Exploring and Analyzing Compression Algorithms in Preprocessing Pipelines

Adrian David Castro Tenemaya
adrian.castro@tum.de
Technische Universität München

ABSTRACT

End-to-end training pipelines nowadays take a lot of time to train due to the rise in model size and the amount of data needed to feed them. Most recent literature focuses on ways to optimize the model training phase, ignoring preprocessing and data handling times in both local and distributed environments. However, preprocessing may take up to 45% of the total time in end-to-end training pipelines. Compression algorithms have proven to be useful in several areas to reduce I/O network bandwidth and storage consumption. In turn, the use of compression leads to additional processing power overhead during compression and decompression stages. It is crucial then to analyze and select appropriate compression strategies depending on the environment, with the goal to find a good balance between compression ratio and speed.

In this work, we show the integration of the Zstandard compression library into the popular machine learning library TensorFlow. We further explore and analyze the effects of data compression on storage and network usage with respect to deep-learning preprocessing pipelines. We also show the impact of different compression options for GZIP and ZLIB compared to default parameters. Lastly, we include observations regarding the impact of different compression algorithms on several datasets and data types (e.g. JPEG vs. PNG).

1 INTRODUCTION

Deep learning is data and computationally intensive, with end-to-end workflows possibly taking weeks or months to complete. Besides, neural networks are trained on expensive dedicated hardware such as GPUs and TPUs to reduce training times. Training can either happen on-premise or on the cloud. The cost of training is generally calculated depending on the time spent using the devices. Because of this, practitioners and researchers have to find ways to maximize their usage in terms of computational power and minimize the amount of time spent training them.

Over the past few years, research in this area has been focused on the optimization of the model training phase, with techniques such as data and model parallelism being the most used. This has allowed deep learning workflows to scale horizontally to models with billions of parameters [21]. With the increase in computational power, data ingestion and preprocessing pipelines need to keep up with the high amount of throughput needed to keep the training hardware as busy as possible. Recent research has shown that these types of pipelines can take up to approximately 45% of the total end-to-end training workflow time [24]. However, the development of techniques and frameworks trying to optimize data preprocessing and ingestion is limited, receiving little to no attention in research.

Recent work [13] has shown potential space for improvement in this area. For example, applying certain transformations during the preprocessing phase may decrease data size. If the dataset then fits into memory, network I/O bottlenecks can be severely decreased due to most operations now happening locally. The use of lossless compression algorithms in preprocessing pipelines has been found to increase throughput under certain conditions.

We reproduced the results shown in [13] and found that the compression algorithms used may increase throughput. However, this comes at a high cost in terms of processing time. ZLIB has several strategies that perform differently according to how data is represented on disk. If not selected, ZLIB defaults to `Z_DEFAULT_STRATEGY` as its compression strategy. In this paper, we further investigate the effects of ZLIB and its different strategies on the same datasets and pipelines as [13] using the PRESTO library.

The `DEFLATE` compression algorithm, used by both ZLIB and GZIP, is quite old and does not reflect current advances in technologies and system architectures. Intuitively, more recent compression algorithms may behave differently and perform better on the same datasets. Zstandard [1] is a lossless compression algorithm developed by Facebook in 2015 and has gained a lot of traction amongst practitioners for its speed and high customizability. We decided to run the same experiments using Zstandard and compared the results against other compression algorithms and strategies. However, TensorFlow's `TFRecord` only supports GZIP and ZLIB. As part of our work, we created a custom TensorFlow implementation to include the lossless compression algorithm Zstandard. Our analysis reveals that Zstandard decreases compression and decompression overhead times by up to 0.81% compared to GZIP and ZLIB in all scenarios.

In this paper, we argue that the selection of a compression algorithm is crucial to minimize compression and decompression overhead and potentially maximize throughput. We further analyze the challenges arising from the implementation of compression in deep learning preprocessing pipelines. In particular, we present what we have learned about the current state-of-the-art regarding the use of compression algorithms in deep learning and their impact on training. Finally, we showcase possible suggestions regarding the use of compression in preprocessing pipelines.

2 SETUP AND BACKGROUND

We start by describing the setup that was used to perform the tests. Our experiments were run using a virtual machine with 40GB DDR4 DRAM, 16 VCPUs on an Intel Xeon E5-2630 v3 16x@2.4 GHz with an Ubuntu 18.04 image on our OpenStack cluster. We use a Ceph cluster as our main storage, backed by HDDs, with a 10 Gb/s uplink and downlink. Ceph is used for storing the intermediate dataset representations and the unprocessed datasets. We repeat each experiment five times, and we drop the page cache after every

[1] https://facebook.github.io/zstd/

run to remove memory caching effects. All experiments are run with 8 threads and are executed with Python 3.7 and TensorFlow 2.7. We included Zstandard 1.5.0 in all experiments that extended `TFRecord`.

