Tensor Decomposition for Compressing Recurrent Neural Network

Andros Tjandra*†, Sakriani Sakti*†, Satoshi Nakamura*†
*Graduate School of Information Science, Nara Institute of Science and Techonology, Japan
† RIKEN, Center for Advanced Intelligence Project AIP, Japan
Email: {andros.tjandra.ai6, ssakti, s-nakamura}@is.naist.jp

Abstract—**In the machine learning fields, Recurrent Neural Network (RNN) has become a popular architecture for sequential data modeling. However, behind the impressive performance, RNNs require a large number of parameters for both training and inference. In this paper, we are trying to reduce the number of parameters and maintain the expressive power from RNN simultaneously. We utilize several tensor decompositions method including CANDECOMP/PARAFAC (CP), Tucker decomposition and Tensor Train (TT) to re-parameterize the Gated Recurrent Unit (GRU) RNN. We evaluate all tensor-based RNNs performance on sequence modeling tasks with a various number of parameters. Based on our experiment results, TT-GRU achieved the best results in a various number of parameters compared to other decomposition methods.**

I. Introduction

In recent years, RNNs have achieved many state-of-the-arts on sequential data modeling task and significantly improved the performance on many tasks, such as speech recognition [1], [2] and machine translation [3], [4]. There are several reasons behind the RNNs impressive performance: the availability of data in large quantities and the advance of modern computer performances such as GPGPU. The recent hardware advance allows us to train and infer RNN models with million of parameters in a reasonable amount of time.

Some devices such as mobile phones or embedded systems only have limited computing and memory resources. Therefore, deploying a model with a large number of parameters in those kind of devices is a challenging task. Therefore, we need to represent our model with more efficient methods and keep our model representational power at the same time.

Some researchers have conducted important works to balance the trade-off between the model efficiency and their representational power. There are many different approaches to tackle this issue. From the low-level optimization perspective, Courbariaux et al. [5] replace neural network weight parameters with binary numbers. Hinton et al. [6] compress a larger model into a smaller model by training the latter on soft-target instead of hard-target. RNNs are composed by multiple linear transformations and followed by non-linear transformation. Most of RNN parameters are used to represent the weight matrix in those linear transformations and the total number of parameters depends on the input and hidden unit size. Therefore, some researchers also tried to represent the dense weight matrices with several alternative structures. Denil et al. [7] employed low-rank matrix to replace the original weight matrix.

Instead of using low-rank matrix decomposition, Novikov et al. [8] used TT format to represent the weight matrices in the fully connected layer inside a CNN model. Tjandra et al. [9] applied TT-decomposition to compress the weight matrices inside RNN models. Besides TT-decomposition, there are several popular tensor decomposition methods such as CP decomposition and Tucker-decomposition.

However, those methods have not been explored for compressing RNN weight matrices, thus we are interested to see the extensive comparison between all tensor-decomposition method performances under the same number of parameters. In this paper, we utilized several tensor decomposition methods including CP-decomposition, Tucker decomposition and TT-decomposition for compressing RNN parameters. We represent GRU RNN weight matrices with these tensor decomposition methods. We conduct extensive experiments on sequence modeling with a polyphonic music dataset. We compare the performances of uncompressed GRU model and three different tensor-based compressed RNN models: CP-GRU, Tucker-GRU and TT-GRU [9] on various number of parameters. From our experiment results, we conclude that TT-GRU achieved the best result in various number of parameters compared to other tensor-decomposition method.

II. Recurrent Neural Network

RNNs are a type of neural networks designed for modeling sequential and temporal data. For each timestep, the RNN calculate its hidden states by combining previous hidden states and a current input feature. Therefore, RNNs are able to capture all previous information from the beginning until current timestep.

A. Elman Recurrent Neural Network

Elman RNNs are one of the earliest type of RNN models [10]. In some cases, Elman RNN also called as simple RNN. Generally, we represent an input sequence as $\mathbf{x} = (x_1, ..., x_T)$, hidden vector sequence as $\mathbf{h} = (h_1, ..., h_T)$ and output vector sequence as $\mathbf{y} = (y_1, ..., y_T)$. As illustrated in Fig. 1, a simple RNN at t-th time-step is can be formulated as:

$$\begin{aligned} h_t &= f(W_{xh}x_t + W_{hh}h_{t-1} + b_h) && (1)\\ y_t &= g(W_{hy}h_t + b_y). && (2) \end{aligned}$$

where W_{xh} represents the weight parameters between the input and hidden layer, W_{hh} represents the weight parameters between the previous and current hidden layers, W_{hy} represents the weight parameters between the hidden and output layer, and b_h and b_y represent bias vectors for the hidden and output layers. Functions $f(\cdot)$ and $g(\cdot)$ are nonlinear activation functions, such as sigmoid or tanh.

