DRAC: a delta recurrent neural network-based arithmetic coding algorithm for edge computing

Bowei Shan¹ · Yong Fang¹

Received: 30 March 2021 / Accepted: 23 June 2021 / Published online: 5 July 2021
© The Author(s) 2021

Abstract

This paper develops an arithmetic coding algorithm based on delta recurrent neural network for edge computing devices called DRAC. Our algorithm is implemented on a Xilinx Zynq 7000 Soc board. We evaluate DRAC with four datasets and compare it with the state-of-the-art compressor DeepZip. The experimental results show that DRAC outperforms DeepZip and achieves 5X speedup ratio and 20X power consumption saving.

Keywords Arithmetic coding · Delta recurrent neural network · Edge computing

Introduction

Due to ubiquitous mobile phones, Internet of Things (IoT) devices, and wireless sensor networks, the size and number of data collected by them has grown exponentially in recent years. Big data era generates many challenges for data process and storage, and the core problem is how to effectively compress many kinds of data, *e.g.*, text, image, video, genomic data, and 3D VR data. In this paper, we only consider the *lossless data compression*, where the reconstructed message is identical to the original message. In Shannon's theory, the minimum compression rate that can be reached by the lossless compression is the entropy of information. Under lossless compression paradigm, a lot of algorithms [1] have been invented, *e.g.*, Huffman coding, and LZ77/78. While, Arithmetic Coding (AC) [2] is the state-of-the-art entropy coding algorithm in terms of compression rate.

AC codec consists of two parts: a probability estimator, which estimates each symbols probability; an AC model, in which encoder repeatedly maps input symbol sequence into a subinterval between the unit interval $[0,1)$ based on the cumulative probability of symbols; and decoder use the code (a real number between interval $[0,1)$) to output the reconstructed symbol sequence based on the cumulative probability of symbols. An accurate estimator is the key for the AC codec.

With the great success of deep learning in recent years, Recurrent Neural Networks (RNN), which is suitable to model the temporal dynamic behavior, has also been considered in the design of AC-related algorithm. [3] combined character-level predictive RNN with statistical coding techniques for text compression. [4] used RNN as a context mixer to mix the opinions of the multiple compressors, and achieved better compression performance. CMIX [5] improved [4] using an Long Short-Term Memory (LSTM) context mixer to mix a total of 2,122 independent models. Recently, [6] presented a DeepZip, which combined RNN predictors with an arithmetic coder and compress a variety of text and genomic datasets losslessly. DeepZip outperforms Gzip on the real and synthetic datasets.

All above algorithms are supposed to be run on a desktop computer, and therefore, the power and memory consumption are not been considered. In recent decades, the great development of IoT has brought new challenges for both computer science and communication societies. When billions of IoT devices are connected and communicated, the data have to be processed in a distributed fashion, which is called the edge computing paradigm [7]. Edge computing processes and analyzes data at the edge of Internet, in close proximity to the mobile devices, sensors, and end users. This novel technology allows to provide scalability and privacy-policy enforcement for the IoT, highly responsive cloud services for mobile computing, and the ability to smooth transient cloud outages. Current edge computing devices, *e.g.*, NVIDIA Jetson Nano, Google EDGE TPU, Intel Neural Compute Stick2, and Raspberry Pi, could

✉ Yong Fang
fy@chd.edu.cn

¹ School of Information Engineering, Chang'an University, Xi'an 710064, Shaanxi, China

provide low-latency, low-power consumption, and limited memory platform for deep neural network models.

The main problem addressed in this paper is how to effectively perform lossless data compression on the edge computing platform with limited power consumption and less computing complexity. Because AC can achieve the best compression rate than other algorithms, we select AC as the codec. In 2020, [8] proposed EdgeDRNN: a delta RNN accelerator for edge inference. Inspired by this idea, we present DRAC: an AC model based on delta RNN algorithm running on edge computing devices. Our main contributions can be summarized as follows:

- **We describe the DRAC: an AC codec based on delta RNN for edge computing device.** To the best of our knowledge, this flexible, low-cost, low latent, and power efficient compressor is first presented by us.
- **We implement DRAC on a Xilinx Zynq 7000 SoC Board.** We use this platform to demonstrate that our DRAC model is suitable for the edge devices.
- **Our DRAC outperforms Deepzip in the experiments.** We use textual genomic datasets to evaluate DRAC, Deepzip, and Gzip. The experimental results show that DRAC achieves the best results than Deepzip and Gzip in terms of compression rate, power consumption, and running time.

