	-0.100	0.062	0.258	0.101	0.341	0.299	1.157	0.612	2.344
Topk5.0p	2.241	4.976	0.060	-0.250	0.059	0.252	0.093	0.334	0.289	1.137	0.582	2.252
Topk2.5p	3.111	7.218	-0.350	-0.710	0.054	0.240	0.090	0.341	0.283	1.131	0.562	2.315
Topk1.75p	3.627	8.545	-0.670	-1.170	0.053	0.235	0.086	0.311	0.280	1.111	0.577	2.286
Topk1.0p	4.486	10.698	-1.420	-2.180	0.051	0.230	0.086	0.311	0.275	1.117	0.541	2.278
Topk0.5p	5.449	13.096	-2.990	-4.700	0.049	0.226	0.089	0.305	0.268	1.083	0.548	2.196
Topk0.25p	6.155	14.910	-5.650	-8.830	0.049	0.220	0.084	0.299	0.264	1.088	0.536	2.173
Topk0.1p	6.725	16.429	-11.900	-16.010	0.047	**0.215**	0.089	**0.294**	**0.262**	1.064	**0.526**	2.142
Topk0.01p	**7.220**	**17.740**	-27.890	-24.380	**0.047**	0.215	**0.082**	0.334	0.265	**1.059**	0.566	**2.123**

Table 4.13: Depicts Top-k with different ks. All evaluations also use SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Row Names, Short Algorithm Names: Topk = Top-K - Number + 'p' is Top-k-Percentage - Number + 'm' is 'Minimum per Layer' - Number + 'l' is 'Loss Adaptive *TkP*' were 'l' is the Loss Transformation Function ($f_{LoTrans}$) (see Figure 4.6) , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Using GCXS for Spare Delta Weights

The compression ratio of the previous section falls behind the theoretical calculations of [PDMM21]. This calculation assumes that only the indices with the values are transmitted since most of the values are empty. 'GCXS' (Section 3.2.6) uses exactly this encoding and shows if such a scheme can help to reduce the memory to said minimum. To have extreme and practical sparsity settings, the extremes of the *TkP* values of Table 4.13

are used in the following experiments which are 15% and 1.75%. Since this is a lossless algorithm, accuracy is not given in the tables.

Table 4.14 shows pure GCXS as well as applying 'Bytewise Segmentation' (Section 3.2.6) for the values and 'Combine Layers' (Section 3.2.6). For 15%, where GCXS should have the least positive effect, applying both 'Combine Layers' and 'Bytewise Segmentation' together can improve all metrics, especially for LSTM. For 1.75%, already raw GCXS outperforms only using SplitFloatAndStackByByteSegments without GCXS in all metrics. In general, applying either 'Combine Layers' and 'Bytewise Segmentation' improves any metric compared to raw GCXS. Using both improves all metrics even further, although the compression time is slightly slower then not using 'Bytewise Segmentation' for 1.75% (+0.001). 'Bytewise Segmentation' on top of GCXS seems to have a larger effect on the compression ratio than using 'Combine Layers'. Especially the decompression time seems to benefit from GCXS and can even outperform the lossless baseline. Again, a smaller TkP leads to overall lower metrics.

Seeing that using GCXS with both 'Combine Layers' and 'Bytewise Segmentation' only has positive effects when using Top-k, it will be applied to all further Top-k evaluations.

Short Algorithm Name Differences	Compression Ratio		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)		Decompression Time Max (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.023	0.037	0.048	0.069	0.207	0.437	0.427	0.932
Topk15.0pGcxxsTsTcFsFd	1.418	3.377	0.039	0.071	0.069	0.115	0.087	0.173	0.171	0.338
Topk15.0pSplbs	1.326	2.664	0.040	0.249	0.063	0.320	0.211	1.092	0.453	2.230
Topk15.0pGcxxsTsFcFsFd	1.326	3.085	0.052	0.106	0.081	0.143	0.190	0.376	0.436	0.818
Topk15.0pGcxxsFsTcFsFd	1.288	3.020	0.040	0.078	0.075	0.103	0.098	0.205	0.208	1.071
Topk15.0pGcxxsFsFcFsFd	1.204	2.806	0.053	0.114	0.095	0.146	0.186	0.360	0.373	0.770
Topk1.75pGcxxsTsTcFsFd	**4.432**	**11.241**	0.028	0.050	0.060	0.099	**0.068**	**0.123**	**0.134**	**0.235**
Topk1.75pGcxxsTsFcFsFd	4.279	10.844	0.029	0.054	0.053	0.090	0.096	0.166	0.186	0.348
Topk1.75pGcxxsFsTcFsFd	4.144	10.456	**0.027**	**0.049**	**0.047**	**0.076**	0.069	0.129	0.138	0.263
Topk1.75pGcxxsFsFcFsFd	4.031	10.054	0.028	0.054	0.053	0.095	0.094	0.174	0.209	0.335
Topk1.75pSplbs	3.511	8.562	0.031	0.213	0.064	0.282	0.198	1.041	0.490	2.107

Table 4.14: Depicts GCXS raw and with compression of Top-k values with minimum and maximum chosen TkP. All evaluations also use Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Row Names, Short Algorithm Names: Topk = Top-K - Number + 'p' is Top-k-Percentage - Number + 'm' is 'Minimum per Layer' - Number + 'l' is 'Loss Adaptive TkP' were 'l' is the Loss Transformation Function ($f_{LoTrans}$) (see Figure 4.6) , Gcxxs = GCXS - Bool + 's' is if apply SplitFloatAndStackByByteSegmentsto full diff and values in GCXS - Bool + 'c' is combine all layers before applying GCXS - Bool + 's' is if apply SplitFloatAndStackByByteSegmentsto indices - Bool + 'd' is if apply differential encoding for indices , Splbs = SplitFloatAndStackByByteSegments, BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStack-ByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Compressing GCXS Indices Table 4.15 shows the effect of applying 'Bytewise Segmentation' for indices and 'Index Differential Encoding'. Both 'Index Differential Encoding' and 'Bytewise Segmentation' have a positive effect on the compression ratio while 'Index Differential Encoding' has a higher effect. Both methods increase the decompression time, especially 'Index Differential Encoding'. For LSTM, both parameters seem to have a negative effect on the compression time (≤ 0.009) while for Conv, their effect on the compression time is inconclusive. When adding 'Bytewise Segmentation' to 'Index Differential Encoding', it can lower its effect on all of the time metrics. This effect is lower with a lower TkP and Conv can even improve the compression time with both compression technics. Generally, the improvement through the indices compression seems to have a higher effect on Conv then on LSTM.

Since both 'Index Differential Encoding' and 'Bytewise Segmentation' on the indices have a positive effect on the compression ratio and only a slight negative effect on the time metrics, they will be used for all further Top-k evaluations.

Short Name Algorithm Differences	Compression Ratio		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)		Decompression Time Max (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.023	0.037	0.048	0.069	0.207	0.437	0.427	0.932
Topk15.0pGcxxsTsTcTsTd	1.668	3.853	0.039	0.080	0.060	0.124	0.092	0.186	0.183	0.361
Topk15.0pGcxxsTsTcFsTd	1.621	3.779	0.041	0.080	0.062	0.120	0.091	0.203	0.192	0.411
Topk15.0pGcxxsTsTcTsFd	1.540	3.452	0.038	0.078	0.068	0.120	0.088	0.182	0.169	0.383
Topk15.0pGcxxsTsTcFsFd	1.418	3.377	0.039	0.071	0.069	0.115	0.087	0.173	0.171	0.338
Topk1.75pGcxxsTsTcTsTd	**4.621**	**11.659**	**0.025**	0.053	**0.043**	0.162	0.071	0.136	0.143	0.259
Topk1.75pGcxxsTsTcFsTd	4.553	11.560	0.027	0.056	0.046	**0.090**	0.071	0.147	0.150	0.285
Topk1.75pGcxxsTsTcTsFd	4.484	11.235	0.027	0.053	0.054	0.091	0.069	0.142	**0.133**	0.276
Topk1.75pGcxxsTsTcFsFd	4.432	11.241	0.028	**0.050**	0.060	0.099	**0.068**	**0.123**	0.134	**0.235**

Table 4.15: Depicts GCXS with compression of indices together with Top-k using minimum and maximum TkP. All evaluations also use Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Row Names, Short Algorithm Names: Topk = Top-K - Number + 'p' is Top-k-Percentage - Number + 'm' is 'Minimum per Layer' - Number + 'l' is 'Loss Adaptive TkP' were 'l' is the Loss Transformation Function ($f_{LoTrans}$) (see Figure 4.6) , Gcxxs = GCXS - Bool + 's' is if apply SplitFloatAndStackByByteSegmentsto full diff and values in GCXS - Bool + 'c' is combine all layers before applying GCXS - Bool + 's' is if apply SplitFloatAndStackByByteSegmentsto indices - Bool + 'd' is if apply differential encoding for indices , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Minimum per Layer

Table 4.16 shows the selected TkP combined with fitting 'Minimum per Layer' (Section 4.8.1) values. Using the 'Minimum per Layer Percentage' ($minLP$) parameter seems to have a positive effect on the accuracy, especially when TkP is lower, while having only a minimal difference for the compression ratio, e.g., TkP of 5% with $minLP$ of 0.1% for LTSM decreases the compression ratio by ≈0.1 while increasing accuracy by ≈0.1. The *minLP* of 0.1 always has a better accuracy than 0.01, except for *TkP* 15%. Using this parameter does not have an effect on the time metrics. 'Minimum per Layer' seems to work generally slightly better for Conv than for LSTM.

The following three configurations were chosen for the determined accuracy bounds: **Top-k 7.5% with *minLP* of 0.1% (accuracy bound 0%)** is close to a 0 accuracy difference for LSTM while having a strong positive impact on Conv. **Top-k 5% with *minLP* of 0.1% (accuracy bound 0%)** is close to a 0.1 accuracy difference for LSTM while still having a positive impact for Conv. **Top-k 1.75% with *minLP* of 0.1% (accuracy bound 1%)** was chosen over *minLP* of 0.01%, which has the best compression ratio below 1%, since 0.1% *minLP* features a better accuracy, especially for Conv, while the compression ratio is only less then 5% lower. These *TkP* and *minLP* value combinations will be used in all further evaluation.

Short Name Algorithm Differences	Compression Ratio		Mean Diff to Lossless Last Accuracy		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)		Decompression Time Max (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.000	0.000	0.030	0.037	0.062	0.079	0.258	0.434	0.524	0.875
Topk15.0p0.1m	1.697	3.840	0.240	**0.050**	0.060	0.187	0.094	0.241	0.153	0.489	0.321	1.003
Topk15.0p0.01m	1.709	3.856	**0.280**	0.030	0.060	0.184	0.103	0.251	0.154	0.488	0.317	1.001
Topk15.0p	1.711	3.860	0.260	0.010	0.059	0.185	0.091	0.245	0.151	0.490	0.312	1.017
Topk7.5p0.1m	2.542	5.991	0.240	0.000	0.056	0.174	0.090	0.238	0.145	0.478	0.303	0.974
Topk7.5p0.01m	2.576	6.039	0.230	-0.030	0.056	0.174	0.085	0.223	0.144	0.475	0.301	0.976
Topk7.5p	2.582	6.054	0.220	-0.080	0.055	0.170	0.088	0.223	0.141	0.468	0.289	0.942
Topk5.0p0.1m	3.077	7.445	0.180	-0.100	0.051	0.171	0.080	0.229	0.132	0.493	0.275	1.039
Topk5.0p0.01m	3.128	7.540	0.150	-0.110	0.052	0.170	0.084	0.225	0.134	0.495	0.282	1.042
Topk5.0p	3.137	7.567	0.120	-0.220	0.052	0.169	0.084	0.225	0.131	0.481	0.270	0.971
Topk2.5p0.1m	4.069	10.024	0.020	-0.370	0.044	0.158	0.075	0.216	0.124	0.465	0.265	0.960
Topk2.5p0.01m	4.201	10.270	-0.180	-0.420	0.045	0.159	0.080	0.216	0.128	0.479	0.269	0.987
Topk1.75p0.1m	4.540	11.140	-0.080	-0.590	0.041	0.148	0.073	**0.199**	0.111	0.460	0.236	0.946
Topk1.75p0.01m	4.750	11.589	-0.410	-0.690	0.040	0.148	**0.068**	0.200	**0.109**	0.460	**0.226**	0.950
Topk2.5p	4.220	10.329	-0.320	-0.710	0.042	0.155	0.079	0.209	0.114	**0.456**	0.236	**0.929**
Topk1.75p	**4.782**	**11.674**	-0.610	-1.120	**0.039**	**0.145**	0.070	0.214	0.109	0.458	0.234	0.936

Table 4.16: Depicts Top-k with different *TkP* and minium per layer values. All evaluations also use Zstandard with compression level 1 , 'Delta Reset Point Creator' with Reset Point Interval of 28 and GCXS with all parameters set to true.

