Improving the Performance of Unitary Recurrent Networks and Applying Them to the Automatic Text Understanding Problem

Ivan Ivanov

under the direction of
Li Jing
Massachusetts Institute of Technology

Research Science Institute
August 2, 2017

Abstract

During a prolonged time execution, deep recurrent neural networks suffer from the so-called *long-term dependency problem* due to their recurrent connection. Although Long Short-Term Memory (LSTM) networks provide a temporary solution to this problem, they have inferior long-term memory capabilities which limit their applications. We use a recent approach for a recurrent neural network model implementing a unitary matrix in its recurrent connection to deal with long-term dependencies, without affecting its memory abilities. The model is capable of high technical results, but due to insufficient implementation does not achieve the expected performance. We optimize the implementation and architecture of the model, achieving time performance up to 5 times better than the original implementation. Additionally, we apply our improved model to the automatic text understanding task and outperform the widely used LSTM model.

Summary

Simulating human thinking on computer systems by means of mathematical models is one of the most challenging problems in the field of Computer Science. A recently developed such model is the artificial neural network, inspired by the neuron structure in the human brain. A limiting factor for the performance of this model is the depth of neural connections that can be established while retaining *learning ability*. We attempt to remove this limitation in the field of automated reading comprehension using a recent neural network model known for achieving high theoretical efficiency. We further optimize the implementation and architecture of that model and exhibit speed improvement up to 5 times better than the original implementation. In addition, we use our improved model for real time text analysis and achieve higher performance than the current state-of-the-art model.

1 Introduction

Human intelligence was recently surpassed by computers in the case of Google's DeepMind AlphaGo program winning against the best human players in the game of Go [1]. Considering the complexity of the game, which unlike chess cannot be approached by the *brute force* method, this achievement displays the potential of artificial intelligence. The algorithm used by the AlphaGo program is predominantly based on the concept of artificial neural networks: robust artificial intelligence devices capable of solving complex problems in computer vision, speech recognition, and natural language processing. The notion of neural networks was coined due to their resemblance of the human nervous system (Figure 1).

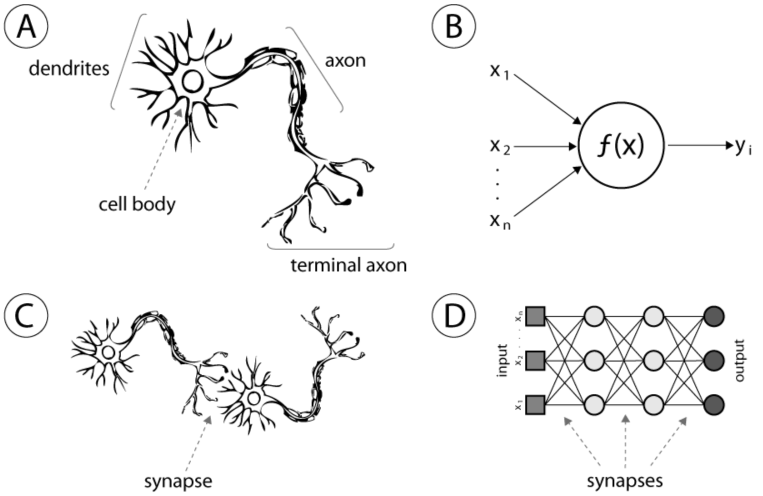
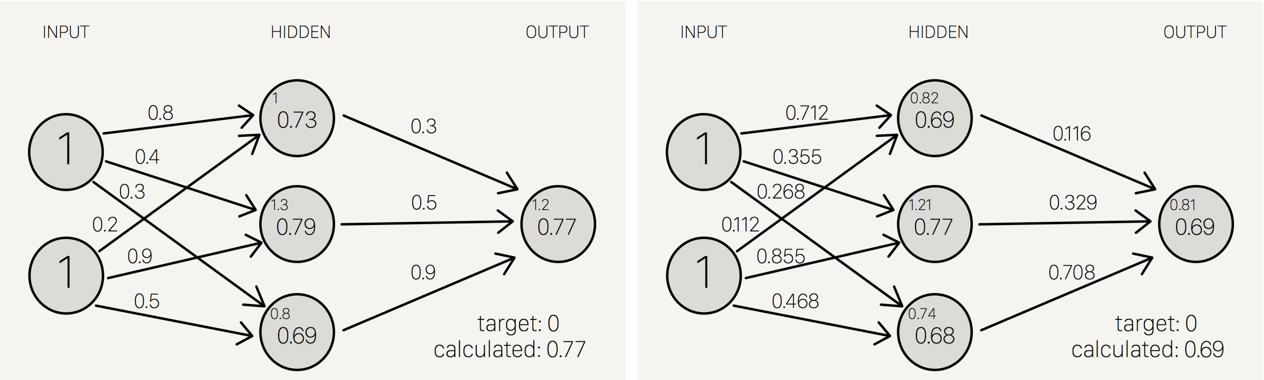


Figure 1: Analogy between the artificial neural network and the human nervous system [2]. Both A (a biology neural cell) and B (an artificial neuron) take several inputs and produce a single output based on a particular function. Additionally, C (a biological neural network) and D (an artificial neural network) illustrate the interneuronal connections highlighting their similar structure.

