DZip: improved general-purpose lossless compression based on novel neural network modeling

Mohit Goyal[γ], Kedar Tatwawadi[*], Shubham Chandak[*] and Idoia Ochoa[γ,†]

[γ]Electrical and Computer Engineering, University of Illinois, Urbana, IL, USA
[*]Department of Electrical Engineering, Stanford University, CA, USA
[†] Department of Electrical Engineering, University of Navarra, Spain
mohit@illinois.edu

Abstract

We consider lossless compression based on statistical data modeling followed by prediction-based encoding, where an accurate statistical model for the input data leads to substantial improvements in compression. We propose DZip, a general-purpose compressor for sequential data that exploits the well-known modeling capabilities of neural networks (NNs) for prediction, followed by arithmetic coding. DZip uses a novel hybrid architecture based on adaptive and semi-adaptive training. Unlike most NN-based compressors, DZip does not require additional training data and is not restricted to specific data types. The proposed compressor outperforms general-purpose compressors such as Gzip (29% size reduction on average) and 7zip (12% size reduction on average) on a variety of real datasets, achieves near-optimal compression on synthetic datasets, and performs close to specialized compressors for large sequence lengths, without any human input. While the main limitation of NN-based compressors is generally the encoding/decoding speed, we empirically demonstrate that DZip achieves comparable compression ratio to other NN-based compressors while being several times faster. The source code for DZip and links to the datasets are available at https://github.com/mohit1997/Dzip-torch.

1 Introduction

There has been a tremendous surge in the amount of data generated in the past years. Along with image and textual data, new types of data such as genomic, 3D VR, and point cloud data are being generated at a rapid pace [1, 2]. Thus, data compression is critical for reducing the storage and transmission costs associated with these data, and has been studied extensively from both theoretical and practical standpoints. In particular, a wide class of (lossless) compressors utilize the "prediction + entropy coding" approach, wherein a statistical model generates predictions for the upcoming symbols given the past and an entropy coder (e.g., arithmetic coder [3]) uses the predicted probabilities to perform compression. In this general framework, a better prediction model directly induces a better compressor.

In past few years, the interest in neural networks (NN) based compression has been growing due to their exceptional performance on several modeling and prediction tasks (e.g., language modeling [4]). NN models can typically learn highly complex patterns in the data much better than traditional finite context and Markov models, leading to significantly lower prediction errors. This has led to the development of several compressors using neural networks as predictors. However, many existing NN based compression methods have been tailored for compression of certain data types (e.g.,

2021 Data Compression Conference (DCC) | 978-1-6654-0333-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/DCC50243.2021.00023

2375-0359/21/$31.00 ©2021 IEEE
DOI 10.1109/DCC50243.2021.00023

text [5] and images [6]), where the prediction model is trained on separate training data of a specific data type. This approach becomes inapplicable in the absence of existing training data and domain knowledge, and thus is not appropriate for general-purpose compression. Nevertheless, there are a few NN-based compressors that serve as general-purpose compressors, such as CMIX (`https://github.com/byronknoll/cmix`) and the recently proposed LSTM-Compress (`https://github.com/byronknoll/lstm-compress`) and NNCP (`https://bellard.org/nncp/`). In parallel to the work on compression, there has been significant progress in language modeling (e.g., BERT [4]). In principle, these can be used for compression leading to significant improvements over the state-of-the-art compressors. However, the NN model is typically quite large and needs vast amounts of data for training, limiting their direct applicability to general-purpose compression.

In this work, we propose a general-purpose lossless compressor for sequential data, DZip, that relies on neural network based modeling. DZip treats the input file as a byte stream and does not require any additional training datasets. Hence, DZip is a standalone compressor capable of compressing any dataset (regardless of the alphabet size), unlike most existing neural network based compressors. We use a novel hybrid training approach which is ideally suited for such a setting and combines elements of adaptive and semi-adaptive modeling.

