Log-Quantization on GRU networks

Sang-Ki Park
Dept. of Electronic Engineering
Hanyang University, Seongdong-gu, Seoul, Republic of Korea
+82-2-2220-4701
pskhanyang@gmail.com

Sang-Soo Park
Dept. of Electronic Engineering
Hanyang University, Seongdong-gu, Seoul, Republic of Korea
+82-2-2220-4701
po092000@hanyang.ac.kr

Ki-Seok Chung
Dept. of Electronic Engineering
Hanyang University, Seongdong-gu, Seoul, Republic of Korea
+82-2-2220-4701
kchung@hanyang.ac.kr

ABSTRACT

Today, recurrent neural network (RNN) is used in various applications like image captioning, speech recognition and machine translation. However, because of data dependencies, recurrent neural network is hard to parallelize. Furthermore, to increase network's accuracy, recurrent neural network uses complicated cell units such as long short-term memory (LSTM) and gated recurrent unit (GRU). To run such models on an embedded system, the size of the network model and the amount of computation need to be reduced to achieve low power consumption and low required memory bandwidth. In this paper, implementation of RNN based on GRU with a logarithmic quantization method is proposed. The proposed implementation is synthesized using high-level synthesis (HLS) targeting Xilinx ZCU102 FPGA running at 100MHz. The proposed implementation with an 8-bit log-quantization achieves 90.57% accuracy without re-training or fine-tuning. And the memory usage is 31% lower than that for an implementation with 32-bit floating point data representation.

CCS Concepts

• Hardware → High-level and register-transfer level synthesis; Hardware accelerators • Computer systems organization → Embedded systems; Neural networks; Heterogenous (hybrid) system.

Keywords

CNN; AI; HW/SW Co-Design; HLS; LeNet-5; FPGA; SDSoC;

1. INTRODUCTION

Convolutional neural network (CNN) and recurrent neural network (RNN) have proved their usefulness in various applications. In image classification and recognition, quite a few CNN models have proven their high accuracy [1], [2]. RNN models are widely used in machine translation, speech recognition, and music composition [3], [4].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Request permissions from Permissions@acm.org.
ICCIP 2018, November 2–4, 2018, Qingdao, China
© 2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6534-5/18/11…$15.00
http://doi.org/10.1145/3290420.3290443

However, neural network models have rapidly increased the number of processing layers and the amount of computation to achieve higher accuracy. The number of parameters and the size of memory requirement are getting bigger, as well. To run these large networks on embedded systems, computational workload needs to be reduced so that they can run on embedded systems in real time. Embedded systems typically have a small amount of memory and limited processing power, and also many of them are very sensitive to energy dissipation. Therefore, it is very important to optimize RNN in terms of processing speed, memory usage and power dissipation.

Many researches have been conducted to reduce the size of parameters. There are several ways to achieve this goal; pruning, factorization, quantization, entropy coding [5]-[9]. For example, Deep Compression [5] uses three techniques: pruning, quantization, and Huffman coding. They reduced the size of parameters in VGG-16 [10] from 522MB to 11.3MB, which is 49 times smaller. And SqueezeNet [6] achieved 80.3 % Top-5 ImageNet accuracy, which is the same as AlexNet [7], with 510 times smaller model size by using Deep Compression. On the other hand, [8] and [9] used quantization rather than compression. [8] used a 12-bit fixed point representation on long short-term memory (LSTM) networks and achieved only 0.3% accuracy drop compared to a model with 32-bit floating point representation.

In this paper, we propose a logarithmic quantization on RNN, specifically, gated recurrent unit (GRU) cell architecture to reduce the total model size, including the amount of data transaction between cells. We use a pre-trained model and apply this method to input vectors, hidden states. And quantization and shift modules are added to convert from the floating-point data type to our 8-bit custom data type.

2. BACKGROUND

2.1 Introduction of RNN

RNN models accept an input vector sequence $x = (x_1; x_2; \dots; x_T)$ and produce output $y = (y_1; y_2; \dots; y_T)$ over time T and each vector has its own time stamp. In time *t*, RNN cell takes the corresponding vector x_t and calculates the hidden state vector, h_t. W_h and W_x are weight matrices for hidden states and input vectors, respectively, and b is a bias vector.

$$h_t = tanh(W_h h_{t-1} + W_x x_t + b_h) \quad (1)$$

The computation of a basic RNN cell is shown in (1). It implies that a hidden state of the current time is influenced by hidden states of the previous times. This data dependency makes RNNs hard to parallelize. Weights and biases are shared throughout one layer. In perspective of weight sharing, one RNN layer with n