Artificial neural networks present one of the greatest and most important paradigms in programming [3]. Conventionally, a computer receives instructions what operations to perform and bigger problems are decomposed into basic operations. In contrast, the approach of artificial neural networks requires the computer to *learn* how to solve a problem after being presented observational data as input. The learning ability of artificial neural networks resides in the weighted connections between the different parts in their structure. These weights are altered during the process of learning, as shown on Figure 2.



(a) Initial weight values

(b) Updated weight values

Figure 2: A neural network learning mechanism [4]. The initially generated weight values are updated based on the discrepancy between the produced and expected result. Thus, when running the network with the same data as input, the produced result is closer to the expected one.

Artificial neural networks are organized in layers: the input, hidden, and output layer which respectively accept input data, process it, and store the produced result. For convenience, we present the interlayer connections in the form of matrices whose elements are the weight values. In a standard model of a network, input data is first presented to the input layer, then subjected to linear combination with the interlayer weight matrices, and at the end the final result is stored in the output layer. To avoid fitting to the input data, an additional non-linear *activation* function is applied to the pieces of data in each neuron cell as they that pass through it. During *training*, the produced result is compared to the expected one and the weight values are adjusted based on the discrepancy, so that the next time the network is presented with similar data as input, it would produce a more accurate result. This series of steps is repeated until the network achieves the desired accuracy. In the general case, to achieve sufficiently high accuracy on a particular task, a neural network must undergo many training iterations. For complex tasks such as speech recognition and natural language processing, the network should furthermore have a large number of weighted connections and ability to access past data. To provide the network with a *memory* ability, we introduce the recurrent connection which connects each hidden layer to itself (Figure 3).

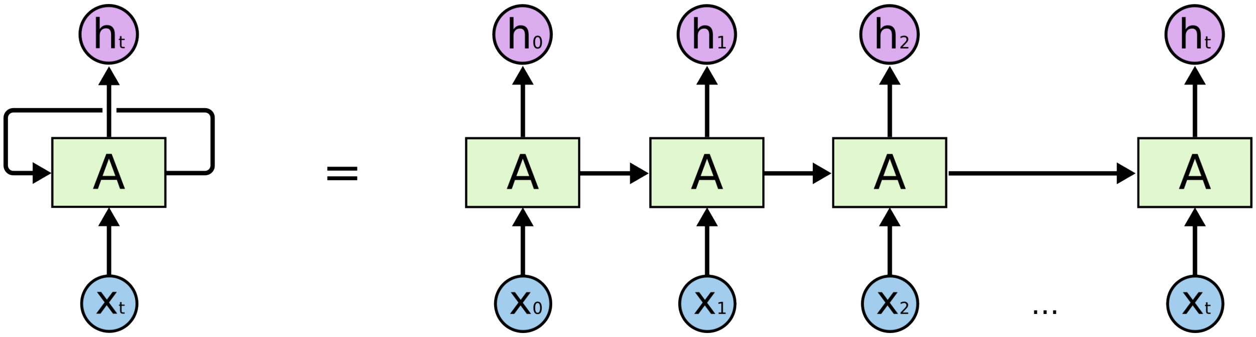


Figure 3: A recurrent neural network (RNN) model and its representation through time [5]. The input layer is marked with x, the output layer is marked with h, the hidden layer (a single one in this case) is marked with A, and the moment in time is marked by t.

If used for a prolonged period of time, the gap between the first and the last neuron cells increases which prevents the network from accurately connecting pieces of previous information and impairs its learning ability. This is known as the *long-term dependency problem*, also referred to in literature as the *exploding or vanishing gradient problem* [6]. Common attempts aiming to solve the problem have decreased the computational time of the model, but imposed serious limitations on the memory ability of the network.

An approach proposed by Jing et al. in 2016 manages to avoid the problem, without affecting the memory ability of the network, by keeping the recurrent connection weight matrix in a unitary state [7]. However, the implementation of Jing et al. does not meet their theoretically expected performance due to low degree of parallelizability. This paper proposes changes to the algorithms by Jing et al. that provide an equivalent computation but are highly parallelizable. Section 2 presents current common approaches dealing with the long-term dependencies problem, and explains in greater detail the unitary RNN model we are optimizing. In Section 3, we elaborate on the performed optimizations. In Section 4, we provide standard benchmarks of our improved implementation. In Section 5, we apply our improved implementation to the automatic text understanding task and analyze the achieved results.

2 Background

In this section, we discuss the previous research in the field of recurrent neural networks (RNNs), presenting the gated and unitary models.

2.1 Gated Recurrent Neural Networks

The Long Short-Term Memory (LSTM) RNN model is a solution to the long-term dependencies problem, first proposed by Hochreiter et al. [8]. The approach introduces three additional structures in the RNN cells called *gates* (Figure 4).