We evaluate DZip on datasets from several domains including text, genomics and scientific floating point datasets, and show that it achieves on average 29% improvement over Gzip (`https://www.gzip.org/`), 33% improvement over LSTM-Compress, 12% improvement over 7zip (`https://www.7-zip.org/`), and 8% improvement over BSC (`http://libbsc.com/`), reducing the gap between general-purpose and specialized compressors. In comparison to state-of-the-art NN-based compressors CMIX and NNCP, we demonstrate that DZip can achieve similar performance on most datasets of sufficiently large length (more than 60 million symbols) while being 3-4 times faster than CMIX and 4 times faster than NNCP in encoding speed. We note that, in contrast to traditional compressors, NN-based compressors suffer from slower encoding and decoding speeds due to the computationally intensive nature of neural networks.

Our results also indicate that for some datasets, the performance of DZip is close to that of specialized compressors, which are highly optimized for the specific data types. In addition, our evaluation on certain synthetic datasets of known entropy highlights the ability of the proposed compressor to learn long-term patterns better than the other general-purpose compressors. DZip is available as an open source tool at `https://github.com/mohit1997/Dzip-torch`, also providing a framework to experiment with several neural network models and training methodologies. We note that DZip is an extension of DeepZip [7][8] (which is similar to the bootstrap only mode discussed in Section 3). An older version of DZip (without GPU based encoding) was presented as a poster at DCC 2020 [9].

2 Background

Consider a data stream $S^N = \{S_1, S_2, \ldots, S_N\}$ over an alphabet $\mathcal{S}$ which we want to compress losslessly. We consider the statistical coding approach consisting of a prediction model followed by an arithmetic coder. For a sequence S^N, the aim of

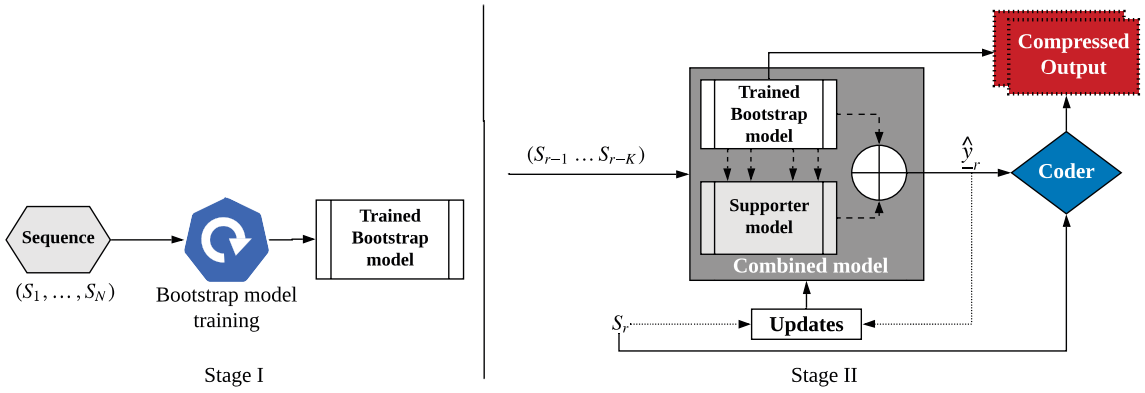

Figure 1: DZip compression overview: In Stage I, the boostrap model is trained by scanning the sequence multiple times. In Stage II, the bootstrap model is combined with the supporter model to predict the conditional probability of the current symbol given the past K symbols ($K = 64$ by default). The current symbol and the predicted probabilities are then fed into the arithmetic coder followed by updates to combined model. The final compressed output consists of the trained bootstrap model and the arithmetic coder's output.

the model is to estimate the conditional probability distribution of the r^{th} symbol S_r based on the previously observed K symbols, denoted as $P(S_r|S_{r-1}, \ldots, S_{r-K})$, where K is a hyperparameter. An estimate of this probability and S_r are then fed into the arithmetic encoding block which recursively updates its state. This state serves for the compressed representation at the end of this process. The compressed size using this approach is equivalent to the cross entropy (CE) loss shown in Eq. 1, where $|\mathcal{S}|$ is the alphabet size, $\underline{y}_r$, $\underline{\hat{y}}_r$ (vectors of size $|\mathcal{S}|$) are the one-hot encoded ground truth and the predicted probabilities, respectively, and N is the sequence length.