Row Names, Short Algorithm Names: Topk = Top-K - Number + 'p' is Top-k-Percentage - Number + 'm' is 'Minimum per Layer' - Number + 'l' is 'Loss Adaptive *TkP*' were 'l' is the Loss Transformation Function ($f_{LoTrans}$) (see Figure 4.6) , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

All Layers Top-K

Table 4.17 compares 'All Layer Top-k' (Section 3.2.2) and Top-k while using 'Minimum per Layer'. A small evaluation showed that the accuracy of 'All Layer Top-k' degrades when not using 'Minimum per Layer'.

'All Layer Top-k' is significantly slower during compression than Top-k since the Top-k selection has to be applied over all layers additionally to applying it to each layer (see Section 3.2.2). 'All Layer Top-k' accuracy seems to very slightly outperform Top-k for the lower two TkP's and especially the lowest TkP (C: +0.05 | L: +0.09). The compression ratios are very similar to each other while none of them are clearly superior to the other. Since the difference is not very significant and reliable while increasing the runtime by up to 50%, 'All Layer Top-k' will not be evaluated in the final run.

Short Name	Algorithm Differences	Compression Ratio		Mean Diff to Lossless Last Accuracy		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)		Decompression Time Max (sec.)	
		Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr		1.000	1.000	0.000	0.000	0.030	0.037	0.063	0.079	0.262	0.434	0.530	0.875
Topk7.5p0.1m		2.542	5.991	**0.200**	0.000	0.056	0.173	0.090	0.234	0.145	0.477	0.301	0.963
Topal7.5p0.1m		2.547	5.951	0.160	**0.020**	0.060	0.230	0.095	0.292	0.124	0.769	0.304	1.572
Topal5.0p0.1m		3.096	7.385	0.170	-0.100	0.040	0.076	0.072	0.145	0.068	0.148	0.141	0.296
Topk5.0p0.1m		3.078	7.445	0.150	-0.100	0.051	0.171	0.080	0.228	0.133	0.493	0.277	1.044
Topal1.75p0.1m		**4.669**	10.665	-0.070	-0.500	**0.036**	**0.067**	**0.064**	**0.119**	**0.065**	**0.139**	**0.138**	**0.271**
Topk1.75p0.1m		4.541	**11.140**	-0.120	-0.590	0.041	0.148	0.070	0.199	0.112	0.463	0.236	0.953

Table 4.17: Depicts Top-k and all Layers Top-k with selected k and minium per layer values. All evaluations also use Zstandard with compression level 1 , 'Delta Reset Point Creator' with Reset Point Interval of 28 and GCXS with all parameters set to true.
Row Names, Short Algorithm Names: Topal = All Layers Top-k + Topk = Top-K - Number + 'p' is Top-k-Percentage - Number + 'm' is 'Minimum per Layer' - Number + 'l' is 'Loss Adaptive TkP' were 'l' is the Loss Transformation Function ($f_{LoTrans}$) (see Figure 4.6) , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Loss Adaptive TkP

Conv and LSTM have vastly different accuracy values which becomes clear when looking at Table 4.17. This can origin from different loss values during the online training. A higher Top-k-Percentage could be appropriate during phases of higher loss and vice versa. Hence, it would be expected that 'Loss Adaptive TkP' (Section 3.2.2) would close the accuracy gap between Conv and LSTM and feature a higher compression ratio with only a minimum accuracy drop.

When a extreme drift occurs, loss is above 0.5 and slowly drops to about 0.1. When the accuracy stabilizes, loss normally stays below 0.1. Hence, 0.5 is chosen as Loss Upper Bound ($LoUB$) and is fixed for any execution. Figure 4.6 shows different Loss Transformation Function ($f_{LoTrans}$) which will be evaluated in this section. A convex function was used to test higher or lower values closer to the extreme ends since especially lower loss values occur more often.

Table 4.18 shows different Loss Transformation Function ($f_{LoTrans}$) settings with Top-k. The rows are grouped by TkP. Due to the lower accuracy when using loss adaptive k, 1.75% was removed while 10% was added to better reflect the desired accuracy range. 'Loss Adaptive TkP' seems to have a better accuracy to compression ratio tradeoff, e.g.,

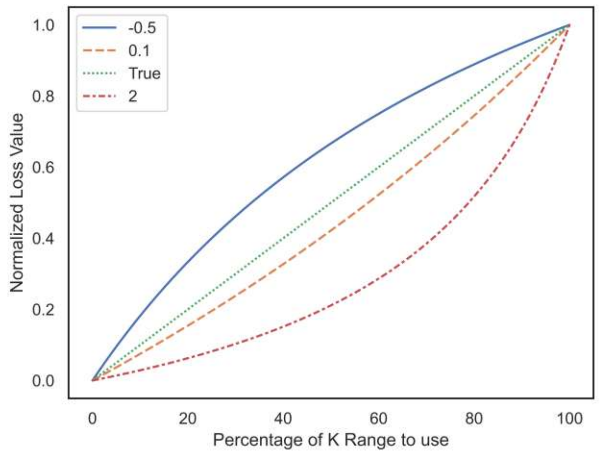

Figure 4.6: Depicts 'Loss Adaptive TkP' curve mapping function 'Loss Transformation Function' ($f_{LoTrans}$) for different curve settings and a linear function. X axis shows Loss Bound and Normalized input, Y axis shows loss bound normalized and transformed output. The function of Loss Transformation Function ($f_{LoTrans}$) is shown in Equation 3.3.

Topk7.5p0.1mTl has a better and more stable accuracy then Topk5.0p0.1m while also having a better compression ratio. It shows that a curve featuring a lower overall Top k-Percentage loss adapted ($TkPLoA$) directly correlates with a lower accuracy and higher compression ratio since each decrease of the overall $TkPLoA$ leads to a lower accuracy and higher compression for both models and for any TkP setting. The accuracies of Conv and LSTM for 'Loss Adaptive TkP' are closer together then for Top-k for a TkP of 5 and 7.5, since all the differences for 'Loss Adaptive TkP' are ≤ 0.04 while it is ≥ 0.2 for Top-k.

All previous selected Top-k settings are replaced by a 'Loss Adaptive TkP' setting since they are outperformed: **Topk7.5p0.1m*0.5l (0% accuracy bound)** outperforms the former chosen Topk7.5p0.1m, especially regarding the compression ratio while also featuring a more stable accuracy difference. **Topk7.5p0.1mTl (0.1% accuracy bound)** is more stable in the accuracy drop compared to Topk5p0.1m and hence, will replace it. Topk5.0p0.1m*0.5l and Topk10.0p0.1m2l both feature a better compression ratio but considering that the accuracy difference is almost tripled compared to the final selection, Topk7.5p0.1mTl was preferred. **Topk5.0p0.1m2l (1% accuracy bound)** has the highest compression ratio while staying under an accuracy difference of 0.5% and having a very stable accuracy drop and hence replaces Topk1.75p0.1m.

4.8.2 Reset Least Significant Bits

Table 4.19 shows Reset Least Significant Bits when Most Significant Bits set (RLSBMSB) (see Section 3.2.4). Checking the Most Significant Bits of Fraction (MSBF) seems to not have any effect since none of the Most Significant Bits of Fraction to Assess (MSBFA)

Short Algorithm Name Differences	Compression Ratio		Mean Diff to Lossless Last Accuracy		Compression Time (sec.)		Compression Time (sec.) Max		Decompression Time (sec.)		Decompression Time (sec.) Max	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.000	**0.000**	0.030	0.037	0.063	0.079	0.262	0.434	0.530	0.875
Topk7.5p0.1m	2.542	5.991	0.200	**0.000**	0.056	0.173	0.090	0.234	0.145	0.477	0.301	0.963
Topk7.5p0.1m*0.5l	3.727	6.838	0.020	-0.010	0.035	0.106	**0.059**	0.151	0.090	0.285	0.193	0.591
Topk7.5p0.1mTl	4.197	7.481	-0.050	-0.030	0.032	0.099	0.062	0.157	0.086	0.269	0.183	0.547
Topk7.5p0.1m0.1l	4.423	7.874	-0.100	-0.060	0.032	0.099	0.061	0.141	0.085	0.272	0.181	0.562
Topk7.5p0.1m2l	5.134	9.481	-0.260	-0.150	0.032	0.095	0.065	0.135	**0.084**	0.266	0.182	0.532
Topk5.0p0.1m	3.078	7.445	0.150	-0.100	0.051	0.171	0.080	0.228	0.133	0.493	0.277	1.044
Topk5.0p0.1m*0.5l	4.271	8.343	-0.100	-0.110	0.036	0.097	0.062	0.135	0.093	0.265	0.197	0.536
Topk5.0p0.1mTl	4.687	8.994	-0.170	-0.160	0.033	0.100	0.063	0.143	0.090	0.272	0.191	0.563
Topk5.0p0.1m0.1l	4.883	9.369	-0.210	-0.170	0.034	0.099	0.062	0.147	0.091	0.274	0.189	0.564
Topk5.0p0.1m2l	**5.451**	**10.761**	-0.380	-0.350	0.033	0.095	0.064	0.134	0.089	0.272	0.187	0.564
Topk10.0p0.1m*0.5l	3.338	5.829	**0.210**	-0.020	0.035	0.066	0.081	0.095	0.092	0.151	0.233	0.303
Topk10.0p0.1m	2.125	5.047	0.170	-0.020	0.038	0.067	0.068	0.102	0.087	0.155	**0.181**	0.312
Topk10.0p0.1mTl	3.839	6.431	0.130	-0.030	0.032	0.062	0.062	0.102	0.084	0.142	0.200	0.289
Topk10.0p0.1m0.1l	4.080	6.815	0.130	-0.040	0.034	0.061	0.079	**0.095**	0.088	0.141	0.201	0.299
Topk10.0p0.1m2l	4.876	8.450	-0.090	-0.100	**0.031**	**0.057**	0.073	0.095	0.085	**0.132**	0.201	**0.265**

Table 4.18: Depicts 'Loss Adaptive TkP' and Top-k grouped by k value. All evaluations also use Zstandard with compression level 1 , 'Delta Reset Point Creator' with Reset Point Interval of 28 and GCXS with all parameters set to true

Row Names, Short Algorithm Names: Topk = Top-K - Number + 'p' is Top-k-Percentage - Number + 'm' is 'Minimum per Layer' - Number + 'l' is 'Loss Adaptive TkP' were 'l' is the Loss Transformation Function ($f_{LoTrans}$) (see Figure 4.6) , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

values seem to have any correlation with the accuracy while the compression ratio does increase with a higher MSBFA. The most extreme case is Remht23s16b, which corresponds to removing the 16 LSBF, which has no signification accuracy difference to any of the other parameter combinations. Hence, just removing a number of bits will be evaluated in the next evaluation which also should decrease the compression time since the starting bits do not have to be checked. Since there was no substantial difference from removing 8 or 16 bits from the this evaluation, the upcoming evaluation will at least remove 16 bits from the lower bit range.

Table 4.20 shows the 'Reset Least Significant Bits' (Section 3.2.4). Setting Least Significant Bits of Fraction to Overwrite with zeros (LSBFO) to 23 leads to a substantial accuracy drop and is equivalent to removing all mantissis bits. Hence, at least a low number of higher bits are required to prevent a substantial accuracy drop. **20 LSBFO** seems to be the only value which does not have any accuracy loss and hence is selected for the final run.

4.8.3 Save Decision through New Model

Table 4.21 shows 'Save Decision through New Model' (Section 3.2.5) with different settings. The two bounds of 0% and 1% accuracy deviation from the model were chosen

Short Algorithm Name Differences	Compression Ratio		Mean Diff to Lossless Last Accuracy		Compression Time (sec.)		Compression Time (sec.) Max		Decompression Time (sec.)		Decompression Time (sec.) Max	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.000	0.000	0.025	0.037	0.059	0.081	0.222	0.434	0.440	0.886
Remht15s8b	1.222	1.559	**0.050**	-0.010	0.043	0.135	0.067	0.198	0.127	0.375	0.262	0.765
Remht19s8b	1.545	1.622	0.040	**0.010**	0.041	0.089	0.073	0.277	0.118	**0.241**	0.313	0.587
Remht23s8b	1.589	1.630	0.010	-0.010	0.043	0.108	0.082	0.159	0.120	0.333	0.278	0.725
Remht11s8b	1.161	1.404	0.000	-0.010	**0.039**	0.084	0.151	0.126	0.124	0.262	0.331	0.927
Remht11s16b	2.641	3.312	0.030	0.000	0.042	0.085	0.083	0.125	0.106	0.249	**0.217**	**0.508**
Remht23s16b	**3.261**	**4.376**	0.020	-0.010	0.042	0.112	0.133	0.161	**0.105**	0.298	0.282	0.595
Remht3s16b	1.035	1.029	0.010	-0.010	0.039	**0.073**	0.077	**0.124**	0.127	0.297	0.333	0.688
Remht7s16b	1.190	1.361	0.000	0.000	0.041	0.137	**0.064**	0.198	0.127	0.403	0.260	0.817

Table 4.19: Depicts Reset Least Significant Bits when Most Significant Bits set with different lengths of bits to check and remove. All evaluations also use SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Row Names, Short Algorithm Names: Remht = Remove higher fraction if lower true ('Reset Least Significant Bits when Most Significant Bits set') - Number + 's' is most significant bit to start the check (Most Significant Bits of Fraction to Assess) - Number + 'b' is lower bits to remove (Least Significant Bits of Fraction to Overwrite with zeros) , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Short Algorithm Name Differences	Compression Ratio		Mean Diff to Lossless Last Accuracy		Compression Time (sec.)		Compression Time (sec.) Max		Decompression Time (sec.)		Decompression Time (sec.) Max	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.000	0.000	0.030	0.037	0.063	0.079	0.262	0.434	0.530	0.875
Flors20r32r	4.667	8.310	**0.020**	**0.020**	0.046	0.048	0.085	0.085	0.255	0.392	0.532	**0.787**
Flors16r32r	3.226	4.425	0.010	-0.030	0.046	0.053	0.084	0.083	0.254	0.402	0.517	0.825
Flors21r32r	5.133	9.807	-0.040	0.000	**0.034**	0.079	**0.058**	0.123	0.221	0.498	0.439	0.995
Flors22r32r	5.620	11.413	-0.090	-0.010	0.034	0.077	0.062	0.121	**0.221**	0.496	**0.437**	0.985
Flors23r32r	**6.047**	**12.894**	-0.320	-0.080	0.038	**0.046**	0.073	**0.080**	0.229	**0.385**	0.465	0.860

Table 4.20: Depicts 'Reset Least Significant Bits' with different Least Significant Bits of Fraction to Overwrite with zeros. All evaluations also use SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Row Names, Short Algorithm Names: Remht = Remove higher fraction if lower true ('Reset Least Significant Bits when Most Significant Bits set') - Number + 's' is most significant bit to start the check (Most Significant Bits of Fraction to Assess) - Number + 'b' is lower bits to remove (Least Significant Bits of Fraction to Overwrite with zeros) , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28 so that 0 and at least 1 sample could be classified wrongly, respectively. The bounds are combined with the option to only evaluate the current data batch instead of using the accuracy of all of the data. Since this Processing Step skips some saves and some evaluations load the same reset point, the decompression times are somewhat skewed. Since two models have to be loaded and executed, the compression runtime exceeds the baseline by up to one order of magnitude which is unacceptable for many scenarios.