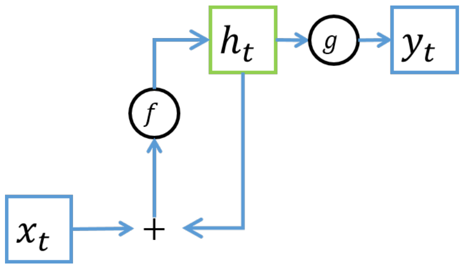

Figure 1. Recurrent Neural Network

B. Gated Recurrent Neural Network

Learning over long sequences is a hard problem for standard RNN because the gradient can easily vanish or explode [11], [12]. One of the sources for that problem is because RNN equations are using bounded activation function such as tanh and sigmoid. Therefore, training a simple RNN is more complicated than training a feedforward neural network. Some researches addressed the difficulties of training simple RNNs. From the optimization perspective, Martens et al. [13] utilized a second-order Hessian-free (HF) optimization rather than the first-order method such as stochastic gradient descent. However, to calculate the second-order gradient or their approximation requires some extra computational steps. Le et al. [14] changed the activation function that causes the vanishing gradient problem with a rectifier linear (ReLU) function. They are able to train a simple RNN for learning long-term dependency with an unbounded activation function and identity weight initialization. Modifying the internal structure from RNN by introducing gating mechanism also helps RNNs solve the vanishing gradient problems. The additional gating layers control the information flow from the previous states and the current input [15]. Several versions of gated RNNs have been designed to overcome the weakness of simple RNNs by introducing gating units, such as Long-Short Term Memory (LSTM) RNN and GRU RNN.

1) Long-Short Term Memory RNN: An LSTM [15] is a gated RNN with memory cells and three gating layers. The gating layers purpose is to control the current memory states by retaining the important information and removing the unused information. The memory cells store the internal information across time steps. As illustrated in Fig. 2, the LSTM hidden layer values at time t are defined by the following equations [2]:

$$
\begin{aligned}
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) && (3)\\
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) && (4)\\
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) && (5)\\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) && (6)\\
h_t &= o_t \odot \tanh(c_t) && (7)
\end{aligned}
$$

where $\sigma(\cdot)$ is sigmoid activation function and i_t, f_t, o_t and c_t are respectively the input gates, the forget gates, the output gates and the memory cells. The input gates retain the candidate memory cell values that are useful for the current memory cell and the forget gates retain the previous memory cell values that are useful for the current memory cell. The output gates retain the memory cell values that are useful for the output and the next time-step hidden layer computation.

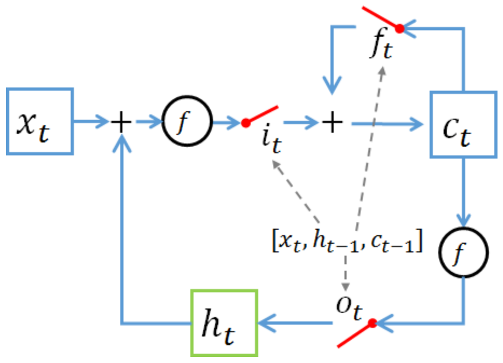

Figure 2. Long Short Term Memory Unit.

2) Gated Recurrent Unit RNN: A GRU [16] is one variant of gated RNN. It was proposed an alternative to LSTM. There are several key differences between GRU and LSTM. First, a GRU does not seperate the hidden states with the memory cells [17]. Second, instead of three gating layers, it only has two: reset gates and update gates. As illustrated in Fig. 3, the GRU hidden layer at time t is defined by the following equations [16]:

$$
\begin{aligned}
r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) && (8)\\
z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) && (9)\\
\tilde{h}_t &= f(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) && (10)\\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t && (11)
\end{aligned}
$$

where $\sigma(\cdot)$ is a sigmoid activation function, $f(\cdot)$ is a tanh activation function, r_t, z_t are the reset and update gates, $\tilde{h}_t$ is the candidate hidden layer values, and h_t is the hidden layer values at time t-th. The reset gates control the previous hidden layer values that are useful for the current candidate hidden layer. The update gates decide whether to keep the previous hidden layer values or replace the current hidden layer values with the candidate hidden layer values. GRU can match LSTM's performance and its convergence speed sometimes surpasses LSTM, despite having one fewer gating layer [17].

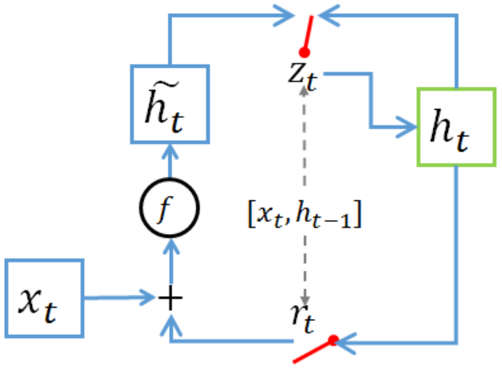

Figure 3. Gated Recurrent Unit

III. Tensor RNN

In this section, we explain our approaches to compress the parameters in the RNN. First, we define the tensorization process to transform the weight matrices inside the RNN model into higher order tensors. Then, we describe two tensor decompositions method called as CANDECOMP/PARAFAC (CP) decomposition and Tucker decomposition. Last, we explain about tensorization and RNN parameters compression with the tensor decomposition methods.

A. Vector, Matrix and Tensor

Before we start to explain any further, we will define different notations for vectors, matrices and tensors. Vector is an one-dimensional array, matrix is a two-dimensional array and tensor is a higher-order multidimensional array. In this paper, bold lower case letters (e.g., **b**) represent vectors, bold upper case letters (e.g., **W**) represent matrices and bold calligraphic upper case letters (e.g., $\mathcal{W}$) represent tensors. For representing the element inside vectors, matrices and tensors, we explicitly write the index in every dimension without bold font. For example, $b(i)$ is the i-th element in vector **b**, $W(p, q)$ is the element on p-th row and q-th column from matrix **W** and $\mathcal{W}(i_1, .., i_d)$ is the $i_1, .., i_d$-th index from tensor $\mathcal{W}$.