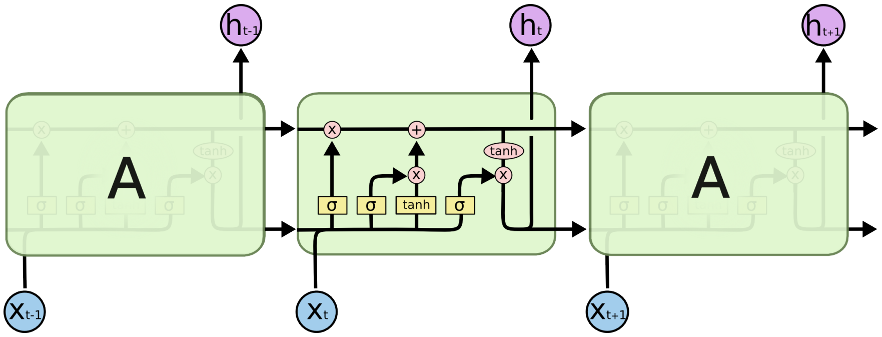


Figure 4: Model of a LSTM network [5]. From left to right, the three rectangles inside the center cell marked by the σ signs show the input, forget, and output gates.

The gates usually utilize the sigmoid function (producing values between 0 and 1) to filter the amount of data flowing through the cell and deal with the long-term dependencies. However, this method also limits memory ability of the network, which in turn lowers its accuracy on important computational tasks [5]. Therefore, LSTM networks present only a partial solution to the long-term dependencies problem because by trading computational intensity they lower their working accuracy.

2.2 The Concept of the Unitary Matrix

A RNN model dealing with the problem that keeps its recurrent connection weight matrix in a unitary state is first proposed by Arjovsky et al. [9]. The approach is based on the fact that all eigenvalues of unitary matrices have absolute values of unity and therefore can be safely raised to large powers, handling long-term dependencies. The main challenge faced by the RNN model is finding an efficient method to retain its unitarity throughout the training process. In the solution proposed by Arjovsky et al., the recurrent connection weight matrix uses only $\mathcal{O}(N)$ parameters which spans an insufficient part of the $\mathcal{O}(N^2)$-dimenstional space of $N \times N$ unitary matrices. This imposes serious constraints on the recurrent connection and limits the learning abilities of the network. In attempt to resolve this, a full-space coverage method is proposed by Wisdom et al. [10]. In Wisdom et al.'s RNN model, the recurrent connection weight matrix can cover the whole space of unitary matrices, but at the expense of N-dimenstional matrix multiplication resulting in computational complexity of $\mathcal{O}(N^3)$.

A mediate solution expanding the operational space of the recurrent connection weight matrix with minor increase in the complexity is proposed by Jing et al. [7]. Jing et al.'s RNN model uses two decompositions of the unitary recurrent connection weight matrix into 2×2 rotation matrices arranged in block diagonal matrices (Figure 5). Each block diagonal matrix is further simplified into two vectors representative of the weighted connections: one containing the elements from the diagonal, and a second containing the remaining non-zero elements. These vectors are multiplied element by element with the data vector and produce the end result of the connection.

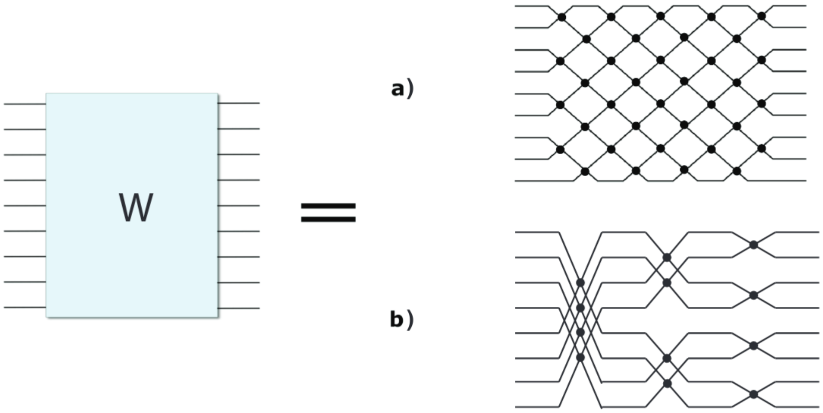


Figure 5: Jing et al.'s unitary model consists of two full-space coverage decompositions of the unitary matrix in terms of 2×2 rotation matrices marked by the points of intersection. The *simple net* model (a) presents a rectangular arrangement consisting of two repeating patterns of columns. The *lightweight* model (b) is an optimized version of the simple net model using the Fast Fourier Transformation (FFT) decomposition method [7].

Even though Jing et al.'s model uses full-space coverage and operation optimization, it is not efficiently parallelized. As a result, its original implementation does not match the theoretically predicted high performance. In the current research, our goal is to propose step-wise refinements and enrichments of the implementation towards the theoretical efficiency of model.

3 Optimizations

In the process of improving the implementation, we focus on the following aspects: parallelizing serial procedures, replacing memory and time consuming operations with lighter alternatives, and expanding the range of acceptable input parameters.

3.1 Implementing Parallelization

Our first step in improving the implementation is to turn the serial code into a parallel one. The idea of parallelization finds increasing use in neural network algorithms because the neurons in a single layer operate independently of one another. For parallelization, we use the TensorFlow library as it is developed especially for devices with multiple threads such as GPUs, and all of its functions are implemented to run in parallel [11]. As its main data type the library uses the tensor data structure. Mathematically, a tensor is defined as a multilinear function; however, in the context of TensorFlow it can be viewed as a multidimensional array.