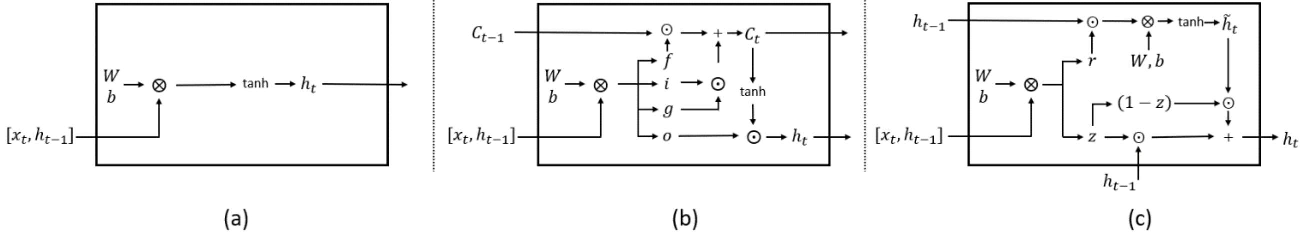

Figure 1. Structure of 3 different RNN cell architectures: (a) Basic RNN cell, (b) LSTM cell, and (c) GRU cell

cells is conceptually the same as unfolded form of one cell over n time periods. We call this n as a hidden size of the layer.

The basic RNN cell, however, may suffer from a problem called vanishing gradient problem; a gradient value is getting smaller and smaller during the training. And it makes difficult to learn contexts with long ranged dependency. Because of this, currently, large networks commonly use LSTM [11] or GRU [12]. An LSTM cell has 3 additional gates; forgot gate (f_t), input gate (i_t), and output gate (o_t). It also adds a new state called cell state (C_t). LSTM computation is described in (2). Weights and biases are distinguished by subscripts; for example, W_{xf} is the weight for x_t in a forgot gate. The operator $\odot$ denotes the element-wise product, and activation function σ is sigmoid.

$$\begin{aligned} f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \\ i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\ o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\ g_t &= tanh\left(W_{xg}x_t + W_{hg}h_{t-1} + b_g\right) \\ C_t &= f_t \odot C_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot tanh(C_t) \end{aligned} \tag{2}$$

By adding these components, LSTM can overcome the vanishing gradient problem. However, the number of parameters is significantly increased. Parameter comparison among three cell architectures will be discussed in the following subsection.

2.2 Architecture of GRU

An LSTM cell has much more computation workload than a basic RNN cell. It has 8 weight matrices and 4 bias vectors whereas a basic RNN cell has 2 weight matrices and one bias vector. Because of its complexity, a new architecture of an RNN cell called GRU has been proposed. In GRU, the gates in LSTM are merged and the cell state output is removed. GRU model equations are described in (3). The reset gate is denoted by r_t and the update gate is denoted by z_t. The next hidden state h_t is calculated by z_t and $\tilde{h}_t$, the candidate hidden state. z_t determines how to combine the new state ($\tilde{h}_t$) with the previous state (h_{t-1}). If the z_t value is 0, then $\tilde{h}_t = h_t$ which means the previous state does not affect the present. If $z_t = 1$, the new hidden state will be the same as the previous hidden state.

$$\begin{aligned} r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\ z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\ \tilde{h}_t &= tanh(W_{xh}x_t + W_{hh}(h_{t-1} \odot r_t) + b_h) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \end{aligned} \tag{3}$$

A GRU cell has 6 weights and 3 biases. Table I shows the number of parameters (weights and biases) of the three different RNN cell architectures, where *X* is the dimension of the input vector and H is the dimension of the hidden state. An LSTM cell and a GRU cell has 4 and 3 times bigger parameter sizes than the basic RNN cell, respectively. Since the basic cell is not recommended if the hidden size of a layer is big, using LSTM and GRU is inevitable.

So, it is important to reduce the size of parameters or decrease the computation workload. It is hard to determine which one is better because the choice of the type of the RNN cell may heavily depend on the dataset and the corresponding task [13]. All three RNN architectures are described in Figure 1. In this paper, we choose the GRU architecture because it can achieve almost the same performance with a relatively smaller model size and less computation workload than LSTM.