LSTM has a worse accuracy compared to Conv. This could be due to the fact that its drift phase is longer than Conv and therefore skipping saves has a higher disadvantage for LSTM. There seems to be only an insignificant difference in the evaluation metrics between using only the current training batch and using all of the data batches for the lower bound of 0%. This could be due to the fact that skipping several saves in a row does not happen often when a 0% bound is set. Skisg1.0pTeTf seems to be very unstable since the accuracies of the models differ strongly. The small accuracy degradation proves that the error rate of the current training data can help to evaluate if the model has to be deployed. For the final run, **Skisg0pFeTf** will be chosen for a 0% accuracy bound option while **Skisg1.0pFeTf** was chosen for a 0.1% accuracy bound option and preferred over Skisg0pTeTf since it is very unstable, as already mentioned.

Short Algorithm Name Differences	Compression Ratio		Mean Diff to Lossless Last Accuracy		Compression Time (sec.)		Compression Time (sec.) Max		Decompression Time (sec.)		Decompression Time (sec.) Max	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.000	**0.000**	0.030	0.037	0.063	0.079	0.262	0.434	0.530	0.875
Skisg0pFeTf	1.508	1.136	**0.220**	**0.000**	0.180	**0.224**	0.284	**1.362**	**0.327**	**0.488**	**0.779**	**1.075**
Skisg0pTeTf	1.524	1.158	0.200	**0.000**	**0.172**	0.234	0.263	1.364	0.334	0.505	0.835	1.109
Skisg1.0pTeTf	1.629	**2.778**	0.210	-0.150	0.173	0.242	**0.259**	1.406	0.340	0.588	0.866	1.443
Skisg1.0pFeTf	**4.383**	2.013	-0.010	-0.050	0.323	0.273	0.771	1.392	0.438	0.637	0.819	1.496

Table 4.21: Depicts removal of lower fraction bits with different lengths of bits and remove. All evaluations also use SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28
Row Names, Short Algorithm Names: Skisg = Skip save when accuracy sill good ('Decide Save by Error rate Decay') - Number + 'p' is the accuracy percentage difference at which the model has to be deployed ('Error Rate Bound')- Bool + 'e' is if True only last batch data will be evaluated - Bool + 'f' is always true and origins from super class , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

4.8.4 Combine Top-k and Reset Least Significant Bits Parameters

Preliminary evaluations showed that the chosen Top-k-Percentage (TkP) of Top-k combined with the chosen Least Significant Bits of Fraction to Overwrite with zeros (LSBFO) of 'Reset Least Significant Bits' lead to a significantly lower accuracy. A reason could be that when selecting less delta values from Top-k, their exact fraction must at least be more detailed. This section determines if an additional value besides 20 LSBFO is required in combination for Top-k.

Table 4.22 shows a LSBFO of 18 and 20 in combination with the selected Top-k settings. Using 18 LSBFO additional to Top-k leads to an inconclusive accuracy change across both models, ranging from -0.05 to +0.09 while having a substantial impact on the compression ratio (C: ≈+1.0–2.0 | L≈+3.3–4.7), especially for LSTM. Hence, additionally using 18 LSBFO with Top-k seems to be advantageous. The accuracy difference between 18 and 20 LSBFO combined with the same TkP is substantial (C: -0.29–0.75 | L: -0.16–0.32), although less so for LSTM, therefore also destabilizing the accuracy difference between the two networks. Using 20 LSBFO in isolation even had a positive effect (see Table 4.20). The compression ratio difference between 18 and 20 LSBFO combined with the same TkP is higher for LSTM than for Conv (C: ≈+0.5–1.0 | L: ≈+1.4–2.8). In conclusion, the 20 LSBFO seems to work better for LSTM then for Conv.

For the final run, **18 LSBFO with Top-k** will be used additionally since there is only an inconclusive accuracy change while having a higher compression ratio and **20 LSBFO with Top-k** because the 1% accuracy bound was not exceeded greatly while having a substantial impact on the compression ratio.

Short Algorithm Name Differences	Compression Ratio		Mean Diff to Lossless Last Accuracy		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)		Decompression Time Max (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.000	0.000	0.030	0.037	0.063	0.079	0.262	0.434	0.530	0.875
Topk7.5p0.1m*0.5lFlors18r32r	5.685	11.515	**0.070**	-0.080	0.037	0.098	0.067	**0.131**	0.068	0.177	0.148	0.344
Topk7.5p0.1m*0.5l	3.727	6.838	0.020	**-0.010**	0.035	0.106	0.059	0.151	0.090	0.285	0.193	0.591
Topk7.5p0.1mTlFlors18r32r	5.864	11.928	0.000	-0.080	0.036	0.098	**0.059**	0.135	0.068	0.176	0.140	0.346
Topk7.5p0.1mTl	4.197	7.481	-0.050	-0.030	**0.032**	0.099	0.062	0.157	0.086	0.269	0.183	0.547
Topk5.0p0.1m2lFlors18r32r	6.421	14.031	-0.300	-0.400	0.035	**0.090**	0.062	0.137	**0.067**	**0.172**	**0.139**	**0.336**
Topk7.5p0.1m*0.5lFlors20r32r	6.669	14.384	-0.360	-0.240	0.060	0.254	0.092	0.310	0.112	0.492	0.236	1.015
Topk5.0p0.1m2l	5.451	10.761	-0.380	-0.350	0.033	0.095	0.064	0.134	0.089	0.272	0.187	0.564
Topk7.5p0.1mTlFlors20r32r	6.745	14.568	-0.520	-0.270	0.065	0.263	0.097	0.361	0.117	0.482	0.258	1.012
Topk5.0p0.1m2lFlors20r32r	**6.935**	**15.422**	-1.050	-0.720	0.058	0.259	0.088	0.331	0.106	0.494	0.225	1.020

Table 4.22: Depicts the combination of selected Top-k settings with removing several lower fraction bits. All evaluations also use SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28
Row Names, Short Algorithm Names: Topk = Top-K - Number + 'p' is Top-k-Percentage - Number + 'm' is 'Minimum per Layer' - Number + 'l' is 'Loss Adaptive TkP' were 'l' is the Loss Transformation Function ($f_{LoTrans}$) (see Figure 4.6) , Flors = Float remove section ('Reset Least Significant Bits')- Number + 'r' is least significant bit position to keep (Least Significant Bits of Fraction to Overwrite with zeros) - Number + 'r' is most significant bit position to keep , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

4.8.5 Selecting Fitting Lossy Configurations

For the final lossy evaluations, three Top-k, two Reset Least Significant Bits and two 'Save Decision through New Model' Processing Steps are used in all configurations. In the following section a maximum of six configurations will be chosen for three different accuracy targets and two different compression time proportions. The section will be divided by the compression speeds, were the slower section will exclusively include 'Save Decision through New Model' since it increases the compression time significantly.

The decompression time max was omitted due to space restrictions of some of the tables but was about double the listed mean decompression time for any of the rows.

Low Compression Time

Table 4.23 shows the configurations with lower compression times which only use Top-k and Reset Least Significant Bits. The compression times are higher than the original lossless configuration but still close enough to the 'Batch Learning Time Baseline' (C: 0.033 | L: 0.094) and are therefore acceptable. Using **Flors20r32rDifrs148f for 0% accuracy bound** features a significantly higher compression ratio and a low compression time. Using **Topk7.5p0.1mTlFlors18r32r for 0.1% accuracy bound** leads to a higher compression ratio compared to the former configuration (C: ≈+1.2 | L: ≈+3.6) while having about the same compression time. Using **Topk7.5p0.1m*0.5lFlors20r32r for 1% accuracy bound** leads to a higher compression ratio compared to the former configuration (C: ≈+0.8 | L: ≈+2.5) while losing only about 0.36% accuracy and having a very stable accuracy drop. All other configurations have a higher accuracy drop while having almost no compression gains.

Short Algorithm Name Differences	Compression Ratio		Mean Diff to Lossless Last Accuracy		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.000	0.000	0.030	0.037	0.063	0.079	0.262	0.434
Topk7.5p0.1m*0.5lFlors18r32r	5.685	11.515	**0.070**	-0.080	0.037	0.098	0.067	0.131	0.068	0.177
Topk7.5p0.1m*0.5l	3.727	6.838	0.020	-0.010	0.035	0.106	0.059	0.151	0.090	0.285
Flors20r32rSplbs	4.667	8.310	0.020	**0.020**	0.046	**0.048**	0.085	**0.085**	0.255	0.392
Topk7.5p0.1mTlFlors18r32r	5.864	11.928	0.000	-0.080	0.036	0.098	**0.059**	0.135	0.068	0.176
Topk7.5p0.1mTl	4.197	7.481	-0.050	-0.030	**0.032**	0.099	0.062	0.157	0.086	0.269
Topk5.0p0.1m2lFlors18r32r	6.421	14.031	-0.300	-0.400	0.035	0.090	0.062	0.137	**0.067**	**0.172**
Topk7.5p0.1m*0.5lFlors20r32r	6.669	14.384	-0.360	-0.240	0.060	0.254	0.092	0.310	0.112	0.492
Topk5.0p0.1m2l	5.451	10.761	-0.380	-0.350	0.033	0.095	0.064	0.134	0.089	0.272
Topk7.5p0.1mTlFlors20r32r	6.745	14.568	-0.520	-0.270	0.065	0.263	0.097	0.361	0.117	0.482
Topk5.0p0.1m2lFlors20r32r	**6.935**	**15.422**	-1.050	-0.720	0.058	0.259	0.088	0.331	0.106	0.494

Table 4.23: Depicts the lower compression time final lossy Configurations. All evaluations also use GCXS with all parameters set to true if Top-k is used, otherwise SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Row Names, Short Algorithm Names: Topk = Top-K - Number + 'p' is Top-k-Percentage - Number + 'm' is 'Minimum per Layer' - Number + 'l' is 'Loss Adaptive TkP' were 'l' is the Loss Transformation Function ($f_{LoTrans}$) (see Figure 4.6) , Flors = Float remove section ('Reset Least Significant Bits')- Number + 'r' is least significant bit position to keep (Least Significant Bits of Fraction to Overwrite with zeros) - Number + 'r' is most significant bit position to keep , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

High Compression Time

Table 4.24 shows the configurations with higher compression times which all use 'Save Decision through New Model'. Combining any Top-k and Skisg1.0pFeTf in any configuration reduces the accuracy strongly. E.g. the comparing the configuration 'Topk7.5p0.1mTlFlors18r32r' from the last section with 'Topk7.5p0.1mTlFlors18r32rSkisg1.0pFeTf' shows that the accuracy drops significantly (C: -0.46 | L: -0.12). This could be due to the fact that the used Loss (Lo) for Top-k does not reflect the change in the network when a save is skipped several times which is the case when using a 1% bound instead of 0%. Besides the mentioned configurations above, adding 'Save Decision through New Model' to the fast execution configuration (see Table 4.23) seems to significantly affect the compression ratio (avg. C: ≈+5.7 | L: ≈+3.1, min. C: +1.7 | L: +0.6) while almost not affecting the accuracy in any combination (avg. C: ≈+0.03 | L: ≈0.03, max. C: -0.07 | L: -0.09).