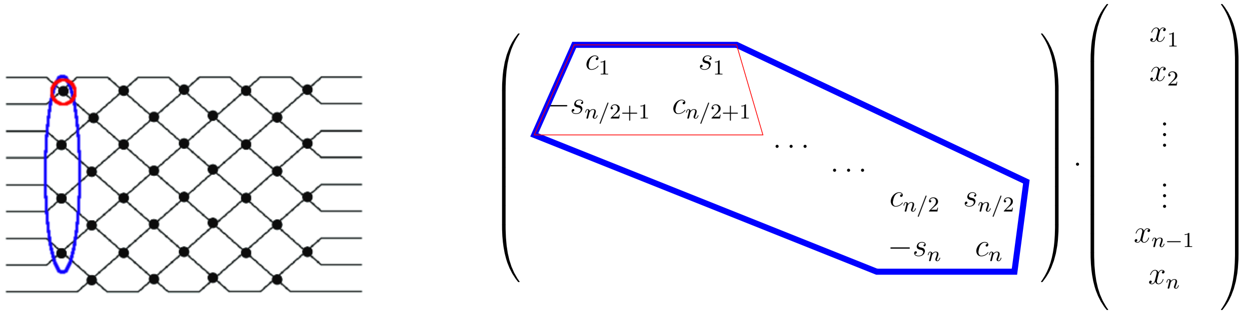
In the original implementation, the construction of the block diagonal matrices and their efficient multiplication (Figure 6 and Figure 7), two important operational optimizations of the algorithm, requires the rotation matrices and the data vector to be permuted based on the hyperparameters of the RNN cell. A downside of this approach is that the generation of the necessary permutations for this operation is implemented via interdependent `for` loops which cannot be parallelized efficiently.

For the simple net decomposition model, we generate the block diagonal matrices as presented on Figure 6b and perform the optimized multiplication method from the original approach as shown on Figure 6d and Figure 6c. Based on the visual representations of the matrices on Figure 6b and Figure 6c, we reorder the vectors according to Figure 6d. Therefore, based on the order of the vectors presented on Figure 6d, we can derive the general form of the required permutations as:

Data vector shuffle pattern
(applied to x_2) $: (1, 0, 3, 2, \dots, 2k-3, 2k-4, 2k-1, 2k-2)$

Rotation matrices shuffle pattern
(applied both to v_1 and v_2) $: (0, k, 1, k+1, \dots, k-2, 2k-2, k-1, 2k-1)$

For efficient parallelization of the generation of the permutations, we propose a method replacing the hardly parallelizable `for` loops with reshaping, reversing, and transposing tensor operations which can be divided in independent parts and therefore easily parallelized. For the data vector permutation generation, we reshape the initial array into two columns, reverse the columns, and reshape the array back into a single row (1). For the rotation matrices permutation generation, we reshape the initial array into two rows, transpose the array,



(a)

$$\begin{pmatrix} c_1 & s_1 & & & \\ -s_{n/2+1} & c_{n/2+1} & & & \\ & & \dots & & \\ & & & \dots & \\ & & & c_{n/2} & s_{n/2} \\ & & & -s_n & c_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}$$

(b)

$$\begin{pmatrix} x_1 c_1 + x_2 s_1 \\ x_2 c_{n/2+1} - x_1 s_{n/2+1} \\ \vdots \\ \vdots \\ x_{n-1} c_{n/2} + x_n s_{n/2} \\ x_n c_n - x_{n-1} s_n \end{pmatrix}$$

(c)

$$\begin{aligned} &\mathbf{F}x = v_1 * x_1 + v_2 * x_2 \\ &v_1 = (c_1, c_{n/2+1}, \dots, c_{n/2}, c_n) \\ &v_2 = (s_1, -s_{n/2+1}, \dots, s_{n/2}, s_n) \\ &x_1 = (x_1, x_2, \dots, x_{n-1}, x_n) \\ &x_2 = (x_2, x_1, \dots, x_n, x_{2n-1}) \end{aligned}$$

(d)