2.1 Datasets

For our experiments, we decided to stick to 4 out of 6 of the datasets analyzed by Isenko et al. [13]. A detailed list of properties of these datasets is represented in Table 1.

2.2 A Brief History of Compression Algorithms

Data compression algorithms have been used almost since the very beginning of telecommunications. A very early example is Morse code. Common English letters "e" and "t" are abbreviated with shorter code words, hence, achieving compression when communicating those letters over long messages. During the late 1970s, with the early development of inter-machine communication over cable came the need to reduce the storage footprint of data and communication overhead, officially kick-starting research into compression algorithms.

We have two macro-types of compression algorithms: *lossless* and *lossy* compression algorithms. The former can be applied to input data to reduce its size, and the result can later go through a process called *decompression*, which restores the original input without any loss of information. The latter is still applied to input data to reduce its size, but in such a way that some original information is lost, and recovering the original input is not possible.

The first compression algorithms were lossless, using the well-known Huffman Coding, invented by David Huffman in 1949. Today, Huffman Coding is still being used in almost every lossless compression algorithm. Most modern lossless compression algorithms, such as LZ77 [2] and LZW [3], started to show up later in the century.

Lossy compression algorithms are a more vague categorization. Image cropping, rotation, down-scaling, and other operations can be seen as examples of lossy compression, as the application of these functions has a result that may not retain all the input's original information. In medical applications, for example, x-rays are generally represented as gray-scale pictures. For this reason, every pixel does not need to be represented by three channels, but just one, with a potential 3x saving in terms of storage space. Lossy compression algorithms have everyday life applications: audio formats (MP3), image formats (JPEG), and more.

In general, compression is used across a wide array of topics. When surfing the web, HTML pages are probably compressed using either GZIP or brotli, and pictures are either losslessly compressed (PNG) or lossy compressed (JPG).

2.3 Preprocessing Pipelines

The preprocessing pipeline is a crucial component of end-to-end deep learning workflows, and defines how data should be acquired and consumed by the model before and during training. They can be split into steps or operations, each either running once, called "offline" steps, or iteratively throughout training, called "online" steps. The order and type of operations differ according to the type of DNN that is being trained and have a huge impact on a model's training performance and accuracy. Offline steps are operations that are static and deterministic, such as image decoding and resizing. Online steps instead are typically non-deterministic and augmentation operations, such as random image cropping or random brightness image augmentation.

Selecting the order of the steps in preprocessing pipelines is a complex task. For this work, we stick to the step selection chosen by [13].

2.4 Distributed Storage

To train deep neural networks, large amounts of data are needed. Preprocessing pipelines need to provide the highest amount of throughput to accelerators to keep them as busy as possible. This translates into the need to serve terabytes or even petabytes of data across the entire training workflow. Network speeds are greater than most storage devices, even SSDs, by at least one order of magnitude. At the time of writing, average network cards installed in servers can provide speeds of up to 100Gb/s, while high-end SSDs can reach speeds of up to 4-5Gb/s. HDDs are even slower, with transfer rates topping 1.2Gb/s, but are also the cheapest viable option for large-scale storage.

Deep learning training generally involves repeatedly read and write operations on shuffled data. However, if data is stored on hard disks, read operations need to perform random access to disks, a notably slow task. Even if solid-state drives were used, speeds would not remotely come close to what a network connection can provide.

For a long time, parallelism techniques have made it possible to exploit high network speeds instead of just relying on sequential file storage access. To achieve high performance in file intensive tasks, distributed storage solutions such as HDFS [20], Ceph [23], TensorFlow `TFRecord` [4] and Amazon S3 [5] exist. Each one of these solutions has its own set of benefits and disadvantages, but employs similar approaches to how to store and serve data to achieve high-performance I/O over the network. Files can be stored on different servers and then later be accessed by a client through the network, achieving higher speeds than sequential access.

In this paper, we employ a Ceph cluster as our main storage solution, with a 10Gb/s uplink and downlink.