B. Tensor decomposition method

Tensor decomposition is a method for generalizing low-rank approximation from a multi-dimensional array. There are several popular tensor decomposition methods, such as Canonical polyadic (CP) decomposition, Tucker decomposition and Tensor Train decomposition. The factorization format differs across different decomposition methods. In this section, we explain briefly about CP-decomposition and Tucker decomposition.

1) CP-decomposition: Canonical polyadic decomposition (CANDECOMP/PARAFAC) [19]–[21] or usually referred to CP-decomposition factorizes a tensor into the sum of outer products of vectors. Assume we have a 3rd-order tensor $\mathcal{W} \in \mathbb{R}^{m_1 \times m_2 \times m_3}$, we can approximate it with CP-decomposition:

$$\mathcal{W} \approx \sum_{r=1}^{R} \mathbf{g}_{1,r} \otimes \mathbf{g}_{2,r} \otimes \mathbf{g}_{3,r} \quad (12)$$

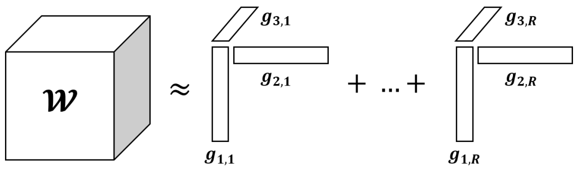

Figure 4. CP-decomposition for 3rd-order tensor $\mathcal{W}$

where $\forall r \in [1..R], \mathbf{g}_{1,r} \in \mathbb{R}^{m_1}$, $\mathbf{g}_{2,r} \in \mathbb{R}^{m_2}$, $\mathbf{g}_{3,r} \in \mathbb{R}^{m_3}$, $R \in \mathbb{Z}^+$ is the number of factors combinations (CP-rank) and $\otimes$ denotes Kronecker product operation. Elementwise, we can calculate the result by:

$$\mathcal{W}(x, y, z) \approx \sum_{r=1}^{R} g_{1,r}(x)\, g_{2,r}(y)\, g_{3,r}(z) \quad (13)$$

In Figure 4, we provide an illustration for Eq. 12 in more details.

2) Tucker decomposition: Tucker decomposition [21], [22] factorizes a tensor into a core tensor multiplied by a matrix along each mode. Assume we have a 3rd-order tensor $\mathcal{W} \in \mathbb{R}^{m_1 \times m_2 \times m_3}$, we can approximate it with Tucker decomposition:

$$\mathcal{W} \approx \mathcal{G}_0 \times_1 \mathbf{G}_1 \times_2 \mathbf{G}_2 \times_2 \mathbf{G}_3 \quad (14)$$

where $\mathcal{G}_0 \in \mathbb{R}^{r_1 \times r_2 \times r_3}$ is the core tensor, $\mathbf{G}_1 \in \mathbb{R}^{m_1 \times r_1}$, $\mathbf{G}_2 \in \mathbb{R}^{m_2 \times r_2}$, $\mathbf{G}_3 \in \mathbb{R}^{m_3 \times r_3}$ are the factor matrices and $\times_n$ is the n-th mode product operator. The mode product between a tensor $\mathcal{G}_0 \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ and a matrix $\mathbf{G}_1 \in \mathbb{R}^{m_1 \times n_1}$ is a tensor $\mathbb{R}^{m_1 \times n_2 \times n_3}$. By applying the mode products across all modes, we can recover the original $\mathcal{W}$ tensor. Elementwise, we can calculate the element from tensor $\mathcal{W}$ by:

$$\mathcal{W}(x, y, z) \approx \sum_{s_1=1}^{r_1} \sum_{s_2=1}^{r_2} \sum_{s_3=1}^{r_3} \mathcal{G}_0(s_1, s_2, s_3) \\ G_1(x, s_1)\, G_2(y, s_2)\, G_3(z, s_3) \quad (15)$$

where $x \in [1, .., m_1], y \in [1, .., m_2], z \in [1, .., m_3]$. Figure 5 gives an illustration for Eq. 14

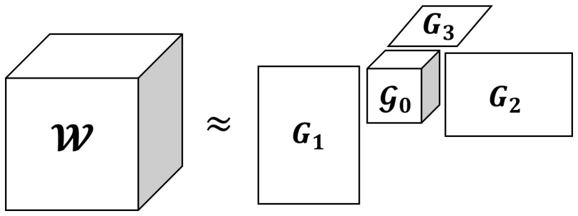

Figure 5. Tucker decomposition for 3rd-order tensor $\mathcal{W}$

C. Tensor Train decomposition

Tensor Train decomposition [18] factorizes a tensor into a collection of lower order tensors called as TT-cores. All TT-cores are connected through matrix multiplications across all tensor order to calculate the element from original tensor.

Assume we have a 3rd-order tensor $\mathcal{W} \in \mathbb{R}^{m_1 \times m_2 \times m_3}$, we can approximate the element at index x, y, z by:

$$\mathcal{W}(x, y, z) \approx \sum_{s_1=1}^{r_1} \sum_{s_2=1}^{r_2} \mathcal{G}_1(x, s_1) \mathcal{G}_2(s_1, y, s_2) \mathcal{G}_3(s_2, z) \quad (16)$$

where $x \in [1, .., m_1], y \in [1, .., m_2], z \in [1, .., m_3]$ and $\mathcal{G}_1 \in \mathbb{R}^{m_1 \times r_1}, \mathcal{G}_2 \in \mathbb{R}^{r_1 \times m_2 \times r_2}, \mathcal{G}_3 \in \mathbb{R}^{r_2 \times m_3}$ as the TT-cores. Figure 6 gives an illustration for Eq. 16.