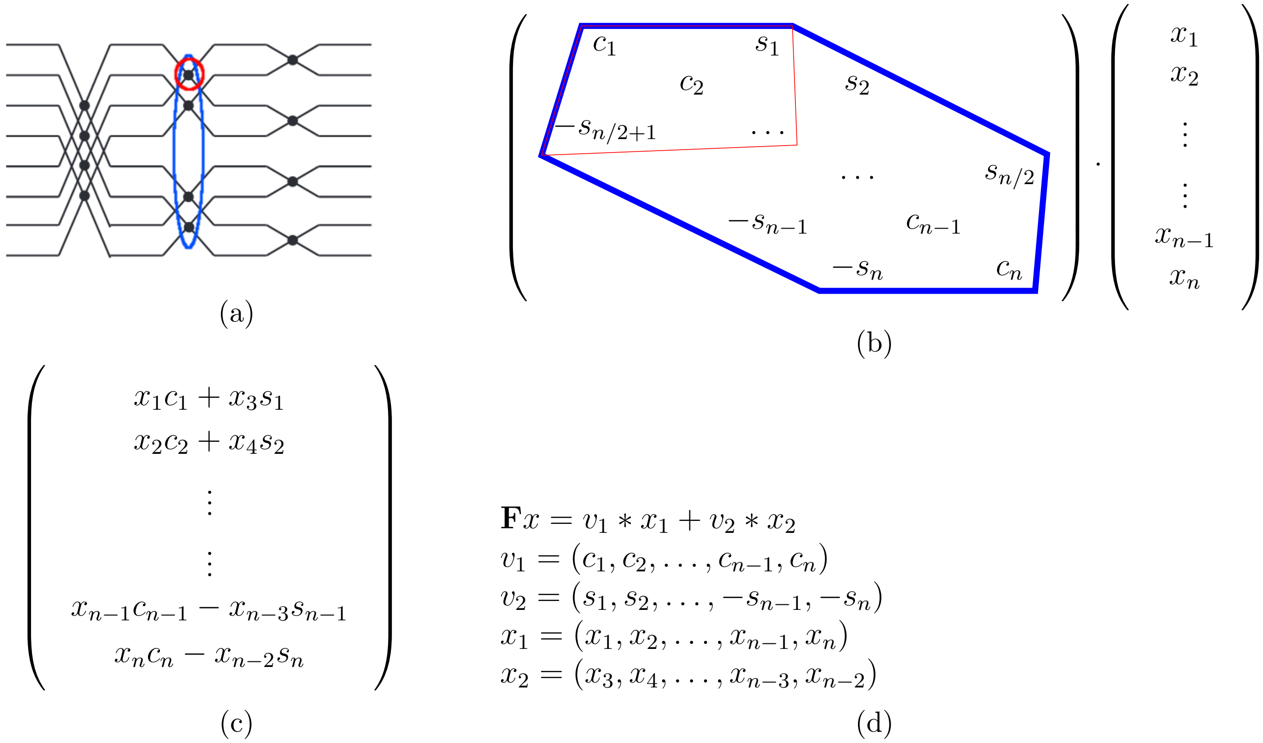
Figure 6: The simple net decomposition model of the unitary matrix [7]. Each point of intersection in (a) is a 2×2 rotation matrix. The rotation matrices are grouped in columns and each column presents the block diagonal matrix in (b). The multiplication process of this matrix with a data vector is illustrated on (b). The result of the multiplication is presented on (c). The mathematical formula for the optimized multiplication together with the four vector structures are given in (d). Based on the result (c) and the vector structures (d) we can determine the general form of the required permutations.

and reshape it back into a single row (2).

$$[a_0, a_1, \dots, a_{2k-2}, a_{2k-1}] \mapsto \begin{bmatrix} a_0 & a_1 \\ \vdots & \vdots \\ a_{2k-2} & a_{2k-1} \end{bmatrix} \mapsto \begin{bmatrix} a_1 & a_0 \\ \vdots & \vdots \\ a_{2k-1} & a_{2k-2} \end{bmatrix} \mapsto [a_1, a_0, \dots, a_{2k-1}, a_{2k-2}] \tag{1}$$

$$[a_0, a_1, \dots, a_{2k-2}, a_{2k-1}] \mapsto \begin{bmatrix} a_0 & \dots & a_{k-1} \\ a_k & \dots & a_{2k-1} \end{bmatrix} \mapsto \begin{bmatrix} a_0 & a_k \\ \vdots & \vdots \\ a_{k-1} & a_{2k-2} \end{bmatrix} \mapsto [a_0, a_k, \dots, a_{k-1}, a_{2k-1}] \tag{2}$$

For the lightweight decomposition model, we generate the block diagonal matrices as



(a)

$$\begin{pmatrix} c_1 & & s_1 & & & \\ & c_2 & & s_2 & & \\ -s_{n/2+1} & & \ldots & & & \\ & & & \ldots & & s_{n/2} \\ & & -s_{n-1} & & c_{n-1} & \\ & & & -s_n & & c_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}$$

(b)

$$\begin{pmatrix} x_1c_1 + x_3s_1 \\ x_2c_2 + x_4s_2 \\ \vdots \\ \vdots \\ x_{n-1}c_{n-1} - x_{n-3}s_{n-1} \\ x_nc_n - x_{n-2}s_n \end{pmatrix}$$

(c)

$$\begin{aligned} &\mathbf{F}x = v_1 * x_1 + v_2 * x_2 \\ &v_1 = (c_1, c_2, \ldots, c_{n-1}, c_n) \\ &v_2 = (s_1, s_2, \ldots, -s_{n-1}, -s_n) \\ &x_1 = (x_1, x_2, \ldots, x_{n-1}, x_n) \\ &x_2 = (x_3, x_4, \ldots, x_{n-3}, x_{n-2}) \end{aligned}$$

(d)

Figure 7: The lightweight decomposition model of the unitary matrix [7]. Each point of intersection in (a) is a 2 × 2 rotation matrix. The rotation matrices are again grouped in columns and each column presents the block diagonal matrix in (b). The multiplication process of this matrix with a data vector is illustrated on (b). The result of the multiplication is presented on (c). The mathematical formula for the optimized multiplication together with the four vector structures are given in (d). Based on the result (c) and the vector structures (d) we can determine the general form of the required permutations.