2.5 Related Work

Efficiently managing computational resources, network I/O and storage consumption is especially important for Internet-of-Things (IoT), mobile, and edge devices. Most of the time, it is not possible to control the hardware characteristics of the target devices, or to even know them. In some cases, consumption and computational power may be bound to several orders of magnitude with respect to regular or dedicated devices. This leads to the need to reduce the amount of computational power being used by algorithms, as well as storage consumption.

Compression techniques to reduce model size. Training and performing inference for deep learning models have typically much higher requirements in terms of computational power than regular

[2] Lempel-Ziv, 1977

[3] Lempel-Ziv-Welch, 1984

[4] https://www.tensorflow.org/tutorials/load_data/tfrecord

[5] https://aws.amazon.com/s3/

Dataset name	Samples	Size in GB	Steps			
Cube++ PNG [10]	4890	85.17	2-read-image	3-decode-image	4-resize-image	5-center-pixel-values
Cube++ JPG [10]	4890	2.54	2-read-image	3-decode-image	4-resize-image	5-center-pixel-values
Librispeech [17]	28K	6.61	2-read-and-decode-flac	3-convert-to-spectrogram		
Commonvoice [4]	12K	0.25	2-read-and-decode-mp3	3-convert-to-spectrogram		

Table 1: List of datasets used for this paper, with number of samples and preprocessing steps.

IoT software. As models get bigger, more computational power is needed to execute operations, more RAM to fit weights and runtime variables, and more storage to house data and the model itself. Consequently, it is simply not possible to store and run state-of-the-art deep learning models, which may contain hundreds of millions, if not billions of parameters. Compression techniques such as pruning and quantization have been applied to model weights to reduce their storage consumption and memory footprint [5, 11, 12].

While in this paper our main focus is on preprocessing pipelines, these issues are currently being tackled using not too dissimilar approaches.

Analysis of image compression formats impact on DNN accuracy. It is fundamental to understand how compression algorithms affect a model's accuracy. Images that may look similar to human vision can be fundamentally different at a data level. In [8], they analyze the effects of different artifacts in JPEG and JPEG-2000 on different computer vision pipelines. According to their findings, models are more sensitive to Gaussian blur and noise with respect to compression level and contrast. Liu et al. [16] present DeepN-JPEG, a JPEG-based image compression format tailored to DNN performance, while maintaining the same accuracy levels.

Data formats for fast I/O. Different data representations on disk can affect access speed at runtime. As shown by Aizman et al. [2], practitioners do not necessarily need to come up with their own data format to achieve great speeds. In their work, the old data format TAR is used by enforcing simple conventions on filenames and directory structure and iterated through their PyTorch client WebDataset. The use of TAR makes their approach intriguing to researchers and practitioners, as it is widely supported across almost every system, and it is simple to integrate into already existing pipelines.

Very recently, the FFCV [14] PyTorch data loader entered the data loaders scene. As shown by their experiments, it is able to speed up data loading times significantly, resulting in much lower prices for dedicated accelerators. It can directly convert indexed PyTorch dataset formats as well as WebDataset.

3 EXTENDING TFRECORDWRITER

In deep learning, we generally load files in batches, which are then consumed by dedicated hardware during model training. File representation on disk plays a major role in how efficiently batches are loaded into memory, especially if data is stored on disk. In this case, randomly accessing binary data can be extremely slow. Google's TensorFlow provides the `TFRecord` file format to deal with these challenges, along with `tf.data` API to access it. `TFRecord` can store several data points in a single or multiple binary files called "shards". This limits random access to disk to a minimum, speeding up file reading from disk. Benefits of sharding have already been studied previously in [2].

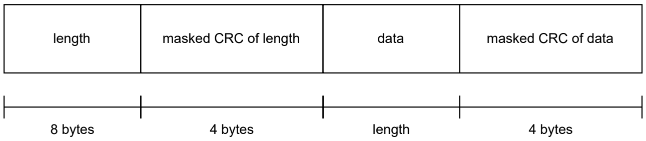


Figure 1: The structure of a TFRecord.

However, it is worth noting that storing files using `TFRecord` comes with the disadvantage that some preprocessing is needed. Specifically, shuffling needs to take place prior to writing to disk, as `TFRecord` does not support random access. These files can then be loaded in parallel, further speeding up read operations.

TensorFlow's release is 2.7, which is the version we started to work with at the beginning of the experiments, has `TFRecord` supporting GZIP and ZLIB as compression techniques. Isenko et al. [13] showed that these implementations are able to achieve higher data throughput under certain circumstances [13]. They compare space-saving and throughput when applying compression to preprocessing steps. When comparing JPG vs. PNG in the Cube++ dataset [9, 10], they found that the PNG pipeline results in much higher space-saving and throughput when compared to JPG.