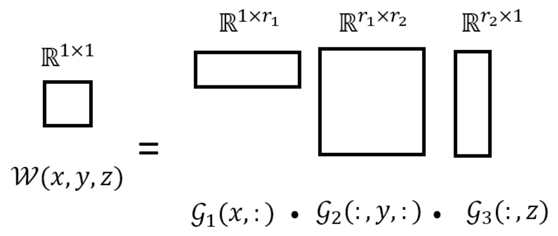
$\mathbb{R}^{1\times1}$
$\mathcal{W}(x,y,z)$
=
$\mathbb{R}^{1\times r_1}$
$\mathbb{R}^{r_1\times r_2}$
$\mathbb{R}^{r_2\times 1}$
$\mathcal{G}_1(x,:) \bullet \mathcal{G}_2(:,y,:) \bullet \mathcal{G}_3(:,z)$

Figure 6. Tensor Train decomposition for 3rd-order tensor $\mathcal{W}$

D. RNN parameters tensorization

Most of RNN equations are composed by multiplication between the input vector and their corresponding weight matrix:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \quad (17)$$

where $\mathbf{W} \in \mathbb{R}^{M \times N}$ is the weight matrix, $\mathbf{b} \in \mathbb{R}^M$ is the bias vector and $\mathbf{x} \in \mathbb{R}^N$ is the input vector. Thus, most of RNN parameters are used to represent the weight matrices. To reduce the number of parameters significantly, we need to represent the weight matrices with the factorization of higher-order tensor. First, we apply tensorization on the weight matrices. Tensorization is the process to transform a lower-order dimensional array into a higher-order dimensional array. In our case, we tensorize RNN weight matrices into tensors. Given a weight matrix $\mathbf{W} \in \mathbb{R}^{M \times N}$, we can represent them as a tensor $\mathcal{W} \in \mathbb{R}^{m_1 \times m_2 \times .. \times m_d \times n_1 \times n_2 \times .. \times n_d}$ where $M = \prod_{k=1}^d m_k$ and $N = \prod_{k=1}^d n_k$. For mapping each element in matrix $\mathbf{W}$ to tensor $\mathcal{W}$, we define one-to-one mapping between row-column and tensor index with bijective functions $\mathbf{f}_i : \mathbb{Z}_+ \rightarrow \mathbb{Z}_+^d$ and $\mathbf{f}_j : \mathbb{Z}_+ \rightarrow \mathbb{Z}_+^d$. Function $\mathbf{f}_i$ transforms each row $p \in \{1, .., M\}$ into $\mathbf{f}_i(p) = [i_1(p), .., i_d(p)]$ and $\mathbf{f}_j$ transforms each column $q \in \{1, .., N\}$ into $\mathbf{f}_j(q) = [j_1(q), .., j_d(q)]$. Following this, we can access the value from matrix $W(p, q)$ in the tensor $\mathcal{W}$ with the index vectors generated by $\mathbf{f}_i(p)$ and $\mathbf{f}_j(q)$ with these bijective functions.

After we determine the shape of the weight tensor, we choose one of the tensor decomposition methods (e.g., CP-decomposition (Sec.III-B1), Tucker decomposition (Sec.III-B2) or Tensor Train [18]) to represent and reduce the number of parameters from the tensor $\mathcal{W}$. In order to represent matrix-vector products inside RNN equations, we need to reshape the input vector $\mathbf{x} \in \mathbb{R}^N$ into a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times .. \times n_d}$ and the bias vector $\mathbf{b} \in \mathbb{R}^M$ into a tensor $\mathcal{B} \in \mathbb{R}^{m_1 \times .. \times m_d}$. Therefore, we can reformulate the Eq. 17 to calculate $y(p)$ elementwise with:

$$\mathcal{Y}(\mathbf{f}_i(p)) = \sum_{j_1, .., j_d} \mathcal{W}(\mathbf{f}_i(p), j_1, .., j_d)\, \mathcal{X}(j_1, .., j_d) + \mathcal{B}(\mathbf{f}_i(p)) \quad (18)$$

by enumerating all columns q position with $j_1, .., j_d$ and $\mathbf{f}_i(p) = [i_1(p), .., i_d(p)]$.

For CP-decomposition, we represent our tensor $\mathcal{W}$ with multiple factors $\mathbf{gm}_{k,r}, \mathbf{gn}_{k,r}$ where $\forall k \in [1..d]\, \forall r \in [1..R]$, $(\mathbf{gm}_{k,r} \in \mathbb{R}^{m_k}, \mathbf{gn}_{k,r} \in \mathbb{R}^{n_k})$. From here, we replace Eq. 18 with:

$$\mathcal{Y}(\mathbf{f}_i(p)) = \sum_{j_1, .., j_d} \left(\sum_{r=1}^{R} \prod_{k=1}^{d} gm_{k,r}(i_k(p)) gn_{k,r}(j_k) \right) \mathcal{X}(j_1, .., j_d) + \mathcal{B}(\mathbf{f}_i(p)). \quad (19)$$

By using CP-decomposition for representing the weight matrix $\mathbf{W}$, we reduce the number of parameters from $M \times N$ into $R * (\sum_{k=1}^d m_k + n_k)$.