$$\mathcal{L} = \sum_{r=1}^{N} CE(\underline{y}_r, \underline{\hat{y}}_r) = \sum_{r=1}^{N} \sum_{k=1}^{|\mathcal{S}|} y_{rk} \log_2 \frac{1}{\hat{y}_{rk}} \tag{1}$$

The model that estimates the probability $P(S_r|S_{r-1}, \ldots, S_{r-K})$, where $r \in \{K+1, \ldots, N\}$, should be trained to minimize the cross entropy loss on the data to be compressed. This training can be performed in three ways [10] as discussed below:

1. Static: Here the model is first trained on some external training data and is made available to both the compressor and the decompressor. This approach requires access to similar training data and is not directly applicable to general-purpose compression.

2. Adaptive: In this case, both the compressor and the decompressor are initialized with the same random model. This model is updated adaptively based on the sequence seen up to some point and does not require additional training data. For complex models, this approach may pose difficulties in training the model in a single pass and adapting to changing statistics (e.g., for non-stationary data).

3. Semi-adaptive: Here the model is first trained on the input sequence and can involve multiple passes through the input data. The trained model parameters are saved as a part of the compressed file, along with the arithmetic coding output, as they need to be used for decompression. The additional cost is expected to be compensated by the better quality of predictions made by the trained model, which

would result in a smaller arithmetic coding output. There is however a trade-off, since a larger model can lead to better compression but the gains might be offset by the size of the model itself, particularly for smaller datasets.

3 Methods

The proposed compressor DZip utilizes a hybrid training scheme that combines semi-adaptive and adaptive training approaches by means of two models, a *bootstrap model* and a *supporter model*, as shown in Figure 1. The bootstrap model is a parameter efficient recurrent neural network (RNN) based model that is trained in a *semi-adaptive* fashion by performing multiple passes on the sequence to be compressed (prior to compression). To further improve the compression, we use a larger randomly initialized supporter model which is combined with the bootstrap model. This *combined* model is updated in an *adaptive* manner during encoding (symmetrically during decoding) and generates the final predictions used for compression. Due to the use of adaptive training, the supporter model parameters do not need to be stored as part of the compressed file.

3.1 Model architecture

1. Bootstrap model: Its architecture is designed keeping in consideration the trade-off between model size and prediction capability, leading to the choice of an RNN-based design with parameter-sharing across time steps. The model is as shown in the top half of Figure 2 and consists of an embedding and two biGRU layers (bidirectional gated recurrent units [11]) followed by linear and dense (fully connected) layers. The output of every m^{th} time step after the biGRU layers is stacked and flattened out into a vector ($m = 16$ by default). Choosing only every m^{th} output helps in reducing the number of parameters in the next layer while still allowing the network to learn long-term dependencies. The output of the dense layer (with ReLU activation) and the flattened vector are added together after linear layers to generate the unscaled probabilities (logits) denoted as $logits_b$ (of dimension equal to the vocabulary size). This dense layer is important for learning long-term relationships in the inputs and showed improved performance on synthetic datasets. The layer widths of the bootstrap model are automatically chosen depending on the vocabulary size of the input sequence, since a higher vocabulary size demands larger input and output sizes. As the vocabulary size varies, the embeddings' dimensionality varies from 8 to 16; hidden state for biGRU varies from 8 to 128; and the dense layer's width (prior to logits) varies from 16 to 256. The above hyperparameters were chosen empirically based on experimental results.