These findings, they argue, may be explained by the fact that PNG is a losslessly-compressed file format, while JPG is lossy-compressed. Converting from other image file formats to JPG introduces artifacts [16]. These artifacts may negatively affect further compression when using ZLIB and GZIP [13]. However, further tests are needed, and other compression algorithms need to be studied to confirm this behavior.

In our paper, we started by wanting to extend these findings by adding another lossless compression algorithm to compare GZIP and ZLIB on the same experiments as [13]. To do this, we had to take a more in-depth look at how `TFRecord` writes to disk and how ZLIB and GZIP are integrated into it to decide which compression algorithm to experiment with.

`TFRecord` binary files are a series of consecutive items, and every item inside contains a header, the actual data, and a footer (see Figure 1). When initializing a new `TFRecord`, an instance of the class `tensorflow::io::WritableFile` is created depending on the selected compression type, or its absence. In the latter case, the writer class will just write contents to the file. Instead, when a compression algorithm has been selected, every new item added to `TFRecord` gets compressed.

Typically, compression algorithms provide APIs to support two different cases: regular compression and streaming compression. Regular compression, put simply, requires immediate knowledge and access to the whole input data and its original size. This speeds up and improves compression ratio, and is generally the most common use case. In some scenarios, however, access to all the information about a dataset is not immediately available. This is the case when dealing with data streaming, as data flows at very high and irregular rates. Streaming compression APIs are built exactly to deal with this problem. Instead of compressing everything at once, an internal state is generally kept across compression runs. This way, every time a new item needs to be compressed, previous information is not lost, and the compression rate improves. Because `TFRecord` does not make assumptions about how big the original dataset is and allows users to programmatically add items to existing records, streaming APIs are used when performing compression.

Following these observations, we chose to add support for Zstandard (ZSTD) to `TFRecord`, using its Streaming API. The library is written in C, and is being actively maintained.

4 EXPERIMENTS

Over the years, compression algorithms have been invented and altered depending on the task at hand. We devised our experiments to shed light on how different compression algorithms may affect preprocessing pipelines' performance. We use the same datasets as [13], described in Table 1, with the exception for Openwebtext and ImageNet. In the original paper, the number of runs per experiment is five. However, for this work, the number of runs has been reduced to three times due to a change in specifications of the virtual machine that runs the experiments.

The compression algorithms we analyzed are ZLIB and ZSTD, with the following configurations:

- **ZLIB compression strategies**. ZLIB uses `Z_DEFAULT_STRATEGY` as the default compression strategy, and it is reportedly good for general applications. However, there exist more strategies that presumably have better performance. We experimented with different ZLIB compression strategies, namely:
  `Z_FIXED`: prevents the use of dynamic Huffman codes. Instead, a fixed table is used [6];
  `Z_FILTERED`: can be used for data produced by a filter, forcing more Huffman encoding and less string matching;
  `Z_HUFFMAN_ONLY`: forces Huffman encoding only, without string matching;
  `Z_RLE`: specifically designed to give better compression for PNG data, but be almost as fast as `Z_HUFFMAN_ONLY` [7]. It uses the Run-Length Encoding [19] as its main compression strategy, backed by a dynamic Huffman table.
- **ZSTD**. We implemented the ZSTD streaming API, using the default compression strategy for `ZSTD_c_strategy = 0`. Multithreading is not currently supported by our implementation but can be added in the future for possible speedups in a multithreading environment.

The compression level remained the default for both ZLIB and ZSTD, respectively at 6 and `ZSTD_CLEVEL_DEFAULT`.

4.1 ZLIB

ZLIB is a very common and general-purpose compression algorithm. In its default strategy `Z_DEFAULT_STRATEGY`, it uses a balanced combination of Huffman Encoding and string matching to perform compression. This is generally good for most applications, but in specific scenarios, it may overperform or underperform significantly. We make the following observations based on Figure 2:

(1) Some strategies have similar performance. The compression strategies `Z_RLE` and `Z_HUFFMAN_ONLY`, when compared, perform very closely to each other in terms of both speed and throughput. The same effect can be seen for `Z_DEFAULT_STRATEGY`, `Z_FILTERED` and `Z_FIXED`. This effect is not coincidental.