The rest of this paper is arranged as follows: “Model description” describes the model of DRAC. The Delta RNN is introduced in “Delta recurrent neural networks”, and the experimental results are demonstrated in “Experimental results”. “Conclusion” concludes this paper.

Model description

Our DRAC consists of two blocks: AC algorithm and probability predictor. Let $Y = \{y_0, y_1, \dots, y_{N-1}\}$ be the data sequence. The probability predictor includes a Delta RNN model which is trained on the Y for multiple epochs. After training is completed, the trained model weights are known by both encoder and decoder. When seeing the previous J symbols, where J is set to 1 in our model, the probability $P(y_i | y_{i-1}, \dots, y_{i-J})$ is estimated by Delta RNN and then fed into the AC encoder to perform the encoding process until the final code is output. For the initial J symbols, a uniform distribution is chosen as prior. This process is illustrated in Fig. 1a. The decoding process is just symmetrical to the encoding process. Decoder takes the code as input and repeatedly output the reconstructed symbols using probability from predictor, which is illustrated in Fig. 1b. We describe the AC algorithm block in “AC algorithm” and probability predictor block in “Probability predictor”.

Fig. 1 Model of DRAC

AC algorithm

We give the encoding process of AC as follows. Let $\mathcal{X} = \{x_0, x_1, \dots, x_{M-1}\}$ be the alphabet of a discrete source and $P = \{p(x_0), p(x_1), \dots, p(x_{M-1})\}$ be the probability of each symbol, which will be predicted by “Probability predictor”. The cumulative probability $C = \{c(x_0), c(x_1), \dots, c(x_{M-1})\}$ is defined as

$$c(x_i) = \sum_{k=0}^i P(x_k), \quad (1)$$

where $c(x_0) \equiv 0$ and $c(M) \equiv 1$. AC encoder recursively maps each symbol of input sequence into a subinterval divided from the current interval. Let $Y = \{y_0, y_1, \dots, y_{N-1}\}$ be the input sequence and $I_0 = [L_0, H_0)$ be the initial interval. The new subinterval $I_j = [L_j, H_j)$ is calculated by

$$H_j = L_{j-1} + (H_{j-1} - L_{j-1}) \times c(y_j + 1) \quad (2)$$

$$L_j = L_{j-1} + (H_{j-1} - L_{j-1}) \times c(y_j), \quad (3)$$

where $c(y_j + 1)$ is the cumulative probability of the subsequent symbol of y_j in the alphabet. Once the encoding

process ends, any real number D in the final interval can be chosen as the code.

In the decoding process, Let $\tilde{Y} = \{\tilde{y}_0, \tilde{y}_1, \dots, \tilde{y}_{N-1}\}$ be the output sequence of AC decoder. Decoder recursively performs the following steps:

- (1) Determine the symbol a_i from C based on current code D_j and output $\tilde{y}_j = x_i$, that is

$$\tilde{y}_j = \{x_i : c(x_i) \leq D_j \leq c(x_i + 1)\}, j \in \{0, 1, \dots, N-1\}. \quad (4)$$

- (2) Update D_j by

$$D_{j+1} = \frac{D_j - c(x_i)}{p(x_i)}. \quad (5)$$

Once the decoding process ends successfully, the output sequence can be acquired.

Probability predictor

Our probability predictor is a delta RNN network. Its architecture is shown in Fig. 2. For a symbol y_i , the past J symbols $y_{i-J}, y_{i-J+1}, \dots, y_{i-1}$ are input to a bi-directional Delta Gated Recurrent Unit (Delta GRU) cell, which will be introduced in “Delta recurrent neural networks”. At the end of Delta GRU cells, a Softmax layer plus a Fully Connected (FC) layer are applied on the hidden states acquired from the Delta GRU cell. Finally, the probability $p(y_i)$ is obtained. The Softmax layer is defined as

$$\text{softmax}(z)_i = p_i = \frac{e^{z_i}}{\sum_{j=1}^{|\mathcal{X}|} e^{z_j}}, \quad (6)$$

where $\mathcal{X}$ is the alphabet of the symbols, and $|\cdot|$ is the cardinality. For input X , the FC layer is defined as

$$F = \sigma(XW^T + B), \quad (7)$$

where W is the weight matrix, B is the bias, and $\sigma(x)$ is the activation function.