For Tucker decomposition, we represent out tensor $\mathcal{W}$ with a tensor core $\mathcal{G}_0 \in \mathbb{R}^{r_1 \times ... \times r_d \times r_{d+1} \times ... \times r_{2d}}$ where $\forall k \in [1..d]$, $r_k < m_k$ and $\forall k \in [1..d]$, $r_{d+k} < n_k$ and multiple factor matrices $\mathbf{GM}_k, \mathbf{GN}_k$, where $\forall k \in [1..d]$, $(\mathbf{GM}_k \in \mathbb{R}^{m_k \times r_k}, \mathbf{GN}_k \in \mathbb{R}^{n_k \times r_{d+k}})$. Generally, the tensor core ranks $r_1, r_2, .., r_d$ are corresponding to the row in tensor index and $r_{d+1}, r_{d+2}, .., r_{2d}$ are corresponding to the column in tensor index. From here, we replace Eq. 18 with:

$$\mathcal{Y}(\mathbf{f}_i(p)) = \sum_{j_1, .., j_d} \left(\sum_{s_1, ..s_d, s_{d+1}, .., s_{2d}}^{r_1, .., r_d, r_{d+1}, .., r_{2d}} \mathcal{G}_0(s_1, ..s_d, s_{d+1}, .., s_{2d}) \prod_{k=1}^{d} GM_k(i_k(p), s_k) GN_k(j_k, s_{d+k}) \right) \mathcal{X}(j_1, .., j_d) + \mathcal{B}(\mathbf{f}_i(p)). \quad (20)$$

By using Tucker decomposition for representing the weight matrix $\mathbf{W}$, we reduce the number of parameters from $M \times N$ into $\sum_{k=1}^d (m_k * r_k + n_k * r_{d+k}) + (\prod_{k=1}^{2d} r_k)$.

For the TT-decomposition, we refer to [9] on how to represent the tensor $\mathcal{W}$ and how to calculate the linear projection to replace Eq. 18.

In this work, we focus on compressing GRU-RNN by representing all weight matrices (input-to-hidden and hidden-to-hidden) with tensors and factorize the tensors with low-rank tensor decomposition methods. For compressing other RNN architectures such as Elman RNN (Sec. II-A) or LSTM-RNN (Sec. II-B1), we can follow the same steps by replacing all the weight matrices with factorized tensors representation.

E. Tensor Core and Factors Initialization Trick

Because of the large number of recursive matrix multiplications, followed by some nonlinearity (e.g, sigmoid, tanh), the gradient from the hidden layer will diminish after several time-step [23]. Consequently, training recurrent neural networks is much harder compared to standard feedforward neural networks.

Even worse, we decompose the weight matrix into multiple smaller tensors or matrices, thus the number of multiplications needed for each calculation increases multiple times. Therefore, we need a better initialization trick on the tensor cores and factors to help our model convergences in the early training stage.

In this work, we follow Glorot et al. [24] by initializing the weight matrix with a certain variance. We assume that our original weight matrix $\mathbf{W}$ has a mean 0 and the variance σ_W^2. We utilize the basic properties from a sum and a product variance between two independent random variables.

Definition III.1. Let X and Y be independent random variables with the mean 0, then the variance from the sum of X and Y is $Var(X+Y) = Var(X) + Var(Y)$

Definition III.2. Let X and Y be independent random variables with the mean 0, then the variance from the product of X and Y is $Var(X * Y) = Var(X) * Var(Y)$

After we decided the target variance σ_w^2 for our original weight matrix, now we need to derive the proper initialization rules for the tensor core and factors. We calculate the variance for tensor core and factors by observing the number of sum and product operations and utilize the variance properties from Def. III.1 and III.2. For weight tensor $\mathcal{W}$ based on the CP-decomposition, we can calculate σ_g as the standard deviation for all factors $\mathbf{gm}_{k,r}, \mathbf{gn}_{k,r}$ with:

$$\sigma_g = \sqrt[4d]{\frac{\sigma_w^2}{R}} \tag{21}$$

and initialize $\mathbf{gm}_{k,r}, \mathbf{gn}_{k,r} \sim \mathcal{N}(0, \sigma_g^2)$.

For weight tensor $\mathcal{W}$ based on the Tucker decomposition, we can calculate σ_g as the standard deviation for the core tensor $\mathcal{G}_0$ and the factor matrices $\mathbf{GM}_k, \mathbf{GN}_k$ with:

$$\sigma_g = \sqrt[(4d+2)]{\frac{\sigma_w^2}{\prod_{k=1}^{2d} r_k}} \tag{22}$$

and initialize $\mathcal{G}_0, \mathbf{GM}_k, \mathbf{GN}_k \sim \mathcal{N}(0, \sigma_g^2)$.

For weight tensor $\mathcal{W}$ based on the Tensor Train decomposition, we refer to [9] for initializing the TT-cores $\mathcal{G}_i$.

IV. Experiments

In this section, we describe our dataset and all model configurations. We performed experiments with three different tensor-decompositions (CP decomposition, Tucker decomposition and TT decomposition) to compress our GRU and also the baseline GRU. In the end, we report our experiment results and finish this section with some discussions and conclusions. Our codes are available at https://github.com/androstj/tensor_rnn.

A. Dataset

We evaluated our models with sequential modeling tasks. We used a polyphonic music dataset [25] which contains 4 different datasets[1]: Nottingham, MuseData, PianoMidi and JSB Chorales. For each active note in all time-step, we set the value as 1, otherwise 0. Each dataset consists of at least 7 hours of polyphonic music and the total is ± 67 hours.