`Z_RLE` comes with the advantages of `Z_HUFFMAN_ONLY`, as it only has to store the Huffman buffer and calculate the Huffman trees [1]. `Z_DEFAULT_STRATEGY`, `Z_FILTERED` and `Z_FIXED` all use string matching, differing only in how they make use of Huffman encoding. Note that both `Z_RLE` and `Z_HUFFMAN_ONLY` have up to 1.4x higher throughput in the Cube++ PNG pipeline compared to no compression but lower throughput than other strategies. In both the Cube++ JPG and Cube++ PNG pipelines, we can observe that `Z_RLE` and `Z_HUFFMAN_ONLY` also execute much faster than `Z_DEFAULT_STRATEGY`, `Z_FILTERED` and `Z_FIXED`.

Interestingly, when looking at Librispeech and Commonvoice, all experimented algorithms show similar performance. Furthermore, `Z_RLE` and `Z_HUFFMAN_ONLY` are even able to outperform the absence of no compression, taking up to 80% as much time. These observations can help us understand how different approaches to data compression can affect how fast data is consumed. In this particular case, the absence of string matching in both `Z_RLE` and `Z_HUFFMAN_ONLY` can explain both the significant speed reduction and the lower throughput.

(2) Storage savings are still not a predictor of throughput. Across every test we analyzed, we could not pinpoint a trend that may correlate storage savings and throughput. We suspect that compression and decompression speed may play a major role in determining throughput. However, we don't have specific data about these metrics, so more testing is needed.

4.2 Zstandard

By default, Zstandard is not integrated into TensorFlow. For this reason, we had to extend TensorFlow's codebase to include Zstandard, in particular, its advanced streaming compression API. No support for multithreading has been added for now, due to the elevated complexity of integrating it into TensorFlow. After constructing a successful implementation, we analyzed its performance against no compression, ZLIB using `Z_DEFAULT_STRATEGY` and GZIP. We make the following observations based on Figure 3:

(1) ZSTD runs much faster in certain scenarios, but not always. Running tests on Cube++ showed that the use of Zstandard improved execution times significantly in both JPEG and PNG scenarios. ZLIB and GZIP are quite slow, taking up to 9x as much as no compression at all (see Figure 3d). Zstandard on the other hand takes only 1.5x as much as no compression. However, this speedup is not consistent across other datasets. In fact, Zstandard is slower than no compression, ZLIB, and GZIP in Librispeech

[6] https://www.rfc-editor.org/rfc/rfc1951
[7] http://optipng.sourceforge.net/pngtech/z_rle.html

and Commonvoice. As noted by [13], this may be due to a CPU bottleneck

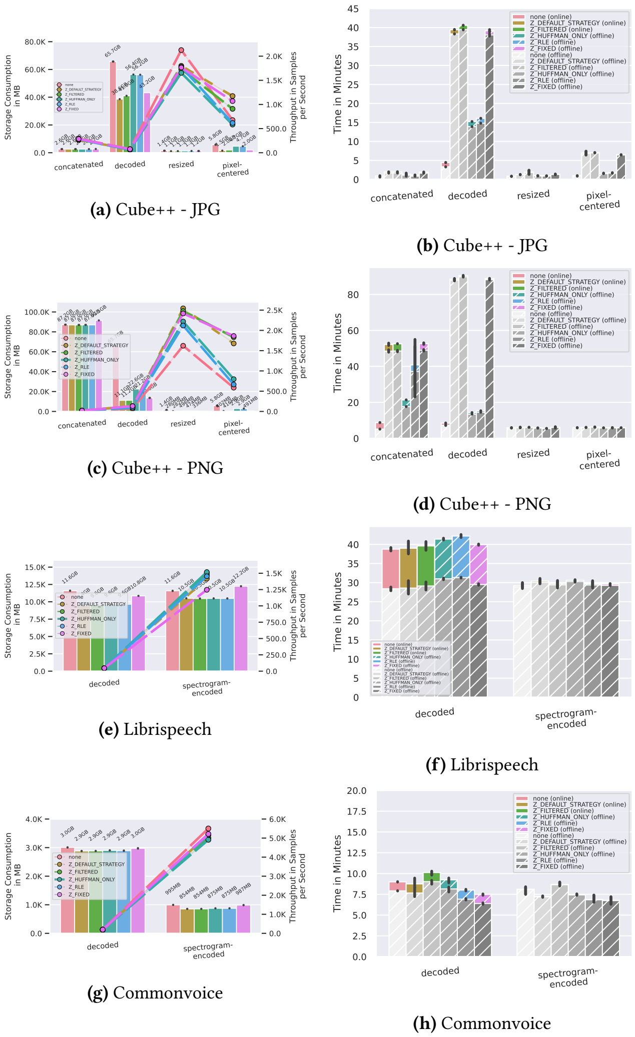
(2) Compression ratio is similar to ZLIB and GZIP. In our tests, we observed that storage savings for ZSTD don't diverge from the average of other compression algorithms. Actually, Zstandard does not aim to provide a better compression ratio than ZLIB or GZIP and instead focuses more on compression and decompression speed.