Table 1. Size of parameters in three different RNN cell architectures

Model	Basic	LSTM	GRU
Parameter size	XH + H^2 + H	4*(XH + H^2 + H)	3*(XH + H^2 + H)

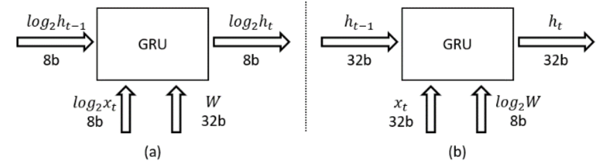

Figure 2. Quantization of the proposed methods

3. RELATED WORKS

Many practical deep learning models employ 32-bit single precision floating point representation. By using the floating-point number representation, highly accurate computation can be achieved, but computation complexity is a lot higher than fixed-point number representation. And the hardware to support floating-point representation is bigger and consumes more power than the hardware for fixed-point representation [14]. So, many studies have proposed to use other data format rather than floating-point representation. In [8], for example, they used 16-bit fixed-point number weights on their LSTM network and achieved 20.7% phone error rate (PER), where the original 32-bit floating point network's PER was 20.4%. They also implemented this network on field programmable gate array (FPGA) with pipelined LSTM techniques. In [5], they used 8-bit and 4-bit quantization on the convolution layer and the fully-connected layer of AlexNet, respectively, without losing accuracy. However, [8] used multipliers and the LSTM is composed of bigger networks than

the GRU. In [5], extra modules were needed to decompress the data which was compressed with Huffman coding.

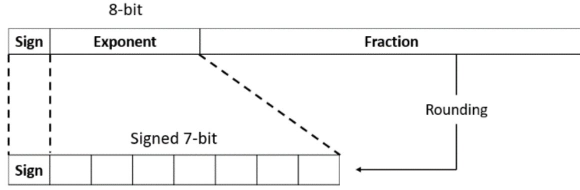

Figure 3. Customized Type Conversion. We remove bias and take 7 bits of exponent part except MSB.

On the other hand, [9] used 3-bit logarithmic quantization on AlexNet and VGG-16. They achieved 89.2% TOP-5 accuracy on VGG-16 using 3-bit quantization compared to 89.8% using 32-bit

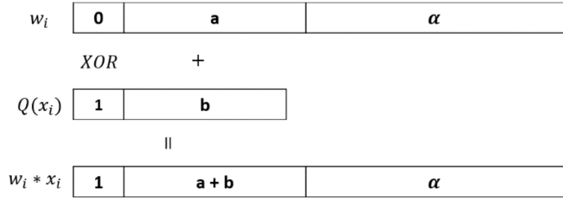

Figure 4. Approximated floating-point multiplication: *ExAdd(wᵢ, Q(xᵢ))*

floating-point number representation. This result is very encouraging because they reached this accuracy by using pre-trained weights rather than re-training. With re-training, they achieved 93.79% on 5-bit log quantization in CIFAR-10 whereas the accuracy of the 32-bit floating point representation is 94.1%.

4. THE PROPOSED QUANTIZATION METHOD

4.1 Quantization and Operation Strength Reduction

In this work, we focus on reducing memory usage, and minimizing the amount of multiplications without any significant accuracy drop. Therefore, we choose the GRU cell in order to make the model size as compact as possible since GRU has less number of parameters than LSTM does. Further, to minimize the amount of multiplications, we replace multiplication with shift and add operations. Detailed explanation will follow in the next subsection.

4.2 Quantization Method

In recurrent network models, every cell in one layer shares the same set of weight matrices. The dimensions of weight matrices for input x_t and that of hidden state vector h_t are $W_x \in \mathbb{R}^X \times \mathbb{R}^H$ and $W_h \in \mathbb{R}^H \times \mathbb{R}^H$, respectively. The number of GRU units in a layer (hidden unit size) is represented as *SEQ*. Then, we can express the number of parameters of one layer of GRU (weights and biases), S_w and the total number of elements in the input vectors and hidden states, S_i as follows:

$$S_w = 3(XH + H^2 + H)$$
$$S_i = SEQ(X + H) \quad (4)$$

Then, the total memory usage of the network would be summation of $u * S_w$ and $v * S_i$ where u and v are the average number of bytes to represent a value. For example, if weights and biases are represented in single precision floating point, u is 4. And the memory usage will be $4 * S_w$. Therefore, we have two choices, reduce u for S_w or reduce v for S_i.