B. Models

We evaluate several models in this paper: GRU-RNN (no compression), CP-GRU (weight compression via CP decomposition), Tucker-GRU (weight compression via Tucker decomposition), TT-GRU [9] (compressed weight with TT-decomposition). For each timestep, the input and output targets are vectors of 88 binary value. The input vector is projected by a linear layer with 256 hidden units, followed by LeakyReLU [26] activation function. For the RNN model configurations, we enumerate all the details in the following list:

1) GRU
   - Input size (N): 256
   - Hidden size (M): 512
2) Tensor-based GRU
   - Input size (N): 256
   - Tensor input shape ($n_{1..4}$): $4 \times 4 \times 4 \times 4$
   - Hidden size (M): 512
   - Tensor hidden shape ($m_{1..4}$): $8 \times 4 \times 4 \times 4$

   a) CP-GRU
      - CP-Rank (R): $[10, 30, 50, 80, 110]$

   b) Tucker-GRU
      - Core ($\mathcal{G}_0$) shape:
        - $(2 \times 2 \times 2 \times 2) \times (2 \times 2 \times 2 \times 2)$
        - $(2 \times 3 \times 2 \times 3) \times (2 \times 3 \times 2 \times 3)$
        - $(2 \times 3 \times 2 \times 4) \times (2 \times 3 \times 2 \times 4)$
        - $(2 \times 4 \times 2 \times 4) \times (2 \times 4 \times 2 \times 4)$
        - $(2 \times 3 \times 3 \times 4) \times (2 \times 3 \times 3 \times 4)$

   c) TT-GRU
      - TT-ranks:
        - $(1 \times 3 \times 3 \times 3 \times 1)$
        - $(1 \times 5 \times 5 \times 5 \times 1)$
        - $(1 \times 7 \times 7 \times 7 \times 1)$
        - $(1 \times 9 \times 9 \times 9 \times 1)$
        - $(1 \times 9 \times 9 \times 9 \times 1)$

In this task, the training criterion is to minimize the negative log-likelihood (NLL). In evaluation, we measured two different scores: NLL and accuracy (ACC). For calculating the accuracy, we follow Bay et al. [27] formulation:

$$ACC = \frac{\sum_{t=1}^{T} TP(t)}{\sum_{t=1}^{T} (TP(t) + FP(t) + FN(t))} \tag{23}$$

where $TP(t), FP(t), FN(t)$ is the true positive, false positive and false negative at time-t.

[1]Dataset are downloaded from: http://www-etud.iro.umontreal.ca/~boulanni/icml2012

For training models, we use Adam [28] algorithm for our optimizer. To stabilize our training process, we clip our gradient when the norm $||\nabla w|| > 5$. For fair comparisons, we performed a grid search over learning rates $(1e{-}2, 5e{-}3, 1e{-}3)$ and dropout probabilities $(0.2, 0.3, 0.4, 0.5)$. The best model based on loss in validation set will be used for the test set evaluation.

C. Result and Discussion

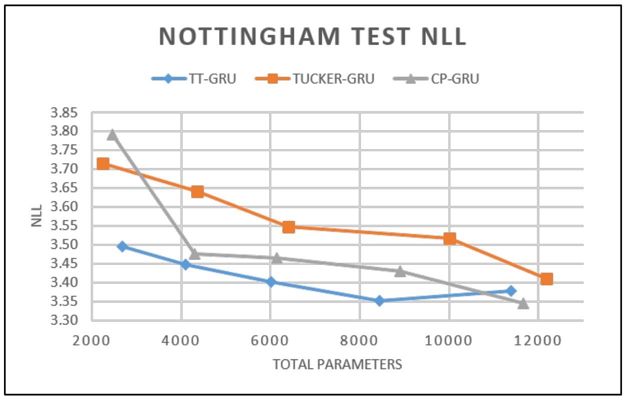

Figure 7. NLL comparison between TT-GRU, Tucker-GRU, and CP-GRU on Nottingham test set

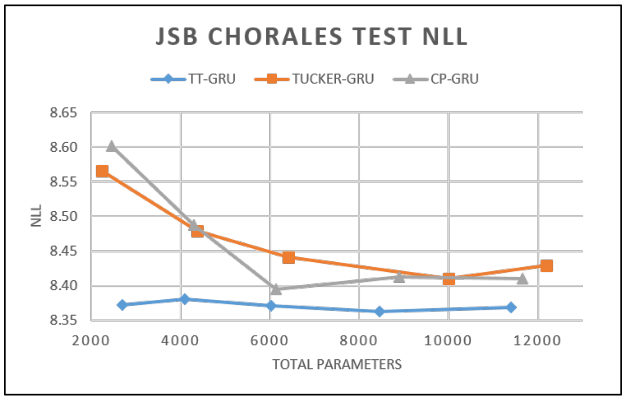

Figure 8. NLL comparison between TT-GRU, Tucker-GRU, and CP-GRU on JSB Chorales test set

We report results of our experiments in Table. I. For the baseline model, we choose standard GRU-RNN without any compression on the weight matrices. For the comparison between compressed models (CP-GRU, Tucker-GRU and TT-GRU), we run each model with 5 different configurations and varied the number of parameters ranged from 2232 up to 12184. In Figure 7-10, we plot the negative log-likelihood (NLL) score corresponding to the number of parameters for each model. From our results, we observe that TT-GRU performed better than Tucker-GRU in every experiments with similar number of parameters. In some datasets (e.g., Piano-Midi, MuseData, Nottingham), CP-GRU has better results compared to Tucker-GRU and achieves similar performance (albeit slightly worse) as TT-GRU when the number of parameters are greater than 6000.