(3) Compression ratio is not a reliable predictor of throughput. Again, we observe how the compression ratio does not seem to predict throughput in every scenario. The compression ratio then should not be taken into consideration when selecting which compression algorithm to choose to obtain higher throughput.

(5) Throughput is similar to GZIP and ZLIB. Looking at previous observations, we noticed how throughput for Zstandard has remained almost unchanged with respect to ZLIB and GZIP. This is curious, as we expected that with much faster compression and decompression also came higher throughput. In fact, throughput for Zstandard runs can be a bit lower than ZLIB and GZIP.

5 CONCLUSIONS

In this paper, we have extended the observations made by Isenko et al. [13] with respect to the effects of compression algorithms in preprocessing pipelines. We ran the PRESTO profiler on a virtual machine with similar characteristics and on the same datasets, except Openwebtext and ImageNet, using Zstandard and different compression strategies for ZLIB. For ZLIB, we found that some compression strategies provide slightly better throughput compared to the `Z_DEFAULT_STRATEGY` and no compression, in particular when dealing with PNG images. Furthermore, some strategies execute



(a) Cube++ - JPG

(b) Cube++ - JPG

(c) Cube++ - PNG

(d) Cube++ - PNG

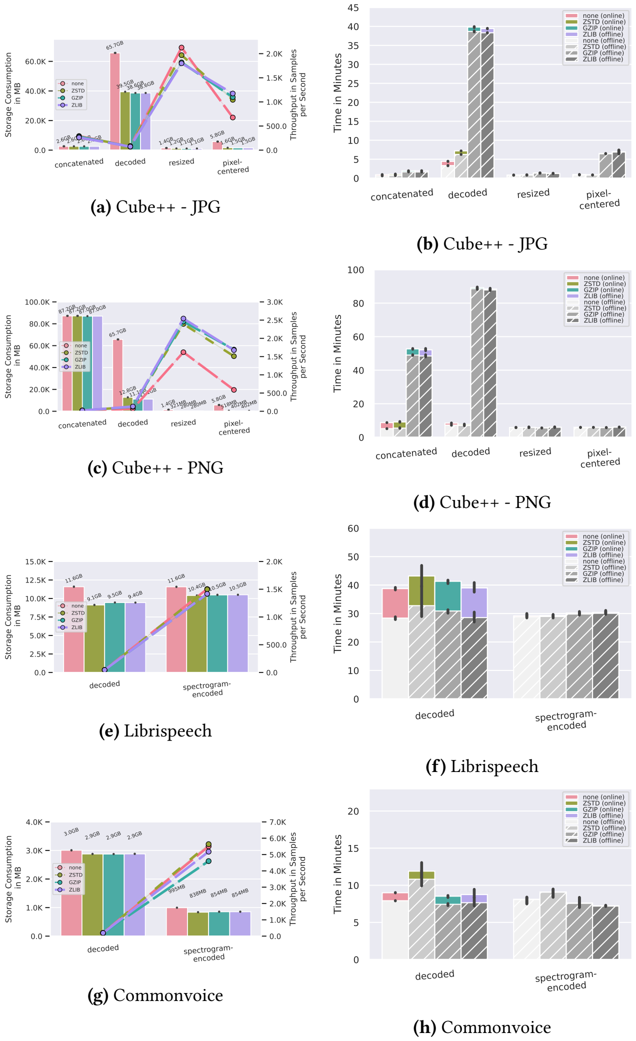
(e) Librispeech

(f) Librispeech

(g) Commonvoice

(h) Commonvoice

Figure 2: (ZLIB) **Left column**: storage consumption compared to throughput. **Right column**: offline (gray bars) and online processing time (colored bars) with compression.