Delta recurrent neural networks

The Gated Recurrent Units (GRU) is a class of gated RNN, and is very popular in the natural language processing area. For RNN, each GRU cell is illustrated in Fig. 3a. Let $\mathbb{R}$ be the set of real number, and $X = \{x_0, x_1, \dots, x_i, x_{N-1}\}$ be the input sequence with length N . For a GRU cell with H neurons, one-dimensional input x_i , and output h_i , the update equations is defined as

$$r_i = \sigma(W_{xr}x_i + W_{hr}h_{i-1} + b_r)$$

Fig. 2 Architecture of probability predictor

$$u_i = \sigma(W_{xu}x_i + W_{hu}h_{i-1} + b_u)$$

$$c_i = \tanh(W_{xc}x_i + r_i \odot (W_{hc}h_{i-1} + b_c))$$

$$h_i = (1 - u_i) \odot c_i + u_i \odot h_{i-1}, \quad (8)$$

where $r, u, c \in \mathbb{R}^H$ are the reset gate, update gate, and cell state, respectively. $W_x \in \mathbb{R}^H$, $W_h \in \mathbb{R}^{H \times H}$ are the weight matrices and $b \in \mathbb{R}^H$ is bias vector. $\sigma(x)$ is the activation function and $\odot$ is the element-wise multiplication of vectors.

To reduce the computing complexity of GRU, [9] proposed the delta recurrent networks. Later, [8] applied DeltaGRU in the edge devices. Inspired by their ideas, we also apply Delta GRU in our model. We define the following variables:

$$\begin{aligned} \hat{x}_i &= \begin{cases} x_i, & |x_i - \hat{x}_{i-1}| \geq \eta_x \\ \hat{x}_{i-1}, & |x_i - \hat{x}_{i-1}| < \eta_x \end{cases} \\ \hat{h}_i &= \begin{cases} h_i, & |h_i - \hat{h}_{i-1}| \geq \eta_h \\ \hat{h}_{i-1}, & |h_i - \hat{h}_{i-1}| < \eta_h \end{cases} \\ \delta x_i &= \begin{cases} x_i - \hat{x}_{i-1}, & |x_i - \hat{x}_{i-1}| \geq \eta_x \\ \hat{x}_{i-1}, & |x_i - \hat{x}_{i-1}| < \eta_x \end{cases} \\ \delta h_i &= \begin{cases} h_i - \hat{h}_{i-1}, & |h_i - \hat{h}_{i-1}| \geq \eta_h \\ \hat{h}_{i-1}, & |h_i - \hat{h}_{i-1}| < \eta_h \end{cases}, \end{aligned} \quad (9)$$

where $\hat{x}_i$ is the input state memory in timestep i , $\hat{h}_i$ the hidden state memory in timestep i , δx_i the delta input state, and δh_i the delta hidden state. η_x and η_h are the thresholds, respectively. In the initial time, i.e., $i = 1$, $\hat{x}_0, h_0$, and $\hat{h}_0$ are all set to zeros. The update equations of Delta GRU are defined as

$$M_{r,i} = W_{xr}\delta x_i + W_{hr}\delta h_{i-1} + M_{r,i-1}$$

Fig. 3 Architecture of GRU cell

$$\begin{aligned}
 M_{u,i} &= W_{xu}\delta x_i + W_{hu}\delta h_{i-1} + M_{u,i-1} \\
 M_{xc,i} &= W_{xc}\delta x_i + M_{xc,i-1} \\
 M_{hc,i} &= W_{hc}\delta h_{i-1} + M_{hc,i-1} \\
 r_i &= \sigma(M_{r,i}) \\
 u_i &= \sigma(M_{u,i}) \\
 c_i &= \tanh(M_{xc,i} + r_i \odot M_{hc,i}) \\
 h_i &= (1 - u_i) \odot c_i + u_i \odot h_{i-1},
 \end{aligned} \tag{10}$$

where $M_{i=0} = b_r$, $M_{u,i=0} = b_u$, $M_{xc,i=0} = b_c$, $M_{hc,i=0} = 0$ are delta memory vectors and $M \in \mathbb{R}^H$.

Comparing GRU operation, the Delta GRU will prune the weights and remove unimportant neuron connections that result in sparse weight matrices. Sparse matrices can be encoded into a sparse matrix format. The sparse matrix-vector multiplication can be accelerated by executing multiply-and-accumulate (MAC) operations only on nonzero weights. By skipping zero elements in delta vectors, whole columns of matrix-vector MAC operations can be skipped. After the model was properly trained, the experiments show that in delta networks, the number of operations can be reduced by 5-to-100 times [8,9].

Experimental results

We report the experimental results of DRAC model in this section.