The proposed method that quantizes the value of x_t and h_t in a GRU network is shown in Figure 2 (a). Each time, a GRU cell takes quantized x_t and h_{t-1} and calculates h_t. Before passing h_t to the next cell, h_t represented as the 32-bit floating-point representation is quantized into an 8-bit fixed-point representation. Our customized quantization method is shown in Figure 3. We extract the exponent part of the floating-point representation to make it as a 7-bit signed integer. This roughly corresponds to apply a logarithm of the floating-point number with base 2. This method will reduce the range of exponent from -127~126 to -64~63. However, since GRU uses sigmoid and *tanh* as the activation function, h_t does not exceed this range. For high accuracy, we take the fraction part of $\log_2 h_t$. into account. Let $Q(x)$ be defined to be a function that returns n' that is the nearest integer to x. When $\log_2 x$ where n is integer and $0 \le \alpha < 1$, $Q(x)$ returns n or $n + 1$.

$$\log_2 x = n + \alpha \quad (5)$$

$$Q(x) = n' \quad (6)$$

We have also tried to quantize weights. As shown in Figure 2 (b), the GRU cell gets 8-bit quantized weight matrices and 32-bit floating-point input vectors and hidden states. Quantization and rounding strategies are the same as the method shown in Figure 3 and Figure 4.

By doing this quantization, we can replace a floating-point multiplication with a set of fixed-point shifts and adds shown in (7) [9], where $Bitshift(x, y)$ is the function that shifts x by y positions in a fixed-point data form, and $Q(x_i) = n'$ by (5) and (6).

$$W^T \mathrm{x} = \sum_i^n w_i * x_i \approx \sum_i^n Bitshift(w_i, n') \quad (7)$$

Multiplication of two floating-point numbers can be approximated by simply adding the exponent part of the two floating-point numbers as described in (8). Specifically, we use the approximation shown in Figure 4 where $ExAdd(x, y)$ is the function that adds y to the exponent part of the x. More details are shown in (9). n and m are integers and $0 \le \alpha,\ \beta < 1$.

$$W^T \mathrm{x} = \sum_i^n w_i * x_i \approx \sum_i^n ExAdd(w_i, n') \quad (8)$$

$$\log_2(w_i * x_i) = \log_2 w_i + \log_2 x_i$$
$$= (n + \alpha) + (m + \beta) \quad \text{(using (5))}$$
$$\approx n + m' + \alpha \quad \text{(using (6))} \quad (9)$$

In (9), the fraction part of w_i does not change, and $Q(x_i)$ is add to the exponent part of w_i. The sign is determined by an XOR operation.

In summary, in the proposed implementation, multiplication operations are replaced with a series of addition and shift operations by applying this proposed quantization method shown in (8) and (9)

5. RESULTS AND DISCCUSION

The GRU-based RNN with the proposed quantization method is implemented on the Zynq UltraSCALE ZC102 FPGA board [15].

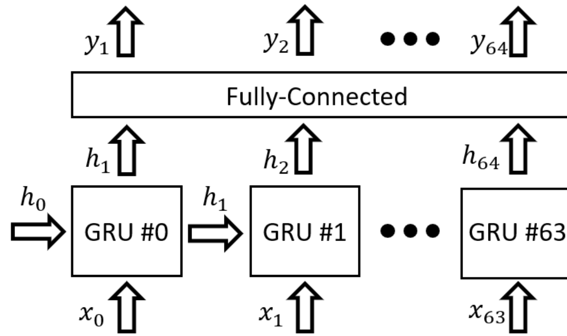

Figure 5. Architecture of the proposed GRU-based RNN

The implementation is running at 100 MHz. The proposed GRU cells are written in C++ and synthesized using a high-level synthesis (HLS) tool. We use a part of the Shakespeare dataset [16] to evaluate the proposed method. The input vector is a sequence of 64-characters and the output vector is the next 64-character sequence. Weights and biases are pre-trained with the TensorFlow framework [17] and there is no re-train or fine-tuning during the experiments. The hardware implement on FPGA is verified by comparing results with a software implementation running on an x86 PC. The hidden unit size is 64, and the dimensions of input vectors and hidden states are both 25 ($X = H = 25$) with 116 batch sizes. Lastly, a fully-connected layer takes 64 hidden state vectors and produces the final output vector y_t using one-hot encoding. The first hidden state h_0 is initialized with 1. The structure for the entire network is described in Figure 5.

Implementations are compared in terms of three metrics: accuracies, cycle counts, and memory usage. The average accuracies are summarized in Table. II. The average accuracy is the average value of accuracies across all batch steps as shown in Figure 6. The cycle count will account for the execution speed. The memory usage is the summation of $u * S_w$ and $v * S_i$.

PROPOSED represents the proposed method that quantizes hidden states and input vectors, and it achieved 90.57% average accuracy with 10% execution time improvement as compared to the conventional method (*CONVENTIONAL* in Table II). There is a 3.33% accuracy drop but *PROPOSED* uses 32% less memory than *CONVENTIONAL*.