(a) Cube++ - JPG

(b) Cube++ - JPG

(c) Cube++ - PNG

(d) Cube++ - PNG

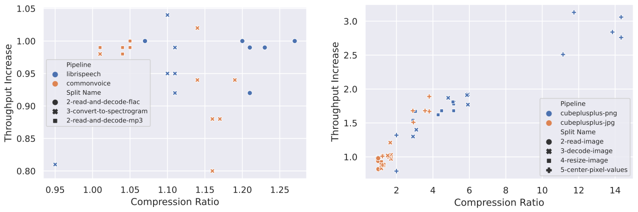
(e) Librispeech

(f) Librispeech

(g) Commonvoice

(h) Commonvoice

Figure 3: (ZSTD) **Left column**: storage consumption compared to throughput. **Right column**: offline (gray bars) and online processing time (colored bars) with compression.



(a) Librispeech and Commonvoice

(b) Cube++

Figure 4: Relation between compression ratio and throughput increase for Librispeech, Commonvoice and Cube++. Colors represent the dataset, shapes represent the name of the steps. We group the data according to the similarity between pipelines. Both Cube++ pipelines seem to have a linear relationship between compression ratio and throughput increase. Librespeech and Commonvoice do not show any particular pattern.

faster than both the default strategy and no compression at all under certain scenarios, with no reduction in throughput.

Zstandard performed significantly better in terms of execution time on Cube++, both JPG and PNG pipelines. However, we didn't find any particular improvement in throughput compared to other compression strategies such as ZLIB and GZIP, sometimes even showing a reduction in throughput. We suggest replacing the use of ZLIB or GZIP with Zstandard as a compression algorithm where the throughput is comparable, and Zstandard presents computational time overhead.

We observed that compression ratio seems to be positively correlated when dealing with Cube++, for both PNG and JPEG images, as shown by Figure 4, regardless of the compression algorithm of choice. On the other hand, throughput for Librispeech and Commonvoice does not seem to be affected by compression ratio in the same way as Cube++ pipelines. This effect is already documented by [13], pointing to CPU bottlenecks.

No compression algorithm is the same. Our work shows that introducing compression in data preprocessing pipelines must be done with care. As each pipeline and step have different properties, one must then select compression algorithms accordingly.

6 FUTURE WORK

In this work, we have decided to use the opinionated `TFRecord` data format for TensorFlow. However, data formats with multi-framework support already exist:

WebDataset [2] for example uses the old data archive format TAR. Although TAR supports different compression algorithms [8], their work does not make use of it. Instead, they support compression of single image files using the Pillow [9] package. In the future, we may want to replicate our findings using WebDataset by adding compression to TAR files. FFCV by Leclerc et al. [14] is a data loading system that has been recently published, and it has reported huge speedups in the preprocessing pipeline. The framework does not mention or support the use of compression. This lack of support opens up further research possibilities in this area, possibly further improving preprocessing pipeline speeds.

`DEFLATE`, the algorithm upon which both ZLIB and GZIP are based, was developed in 1996, and has remained virtually unchanged since then. Zstandard was developed in 2015 by Facebook with more modern hardware in mind and is able to exploit multithreading using its streaming API. Our implementation does not make use of the multithreading, and in future work, it may be possible to further modify our implementation to introduce it.

In this paper, we have observed how the presence of certain components in a compression algorithm such as string matching can affect both consumption speed and throughput. It might be beneficial to develop a compression algorithm that combines several of these components, turning them on or off depending on the data properties. For example, Burtscher and Ratanaworabhan [6] introduces gFPC, a self-tuning compression algorithm for floating-point data. When executing operations inside a GPU, swapping tensors can be an expensive task. Chen et al. [7] address this issue with CSWAP, dynamically choosing between four compression algorithms: zero-value compression (ZVC) [18], Run-Length Encoding (RLE) [19], Compressed Sparse Row (CSR) [3] and LZ4.

Using different lossless data formats can help reduce storage consumption footprint and speed up the preprocessing pipeline. The Quite OK (QOI) [22] data format has recently received a lot of attention in this regard, claiming 20x-50x faster encoding and 3x-4x faster decoding with respect to PNG, while offering a similar compression ratio. It can also be paired with other compression algorithms to achieve possibly achieve a higher compression ratio for limited storage scenarios. Other attempts to have dedicated data formats exist, such as DeepN-JPEG Liu et al. [16], a format based on JPEG but dedicated to deep learning environments.

Until now, we have only taken a look at lossless compression algorithms. However, there might be even more advantages in using lossy compression algorithms instead, without necessarily sacrificing accuracy or training time. ZFP [15] is a lossy compression algorithm for n-dimensional floating-point data, with options for error-bounding the compression. In general, lossy compression algorithms may be used to reduce storage consumption and have been shown to maintain similar accuracy scores [8, 16].