The configurations **Flors20r32rSkisg0pFeTfSplbs for 0% accuracy bound** was chosen since it is the only configuration fulfilling this bound. The configuration **Flors20r32rSkisg1 for 0.1% accuracy bound** was chosen since it has by far the hightest compression ratio below this bound will having a stable accuracy drop for both models.
Topk7.5p0.1m*0.5lFlors20r32rSkisg1.0pFeTf for 1% accuracy bound was chosen since it the option with the hightest compression ratio with an accuracy below the bound with no other option having a comparable compression ratio while achieving a significantly better accuracy.

Slower General Compression

This section will evaluate if a general compression algorithm with a higher compression time has a larger impact on the compression ratio due to the used lossy compression algorithms. To evaluate this hypothesis, the chosen configurations from the previous sections with the two highest compression ratios are combined with the general compression algorithm LZMA which was also used in 'Final Evaluation Lossless Configurations' (Section 4.7.5)

Table 4.25 shows the results when comparing ZStandard 1 with LZMA for different combinations from the final selection of fast and slow algorithms. Mostly, using LZMA increases the overall compression time strongly (C: ≈0.5–1.1 | L: ≈0.5–3.3), depending if the overall runtime is already very slow. The compression ratio difference between LZMA and Zst 1 is significantly higher (C: ≈0.1–1.4 | L: ≈0.4–2) then for 'Final Evaluation Lossless Configurations' (Section 4.7.5). Due to the increased compression times, especially for fast configurations, LZMA will not be used in any configurations since the compression ratio increases are not high enough to support justify its use.

Short Algorithm Name Differences	Compression Ratio		Mean Diff to Lossless Last Accuracy		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.000	0.000	0.030	0.037	0.063	0.079	0.262	0.434
Flors20r32rSkisg0pFeTfSplbs	7.706	9.425	**0.290**	**0.030**	0.193	0.424	0.277	1.775	0.297	0.959
Topk7.5p0.1m*0.5lSkisg0pFeTf	5.491	7.482	0.170	-0.050	0.189	0.383	0.271	1.669	0.141	0.530
Topk7.5p0.1m*0.5lFlors18r32rSkisg0pFeTf	9.870	13.143	0.080	-0.080	0.167	**0.321**	0.264	1.517	0.082	**0.247**
Topk7.5p0.1mTlSkisg0pFeTf	6.378	8.226	0.050	-0.080	0.194	0.398	0.276	1.746	0.152	0.508
Topk7.5p0.1mTlFlors18r32rSkisg0pFeTf	10.327	13.694	0.010	-0.110	0.171	0.326	**0.241**	1.493	0.085	0.254
Flors20r32rSkisg1.0pFeTfSplbs	19.801	14.435	-0.080	-0.050	0.295	0.453	0.680	1.781	0.368	0.903
Topk7.5p0.1m*0.5lSkisg1.0pFeTf	12.984	10.362	-0.240	-0.100	0.310	0.423	0.913	1.673	0.223	0.653
Topk7.5p0.1m*0.5lFlors18r32rSkisg1.0pFeTf	27.239	21.080	-0.280	-0.110	0.256	0.355	0.575	**1.463**	0.156	0.290
Topk5.0p0.1m2lSkisg0pFeTf	9.180	13.335	-0.300	-0.440	0.189	0.395	0.274	1.730	0.123	0.617
Topk5.0p0.1m2lFlors18r32rSkisg0pFeTf	11.797	19.407	-0.310	-0.420	**0.164**	0.331	0.369	1.528	**0.072**	0.285
Topk7.5p0.1mTlSkisg1.0pFeTf	14.770	11.696	-0.330	-0.180	0.331	0.443	0.952	1.763	0.278	0.586
Topk7.5p0.1m*0.5lFlors20r32rSkisg0pFeTf	13.745	18.328	-0.400	-0.260	0.200	0.459	0.318	1.783	0.122	0.514
Topk7.5p0.1mTlFlors18r32rSkisg1.0pFeTf	28.576	23.496	-0.460	-0.130	0.247	0.367	0.503	1.498	0.138	0.320
Topk7.5p0.1mTlFlors20r32rSkisg0pFeTf	14.013	18.580	-0.560	-0.300	0.208	0.475	0.297	1.821	0.123	0.594
Topk5.0p0.1m2lSkisg1.0pFeTf	21.994	20.171	-0.740	-0.550	0.319	0.455	0.729	1.715	0.231	0.530
Topk5.0p0.1m2lFlors18r32rSkisg1.0pFeTf	31.860	32.906	-0.760	-0.570	0.233	0.380	0.477	1.511	0.138	0.263
Topk7.5p0.1m*0.5lFlors20r32rSkisg1.0pFeTf	34.729	34.610	-0.850	-0.360	0.331	0.508	0.709	2.077	0.217	0.583
Topk7.5p0.1mTlFlors20r32rSkisg1.0pFeTf	35.867	35.425	-1.120	-0.410	0.354	0.537	0.743	1.822	0.213	0.542
Topk5.0p0.1m2lFlors20r32rSkisg0pFeTf	15.770	22.065	-1.120	-0.760	0.204	0.495	0.425	1.825	0.121	0.534
Topk5.0p0.1m2lFlors20r32rSkisg1.0pFeTf	**38.085**	**39.735**	-1.650	-0.930	0.317	0.559	0.619	1.840	0.157	0.441

Table 4.24: Depicts the higher compression time final lossy configurations. All evaluations also use GCXS with all parameters set to true if Top-k is used, otherwise SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Row Names, Short Algorithm Names: Topk = Top-K - Number + 'p' is Top-k-Percentage - Number + 'm' is 'Minimum per Layer' - Number + 'l' is 'Loss Adaptive TkP' were 'l' is the Loss Transformation Function ($f_{LoTrans}$) (see Figure 4.6) , Flors = Float remove section ('Reset Least Significant Bits')- Number + 'r' is least significant bit position to keep (Least Significant Bits of Fraction to Overwrite with zeros) - Number + 'r' is most significant bit position to keep , Skisg = Skip save when accuracy sill good ('Decide Save by Error rate Decay') - Number + 'p' is the accuracy percentage difference at which the model has to be deployed ('Error Rate Bound')- Bool + 'e' is if True only last batch data will be evaluated - Bool + 'f' is always true and origins from super class , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Final Lossless Configurations with Different Reset Point Interval Settings

Table 4.26 shows the selected configurations from the slower and higher compression time section with the Reset Point Interval (RPI) of 148 compared to 28 which was used until now. When combining 'Save Decision through New Model' with the maximum RPI of 148, the reset point at the end of the run will be omitted since less saves in total are executed. Hence, only one reset point at the start will be present for theses evaluations. This somewhat skews the compression ratio and especially the decompression time since all difference recreations have to be started from the first reset point. As the resulting online training weights slightly deviate (see Section 4.6), the accuracy can deviate per run. The results show that some configurations can have vastly different accuracies for the different reset point settings. This is due to the fact that some Processing Steps are more unstable than others. E.g. a rough analysis has shown that the skipped saves can vastly

Short Name Algorithm Differences	Compression Ratio		Compression Time (sec.)		Compression Time Max (sec.)		Decompression Time (sec.)		Decompression Time Max (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-LosslessCompr	1.000	1.000	0.030	0.037	0.063	0.079	0.262	0.434	0.530	0.875
Topk7.5p0.1m*0.5lSkisg1.0pFeTfZstwm1c	**34.729**	34.610	0.331	0.508	0.709	2.077	0.217	0.583	0.331	1.307
Topk7.5p0.1m*0.5lSkisg1.0pFeTfLzmwm	33.322	**35.244**	0.248	0.373	0.886	2.575	0.151	0.277	0.234	0.554
Skisg1.0pFeTfSplbsLzmwm	20.524	16.419	0.281	0.892	1.193	3.739	0.508	0.660	0.825	1.437
Skisg1.0pFeTfSplbsZstwm1c	19.830	14.435	0.308	0.453	0.643	1.781	0.377	0.903	0.625	1.988
Topk7.5p0.1m*0.5lLzmwm	6.771	14.665	0.081	0.214	1.022	2.317	**0.086**	**0.198**	**0.154**	**0.373**
Topk7.5p0.1m*0.5lZstwm1c	6.669	14.384	0.060	0.254	0.092	0.310	0.112	0.492	0.236	1.015
SplbsLzmwm	5.010	9.569	0.301	1.220	1.171	3.398	0.275	0.607	0.548	1.173
SplbsZstwm1c	4.667	8.310	**0.046**	**0.048**	**0.085**	**0.085**	0.255	0.392	0.532	0.787

Table 4.25: Depicts all selected final evaluations with a higher compression ratio with different compression algorithms. All evaluations also use GCXS with all parameters set to true if Top-k is used, otherwise SplitFloatAndStackByByteSegments, 'Reset Least Significant Bits' with a LSBFO of 20, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

Row Names, Short Algorithm Names: Topk = Top-K - Number + 'p' is Top-k-Percentage - Number + 'm' is 'Minimum per Layer' - Number + 'l' is 'Loss Adaptive TkP' were 'l' is the Loss Transformation Function ($f_{LoTrans}$) (see Figure 4.6) , Zst = ZStandard - Numer + 'c' is compression level parameter Lzmwm = LZMA , Skisg = Skip save when accuracy sill good ('Decide Save by Error rate Decay') - Number + 'p' is the accuracy percentage difference at which the model has to be deployed ('Error Rate Bound')- Bool + 'e' is if True only last batch data will be evaluated - Bool + 'f' is always true and origins from super class , BL-LosslessCompr = Lossless Compression Baseline - SplitFloatAndStackByByteSegments, Zstandard with compression level 1 and 'Delta Reset Point Creator' with Reset Point Interval of 28

differ per run and lead to a chain reaction also affecting all of the upcoming save decisions. The great deviation can be seen for Flors20rSkisg1.0pFeTf where accuracy deviation of Conv exceeds 0.1%. Configuration with 'Save Decision through New Model' a high compression ratio only slightly increases it with a higher RPI since they already have a low number of reset points because they have a low number of saves in general. The real effect of the higher RPI on the compression ratio can therefore only be analyzed if the evaluation has more batches and hence, the number of skipped saves is less meaningful. The metrics of the configurations using RPI of 148 will be used since all decompression times are in an acceptable range.

4.9 Summary

This section will summarize which Processing Step Parameters to use, which Processing Steps were unsuccessful and which configurations to use. All recommendations are only based on the two used models. 'Summary of Utility of Assessed Processing Steps' (Section 6.1.5) summarizes the most impactful Processing Steps.

Short Algorithm Name Differences	Compression Ratio		Mean Diff to Lossless Last Accuracy		Compression Time (sec.)		Compression Max Time (sec.)		Decompression Time (sec.)	
	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM	Conv	LSTM
BL-GeneralCompr	1.000	1.000	0.000	0.000	0.012	0.024	0.035	0.043	0.011	0.024
Flors20rSkisg0pFeTfDifrs28f	28.523	14.208	**0.190**	**0.010**	0.193	0.424	0.277	1.775	0.297	0.959
Flors20rSkisg0pFeTfDifrs148f	46.197	21.856	0.130	-0.030	0.159	0.274	0.483	1.435	1.081	4.323
Flors20rDifrs148f	29.944	18.017	-0.020	-0.030	**0.028**	0.085	**0.046**	0.120	0.931	2.476
Flors20rDifrs28f	17.275	12.528	-0.080	0.000	0.046	**0.048**	0.085	**0.085**	0.255	0.392
Topk7.5p0.1mTlFlors18rDifrs148f	46.405	32.109	-0.080	-0.120	0.037	0.123	0.058	0.152	0.311	1.103
Topk7.5p0.1mTlFlors18rDifrs28f	21.704	17.981	-0.100	-0.100	0.036	0.098	0.059	0.135	**0.068**	**0.176**
Flors20rSkisg1.0pFeTfDifrs28f	73.293	21.762	-0.180	-0.070	0.295	0.453	0.680	1.781	0.368	0.903
Flors20rSkisg1.0pFeTfDifrs148f	82.877	29.074	-0.260	-0.100	0.236	0.297	0.473	1.387	0.458	2.516
Topk7.5p0.1m*0.5lFlors20rDifrs28f	24.685	21.684	-0.460	-0.260	0.060	0.254	0.092	0.310	0.112	0.492
Topk7.5p0.1m*0.5lFlors20rDifrs148f	62.550	46.195	-0.490	-0.330	0.039	0.094	0.066	0.140	0.326	0.774
Topk7.5p0.1m*0.5lFlors20rSkisg1.0pFeTfDifrs28f	128.547	52.176	-0.940	-0.380	0.331	0.508	0.709	2.077	0.217	0.583
Topk7.5p0.1m*0.5lFlors20rSkisg1.0pFeTfDifrs148f	**129.692**	**76.917**	-0.980	-0.400	0.243	0.334	0.426	1.466	0.121	0.579

Table 4.26: Depicts the final lossy configurations with different reset points frequencies. All evaluations also use GCXS with all parameters set to true if Top-k is used, otherwise SplitFloatAndStackByByteSegments and Zstandard with compression level 1

Row Names, Short Algorithm Names: Topk = Top-K - Number + 'p' is Top-k-Percentage - Number + 'm' is 'Minimum per Layer' - Number + 'l' is 'Loss Adaptive TkP' were 'l' is the Loss Transformation Function ($f_{LoTrans}$) (see Figure 4.6) , Flors = Float remove section ('Reset Least Significant Bits')- Number + 'r' is least significant bit position to keep (Least Significant Bits of Fraction to Overwrite with zeros) - Number + 'r' is most significant bit position to keep , Skisg = Skip save when accuracy sill good ('Decide Save by Error rate Decay') - Number + 'p' is the accuracy percentage difference at which the model has to be deployed ('Error Rate Bound')- Bool + 'e' is if True only last batch data will be evaluated - Bool + 'f' is always true and origins from super class , BL-GenComp = General Compression Baseline - Zstandard with compression level 1

4.9.1 Summary Processing Steps

'General Compression': **Zstandard 1** has proven have the best tradeoff between compression ratio (-0.013 compared to best) while having the lowest compression time (max. 0.024s). Others were repeatedly reassessed, especially LZMA, but everytime had a high compression time while not improving the compression ratio considerably.