presented on Figure 7b and perform the optimized multiplication method from the original approach as shown on Figure 7d and Figure 7c. Based on the visual representations of the matrices on Figure 7b and Figure 7c, we reorder the vectors as illustrated on Figure 7d. Therefore, based on the order of the vectors presented on Figure 7d, we can derive the general form of the required permutations as:

Data vector shuffle pattern
(applied to x_2)

$$:(k, \dots, 2k-1, 0, \dots, k-1) \text{ for } s = 0$$
$$(k/2, \dots, k, 0, \dots, 2k-1, k/2, \dots, 3k/2-1) \text{ for } s = 1$$
$$\dots$$
$$(1, 0, \dots, 2k-1, 2k-2) \text{ for } s = log_2 s - 1$$

Rotation matrices shuffle pattern
(applied to v_1 and v_2)

$$:(0, 1, \dots, 2k-2, 2k-1) \text{ for } s = 0$$
$$(0, 2, \dots, 2k-2, 1, 3, \dots, 2k-3, 2k-1) \text{ for } s = 1$$
$$\dots$$
$$(0, k, \dots, k-1, 2k-1) \text{ for } s = log_2 s - 1$$

Based on the similarity between the general forms of the permutations used in the simple net decomposition model and the lightweight decomposition model, we use similar strategy for implementing the parallelization scheme of the latter. For the generation of the data vector permutation, we reshape the initial array into two columns whose elements are arrays of length $x = N/2^{s+1}$, depending on the column number s. Then, we reverse these columns and reshape the array back to its original shape (3). For the generation of the second permutation, we reshape the initial array into 2^s rows depending on the column number s. Then, we transpose the array and reshape it back to its original shape (4).

$$\begin{aligned}
[a_0, a_1, \dots, a_{2k-2}, a_{2k-1}] &\mapsto \begin{bmatrix} [a_0, \dots, a_{x-1}] & [a_x, \dots, a_{2x}] \\ \vdots & \vdots \\ [a_{2k-3x-1}, \dots, a_{2k-2x-1}] & [a_{2k-2x}, \dots, a_{2k-1}] \end{bmatrix} \mapsto \\
&\mapsto \begin{bmatrix} [a_x, \dots, a_{2x}] & [a_0, \dots, a_{x-1}] \\ \vdots & \vdots \\ [a_{2k-2x}, \dots, a_{2k-1}] & [a_{2k-3x-1}, \dots, a_{2k-2x-1}] \end{bmatrix} \mapsto \\
&\mapsto [a_x, \dots, a_{2x}, a_0 \dots, a_{x-1}, \dots, a_{2k-3x-1}, \dots, a_{2k-2x-1}]
\end{aligned} \tag{3}$$

$$
\begin{aligned}
[a_0, a_1, \ldots, a_{2k-2}, a_{2k-1}] &\mapsto \begin{bmatrix} a_0 & a_1 & \ldots & a_{d-2} & a_{d-1} \\ a_d & & \ldots & & a_{2d-1} \\ \vdots & & & & \vdots \\ a_{2k-2d-1} & & \ldots & & a_{2k-d-2} \\ a_{2k-d-1} & a_{2k-d} & \ldots & a_{2k-2} & a_{2k-1} \end{bmatrix} \mapsto \\
&\mapsto \begin{bmatrix} a_0 & a_d & \ldots & a_{2k-2d-1} & a_{2k-d-1} \\ a_1 & & \ldots & & a_{2k-d} \\ \vdots & & & & \vdots \\ a_{d-2} & & \ldots & & a_{2k-2} \\ a_{d-1} & a_{2d-1} & \ldots & a_{2k-2-d} & a_{2k-1} \end{bmatrix} \mapsto \\
&\mapsto [a_0, a_d, \ldots, a_{2k-d-1}, a_1, a_{d+1} \ldots, a_{d-1}, a_{2d-1}, \ldots, a_{2k-1}]
\end{aligned} \tag{4}
$$

Hence, we have transformed the inefficient serial generation of permutations, necessary for the approach, into highly parallelizable TensorFlow operations which accomplishes our goal of implementing a parallelization architecture. A comparison between the original implementation and our improved implementation is presented in Appendix A.

3.2 Reducing Operations Complexity

Even though we optimized the generation of the template permutations, this process is executed only once during the initialization of the RNN model. As a result, its optimization alone would not lead to noticeable improvements in the long-run operation of the network. However, our improvements can prove extremely useful in the permutation process of the rotation matrices and data vectors executed multiple times for each RNN cell during a single iteration. In the original implementation, the permutation of the rotation matrices and data vectors is accomplished through the use of memory and time consuming functions such as the *gather* function used for rearranging a given tensor according to a particular permutation. As the function is designed for operation in the general case with an arbitrary permutation as parameter, it employs additional resources which increase the runtime memory consumption of the implementation without accounting for sufficient performance change. For example, in its working process the *gather* function stores multiple copies of each the different layers of the tensor object which significantly increases the memory usage especially if large tensor objects are used. The context in which the operation is used is presented in Algorithm 1.