Datasets

Four datasets are used in our experiments, which are listed in Table 1. The *webster* [10] and *enwiki9* [11] are text data downloaded from web resource, in which the symbols are from ASCII and the alphabet size are 96 and 206, respectively. The *h.chr1* [12] and *c.e.genome* [13] are genomic data, in which the symbols are $\{A, C, G, T\}$ and the alphabet size is 4. All these four datasets are from real data. In [6], the model is also evaluated on the synthetic datasets, which are the artificial data generated from synthetic sources of known entropy. Due to the limited length of the paper, we will experiment synthetic datasets in the future research.

Training process

To train the delta RNN based model, we take the categorical cross entropy as the loss function, which is defined as

$$L = \frac{1}{N} \sum_{n=1}^N \sum_{m=1}^M p_m \log_2 \frac{1}{\hat{p}_m}, \quad (11)$$

where N is the sequence length, M is the alphabet size, p_m is the one-hot encoded ground truth, and $\hat{p}_m$ is the predicted probability. An Adam [14] optimizer is used to minimize the categorical cross entropy L . The sequence is divided into $\lfloor \frac{N}{K+1} \rfloor$ segments, and the length of each segment is $K+1$. In each segment, let the first K symbols be the input and last one be the output. We set K at 64 and the batch size at 1024. In the optimization, the maximum epochs is 10 and the learning rate is $3e-4$. When there is no further improvement before 10 epochs, the training is terminated. In every epoch, if there is a decrease in L from the previous minima, the model is updated. This training process is similar to that introduced in [6,8].

Implementation

The arithmetic coding block in DRAC is borrowed from an open-source code: Arithmetic Coding Library [15], which is implemented by Python. We made very little modification to enable it to be integrated with predictor block. The Delta RNN block is implemented with Pytorch 1.2.0 framework. The training process was performed on one NVIDIA TITAN V GPU with CUDA 10.0 and cuDNN 7.6.

Our edge device is a Xilinx Zynq-7000 SoC on a customized baseboard. It has a dual ARM Cortex-A9 CPU, a Kintex-7 XC7Z100 FPGA, and 1 GB DDR3 SDRAM. Its operating system is an embedded Linux OS by Xilinx called PetaLinux. Once the DRAC is trained, we use a Python script to convert the Python network modules into C++ header files, which include the network parameters for Delta RNN. The application is compiled with the standard cross compiler and the resulting image is downloaded into the flash of the board.

Delta threshold

Comparing RNN, the major improvement made by Delta RNN is the introduce of delta threshold η_x and η_h . In this section, we will use the experiments to investigate the effects of η ($\eta = \eta_x = \eta_h$). The Delta RNN was trained by the datasets with different delta thresholds η ranging from 0x00 to 0x80 (η is in 16bit Q8.8 format. 0x80 in Q8.8 format corresponds to 0.5 in decimal numbers). The predicted probability error is calculated to evaluate the accuracy. The predicted probability accuracy as the function of delta threshold is shown

Fig. 4 Predicted probability accuracy as the function of delta threshold

in Fig. 4. We find that at small η , the accuracy keeps almost 100%. When η is bigger than 0x80, the accuracy greatly decreases.

We take a sample data from *webster* with the length of 10M and use it to measure the calculation time of the prediction. Figure 4 shows that with the increase of η , calculation time gradually decreases. This is because big delta threshold will allow many MAC operations to be skipped, and reduced operation will result in reduced power consumption and calculation time. We make the compromise between accuracy and calculation time, and choose $\eta = 0x80$ as the optimal value in the following experiments.

Performance comparison

To investigate the performance of our model, we compare DRAC with DeepZip in terms of compression rate, running time, and power consumption.

Four datasets (Table 1) were used in the experiment. We choose two versions of DeepZip [6], *i.e.*, DeepZip-b, whose neural network is implemented by a bi-directional variant of the GRU and DeepZip-l, whose neural network is made by multiple LSTM cells. This setup is because DeepZip-b and DeepZip-l are the state-of-the-art lossless compressor. All DeepZip models are tested on a desktop server with Intel i7-6800K CPU, and an NVIDIA TITAN V GPU. The CPU power is measured by the Stress Terminal UI monitoring tool [16] and the GPU power is measured using the nvidia-smi utility. The DRAC model is performed on Zynq-7000 board and the total power is measured by the wall plug power meter. The running time are also measured. All experimental results are reported in Table 2, in which the best results are boldfaced.