'Delta Creator': It significantly improves the compression ratio (min. +0.3) while almost having no effect on the compression time (max. +0.008). The 'Reset Point Interval' (RPI) can be set higher then the maximum of 148 since the decompression time maximum over all runs was 2.64s.

'Bytewise Segmentation for Float Matrices': **SplitFloatAndStackByByteSegments** is superior to SplitFloatAndStackByByteSegmentsSplitLater and has a higher compression ratio (min. 0.011) compared to SplitFloatAndStack but featuring a higher decompression time (max. diff of max. 0.044s (max. abs. 0.076)).

'Top-K': It has a significant compression ratio increase (+1.5-10.8), can be adapted to an accuracy bound and can have a noticable effect on the compression time (max. +0.07). The range of 'Top-k-Percentage' (TkP) should be between 15% and 1.75% since otherwise the accuracy difference is not noticable or too high. 'Minimum per Layer' is always recommended to improve accuracy and 'Minimum per Layer Percentage' ($minLP$)

should range for the recommended TkP between 0.1% and 0.01%. 'GCXS' should be used with all optimizations since all of them are benefitial for the given TkP range and improve the compression ratio and compression speed. We partially recommend 'Loss Adaptive TkP' since it improves the accurcay and compression ratio but has to be well chosen together with the two parameters TkP and $minLP$ and hence, is harder to configure. To reduce the search space, we recommend to use a linear function for 'Loss Transformation Function' ($f_{LoTrans}$) and to only adapt the other two parameters. Combining it with 'Decide Save by Error rate Decay' can lead to degredation of accuracy due to the loss value is not summarized (see Section 6.3.1). We do not recommend 'All Layer Top-k' since it has almost no affect on the accuracy or compression ratio while having a higher compression time.

'Reset Least Significant Bits': It features a very significant compression ratio increase (+3.7-7.3) without almost any accuracy deviation and a mild effect on the compression time (max. +0.016). We recommend using a 'Least Significant Bits of Fraction to Overwrite with zeros' (LSBFO) between 18-20. A LSBFO value in the lower end of this range is recommended by us when combining it with 'Top-K'.

'Decide Save by Error rate Decay': Is only partially recommended by us since it has a low compression ratio increase (+0.14-4.4) compared to the other lossy Processing Steps and has substantial drawbacks like the very slow execution time (+0.196), requiring a lot of memory and using the neural network processing unit but could be essential to save storage space in a different scenario were there is only little drift. To reduce the required memory, we recommend to only use the data of the current batch for evaluation. The two used percentages values 0% and 1% should be optimal to reflect no and a little change in the error rate.

4.10 Drawbacks of Retraining for Model Recreation System

'Retraining for Model Recreation System' (Section 2.1) shortly discussed the option to instead retrain the model with the saved data instead of saving every deployed version. This section compares it with the resulting VSOL from a standpoint of the given research questions and gained insight of the research since it is the most obvious alternative to an VSOL like system.

Having a deterministic learning process, which would be required for retraining a model exactly, is either not implemented or can significantly increase the 'Batch Learning Time Baseline' which would strongly impact 'RQ1: Impact Reduction on Online Learning' (Section 1.2) for the machine learning framework tensorflow, as discussed in 'Undeterministic Learning Process' (Section 4.6). This downside could become obsolete in the future when more neural network layer functions are deterministic and the deterministic implementations are as performant as the non-deterministic implementations.

Storing one batch of online training data requires significantly more storage space (RQ2) then storing one compressed weight delta of the models we have encountered during the research. The large amount of data is explicitly listed as one of the limitations of online learning, as described in 'Online Learning Setting' (Section 2.3).

A Retraining for Model Recreation System (RMRS) would require tracking metadata such as the model architecture. This system has the same requirements as a metadata system, described in Section 2.1, which would increase the integration requirements (RQ3) and overall complexity significantly. The VSOL integration requirements are discussed in 'Portability to other Frameworks' (Section 3.4.8).

To keep the decompression time (RQ4) of a RMRS in an acceptable range, reset points as with the VSOL discussed in 'Delta Reset Point Creator' (Section 3.2.8) would be required. Different to the VSOL, the weights cannot be loosely compressed since they have to match the training weights exactly which would affect the compression ratio (RQ2).

In conclusion, due to the limitations listed above, a RMRS would only be an option to use instead of the VSOL when the listed downsides are addressed.

CHAPTER 5

Versioning System for Online Learning systems Architecture

5.1 Code for Integration to Training Pipeline with Final Compression Ratios

Listing 5.1 shows the code for adding the Versioning System for Online Learning systems (VSOL) to a training pipeline. Any of the depicted code is available on GitHub[1]. Line 8 shows how the preferences for chosing a certain configuration are set. A set of settings is then mapped to a preselected configuration. Listing 5.3 shows the documentation of the PerformanceRequirementSettingsData class. It shows in detail which configurations have which affect on the compression ratio and describes each value in detail. A user then can easily choose which tradeoffs are acceptable to reach a desired compression ratio. The compression ratios from the tables result from the final runs of the lossless (see Table 4.12) and the lossy (see Table 4.26) settings. Having a separate configuration object helps to break down the integration process into smaller tasks were the user can finish the configuration fully before having to deal with adding the rest of the required objects for the VSOL. It also reduces the number of required input parameters for the VSOL. The configuration is provided with defensive default settings, being the only lossy option, which value consistency of results over compression ratio. A separate object makes it also easily sharable across several VSOLs.

We recommend using the fast lossy configuration with zero accuracy drop shown in Listing 5.2 since it features a ≈7.5 times higher compression ratio then the lossy confiugration but also has no accuracy degredation. It's compression time is between 0.000 and 0.035s slower then the lossy variant and it's decompression time is about the same (-0.054-+0.17).

[1]`https://github.com/christopheitenberger/VSOL`

The 'max_decompression_time_in_s' should be set to the maximum acceptable value to further increase the compression ratio.

Line 15 shows the creation of the VSOL. The model and the data sequence is required for 'Save Decision through New Model' (Section 3.2.5) to create models and evaluate the data of the last batch since Keras does not provide the training data through another interface. They are also used as is in the training step and should therefore already be present. The default setting shown in Line 5 is set as default value for the VSOL so that no configuration has be created separately, reducing the full configuration to one line and three parameters.

Line 24 shows how the VSOL is integrated into the training process. It implements a callback interface from Keras which is called before and after each batch. The training data source 'data_seq_online_run' is implemented to have one batch per epoch, hence the number of batches are given as the number of epochs. This is valid in an online learning setting since epochs are not used when only new data is ingested and not repeated. The VSOL does not require this approach but it needs to be able to acquire the last used batch from the data sequencer. When previously saved model weights are loaded, the timestamp is then mapped to a run number through a list containing the timestamps when each training run was saved. The position in the table then corresponds to the run number of the saved weights. Through the run number, the differential files and the nearest full save can be identified. Each differential and full save is then decompressed through the pipeline to recreate the saved weights.

Those weights are then either returned directly, as with in line 35, or loaded into the cloned model, as in line 39. Either loading process only requires one line.

Since training and evaluation use different weights and probably are executed in parallel, the saved weights have to be rolled out the the evaluating model. This is not integrated since deployment is not a focus of this thesis. This could be implemented with the callback pattern. A list of functions accepting the new model weights as parameters could be handed over to the VSOL.

In conclusion, the VSOL can be easily integrated by a callback into a preexisting codebase by only adding one argument. The setup requires at most two lines were the configuration of the VSOL is separated and the class itself only requires values which are already present for the learning process. Including the VSOL in the learning process and loading the weights for a certain timestamp only requires one line and no additional objects. Finally, answering 'RQ3: Reduce Integration Effort' (Section 1.2), it can be said that with Keras it is possible to integrate an VSOL with very little effort.

5.2 Keras Usage and Integration

Figure 5.4 shows a UML class diagram of how the Versioning System for Online Learning systems (VSOL) is integrated and coupled with keras. All classes have several more parameters and methods but only the relevant ones for the integration into keras are shown here.

The core VSOL is the 'CompressionPipelineRunner' below. It wraps all of the used Processing Steps called 'runners' and conducts their interaction with each other. It exposes all of the required functionality of the VSOL, namely processing the next trained weights and loading the saved weights from a certain timestamp. Some of the runners required the used keras model which is provided by the 'CompressionPipelineRunner'. They use 'get_weights' to acquire the dimensions of each layer beforehand. The save decision configurations (see Section 3.2.1) use a separate model to evaluate if the newer model has to be saved. Hence, it needs to copy the training model architecture and evaluate the given training data. It therefore requires the methods 'keras.model.clone_model', 'Model.set_weights' and 'Model.evaluate'.

'CompressModel' integrates 'CompressionPipelineRunner' with the keras training process through a callback mechanism. It implements the before and after batch functions of the 'keras.callbacks.Callback' class as a subclass. It also exposes the used model of the training run through a class attribute. After each batch it then calls the 'Compression-PipelineRunner' and passes on the new weights, the loss value of the training run and the training data sample. To receive the later, '___get_item___' from 'keras.utils.Sequence' is used which is an attribute of the class and is also used as a batch generator during training. The implementation of 'Sequence' for generating the online learning data stream, not shown in this uml diagram, was implemented to return the last used data sample when '___get_item___' is called with the current batch number. This must also be guaranteed by any other implementation of 'Sequence' to be used with this class. This class is also mainly used be the evaluation process during the training evaluation.

'OnlineLearningVersionSaver' then uses 'performance_requirement_settings' to select the runners the 'CompressionPipelineRunner' object and hands them over to the base class 'CompressionModel'. It also exposes the retrieval of weights of the 'Compression-PipelineRunner' object and streamlines creating an evaluation model with those weights. In summary it acts as the callback function as well as a factory for the VSOL.

Keras was chosen since it has the well defined interfaces shown in the UML diagram through which it is decoupled from the training implementation and can be added without having to consider how and where to integrate it into an existing pipeline.

```python
model, data_seq_online_run, batch_size, number_of_online_batches = (
    get_mnist_model_with_data_and_settings_for_online_run_example())

# choosing the default defensive settings to get started quickly
settings_default = PerformanceRequirementSettingsData()

# choosing a specific value for each type of setting
settings_for = PerformanceRequirementSettingsData(
    max_decompression_time_in_s=15,
    lossy=LossSetting.LOSSY,
    compression_speed=CompressionSpeed.FAST,
    accuracy_limit_percent=AccuracyLimitPercent.ZERO_POINT_ONE)

# creating online learning saver from chosen settings
saver = OnlineLearningVersionSaver(
    model,
    data_seq_online_run,
    path_to_overall_model_folder,
    settings_for)

# to simulate a classification timestamp, a timestamp during training is needed
# this is done by creating a timestamp before training and adding 5 seconds
timestamp_before_run = time.time_ns()

# using the online learning saver is as easy as adding
# it as a callback function with the learning function
# the 'saver' internally recorded the deployment timestamp
model.fit(data_seq_online_run,
          batch_size=batch_size,
          epochs=number_of_online_batches,
          verbose=1,
          shuffle=False,
          callbacks=[saver]
          )

# adding 5 seconds to the timestamp of the beginning of the training results
# in a timestamp during training, simulating a classification timestamp
# the deployment timestamp is recorded by the 'saver
simulated_classification_timestamp = timestamp_before_run + (5 * pow(10, 9))

# the weights can be loaded simply by using a timestamp
loaded_weights = saver.load_weights_at_timestamp(simulated_classification_timestamp)

# the model can be also cloned and loaded
# with the weights from the timestamp
model = saver.clone_model_and_load_weights_at_timestamp(
    simulated_classification_timestamp, model)
```

Listing 5.1: The tutorial code for adding the Versioning System for Online Learning systems to an existing online learning training pipeline.

```
recommended_setting = PerformanceRequirementSettingsData(
    # should be set by user as high as acceptable to increase compression ratio
    max_decompression_time_in_s=15,
    lossy=LossSetting.LOSSY,
    compression_speed=CompressionSpeed.FAST,
    accuracy_limit_percent=AccuracyLimitPercent.ZERO
)
```