Algorithm 1 Original *permute* function

```
Input: update vector $x$, permutation $ind$
Output: shuffled update vector $step3$
  function PERMUTE($x, ind$)
      step1 ← transpose($x$)
      step2 ← gather(step1,$ind$)
      step3 ← transpose(step2)
      return step3
  end function
```

Algorithm 2 Our *permute* function

```
Input: update vector $x$
Output: shuffled update vector $x$
  function PERMUTE($x$)
      x ← reshape(x,[-1,2])
      x ← transpose(x,1)
      x ← reshape(x,[-1])
      return x
  end function
```

Using our previously developed methods for permutation generation, we can completely avoid the *gather* function by applying the permute operations directly to the tensor object instead of to additional arrays, as shown in Algorithm 2. Therefore by introducing this method we accomplish our goal of reducing the complexity of used operations.

3.3 Expanding Hyperparameter Range

As one of the goals of this research is to apply the designed model in practice, the provided RNN implementation should be compatible with a wide variety of hyperparameters. However, the original implementation has significant limitations on its range of parameters. In particular, the simple net decomposition model was defined only for even number of columns and even hidden layer size and the lightweight decomposition model was only applicable for hidden layer sizes which are powers of two. These are important architecture problems which limit the possible configurations of the RNN cell and our last contribution is to improve the architecture of Jing et al.'s model and expand its range of acceptable hyperparameters. This enhancement will not lead to better performance results, but will make the model more customizable and therefore more applicable in the real world.

Our approach for expanding the range of acceptable hyperparameters focuses on changing the arrangement of the rotation matrices in the block diagonal matrices based on the hyperparameters characteristics. Thus, for an odd hidden layer size, or odd number of columns, the rotation matrices in some of the block diagonal matrices shift their positions up, or down, leaving an empty row, or column. Then, upon multiplication with the data vector, the empty rows of these matrices are added to the non-empty rows of the remaining matrices and form the final product. For expanding the range of acceptable hyperparameters for the lightweight decomposition model, we applied Genz et al.'s method for generating random orthogonal matrices which is used by Jing et al. for the lightweight decomposition model itself [12]. The method consists of adding an extra column (block diagonal matrix) to the decomposition model for the additional rotation matrices and filling its empty cells with control values of 0 and 1. Then upon, multiplication the control values add to the rotation matrices values in the other block diagonal matrices and produce the final product.

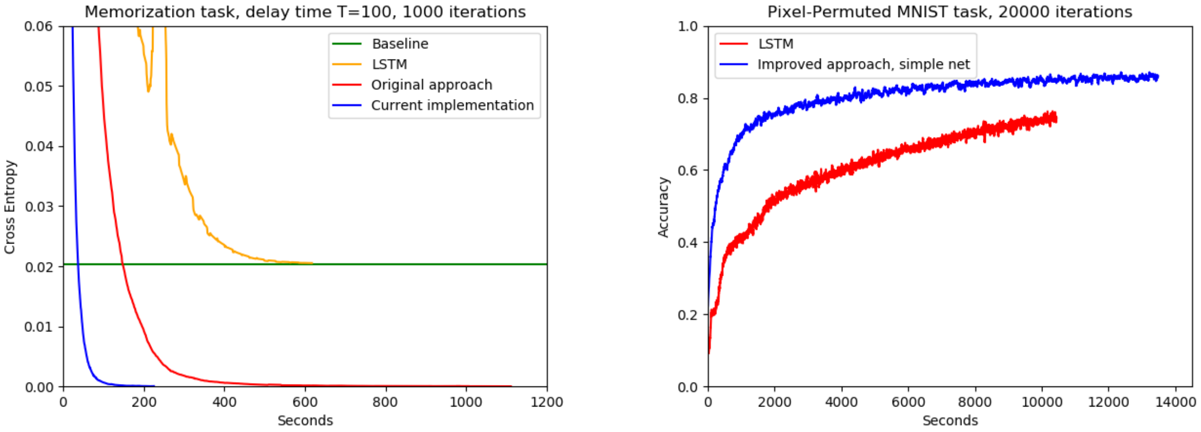
Although we made architecture improvements expanding the hyperparameter range of the lightweight decomposition, we have increased the computational time of the RNN model. This is a problem we would like to address in the future by finding a more effective way for expanding the hyperparameter range.

4 Results and Analysis

In this section, we test implementation of Jing et al.'s model on two standard benchmarking tasks for RNNs, comparing its performance to the original implementation, as well as to other commonly used RNN models. In the end, we include a brief analysis of the results. We also provide a TensorFlow implementation of our improved model available on `https://github.com/vanjo9800/EUNN-tensorflow`.

The first benchmarking task we use is the memorization task, defined by [8], [9], and [13].

It tests the basic ability of the RNN to recall information presented T steps earlier in time. For better representation of the learning behavior, we draw a baseline marking the *memoryless* strategy. The second benchmarking task upon which our improved implementation is tested is the pixel-permuted MNIST task which assesses the learning ability of the network in greater detail. The testing process consists of feeding the network with pixel-by-pixel handwritten digit samples from the MNIST dataset and determining the answer based on the probability distribution, quantifying the digit prediction returned at the end. For creating more long-range patterns and therefore higher complexity, we additionally apply a fixed permutation to each sample before feeding it into the network.