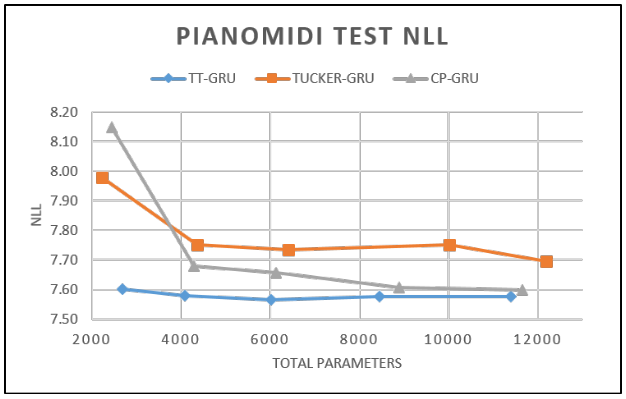

Figure 9. NLL comparison between TT-GRU, Tucker-GRU, and CP-GRU on PianoMidi test set

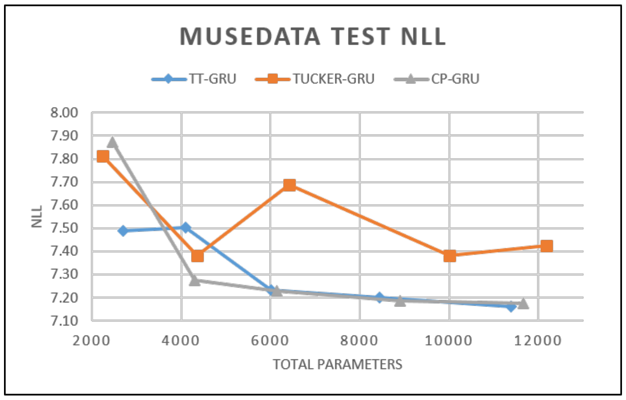

Figure 10. NLL comparison between TT-GRU, Tucker-GRU, and CP-GRU on MuseData test set

V. Related Work

Compressing neural network has been studied intensively in the recent years. Some works have been proposed to reduce the number of bits needed to represent neural network weight values. Instead of using full precision 32-bit floating points, Courbariaux et al. [29] and Gupta et al. [30] half precision floating points is sufficient to represent the neural network weights. Later, Courbariaux et al. [5] represented the floating point numbers in the weight matrices into the binary values and replace most arithmetic operations with bit-wise operations.

"Distilling" the knowledge from a larger model into a smaller model is popularized by Hinton et al. [6]. There are several steps for knowledge distillation: 1) Train a large neural network model with hard labels as the output target, 2) Using a trained large neural network, generate the soft label from each input by taking the last softmax output with higher temperature, 3) Train a smaller neural network with the soft target as the output target. Tang et al. [31] adapt knowledge distillation by using large DNN soft-targets to assist the RNN model training. Kim et al. [32] proposed sequence-level knowledge distillation for compressing neural machine translation models.

Table I
COMPARISON BETWEEN ALL MODELS AND THEIR CONFIGURATIONS BASED ON THE NUMBER OF PARAMETERS, NEGATIVE LOG-LIKELIHOOD AND ACCURACY OF POLYPHONIC TEST SET

Model	Config	Param	Dataset							
			Nottingham		JSB		PianoMidi		MuseData	
			NLL	ACC	NLL	ACC	NLL	ACC	NLL	ACC
GRU IN:256 OUT:512		1181184	3.369	71.1	8.32	30.24	7.53	27.19	7.12	36.30
CP-GRU IN: 4,4,4,4 OUT: 8,4,4,4	**Rank**									
	10	2456	3.79	67.51	8.60	27.29	8.15	19.03	7.87	27.32
	30	4296	3.48	69.85	8.49	28.33	7.68	25.03	7.27	36.19
	50	6136	3.46	69.56	8.40	28.47	7.66	26.18	7.23	36.34
	80	8896	3.43	69.73	8.41	27.88	7.61	28.28	7.19	36.57
	110	11656	3.34	70.42	8.41	29.45	7.60	27.36	7.18	36.89
TUCKER-GRU IN: 4,4,4,4 OUT: 8,4,4,4	**Cores**									
	2,2,2,2	2232	3.71	68.30	8.57	27.28	7.98	20.79	7.81	29.94
	2,3,2,3	4360	3.64	68.63	8.48	28.10	7.75	24.92	7.38	34.20
	2,3,2,4	6408	3.55	69.10	8.44	28.06	7.73	25.66	7.69	32.50
	2,4,2,4	10008	3.52	69.18	8.41	27.70	7.75	24.46	7.38	35.58
	2,3,3,4	12184	3.41	70.23	8.43	29.03	7.69	25.26	7.43	33.63
TT-GRU IN: 4,4,4,4 OUT: 8,4,4,4	**TT-rank**									
	1,3,3,3,1	2688	3.49	69.49	8.37	28.41	7.60	26.95	7.49	34.99
	1,5,5,5,1	4096	3.45	69.81	8.38	28.86	7.58	27.46	7.50	33.37
	1,7,7,7,1	6016	3.40	70.72	8.37	28.83	7.57	27.58	7.23	36.53
	1,9,9,9,1	8448	3.35	70.82	8.36	29.32	7.58	27.62	7.20	37.81
	1,11,11,11,1	11392	3.38	70.51	8.37	29.55	7.58	28.07	7.16	36.54

Low-rank approximation for representing the weight parameters in neural network has been studied by [33], [7], [34]. The benefits from low-rank approximation are reducing the number of parameters as well as the running time during the training and inference stage. Novikov et al. [8] replaced the weight matrix in the convolutional neural network (CNN) final layer with Tensor-Train [18](TT) format. Tjandra et al. [9] and Yang et al. [35] utilized the TT-format to represent the RNN weight matrices. Based on the empirical results, TT-format are able to reduce the number of parameters significantly and retain the model performance at the same time. Recent work from [36] used block decompositions to represent the RNN weight matrices.