The experimental results show that, in terms of compression size, DRAC nearly achieves same performance with Deepzip, but in terms of running time and power consump-

Table 1 Datasets

Name	Alphabet size	Length	Description
<i>webster</i>	96	41.1M	HTML data of the 1913 Webster dictionary from the Silesia corpus
<i>enwiki9</i>	206	500M	First 500M of the English Wikipedia dump on 2006
<i>h.chr1</i>	4	249M	Chromosome 1 of H. Sapiens GRCh38 sequence
<i>c.e.genome</i>	4	100M	C. elegans whole genome sequence

Table 2 Experimental results

Model		<i>webster</i>	<i>enwiki9</i>	<i>h.chr1</i>	<i>c.e.genome</i>
Deepzip-b	Compression size (M)	8.42	167.27	48.80	23.13
	Running time (s)	48.2	769.5	244.7	154.2
	Power consumption (W)	300.2	311.4	308.1	304.2
Deepzip-l	Compression size (M)	8.66	165.22	48.56	23.41
	Running time (s)	49.3	772.8	256.1	156.2
	Power consumption (W)	301.7	321.5	309.6	303.4
DRAC	Compression size (M)	8.84	173.47	48.21	23.57
	Running time (s)	8.7	159.2	50.2	33.1
	Power consumption (W)	15.4	16.8	15.7	15.9

tion, DRAC greatly outperforms Deepzip. DRAC nearly achieves 5X speedup ratio and 20X power consumption saving than Deepzip. This improvement is due to the introduction of Delta RNN. The experiment shows that DRAC is suitable for the edge computing framework.

Conclusion

We present the DRAC: an AC codec based on Delta RNN for lossless compression, and implement it on Zynq-7000 SoC board. This setup makes it suitable for the edge computing environment. We compare DRAC with DeepZip under four datasets, and DRAC outperforms DeepZip.

This work has two main limits: the selection of the delta threshold η is in a heuristic manner, and the datasets are not plentiful enough. In the future research, we hope to present a guided searching algorithm to quickly converge onto optimal η by exploiting a dynamic trade-off of accuracy versus running time. We will also evaluate DRAC on more real and synthetic datasets.

Acknowledgements The authors would like to thank Z.-P. Wang, a professor with Beihang University, China, and F. Jiang, a professor with Xian University of Posts and Telecommunications, China, for their financial supports.

Funding Information The work was supported by the Natural Science Basic Research Program of Shaanxi Province, China (grant no. 2021JM-188), the National Key Research and Development Program of China (grant no. 2020YFB0505602), and the Open Research Fund of

the Shaanxi Province Key Laboratory of Information Communication Network and Security, China (grant no. ICNS202002).

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Sayood Khalid (2017) Introduction to data compression. Morgan Kaufmann, Morgan Kaufmann, Cambridge, MA, United States
2. Witten IH, Neal R, Cleary JG (1987) Arithmetic coding for data compression. Commun ACM 30:520–540
3. Schmidhuber Jürgen, Heil Stefan (1996) Sequential neural text compression. IEEE Trans Neural Netw 7(1):142–146
4. Mahoney Matthew V (2000) Fast text compression with neural networks. In: Proceedings of the thirteenth international Florida

- artificial intelligence research society conference. pp. 230–234, AAAI Press
5. Byron Knoll, “CMIX,” <http://www.byronknoll.com/cmixon.html>
 6. Goyal M, Tatwawadi K, Chandak S, Ochoa I, “DeepZip: Lossless Data Compression Using Recurrent Neural Networks”, (2019) Data Compression Conference (DCC). Snowbird, UT, USA 2019:575
 7. Satyanarayanan M (2017) The emergence of edge computing. *Computer* 50(1):30–39
 8. Gao C, Rios-Navarro A, Chen X, Liu S-C, Delbruck T (2020) Edge-DRNN: recurrent neural network accelerator for edge inference. *IEEE J Emerging Selected Topics Circuits Syst* 10(4):419–432
 9. Neil D, Lee J, Delbrück T, Liu S-C (2017) Delta networks for optimized recurrent network computation. In: *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, Sydney, NSW, Australia, Aug, pp. 2584–2593
 10. <http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia>
 11. <http://www.mattmahoney.net/dc/textdata.html>
 12. <ftp://hgdownload.cse.ucsc.edu/goldenPath/hg38/chromosomes/chr20.fa.gz>
 13. [ftp://ftp.ensembl.org/pub/release-97/fasta/caenorhabditis\\$_\\$elegans/dna/Caenorhabditis\\$_\\$elegans.WBcel235.dna.toplevel.fa.gz](ftp://ftp.ensembl.org/pub/release-97/fasta/caenorhabditis/$elegans/dna/Caenorhabditis$_$elegans.WBcel235.dna.toplevel.fa.gz)
 14. Kingma Diederik P, Ba Jimmy (2014) Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980
 15. Arithmetic Coding Library,” <https://www.nayuki.io/page/reference-arithmetic-coding>
 16. <https://github.com/amanusk/s-tui>

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.