2. Supporter model: The supporter model architecture is designed to adapt quickly and provide better probability estimates than the bootstrap model, without any constraints on the model size itself. The input to this model consists of the embeddings and the intermediate representations from the bootstrap model (see Figure 2). The supporter model consists of three sub-NNs which act as independent predictors of varying complexity. The first sub-NN is linear, the second sub-NN has two dense layers and the third sub-NN uses residual blocks [12] for learning more complex pat-

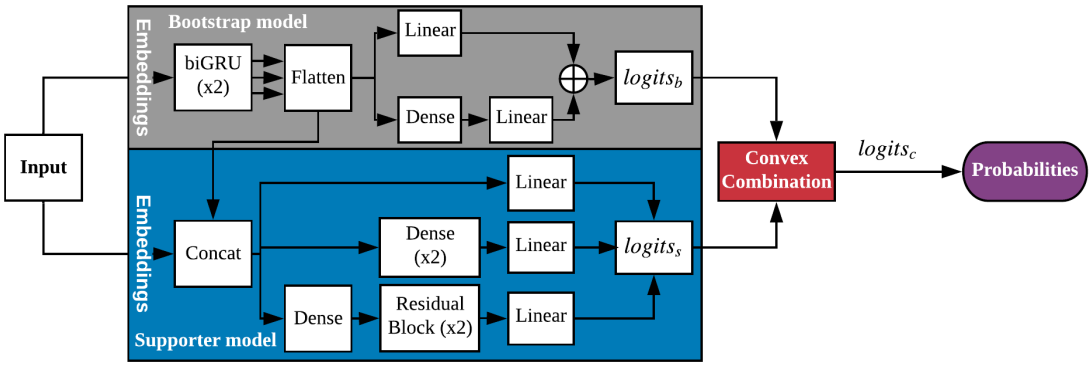

Figure 2: Combined model architecture consisting of bootstrap and supporter models. Dense layers correspond to fully connected layers with ReLU activation. Linear layers are also fully connected layers but do not incorporate a non linear transformation. Concat block denotes concatenation of all the input vectors.

terns [4]. We employ ReLU activation in all dense layers and the residual blocks. Then, each of the output vectors from these sub-NNs are linearly downsized into a vector of dimensionality equal to the vocabulary size and added together. The result of this operation is interpreted as the logits for the supporter model, denoted as $logits_s$. Based on empirical evaluation, the widths for the dense and residual layers are automatically set to 1024 or 2048 depending on the vocabulary size.

3. Combined model: The combined model takes the logits from the bootstrap model ($logits_b$) and the output logits from the supporter model ($logits_s$) to generate the final logits ($logits_c$) through a convex sum as shown below.

$$logits_c = \lambda * logits_b + (1 - \lambda) * logits_s \text{ , s.t. } \lambda \in [0, 1],$$

where λ is a learnable parameter (restriced to $[0, 1]$ through sigmoid activation). This allows the combined model to weigh the logits from the two models appropriately [13]. Since the prediction of supporter model can be expected to be poor initially, this combination allows the model to give more weight to $logits_b$ as compared to $logits_s$. The final output $logits_c$ is scaled to probabilities through softmax activation, and then input to the arithmetic coding.

3.2 Model training

The first stage of DZip involves reading the input file byte-by-byte and, based on the vocabulary size[1], automatically selecting the hyperparameters for the bootstrap and supporter models. The second stage consists of training the bootstrap model by performing multiple passes through the sequence. The model is trained for 8 epochs with a batch size of 2048, gradient clipping, context length $K = 64$ and Adam optimizer [14] (learning rate 0.005, $\beta_1 = 0.9$ and $\beta_2 = 0.999$) with learning rate decay, while minimizing categorical cross entropy loss (Eq. 1). Once the training is finalized, this model is saved as part of the compressed file after being losslessly compressed with general-purpose compressor BSC. DZip can then be used in two modes which trade-off compression ratio with encoding/decoding speed.

[1]In this case, the maximum vocabulary size can be 256 which would correspond to byte symbols.

Combined mode (hybrid): Here, the prediction is done using the combined model, where the trained bootstrap model serves as a prior. The supporter model parameters are pseudo-randomly initialized whereas the bootstrap model parameters remain fixed. For fast encoding, we divide the sequence into 64 equally sized parts, and predictions for all parts are generated simultaneously in a batch. After encoding symbol $\underline{y}$ (a one-hot vector), the combined model parameters are optimized by minimizing the following loss,

$$\mathcal{L}_{com} = CE(\underline{y}, f_s(logits_c)) + CE(\underline{y}, f_s(logits_s)),$$

where f_s denotes the softmax activation, and CE is the cross entropy loss defined earlier. The second term in this loss function forces the supporter model to learn even if the $logits_c$ are assigning more weight to the $logits_b$. The weight updates are performed after encoding/decoding every 20 symbols (per part) with a learning rate of 0.0005 using Adam optimizer with $\beta_1 = 0$ and $\beta_2 = 0.999$ to quickly adapt to the non-stationary sequence statistics. This is the default mode of DZip.