As aforementioned, we have also attempted to quantize only the weights as shown in Figure 2 (b), and *ATTEMPT* represents this case. Accuracies of *ATTEMPT* are very poor. Although *ATTEMPT* runs faster and requires less memory than the other two methods, the average accuracy of 22.06% is too low to get any meaningful result. Therefore, re-training and fine-tuning are needed in case of quantizing the weights.

Table 2. Evaluation result comparisons

Model	CONVENTIONAL	PROPOSED	ATTEMPT
Avg. Accuracy	93.90%	90.57%	22.06%
# Cycles	9783.26M	8857.33M	8597.28M
Cycle ratio	1	0.91	0.88
Memory Use	29.88 KB	20.51 KB	16.85 KB
Memory ratio	1	0.69	0.56

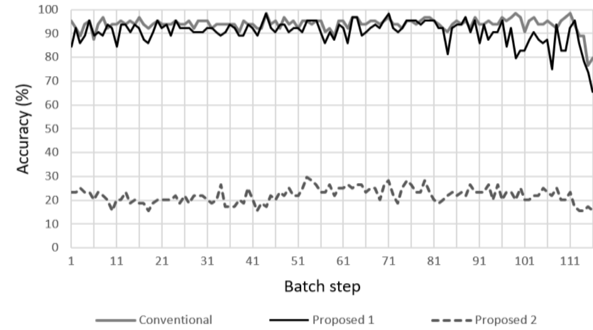

Figure 6. Test accuracy curves

6. CONCLUSION

In this paper, we proposed a method that quantizes the GRU vectors into fixed-point numbers using an 8-bit logarithmic quantization. By using the log-quantization, we can replace floating-point multiplications with a series shifts and adds. For an RNN with 64 hidden units, the proposed method that quantizes hidden states and input vectors achieved 90.57% accuracy without re-training. Using 8-bit quantized weights, however, decreases the accuracy down to 22.06%. In this case, re-training is essential.

7. ACKNOWLEDGMENTS

This work was supported by Institute for Information \& communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (R7119-16-1009, Development of Intelligent Semiconductor Core Technologies for IoT Devices based on Harvest Energy).

8. REFERENCES

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun 2016. Deep Residual Learning for Image Recognition. *The IEEE Conference on Computer Vision and Pattern Recognition* (June 2016), 770-778 DOI= https://doi.org/10.1109/CVPR.2016.90

[2] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre S., et al. 2015. Going Deeper With Convolutions. The IEEE Conference on Computer Vision and Pattern Recognition, (June 2015), 1-9, DOI= https://doi.org/10.1109/CVPR.2015.7298594

[3] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. *In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, 3104-3112 (Dec. 2014) DOI= https://dl.acm.org/citation.cfm?id=2969033.2969173

[4] Douglas Eck and Juergen Schmidhuber. 2002. *A First Look at Music Composition Using LSTM Recurrent Neural Networks*. Technical Report. Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale.

[5] Song Han, Huizi Mao, William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations* (May 2016), DOI= https://arxiv.org/pdf/1510.00149v5.pdf

[6] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer

parameters and <0.5MB model size. DOI= https://arxiv.org/abs/1602.07360

[7] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems* (Dec. 2012), 1097-1105, DOI= https://dl.acm.org/citation.cfm?id=2999257

[8] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. *In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Feb. 2017), 75-84 DOI= https://doi.org/10.1145/3020078.3021745

[9] Daisuke Miyashita, Edward H. Lee and Boris Murmann. 2016. Convolutional Neural Networks using Logarithmic Data Representation. DOI= https://arxiv.org/abs/1603.01025

[10] Karen Simonyan and Andrew Zisserman. 2014 Very Deep Convolutional Networks for Large-Scale Image Recognition. DOI= https://arxiv.org/abs/1409.1556

[11] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. Neural Comput. 9, 8 (November 1997), 1735-1780. DOI= http://dx.doi.org/10.1162/neco.1997.9.8.1735

[12] Cho Kyunghyun, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. DOI= https://arxiv.org/abs/1406.1078

[13] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *Advances in Neural Information Processing Systems* (Dec. 2014), DOI= https://arxiv.org/abs/1412.3555

[14] Song Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, W. J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *ACM/IEEE 43rd Annual International Symposium on Computer Architecture* (June 2016), 243-254, DOI= https://doi.org/10.1109/ISCA.2016.30

[15] ZCU102 Board User Guide, https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf

[16] William Shakespeare Plays Datasets, https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt

[17] Martín Abadi, Paul Barham, Jianmin Chen et al. 2016. TensorFlow: a system for large-scale machine learning. *In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation* (Nov. 2016), 265-283, https://www.tensorflow.org