Listing 5.2: The code for our recommended configuration.

```python
@dataclass
class PerformanceRequirementSettingsData:
    """
    The performance requirement settings for the
    OnlineLearningVersionSaver class.
    To try out the online version saver, no configuration is required since
    all parameters have a default and preserve the trained values.

    Changing the settings leads to a higher compression ratio
    while effecting runtime and accuracy.

    The table below shows which settings possibly lead to which compression ratios.
    The values can differ since they result from testing
    two different neural networks.
    The values result from a very low max_decompression_time_in_s
    and can be significantly higher with a higher value.

    Compression Ratios for settings
    -------------------------------

    lossy=LOSSLESS:
    -------
    1.6-4.2
    -------

    lossy=LOSSY:
    --------------------------------------------------------------
    AccuracyLimitPercent\CompressionSpeed |    FAST    |    SLOW
    --------------------------------------------------------------
    ZERO                                  | 12.5-30.0 | 14.3-46.2
    ZERO_POINT_ONE                        | 18.0-46.4 | 21.8-82.9
    ONE                                   | 21.7-62.6 | 52.2-129.7

    Attributes
    ----------
    max_decompression_time_in_s: float
        The maximum acceptable decompression time.
        Effects after how many saves a reset point is generated.
        A higher value leads to a higher compression ratio, has a high impact.
        A large neural network requires a higher setting to achieve the same
        compression ratio since compressing and decompressing requires
        more time with an increasing size.
    lossy: LossSetting
        If the compression pipline should be lossless or lossy.
        LOSSLESS saves the weights after each online batch as is.
        LOSSY can alter weights before they are saved and rolled out
        or even skip the rollout of a set of new weights.
        The runtime of LOSSLESS is close to the training time of the executed batch.
        LOSSY leads to a higher compression ratio. See table above for details.
        The attributes compression_speed and accuracy_limit_percent
        are only relevant if lossy is set to LOSSY.
    compression_speed: CompressionSpeed
        If the mean compression speed should be FAST or SLOW.
        Only relevant if lossy=LOSSY.
        SLOW leads to a higher compression ratio. See table above for details.
        The runtime of FAST is close to the training time of the executed batch.
        The runtime of SLOW is close to 3-5x the training time of
        the executed batch.
    accuracy_limit_percent: AccuracyLimitPercent
        The acceptable average accuracy drop compared to a lossless run.
        Only relevant if lossy=LOSSY.
        A higher value leads to a higher compression ratio.
        See table above for details.
    """
    max_decompression_time_in_s: float = 10
    lossy: Lossy = Lossy.LOSSLESS
    compression_speed: CompressionSpeed = CompressionSpeed.FAST
    accuracy_limit_percent: AccuracyLimitPercent = AccuracyLimitPercent.ZERO
```

Listing 5.3: The code with documentation for the 'PerformanceRequirementSettingsData'. Describes its attributes and their effect in detail and includes the final summary table for the compression ratio.

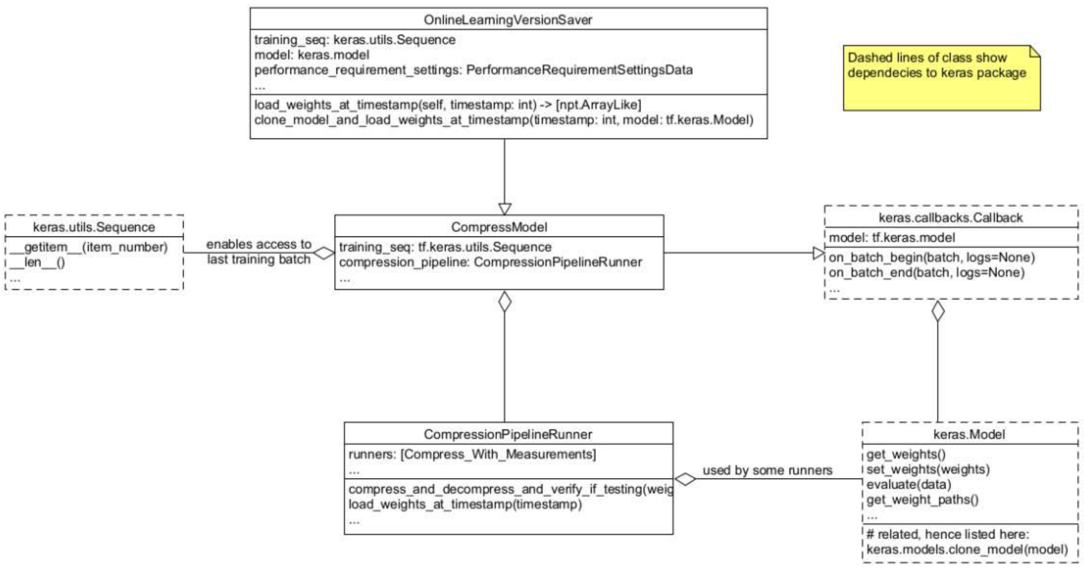
OnlineLearningVersionSaver
training_seq: keras.utils.Sequence
model: keras.model
performance_requirement_settings: PerformanceRequirementSettingsData
...
load_weights_at_timestamp(self, timestamp: int) -> [npt.ArrayLike]
clone_model_and_load_weights_at_timestamp(timestamp: int, model: tf.keras.Model)
Dashed lines of class show dependecies to keras package
keras.utils.Sequence
__getitem__(item_number)
__len__()
...
enables access to last training batch
CompressModel
training_seq: tf.keras.utils.Sequence
compression_pipeline: CompressionPipelineRunner
...
keras.callbacks.Callback
model: tf.keras.model
on_batch_begin(batch, logs=None)
on_batch_end(batch, logs=None)
...
CompressionPipelineRunner
runners: [Compress_With_Measurements]
...
compress_and_decompress_and_verify_if_testing(weig
load_weights_at_timestamp(timestamp)
...
used by some runners
keras.Model
get_weights()
set_weights(weights)
evaluate(data)
get_weight_paths()
...
# related, hence listed here:
keras.models.clone_model(model)

Figure 5.4: UML class diagram showing how the Versioning System for Online Learning systems was integrated into the Keras learning flow.

CHAPTER 6

Conclusion

6.1 Main Results

The final Versioning System for Online Learning systems (VSOL), discussed and developed in this thesis, has seven different configurations designed for different processing time and error rate requirements, as summarized in Listing 5.3. The configurations were tested with the two neural networks Conv and LSTM (see Section 4.4) under a simulated drift. The following section will summarize to which degree the different configurations met the research questions (RQ).

6.1.1 Assessment of RQ1: Impact Reduction on Online Learning

RQ1 (see Section 1.2) concerns the performance impact of the VSOL on the training process. Relevant metrics for assessing the impact are the error rate, measured by the 'Prequential Accuracy' (Section 4.3.1), and processing impact, measured by the 'Compression Time' (Section 4.3.3). Table 4.7 shows the 'Batch Learning Time Baseline' for the compression time. RQ1 and RQ2 are tradeoffs between each other, as discussed in 'RQ2: Reduce Required Storage' (Section 1.2). Hence, the following section split the resulting configurations into fulfilling RQ1 fully and partially, whereat the latter is required to excel in RQ2.

VSOL Configurations Fully Satisfying RQ1 Two of the seven configurations fully satisfied RQ1. Table 4.12 shows the final selected lossless configuration 'SplbsZstwm1cDifrs148f' with the compression time (C: 0.028 | L: 0.050), which is below the 'Batch Learning Time Baseline' (C: 0.033 | L: 0.094) and saves the weights losslessly. Table 4.26 shows the lossy configuration 'Flors20r32rDifrs148f', which has no error rate drop compared to the lossless baseline and is also is below the 'Batch Learning Time Baseline' (C: 0.033 | L: 0.094), having a compression time of (C: 0.028 | L: 0.085). In conclusion, it is possible to reduce the impact on the learning process to a minimum.

VSOL Configurations Partially Satisfying RQ1 Five of the seven VSOL configurations do not satisfy the RQ since they exceed either the 'Batch Learning Time Baseline' (C: 0.033 | L: 0.094) or the accuracy baseline of close to 0. As already mentioned in the RQ1 (see Section 1.2), those configurations will still be considered to see how much the configurations can excel in compression, relevant for 'RQ2: Reduce Required Storage' (Section 1.2). 'Comparison to other Drift Simulations' (Section 4.1.4) shows that the used drift simulation has an overall higher degree of change and the measured error rate could be on average considerably lower under less extreme circumstances. All of the following scenarios are lossy and depicted in Table 4.26. The deviation of accuracy between the two neural networks and between the configurations has no recognizable correlation. Hence, the deviation is unpredictable for other neural networks. However, the maximum of the two deviations has always been chosen to assume a worst-case scenario.

The lossy configurations are categorized into two groups, one with a slow compression time and one with a fast compression time, as shown in the table in the Code Documentation (5.3) .

One of the three fast configurations ('Flors20r32rDifrs148f') has fully satisfied RQ1. The accuracy of the two remaining faster configurations 'Topk7.5p0.1mTlFlors18r32r' (compr. time C: 0.037| L: 0.123) and 'Topk7.5p0.1m*0.5lFlors20r32r' (compr. time C: 0.039| L: 0.094) stays below the accuracy drop thresholds of 0.1% and 1% respectively while only exceeding 'Batch Learning Time Baseline' (C: 0.033 | L: 0.094) of the compression time by about 30ms (25%).

The slower configurations use a Processing Step (see Section 3.2.5) which decides if a model is deployed based on classifing the current data. The higher processing time can indirectly lead to a lower accuracy, as discussed in 'Processing Time affecting Error Rate' (Section 3.3) while the usage of the neural network processing unit can increase the processing time beyond the measured processing time, as discussed in 'Classifying Data Impacting Training' (Section 3.4.2).

The accuracy of the three slow configurations 'Flors20r32rSkisg0pFeTfSplbs' (compr. time C: 0.159 | L: 0.274), 'Flors20r32rSkisg1' (compr. time C: 0.236 | L: 0.297) and 'Topk7.5p0.1m*0.5lFlors20r32rSkisg1.0pFeTf' (compr. time C: 0.243 L: 0.334) stays below the accuracy drop thresholds of 0%, 0.1% and 1%, respectively. Their maximum average processing time exceeds the baseline average by about 250% while the highest value of their maximum processing times exceeds the baseline maximum by 990%.

Depending on the requirements of the user, the configurations can be considered an acceptable tradeoff and indirectly fulfill RQ1 since their training process is only affected in an acceptable range.

6.1.2 Assessment of RQ2: Reduce Required Storage

RQ2 (see Section 1.2) concerns the storage amount used by VSOL. The relevant metric for assessing the impact is the storage amount, measured by the

’Compression Ratio’ (Section 4.3.2) compared to the baseline. Different measurements, presented in Table 4.26, show that the compression ratio between the neural networks Conv and LSTM can vary strongly. Their difference also depends on the chosen configuration and hence, is very unstable and unpredictable for other neural networks, same as with the accuracy deviation in ’Assessment of RQ1: Impact Reduction on Online Learning’ (Section 6.1.1). The table in the Code Documentation (5.3) shows the range of compression ratios for the different configurations.

The compression ratios of the lossless configurations are 1.6 and 4.2 for Conv and LSTM respectively while those of the lossy configurations are 12.5 and 30.0. The compression ratios of the configurations only partially fulfilling RQ1 range from 14.3 to 129.7.

Contextualizing the results is difficult since, to our knowledge, no other work considered these tradeoffs and metrics. While [MLDD17b] does not change the training process - like this thesis - it is not evaluated for online learning and though their results are not clearly comprehensible, their best lossless compression ratio is 2.7. [PDMM21] considers online learning with four major differences to this thesis while stating a theoretical compression ratio of 4590. They change the learning process, have a setup with considerably lower Weight Change Degree, us a considerably larger network which could have more redundancy and only deploy the weights without saving them. While both works can act as a lower and upper bound, [MLDD17b] is more comparable since the given limitations are closer to this thesis while the online learning simulation of [PDMM21] is limited. Considering the smaller sizes of the chosen neural networks, shown in Table 4.1, the requirement to not change the training process, described in ’RQ3: Reduce Integration Effort’ (Section 1.2), the stronger change of the neural network due to the more extreme simulation compared to other drift simulations, described in ’Comparison to other Drift Simulations’ (Section 4.1.4), the required fast execution time, described in ’RQ1: Impact Reduction on Online Learning’ (Section 1.2), the novelty of the system, described in ’Model Recreation’ (Section 2.1) and additionally requiring a materialization as reset point due to ’RQ4: Reduce Retrieval Time’ (Section 1.2), described in ’Delta Reset Point Creator’ (Section 3.2.8), a compression ratio which fastly exceeds [MLDD17b] is impressive.

6.1.3 Assessment of RQ3: Reduce Integration Effort

’RQ3: Reduce Integration Effort’ (Section 1.2) concerns the easy of integration. It is assessed by the requirements listed in the RQ and an self assessed anaylsis of the parameters and lines of code.

The designed and implemented VSOL does not change the learning process and does not require any external resources besides a storage location and therefore fulfills all listed requirements. ’Processing Pipeline of the VSOL’ (Section 3.2) lists all used Processing Steps and explains how they only work on the post weight trainings.

’Portability to other Frameworks’ (Section 3.4.8) describes the prerequisits to integrate the VSOL into another machine learning framework and

'Keras Usage and Integration' (Section 5.2) describes the implemented Keras integration which together helps in comprehending how to integrate the VSOL into a different machine learning framework besides Keras.