Bootstrap only mode: The sequence is divided into 1024 parts and the first K symbols of each part are encoded using uniform probabilities. Then we encode each part, starting from the $(K+1)^{\text{th}}$, symbol using the probability estimates from the bootstrap model, where the prediction for each part is done in a single batch (with a batch size of 1024). This procedure is repeated until all parts are successfully encoded.

Reproducibility: Since DZip utilizes a GPU to reduce runtime, its current implementation requires encoding and decoding to be performed on the same hardware (a limitation of the PyTorch library).[2] DZip can be fairly easily adapted to the appropriate deep learning framework once reproducibility across GPUs becomes available.

4 Experiments

We benchmark the performance of our neural network based compressor DZip on real and synthetic datasets, and compare it with state-of-the-art general-purpose compressors Gzip, BSC (BWT-based compressor), 7zip, and ZPAQ (`http://mattmahoney.net/dc/zpaq.html`), as well as with RNN-based compressors LSTM-Compress, NNCP and CMIX. ZPAQ is a general purpose compressor which is specialized for text data, where it achieves better performance. LSTM-Compress uses an LSTM (Long Short Term Memory Cells) model to adaptively learn the source distribution while encoding with an arithmetic coder. CMIX, the current state-of-the-art NN-based general-purpose lossless compressor, uses several thousand context models followed by an LSTM byte level mixer (to combine predictions) and a bit level NN-based context mixer. The context models and the mixers are then trained through backpropagation adaptively while encoding the input data. CMIX is specialized for text and executable data. NNCP is also an LSTM-based compressor which adaptively compresses the input sequence while simultaneously updating the weights of the RNN. NNCP uses seven stacked LSTM layers which incorporate feature normalisation layers, further adding to the overall runtime. Moreover, the compressor only supports

[2]See `https://pytorch.org/docs/stable/notes/randomness.html`. Nevertheless, the DZip framework also supports CPU based encoding and decoding.

Table 1: Bits per character (bpc) achieved by the tested compressors on the real datasets. Best results are boldfaced. $\log_2 |\mathcal{S}|$ represents the bpc achieved assuming an independent uniform distribution over the alphabet of size $|\mathcal{S}|$. For DZip, we specify the total bpc and the size of the model (in % space occupied). Spec. Comp. stands for specialized compressor.

File	Len/$\log_2\|\mathcal{S}\|$	Gzip	BSC	7zip	ZPAQ	LSTM Compress	NNCP	CMIX	DZip		Spec. Comp.
									bpc	Model	
webster	41.1M/6.61	2.32	1.29	1.70	1.09	1.23	0.98	**0.83**	1.44	31.33%	0.83
mozilla	51.2M/8.00	2.97	2.52	2.11	1.88	2.05	1.63	**1.39**	2.15	25.37%	1.39
h. chr20	64.4M/2.32	2.05	1.73	1.77	1.68	7.82	1.66	**1.62**	1.63	0.92%	1.62
h. chr1	100M/2.32	2.14	1.78	1.83	1.74	7.36	**1.67**	**1.67**	**1.67**	0.58%	1.65
c.e. genome	100M/2.00	2.15	1.87	1.89	1.80	7.51	1.80	**1.74**	1.81	0.53%	1.72
ill-quality	100M/2.00	0.50	0.35	0.35	0.34	6.48	0.34	**0.33**	0.34	2.79%	0.51
text8	100M/4.75	2.64	1.68	1.93	1.52	1.76	1.48	**1.31**	1.74	9.38%	1.31
np-bases	300M/2.32	2.16	1.86	1.93	1.79	7.34	**1.70**	1.73	1.73	0.19%	1.75
np-quality	300M/6.51	5.95	5.69	5.71	5.53	5.51	5.50	**5.49**	5.56	1.13%	5.35
enwiki9	500M/7.69	2.72	1.64	1.94	1.43	1.66	1.21	**1.05**	1.47	3.67%	1.05
num-control	159.5M/8.00	7.57	7.66	7.41	6.96	6.82	6.72	**6.63**	6.83	2.67%	7.12
obs-spitzer	198.2M/8.00	6.50	2.51	2.27	2.20	2.87	1.73	**1.58**	2.18	6.70%	7.74
msg-bt	266.4M/8.00	7.08	6.96	5.76	6.29	6.22	5.36	5.24	**5.21**	2.08%	6.67
audio	264.6M/8.00	5.75	4.63	4.98	4.17	4.92	3.49	3.44	**3.40**	3.29%	N/A