REFERENCES

[1] 2000. New zlib strategy proposal: Z_RLE. http://optipng.sourceforge.net/pngtech/z_rle.html.

[2] Alex Aizman, Gavin Maltby, and Thomas M. Breuel. 2020. High Performance I/O For Large Scale Deep Learning. *CoRR* abs/2001.01858 (2020). arXiv:2001.01858 http://arxiv.org/abs/2001.01858

[3] Ashwaq Alabaichi, Amjad Alhusiny, and Elham Alsaadi. 2017. A Novel Compressing a Sparse Matrix using Folding Technique. *Research Journal of Applied Sciences, Engineering and Technolog* 14 (08 2017), 310–319. https://doi.org/10.19026/rjaset.14.4955

[4] Rosana Ardila, Megan Branson, Kelly Davis, Michael Henretty, Michael Kohler, Josh Meyer, Reuben Morais, Lindsay Saunders, Francis M. Tyers, and Gregor Weber. 2019. Common Voice: A Massively-Multilingual Speech Corpus. *CoRR* abs/1912.06670 (2019). arXiv:1912.06670 http://arxiv.org/abs/1912.06670

[5] Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. 2020. What is the State of Neural Network Pruning? *CoRR* abs/2003.03033 (2020). arXiv:2003.03033 https://arxiv.org/abs/2003.03033

[6] Martin Burtscher and Paruj Ratanaworabhan. 2010. gFPC: A Self-Tuning Compression Algorithm. In *2010 Data Compression Conference*. 396–405. https://doi.org/10.1109/DCC.2010.42

[7] Ping Chen, Shuibing He, Xuechen Zhang, Shuaiben Chen, Peiyi Hong, Yanlong Yin, Xian-He Sun, and Gang Chen. 2021. CSWAP: A Self-Tuning Compression Framework for Accelerating Tensor Swapping in GPUs. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 271–282. https://doi.org/10.1109/Cluster48925.2021.00019

[8] gzip, bzip2, xz, lzip, lzma, lzop, zstd

[9] https://pypi.org/project/Pillow/

[8] Samuel F. Dodge and Lina J. Karam. 2016. Understanding How Image Quality Affects Deep Neural Networks. *CoRR* abs/1604.04004 (2016). arXiv:1604.04004 http://arxiv.org/abs/1604.04004

[9] Egor Ershov, Alexander Belokopytov, and Alex Savchik. 2020. Problems of dataset creation for light source estimation. *arXiv preprint arXiv:2006.02692* (2020).

[10] Egor Ershov, Alexey Savchik, Illya Semenkov, Nikola Banić, Alexander Belokopytov, Daria Senshina, Karlo Koščević, Marko Subašić, and Sven Lončarić. 2020. The Cube++ Illumination Estimation Dataset. *IEEE Access* 8 (2020), 227511–227527.

[11] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv:arXiv:1510.00149

[12] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 241 (2021), 1–124. http://jmlr.org/papers/v22/21-0366.html

[13] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. 2022. Where Is My Training Bottleneck? Hidden Trade-Offs in Deep Learning Preprocessing Pipelines. (2022). https://doi.org/10.1145/3514221.3517848 arXiv:arXiv:2202.08679

[14] Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Madry. 2022. ffcv. https://github.com/libffcv/ffcv/.

[15] Peter Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (12 2014). https://doi.org/10.1109/TVCG.2014.2346458

[16] Zihao Liu, Tao Liu, Wujie Wen, Lei Jiang, Jie Xu, Yanzhi Wang, and Gang Quan. 2018. DeepN-JPEG: A Deep Neural Network Favorable JPEG-based Image Compression Framework. *CoRR* abs/1803.05788 (2018). arXiv:1803.05788 http://arxiv.org/abs/1803.05788

[17] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5206–5210. https://doi.org/10.1109/ICASSP.2015.7178964

[18] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W. Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 78–91. https://doi.org/10.1109/HPCA.2018.00017

[19] A.H. Robinson and C. Cherry. 1967. Results of a prototype television bandwidth compression scheme. *Proc. IEEE* 55, 3 (1967), 356–364. https://doi.org/10.1109/PROC.1967.5493

[20] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[21] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zheng, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *CoRR* abs/2201.11990 (2022). arXiv:2201.11990 https://arxiv.org/abs/2201.11990

[22] Dominic Szablewski. 2021. QOI — The Quite OK Image Format. https://qoiformat.org/.

[23] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*.

[24] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. 2639–2652. https://doi.org/10.1145/3448016.3457566