'Code for Integration to Training Pipeline with Final Compression Ratios' (Section 5.1) describes the integration process on the basis of the tutorial code and argues in detail why the integration process is simple. Summarizing the code analysis, integration only requires two to three lines. Settings have practical defaults, and parameters are adequately explained in code comments and have practical defaults.

Hence, it was easy to integrate the VSOL into Keras and the VSOL Keras integration can be easly added to a learning process by a user. In conclusion, it can be said that RQ3 is fully satisfied.

6.1.4 Assessment of RQ4: Reduce Retrieval Time

'RQ4: Reduce Retrieval Time' (Section 1.2) concerns the reduction of the model retrieval time and should only bound it without optimizing for it. The relevant metric for assessing the impact is the model retrieval time, measured by decompression time (see Section 4.3.4). Different to RQ2, the decompression time is less affected by the lossy configurations since the lossy precision reduction is executed during compression and only memory reordering connected to the decompression have to be executed. The aimed for decompression time can be configured, as described in 'Code for Integration to Training Pipeline with Final Compression Ratios' (Section 5.1). Table 4.12 and 4.26 show the measured decompression times of the lossless and lossy configurations, respectively. The mean decompression time is below 4.5s for any of the configurations while the maximum decompression time is below 10s. In practice, these values should be considerably lower since they do not have a reset point at the end, as mentioned in 'Final Lossless Configurations with Different Reset Point Interval Settings' (Section 4.8.5). This decompression time should be acceptable when waiting for the recreation of the model.

6.1.5 Summary of Utility of Assessed Processing Steps

The following list, shows the Processing Steps with the most utility, according to our weighting and in descending order:

- **'Reset Least Significant Bits'** features a very significant compression ratio increase (+3.7-7.3) without almost any accuracy deviation and a mild effect on the compression time (max. +0.016). It requires SplitFloatAndStackByByteSegments to achieve these compression ratios.
- **'Top-K'** has a significant compression ratio increase (+1.5-10.8), can be adapted to an accuracy bound and can have a noticable effect on the compression time (max. +0.07). It should be used with a 'Minimum per Layer' setting to increase the accuracy and a 'Loss Adaptive TkP' setting to increase the speed and compression ratio.

- **Zstandard 1** has repeatedly proven to be the fastests lossless general compression algorithm (max. 0.024s) of the assessed general compression algorithms while also almost having the best compression ratio (-0.013 compared to best). 'GCXS' with all optimizations should be used with it since it only features improvments of speed and compression ratio

- **SplitFloatAndStackByByteSegments** is lossless, features a mild but stable compression ratio improvement (min. +0.95), has almost no effect on the compression time (max. +0.004), is the best implementation of 'Bytewise Segmentation for Float Matrices' (Section 3.2.6) and enables a higher compression ratio when paired with entropy reduction algorithms.

- **'Delta Creator'** is lossless, has a significantly improves the compression ratio (min. +0.3), almost having no effect on the compression time (max. +0.008) and being able to have a high Reset Point Interval (RPI) (>148) while retaining an acceptable maximum decompression time for any run (max. 2.64s).

- **'Decide Save by Error rate Decay'** has a low compression ratio increase (+0.14-4.4) compared to the other lossy Processing Steps and has substantial drawbacks like the very slow execution time (+0.196), requiring a lot of memory and using the neural network processing unit but could be essential to save storage space in a different scenario were there is only little drift.

6.1.6 Online Learning Simulation

Since drift is the main aspect of online learning, it is required to check if the VSOL can deal with it, as described in more detail in the intro of 'Online Learning Virtual Drift Simulation' (Section 4.1). 'Comparison to other Drift Simulations' (Section 4.1.4) describes why other used simulations or data sets are insufficient for simulating drift for the VSOL. 'Online Learning Virtual Drift Simulation' (Section 4.1) describes how the drift can be automatically simulated for any dataset with classes and which parameters it requires.

'Determine Data Split Parameter Settings for Virtual Drift Simulation' (Section 4.5) optimizes the drift simulation parameters to curve fit two drifts from different data sets and concludes that it possible to reduce the suggested metric Normalized Last Accuracy mean Difference to 0.102, which is close to optimal by our opinion. 'Effectiveness of Virtual Drift Conversion' (Section 4.5.4) summarizes the impact per parameter and concludes that the given optimizations were limited by the availabile data of the data sets.

In conclusion, it is possible to automatically simulate drift for any dataset with labels and to fit the drift cureves of the given data sets for the given limitations.

6.2 Limitations

The biggest limitation is how the Versioning System for Online Learning systems (VSOL) would work in a production setting. 'Limitations of Implemented Drift System' (Section 4.1.4) states that the Implemented Drift System features a high Weight Change Degree, decreasing the compression ratio compared to a real setting with less drift, and a very specific simulation which can be different in a real setting, having unknown consequences on the metrics of the VSOL. 'Model Selection Process and Limitations' (Section 4.4) mentions the limitations of the chosen models, making it unclear how well a model which has online specific settings, a larger size, a different or more complex architecture would work with the VSOL. 'Evaluation Lossless Configurations' (Section 4.7) and 'Lossy Algorithms' (Section 4.8) show that all evaluation metrics are unstable when comparing them between the used models which indicates that the same is true for other models.

Two configurations fully and five partially fulfill 'RQ1: Impact Reduction on Online Learning' (Section 1.2), as mentioned in 'Main Results' (Section 6.1), impacting 'RQ2: Reduce Required Storage' (Section 1.2) to different extents. Hence, none of the configurations work well for both RQs, letting users decide what advantages and drawbacks are more important for them. 'Limitations of Implementation' (Section 3.4) lists several limitations of the VSOL, most importantly the impact of the slower configurations on the learning process.

In conclusion, the main limitations concern the unforeseeable consequences of a deviating production setting on the evaluation metrics and the tradeoff between the two first RQs.

6.3 Future Works

6.3.1 Further Optimizations on VSOL

Increasing Accuracy for Combinations of Loss Dependent Top-K and Save Decision through New Model

Section 3.2.2 explains loss adaptive Top-k while Section 3.2.5 explains 'Save Decision through New Model'. When 'Save Decision through New Model' prevents saving a version, the loss or delta values between the last saved version and the next set of training weights cumulate. For loss adaptive Top-k handle the increased delta values, the loss value parameter handed over to Top-k should be appropriately adapted. This should incrase the accuracy not only through the higher number of delta values selected by Top-k but also due to the fact that the increased accuracy could lead to outperforming the last saved version which leads to saved version since 'Save Decision through New Model' has detected the degredation of the last saved model.

Although this increases the storage amount for the configuration, it could be used for lower accuracy cutoff requirements, e.g. when it can be used for a 0.1% accuracy decrease requirement instead of a 1% requirement and therefore decrease the required memory since the former combination required even more storage. Table 4.26 shows that two of the slow selected configurations use Top-k and 'Save Decision through New Model' in combination and could maybe replace the configuration with the next lower accuracy threshold.

Since loss is not additive as the same weight between different runs can increase and decrease, the weight of the loss values of the unsaved runs when adding them must be tuned. Additionally, the upper k limit from Top-k must also be increased when several versions have not been saved and therefore the buildup difference cannot be sufficiently represented by a smaller k.

Increase all Metrics through Classification Independent Drift Detection

'Decide Save by Error rate Decay' (Section 3.2.5) has shown to reduce the storage amount and is used in three selected configurations as shown in Table 4.26 while only slightly affecting the accuracy but was ruled out when fast executing is required.

Drift detection was discussed Section 2.3.3 and is broadly discussed in the scientific literature. Implementing and using a drift detection method that only relies on the data without any classification model could make the usage more viable since it executes fastly and would also reduce the decision space of the configurations by one parameter since a slower running configuration could then be unnecessary. This faster drift detection would then be in the 'save decision' Stage (see Section 3.2.1) of an online learning saving system. Experimenting with other drift detection methods using classification could lead to better decisions which could further increase the storage amount or accuracy when the save decision is executed more precisely.

Reduce Decompression Time by Parallelizing File Retrieval

'Delta Creator' (Section 3.2.3) describes how each delta is created while 'Delta Reset Point Creator' (Section 3.2.8) describes how resetpoints are created. Each delta is saved as a separate file and the decompression process loads and decompresses them sequentially.

This process can be parallelized because each delta file is selfcontaining. Merging uncompressed xor delta can even be executed out of order right after the decompression has finished since the xor operation is associative which prevents several uncompressed xor delta from remaining in memory. The reset point and xor deltas can also be combined out of order.

This should significantly decrease the decompression time and can even be implemented retroactively since no additional data is required during the saving process.

Quantization

'Float Compression' (Section 2.2.3) describes the usage of quantization with neural networks. Quantization approximates the weights by assigning each weight a single value from a list of predetermined values which reduces the required memory since each value only needs to address the index of the list. Instead of using quantization, the methods 'Bytewise Segmentation for Float Matrices' (see Section 3.2.6) and 'Lower Bits Reduction'(see Section 3.2.4) were used. Section 3.2.4 argues why this method was preferred over quantization, mainly due to the processing speed.

Table 4.26 shows that some configurations reduce the precision by 20 bits, leaving only 12 bits. As seen in Table 4.20, in practice, the compression factor of reducing 20 bits is not 2.6 but between 4.6 and 8.3. [BNS19] shows that a 4 bit quantization reduces the accuracy at least more the 0.1% and often more than 1%, hence, due to 'RQ1: Impact Reduction on Online Learning' (Section 1.2), a higher number of bits could be required. In conclusion, trying quantization instead of the used Processing Steps mentioned above could, in theory, reduce the required storage amount additionally two-fold if the quantization process can be executed quick enough during compression and decompression.

6.3.2 Optimizing Storage of saved Weight Sets

The following section describes how the design of the Delta Creator (see Section 3.2.3) and the Delta Reset Points (see Section 3.2.8) can be harnessed to retrospectively reduce the storage amount or decrease the decompression time when loading a weight set. Deltas are used in all resulting VSOL configurations. Delta Reset Points can be removed or added since they are reproduceable by using another close reset point and the deltas in between both reset points. The implemented VSOL does not support this since it relies on its created reset point in a certain interval but it is retrospectively possible, as described in 'Delta Reset Point Creator' (Section 3.2.8).

Removing Reset Points

Reset Points are only used to reduce the decompression time. One reset point would be enough to load any set of weights through other deltas. Hence, removing selected reset points can be an acceptable trand-off between a lower storage amount and a higher decompression time for the weights before or after the reset point. An example for a selection technique is the prediction of the access frequency of its surrounding weights. The age and the past access frequency can be used for such a prediction.

Removing Unused Weights

In a scenario where the usage of the deployed model is infrequent, versions which where never used to process any data could be removed completely in hindsight. This must involve the merger between the deleted delta and its successor. Since all timestamps of when data instances were being classified have to be saved, all of the information should be present to implement this later on.

Adding Reset Points

If a certain set of weights in time or a certain time period is requested more often, reset points can be recreated to reduce the decompression time in retrospective. A higher storage amount would then justify the decreased decompression time.

6.3.3 Measuring Error Rate Impact by Delayed Deployment

'Processing Time affecting Error Rate' (Section 3.3) discusses how delaying the deployment of a newer and drift adapted model leads to data instances being misclassified which affects the error rate. To our knowledge, this topic is not discussed although it remains unclear if this is a real issue in practice. In the papers viewed by us and the simulation conducted in this thesis, the prequential accuracy (see Section 4.3.1) measurements assume that the upcoming batch can be delayed until the newer model is deployed and is then processed. In a scenario where a delay cannot be accepted due to a time sensitive task, the measurement must assume that the next test batch is classified by the old model until the newer model is finally deployed. A real time simulation would involve data were each instance has a classification and label arrival timestamp and two models, one for training and one for classification. The classification model is then replaced by the training model only when it has finished processing the labeled data and is ready to be deployed. An upper limit baseline in this simulation would be to assume that the time is halted when a new labelled data batch is read for training until the training model has finished processing the batch and is deployed. This baseline is similar to the current prequential accuracy simulations except that the datas timestamps could be out of order whereas the prequential accuracy assumes the classification and arrival timestamp to be after each other and no other timestamp of another data instance being inbetween.

6.4 Summary

It was shown that it is possible to generically create a virtual drift from labeled data and align the drift curves of two different datasets with different neural networks to maintain comparability.

This thesis has shown that it is possible to build an easily integrable Versioning System for Online Learning systems (RQ3) bounded by limitations of not affecting the training process (RQ1) and retaining an acceptable retrieval time (RQ4) while achieving a compression ratio between 1.6 and 30.0 (RQ2). If compromises on the training process are acceptable, the compression ratio can even range between 14.3 to 129.7.

The source code of both the VSOL and its evaluation is available on GitHub[1].