CPU based training and inference, resulting in extremely slow encoding speed. We also provide a comparison with specialized compressors for the real datasets when available. Unless otherwise stated, all results corresponding to DZip are obtained using the combined model (default setting). DZip results are reported on a 16 GB NVIDIA Tesla P100 GPU. Gzip, BSC, 7zip, ZPAQ, LSTM-Compress, NNCP (on 8 cores) and CMIX results are reported on Intel Xeon Gold 6146 CPUs.

Datasets: We consider a wide variety of real datasets with different alphabet sizes and sequence lengths, including genomic data (*h. chr1, h. chr20, c.e. genome, np-bases, np-quality, ill-quality*), text (*webster*, *text8, enwiki9*), executable files (*mozilla*), double precision floating point data (*num-control, obs-spitzer, msg-bt*), and audio data (*audio*). Furthermore, we also test it on synthetic datasets with known entropy rate and increasing complexity. Dataset sources and descriptions are provided at `https://github.com/mohit1997/Dzip-torch/blob/master/Datasets.md`.

Results on real data: We first analyze the performance of DZip on the real datasets (Table 1). On each dataset, we include results for specialized compressors (when available) as their performance serve as a baseline for achievable compression. In particular, we use CMIX for *webster, mozilla, text8*, and *enwiki9*, GeCo [15] for *h. chr20, h. chr1, c.e. genome*, and *np-bases*[3], DualCtx [16] for *np-quality*[4], QVZ [17] for *ill-quality*[5], and FPC [18] for *msg-bt, num-control*, and *obs-spitzer* datasets. Since ZPAQ is also specialized for text data, we discuss its comparison with DZip on *webster, text8* and *enwiki9*. We do not consider any specialized compressors for *audio* since we generate the binary audio file by concatenating multiple audio files together.

When compared against traditional compressors Gzip, BSC and 7zip, DZip offers the best compression performance for all datasets except for *webster* and *mozilla*, in which BSC or 7zip perform better than DZip. Specifically, DZip offers on average

[3]GeCo is not optimized for nanopore genomic read bases (*np-bases*).

[4]Unlike DZip, DualCtx uses read bases as an additional context for quality value compression.

[5]QVZ is optimized for lossy compression, but also provides a mode for lossless compression.

about 29% improvement over Gzip, 12% improvement over 7zip and 9.4% improvement over BSC. ZPAQ, which is specialized for text datasets, consistently outperforms DZip on *webster, mozilla, text8* and *enwiki9*. Note however that the gap decreases with the increase in length of the sequence, as the overhead of storing the bootstrap model is reduced. On other datasets, DZip always outperforms ZPAQ (except for *np-quality* and *c.e. genome* where the performance is similar).

With respect to NN-based compressors, we observe that DZip consistently outperforms LSTM-Compress (33% improvement) on datasets of length larger than 60 million. In comparison to NNCP and CMIX, DZip performs either worse or similar on most datasets. Specifically, NNCP performs better than DZip on the text and executable datasets with roughly 20% improvement, while the performance on the remaining datasets is near-identical. Finally, CMIX was observed to perform consistently better than all other compressors on all datasets except *msg-bt* and *audio*, where DZip performs slightly better. Since CMIX is specialized for text and executable files, the bpc is significantly better for *webster, mozilla, text8* and *enwiki9* datasets. On genomic datasets, CMIX provides on average approximately 3-4% improvement over DZip. Note that CMIX is expected to outperform DZip, as it uses several thousand context models specialized for specific data types. In contrast, DZip uses a single model and does not incorporate any data type specific knowledge.