Besides the tensor train, there are several tensor decomposition methods that are also popular such as CP and Tucker decomposition. However, both the CP and the Tucker decomposition have not yet been explored for compressing the RNN model. In this paper, we utilized the CP and the Tucker decomposition to compress RNN weight matrices. We also compared the performances between the CP, Tucker and TT format by varying the number of parameters at the same task.

VI. Conclusion

In this work, we presented some alternatives for compressing RNN parameters with tensor decomposition methods. Specifically, we utilized CP-decomposition and Tucker decomposition to represent the weight matrices. For the experiment, we run our experiment on polyphonic music dataset with uncompressed GRU model and three tensor-based RNN models (CP-GRU, Tucker-GRU and TT-GRU). We compare the performance of between all tensor-based RNNs under various number of parameters. Based on our experiment results, we conclude that TT-GRU has better performances compared to other methods under the same number of parameters.

Acknowledgment

Part of this work was supported by JSPS KAKENHI Grant Numbers JP17H06101 and JP17K00237. The authors would like to thank Shiori Yamaguchi for helpful discussions and comments.

References

[1] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 6645–6649.

[2] A. Graves, N. Jaitly, and A.-r. Mohamed, "Hybrid speech recognition with deep bidirectional LSTM," in *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*. IEEE, 2013, pp. 273–278.

[3] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[4] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[5] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[6] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[7] M. Denil, B. Shakibi, L. Dinh, N. de Freitas *et al.*, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems*, 2013, pp. 2148–2156.

[8] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 442–450.

[9] A. Tjandra, S. Sakti, and S. Nakamura, "Compressing recurrent neural network with tensor train," in *Neural Networks (IJCNN), 2017 International Joint Conference on*. IEEE, 2017, pp. 4451–4458.

[10] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[11] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Neural Networks, IEEE Transactions on*, vol. 5, no. 2, pp. 157–166, 1994.

[12] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, “Gradient flow in recurrent nets: the difficulty of learning long-term dependencies,” 2001.

[13] J. Martens and I. Sutskever, “Learning recurrent neural networks with Hessian-free optimization,” in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 1033–1040.

[14] Q. V. Le, N. Jaitly, and G. E. Hinton, “A simple way to initialize recurrent networks of rectified linear units,” *arXiv preprint arXiv:1504.00941*, 2015.

[15] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[16] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.

[17] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *arXiv preprint arXiv:1412.3555*, 2014.

[18] I. V. Oseledets, “Tensor-train decomposition,” *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011. [Online]. Available: http://dx.doi.org/10.1137/090752286

[19] R. A. Harshman, “Foundations of the parafac procedure: Models and conditions for an” explanatory” multimodal factor analysis,” 1970.

[20] H. A. Kiers, “Towards a standardized notation and terminology in multiway analysis,” *Journal of chemometrics*, vol. 14, no. 3, pp. 105–122, 2000.

[21] T. G. Kolda and B. W. Bader, “Tensor decompositions and applications,” *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[22] L. R. Tucker, “Some mathematical notes on three-mode factor analysis,” *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[23] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” in *International Conference on Machine Learning*, 2013, pp. 1310–1318.

[24] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS’10). Society for Artificial Intelligence and Statistics*, 2010.

[25] N. Boulanger-lewandowski, Y. Bengio, and P. Vincent, “Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription,” in *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, J. Langford and J. Pineau, Eds. New York, NY, USA: ACM, 2012, pp. 1159–1166. [Online]. Available: http://icml.cc/2012/papers/590.pdf

[26] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models,” in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*.

[27] M. Bay, A. F. Ehmann, and J. S. Downie, “Evaluation of multiple-f0 estimation and tracking systems.” in *2009 International Society for Music Information Retrieval Conference (ISMIR)*, 2009, pp. 315–320.

[28] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[29] M. Courbariaux, J.-P. David, and Y. Bengio, “Training deep neural networks with low precision multiplications,” *arXiv preprint arXiv:1412.7024*, 2014.

[30] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, 2015, pp. 1737–1746.

[31] Z. Tang, D. Wang, and Z. Zhang, “Recurrent neural network training with dark knowledge transfer,” in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2016, pp. 5900–5904.

[32] Y. Kim and A. M. Rush, “Sequence-level knowledge distillation,” in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016, pp. 1317–1327.

[33] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, “Low-rank matrix factorization for deep neural network training with high-dimensional output targets,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 6655–6659.

[34] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, “Exploiting linear structure within convolutional networks for efficient evaluation,” in *Advances in neural information processing systems*, 2014, pp. 1269–1277.

[35] Y. Yang, D. Krompass, and V. Tresp, “Tensor-train recurrent neural networks for video classification,” in *International Conference on Machine Learning*, 2017, pp. 3891–3900.

[36] J. Ye, L. Wang, G. Li, D. Chen, S. Zhe, X. Chu, and Z. Xu, “Learning compact recurrent neural networks with block-term tensor decomposition,” *arXiv preprint arXiv:1712.05134*, 2017.