[1] https://github.com/christopheitenberger/VSOL

List of Figures

List of Tables

Acronyms

Lo Loss. 21, 22, 78

$LoBN$ Loss Bound and Normalized. 22, 72, 103

$LoBNT$ loss bound normalized and transformed. 22, 72, 103

$LoUB$ Loss Upper Bound. 22, 71

$MDTT$ Maximum Decompression Time Threshold. 30, 31, 32

MtA Move to Axis. 22

Pl number of Parameters per Layer. 20, 21

$Plmax$ number of Parameters of layer with the most parameters. 20, 21

RPI Reset Point Interval. 30, 31, 59, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 73, 74, 75, 76, 77, 79, 80, 81, 96, 97, 103, 105, 106, 107, 108, 109

$TkAbMin$ Top k Absolute number of Parameters for Layer Minimum bounded. 20, 21

$TkAbPl$ Top k Absolute number of Parameters for Layer. 20, 21

TkP Top-k-Percentage. 19, 20, 21, 22, 66, 67, 68, 69, 70, 71, 72, 73, 75, 76, 77, 79, 80, 81, 82, 96, 103, 106, 107, 108, 109, 110

$TkPLoA$ Top k-Percentage loss adapted. 22, 72

$f_{LoTrans}$ Loss Transformation Function. 22, 67, 68, 69, 70, 71, 72, 73, 76, 77, 79, 80, 81, 82, 103, 106, 107, 108, 109, 110

$max(t_c)$ Maximum Compression Time measured. 31

$minLP$ Minimum per Layer Percentage. 20, 21, 69, 70, 81, 82

$minPl$ Minimum number of Parameters per Layer. 20, 21, 22

DSED Decide Save by Error rate Decay. 26, 37, 75, 79, 80, 81, 82, 97, 99, 108, 109, 110

IDS Implemented Drift System. 44, 45, 51, 98

LSBF Least Significant Bits of Fraction. 24, 25, 73

LSBFO Least Significant Bits of Fraction to Overwrite with zeros. 25, 26, 73, 74, 75, 76, 77, 79, 80, 81, 82, 108, 109, 110

MSBF Most Significant Bits of Fraction. 24, 25, 26, 72

MSBFA Most Significant Bits of Fraction to Assess. 25, 72, 73, 74, 108

NLAcc Normalized Last Accuracy. 43, 53, 54, 55, 56, 57, 103, 105

NLAccDiff Normalized Last Accuracy mean Difference. 43, 53, 54, 55, 56, 97, 105

POff Percentage Offline data. 42, 43, 53

RLSB Reset Least Significant Bits. 25, 26, 37, 72, 73, 74, 75, 76, 77, 79, 80, 81, 82, 96, 108, 109, 110

RLSBMSB Reset Least Significant Bits when Most Significant Bits set. 24, 26, 37, 72, 74, 108

RMRS Retraining for Model Recreation System. xiv, 7, 8, 82, 83

TBOn Total Batches Online data. 42, 43, 44, 45, 53, 54, 55, 56, 57, 61, 103, 105

UPMOn Underrepresented label Percentage Maximum Online data. 42, 43, 45, 53, 54, 55, 56, 57, 103, 105

UPOff Underrepresented label Percentage Offline data. 42, 43, 53, 54, 55, 56, 57, 103, 105

VSOL Versioning System for Online Learning systems. ix, xi, xiii, xiv, 2, 5, 7, 8, 9, 12, 13, 15, 16, 17, 18, 19, 21, 23, 24, 25, 26, 27, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39, 44, 45, 46, 47, 48, 49, 50, 51, 64, 82, 83, 85, 86, 87, 88, 90, 91, 93, 94, 95, 96, 97, 98, 100, 101, 103, 104

Bibliography

[AGL$^+$17] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *Conference on Neural Information Processing Systems*, volume 30, pages 1709–1720. Curran Associates, Inc., 2017.

[Ass19] Yahia Assiri. Stochastic Optimization of Plain Convolutional Neural Networks with Simple methods. In *International Conference on Machine Learning and Data Mining*, volume II, pages 833–844. ibai Publishing, July 2019.

[BCH$^+$15] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proceedings of the VLDB Endowment*, 8(12):1346–1357, August 2015.

[BKD21] Adam Byerly, Tatiana Kalganova, and Ian Dear. No routing needed between capsules. *Neurocomputing*, 463:545–553, November 2021.

[BM08] Stephen H. Bach and Marcus A. Maloof. Paired Learners for Concept Drift. In *IEEE International Conference on Data Mining*, volume 8, pages 23–32. IEEE Computer Society, December 2008.

[BNS19] Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Conference on Neural Information Processing Systems*, volume 32, pages 7948–7956. Curran Associates, Inc., 2019.

[BS18] Roberto Souto Maior Barros and Silas Garrido T. Carvalho Santos. A large-scale comparison of concept drift detectors. *Information Sciences*, 451–452:348–370, July 2018.

[CAB18] Steven Claggett, Sahar Azimi, and Martin Burtscher. SPDP: An Automatically Synthesized Lossless Compression Algorithm for Floating-Point Data. In *Data Compression Conference*, volume 28, pages 335–344. IEEE, March 2018.

[CCB^{+}18] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. AdaComp : Adaptive Residual Gradient Compression for Data-Parallel Distributed Training. *AAAI Conference on Artificial Intelligence*, 32(1):2827–2835, April 2018.

[CCD^{+}20] Andrew Chen, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Avesh Singh, Fen Xie, Matei Zaharia, Richard Zang, Juntai Zheng, and Corey Zumar. Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *Workshop on Data Management for End-To-End Machine Learning*, volume 4, pages 1–4. ACM, June 2020.

[CLC^{+}19] Ryan Chard, Zhuozhao Li, Kyle Chard, Logan Ward, Yadu Babuji, Anna Woodard, Steven Tuecke, Ben Blaiszik, Michael J. Franklin, and Ian Foster. DLHub: Model and Data Serving for Science. In *IEEE International Parallel and Distributed Processing Symposium*, volume 33, pages 283–292. IEEE, May 2019.

[DKS19] Hariharan Devarajan, Anthony Kougkas, and Xian-He Sun. An Intelligent, Adaptive, and Flexible Data Compression Framework. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, volume 19, pages 82–91. IEEE, May 2019.

[DLF20] Jacob Dexe, Jonas Ledendal, and Ulrik Franke. An Empirical Investigation of the Right to Explanation Under GDPR in Insurance. In *Trust and Privacy in Digital Business*, volume 17, pages 125–139. Springer, 2020.

[DMRM19] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Tilmann Rabl, and Volker Markl. Continuous Deployment of Machine Learning Pipelines. In *International Conference on Extending Database Technology*, volume 22, pages 397–408. OpenProceedings.org, March 2019.

[GBEB18] Heitor Murilo Gomes, Jean Paul Barddal, Fabrício Enembreck, and Albert Bifet. A Survey on Ensemble Learning for Data Stream Classification. *ACM Computing Surveys*, 50(2):1–36, March 2018.

[GŽB^{+}14] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys*, 46(4):Article 44 1–37, March 2014.

[HPC12] T. Ryan Hoens, Robi Polikar, and Nitesh V. Chawla. Learning from streaming data with concept drift and imbalance: An overview. *Progress in Artificial Intelligence*, 1(1):89–101, April 2012.

[HRFS16] Seyyed Hossein Hasanpour, Mohammad Rouhani, Mohsen Fayyaz, and Mohammad Sabokrou. Lets keep it simple, Using simple architectures to outperform deeper and more complex architectures. *arXiv:1608.06037*, August 2016.

[HSLZ21] Steven C. H. Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. Online learning: A comprehensive survey. *Neurocomputing*, 459:249–289, October 2021.

[IKJS17] Vamsi K. Ithapu, Risi Kondor, Sterling C. Johnson, and Vikas Singh. The Incremental Multiresolution Matrix Factorization Algorithm. In *Conference on Computer Vision and Pattern Recognition*, pages 692–701. IEEE, July 2017.

[kag17] Kaggle Notebook - Convolutional Neural Network Keras for MNIST Dataset. `https://kaggle.com/code/yassineghouzam/introduction-to-cnn-keras-0-997-top-6`, August 2017. (accessed 2023-07-26).

[kag21] Kaggle Notebook - LTSM Neural Network Keras for AG News Classification Dataset. `https://kaggle.com/code/ishandutta/ag-news-classification-lstm`, November 2021. (accessed 2023-07-27).

[KTF21] Fabian Knorr, Peter Thoman, and Thomas Fahringer. Ndzip: A High-Throughput Parallel Lossless Compressor for Scientific Data. In *Data Compression Conference*, volume 31, pages 103–112. IEEE, March 2021.

[LDG$^+$17] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, and Feng Sun. Model Ensemble for Click Prediction in Bing Search Ads. In *The Web Conference*, volume 26, pages 689–698. ACM, April 2017.

[LHM$^+$20] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *International Conference on Learning Representations*, volume 6. OpenReview.net, June 2020.

[Lin14] Peter Lindstrom. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, December 2014.

[LLD$^+$19] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, João Gama, and Guangquan Zhang. Learning under Concept Drift: A Review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346–2363, December 2019.

[MD18] Hui Miao and Amol Deshpande. ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows. *IEEE Data Engineering Bulletin*, 41(4):26–38, 2018.

[MLDD17a] Hui Miao, Ang Li, Larry S. Davis, and Amol Deshpande. ModelHub: Deep Learning Lifecycle Management. In *International Conference on Data Engineering*, volume 33, pages 1393–1394. IEEE Computer Society, April 2017.

[MLDD17b] Hui Miao, Ang Li, Larry S. Davis, and Amol Deshpande. Towards Unified Data and Lifecycle Management for Deep Learning. In *International Conference on Data Engineering*, volume 33, pages 571–582. IEEE Computer Society, April 2017.

[pap23a] Papers with Code - AG News Benchmark (Text Classification). `https://paperswithcode.com/sota/text-classification-on-ag-news`, July 2023. (accessed 2023-07-25).

[pap23b] Papers with Code - MNIST Benchmark (Image Classification). `https://paperswithcode.com/sota/image-classification-on-mnist`, July 2023. (accessed 2023-07-26).

[PDMM21] Ioannis Prapas, Behrouz Derakhshan, Alireza Rezaei Mahdiraji, and Volker Markl. Continuous Training and Deployment of Deep Learning Models. *Datenbank-Spektrum*, 21(3):203–212, November 2021.

[PFG18] Beatriz Pérez-Sánchez, Oscar Fontenla-Romero, and Bertha Guijarro-Berdiñas. A review of adaptive online learning for artificial neural networks. *Artificial Intelligence Review*, 49(2):281–299, February 2018.

[PPA18] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. In *International Conference on Learning Representations*, volume 6. OpenReview.net, February 2018.

[RGZ$^+$21] Andreas Rauber, Bernhard Gößwein, Carlo Maria Zwölf, Chris Schubert, Florian Wörister, James Duncan, Katharina Flicker, Koji Zettsu, Kristof Meixner, Leslie D. McIntosh, Reyna Jenkyns, Stefan Pröll, Tomasz Miksa, and Mark A. Parsons. Precisely and Persistently Identifying and Citing Arbitrary Subsets of Dynamic Data. *Harvard Data Science Review*, 3(4), October 2021.

[SBK$^+$17] Sebastian Schelter, Joos-Hendrik Böse, Johannes Kirschnick, Thoralf Klein, and Stephan Seufert. Automatically Tracking Metadata and Provenance of Machine Learning Experiments. In *Machine Learning Systems Workshop at the Conference on Neural Information Processing Systems*, volume 30. learningsys.org, 2017.

[SCG$^+$22] Jun Sun, Tianyi Chen, Georgios B. Giannakis, Qinmin Yang, and Zaiyue Yang. Lazily Aggregated Quantized Gradient Innovation for Communication-Efficient Federated Learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(4):2031–2044, April 2022.

[SH15] Md Abu Hanif Shaikh and K.M. Azharul Hasan. Efficient storage scheme for n-dimensional sparse array: GCRS/GCCS. In *International Conference on High Performance Computing & Simulation*, pages 137–142. IEEE, July 2015.

[Sik97] T. Sikora. MPEG digital video-coding standards. *IEEE Signal Processing Magazine*, 14(5):82–100, September 1997.

[spa21] Sparse Python Package (0.13.0) - GCXS Documentation. `https://sparse.pydata.org/en/0.13.0/generated/sparse.GCXS.html#sparse.GCXS`, March 2021. (accessed 2023-10-19).

[SZS19] Devendra Singh Sachan, Manzil Zaheer, and Ruslan Salakhutdinov. Revisiting LSTM Networks for Semi-Supervised Text Classification via Mixed Objective Function. In *AAAI Conference on Artificial Intelligence*, volume 33, pages 6940–6948. AAAI Press, July 2019.

[ten22] Tensorflow Python Package Configuration (2.10.1) - 'enable_op_determinism' Documentation. `https://github.com/tensorflow/tensorflow/blob/v2.10.1/tensorflow/python/framework/config.py#L925`, February 2022. (accessed 2023-09-08).

[TMK16] Nedelina Teneva, Pramod K Mudrakarta, and Risi Kondor. Multiresolution Matrix Compression. In *International Conference on Artificial Intelligence and Statistics*, volume 51, pages 1441–1449. JMLR.org, May 2016.

[VSL+16] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. ModelDB: A system for machine learning model management. In *Workshop on Human-In-the-Loop Data Analytics*, page 14. ACM, 2016.

[YDY+19] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Conference on Neural Information Processing Systems*, volume 32, pages 5754–5764. Curran Associates, Inc., 2019.