With respect to specialized compressors, we observe that they outperform DZip in all cases, except for the genomic files and scientific floating point datasets where DZip offers improved performance. This may be explained because QVZ, GeCo and FPC are not optimized for lossless compression of quality data, nanopore bases and scientific floating point data, respectively.

Finally, it is important to note that the performance of DZip is sensitive to the alphabet size and the sequence length because of the overhead associated with storing the bootstrap model parameters. Specifically, for small length and large alphabet sequences such as *webster* and *mozilla* datasets, the bootstrap model size occupies 31% and 25% of the compressed file size, respectively, hurting the overall compression ratio. However, as the sequence length increases, this model size contribution gets amortized. For example, on *np-bases, np-quality, msg-bt* and *audio* datasets, DZip is able to achieve a performance close to that of NNCP and CMIX.

Results on synthetic data: We also evaluate DZip on two sequence classes with known entropy rates, *XOR-k* (entropy rate 0) and *HMM-k* (entropy rate 0.469), where k represents the memory of the sequence. We consider values of k ranging from 20 to 70. We observe that DZip achieves the best compression performance in all cases, almost achieving the entropy rate of the corresponding sequences when $k < 70$, with slight overhead due to the bootstrap model size. Since DZip uses 64 previous symbols for prediction, it becomes impossible to learn dependencies beyond this range. Note that this hyperparameter can be changed at the cost of increased encoding/decoding runtime. On the contrary, Gzip, BSC, ZPAQ, 7zip, LSTM-Compress, NNCP and CMIX fail to achieve meaningful compression, except for $k = 20$ in which case BSC and ZPAQ are able to capture the dependency to some extent.[6]

[6]Results not shown due to space constraints.

Table 2: Compression in bpc obtained by (i) only the bootstrap model and (ii) DZip (combined model). Improv. stands for the improvement of the combined model with respect to the bootstrap model (in bpc).

File	*webster*	*mozilla*	*h. chr1*	*text8*	*np-bases*	*np-quality*	*enwiki9*	*obs-spitzer*	*msg-bt*	*audio*
Length	41.1M	51.2M	100M	100M	300M	300M	500M	198.2M	266.4M	264.6M
Bootstrap Only	1.474	2.233	1.720	1.789	1.755	5.588	1.596	2.445	5.259	3.405
DZip	1.443	2.150	1.673	1.737	1.725	5.562	1.470	2.181	5.214	3.389
Improv.	0.031	0.083	0.047	0.052	0.03	0.026	0.126	0.264	0.045	0.016

Trade-off between "bootstrap only" and combined modeling approaches: As an ablation study, we compare the two compression modes: (i) using only the trained bootstrap model and (ii) using the combined DZip hybrid model with adaptive training (default setting) on selected real datasets (Table 2). On average, we observe that using the proposed combined model improves the compression by 0.072 bpc on the real datasets. DZip in bootstrap only mode still outperforms Gzip, 7zip, BSC and LSTM-Compress on most of the selected datasets, while being more practical due to its reduced running time. For example, we observe 27% improvement with respect to Gzip, 31% improvement with respect to LSTM-Compress, 9% improvement with respect to 7zip, and 6% improvement over BSC. Hence, depending on speed and performance requirements, one mode may be preferred over the other.

Computational requirements: The first stage of DZip, which consists on training the bootstrap model, requires 2-5 minutes/MB (depending on the alphabet size). In the bootstrap only/combined mode, the encoding and decoding typically take 0.4/2.5 minutes per MB and 0.7/2.8 minutes per MB, respectively. DZip outperforms other NN-based compressors in terms of computational performance because it relies on simpler models and it uses GPU along with various parallelization schemes adopted during both training and encoding. LSTM-Compress is 2/3 times faster in encoding speed and 5/1.3 times slower in decoding speed than DZip in bootstrap only/combined mode. Compared to NNCP, DZip compresses 5-6/3-4 times faster and decompresses 60/15 times faster in bootstrap only/combined mode. We also implemented NNCP compressor on GPU and observed that NNCP still encodes 2/4 times slower and decodes 20/15 times slower as compared to bootstrap only/combined mode for DZip. CMIX uses specific preprocessing transformations based on the data type and has variable encoding/decoding speed. On average, DZip compresses more than 4/5 times and decompresses 25/10 times faster than CMIX in bootstrap only/combined mode. In comparison, Gzip, 7zip, BSC and ZPAQ take just a few seconds per MB for compression and less than a second per MB (except ZPAQ which employs a few seconds) for decompression. The difference in compression speeds is expected since training and inference for NNs are expensive, but they can provide better compression rates due to superior modeling capabilities.

5 Conclusion

In this work, we introduce a general-purpose neural network prediction based framework for lossless compression of sequential data. The proposed compressor DZip uses a novel NN-based hybrid modeling approach that combines semi-adaptive and adap-

tive modeling. We show that DZip achieves improvements over Gzip, 7zip, BSC, ZPAQ and LSTM-Compress for a variety of real datasets and near optimal compression for synthetic datasets. DZip also compares favorably with the state-of-the-art NN-based compressors CMIX and NNCP, achieving similar compression while being substantially faster. Although the practicality of DZip is currently limited due to the required encoding/decoding time, we believe the proposed framework and experiments can shed light into the potential of neural networks for compression, as well as serve to better understand the neural network models themselves.

References

[1] Adam Auton et al., "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, 2015.

[2] I. Armeni et al., "Joint 2D-3D-Semantic Data for Indoor Scene Understanding," *ArXiv e-prints*, Feb. 2017.

[3] Ian H. Witten et al., "Arithmetic Coding for Data Compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, June 1987.

[4] Jacob Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[5] Q. Liu, Y. Xu, and Z. Li, "DecMac: A Deep Context Model for High Efficiency Arithmetic Coding," in *2019 (ICAIIC)*, Feb 2019, pp. 438–443.

[6] Friso H. Kingma, Pieter Abbeel, and Jonathan Ho, "Bit-swap: Recursive bits-back coding for lossless compression with hierarchical latent variables," *CoRR*, vol. abs/1905.06845, 2019.

[7] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa, "Deepzip: Lossless data compression using recurrent neural networks," *arXiv preprint arXiv:1811.08162*, 2018.

[8] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa, "Deepzip: Lossless data compression using recurrent neural networks," in *2019 Data Compression Conference (DCC)*, 2019, pp. 575–575.

[9] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa, "Dzip: Improved general-purpose lossless compression based on novel neural network modeling," in *2020 Data Compression Conference (DCC)*, 2020, pp. 372–372.

[10] Timothy Bell et al., "Modeling for Text Compression.," *ACM Comput. Surv.*, vol. 21, pp. 557–591, 12 1989.

[11] K. Cho et al., "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *2014 (EMNLP)*, Doha, Qatar, Oct. 2014, pp. 1724–1734.

[12] Kaiming He et al., "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[13] Matthew V. Mahoney, "Adaptive weighing of context models for lossless data compression," in *Tech. Rep. CS-2005-16*, Florida Tech., 2005.

[14] Diederik P. Kingma and Jimmy Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[15] Diogo Pratas et al., "Efficient compression of genomic sequences," in *2016 Data Compression Conference (DCC)*. IEEE, 2016, pp. 231–240.

[16] Guillermo Dufort y Álvarez et al., "Compression of Nanopore FASTQ Files," in *Inter. Work-Conference on Bioinformatics and Biomedical Engineering*. Springer, 2019.

[17] Greg Malysa et al., "QVZ: lossy compression of quality values," *Bioinformatics*, vol. 31, no. 19, pp. 3122–3129, 2015.

[18] M. Burtscher and P. Ratanaworabhan, "FPC: A High-Speed Compressor for Double-Precision Floating-Point Data," *IEEE Transactions on Computers*, vol. 58, 2009.