(a) Results of the two unitary implementations and the LSTM model on the memorization task performed with a decay rate of 0.5 and learning rate of 0.001. The cross entropy function marks the loss experienced by the RNN model; the lower the loss, the better is the learning performance of the model.

(b) Results of the LSTM model and our improved implementation on the pixel-permuted MNIST task performed with a decay rate of 0.9 and learning rate of 0.0001.

On the memorization benchmark, both unitary implementations express sufficient learning behavior beating the baseline. Our improved implementation outperforms the original one by a factor of 5 in terms of execution time (Figure 8a). In particular cases during the process of testing, the time performance improvement reached up to 12 times better than the original implementation. In the general case, our enhanced implementation performs at least 20% faster than the original implementation for all test cases. Compared with the LSTM model, our improved unitary implementation performs better both in terms of time and accuracy, as the LSTM network barely reaches the baseline and is not able to beat it. Additionally, tests of our improved implementation in an environment under load reveal stable execution time making our implementation of the unitary model highly applicable in academic environments where multiple algorithms are often executed on a single machine. This behavior can be explained by the replacement of the previously used *gather* function which reduced the cache memory dependency of our improved implementation.

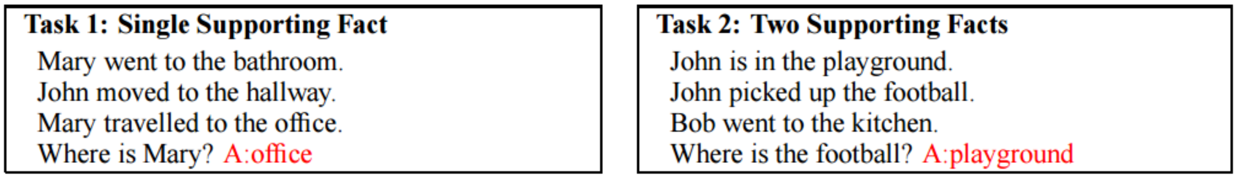
On the pixel-permuted MNIST task, our unitary model implementation achieves higher accuracy compared to the LSTM model from the early iterations and later converges to its maximum accuracy of around 80%. The LSTM model implementation demonstrates slower learning pattern and is not able to reach the accuracy of our unitary model implementation for the given time interval. Therefore, our improved unitary model implementation is considered more efficient and should present similar results when executed in practice.

5 Reading Comprehension Performance

This section evaluates the new implementation with this example task of automatic text understanding and reasoning.

5.1 The bAbI Dataset

As a proof of concept for the performance of the improved model on the automatic text understanding task, we run our enhanced implementation on the bAbI dataset provided by Facebook [14]. It consists of 150 words used in 20 different tasks for automatic text understanding and reasoning each testing different capabilities of the RNN model.



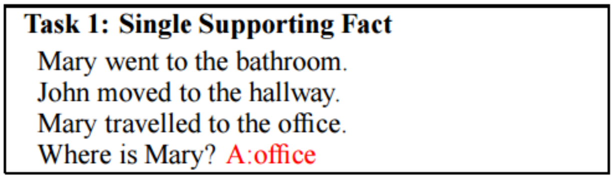
Task 1: Single Supporting Fact
Mary went to the bathroom.
John moved to the hallway.
Mary travelled to the office.
Where is Mary? A:office

Task 2: Two Supporting Facts
John is in the playground.
John picked up the football.
Bob went to the kitchen.
Where is the football? A:playground

Figure 9: Task 1 and 2 from the bAbI dataset [14].

5.2 The Testing Process

As the automatic text understanding task is more complicated than the memorization and pixel-permuted MNIST benchmarks, certain preparation of the input data is required before its insertion into the RNN (Figure 10 and 11).



Task 1: Single Supporting Fact
Mary went to the bathroom.
John moved to the hallway.
Mary travelled to the office.
Where is Mary? A:office

(a) A sample test from Task 1 in the bAbI dataset [14].

Mary	went	to	...	office	Where	is
0	1	2	...	9	10	11

(b) The dictionary of the used vocabulary.

$$\begin{tikzcd} \textit{Where is Mary?} \arrow[r] & (10,\ 11,\ 0) \arrow[d] \\ & ([0,0,\ldots,1,0],\ [0,0,\ldots,0,1],\ [1,0,\ldots,0,0]) \end{tikzcd}$$

(c) Mapping the question into the one-hot vector data type used by the network.

Figure 10: The words from the test case are inserted into a dictionary and each of them receives its unique numeric code. The statements are reduced into number arrays which are later transferred into one-hot vectors.

For maximizing the accuracy of the neural network, we have placed our model in Mostafa Samir's TensorFlow implementation of DeepMind's Differential Neural Computer (DNC) [15]. The DNC is a device that strengthens the memory and learning capabilities of a RNN model by memorizing particular states of the network and retrieving them when needed. The implementation originally uses the standard TensorFlow implementation of an LSTM RNN cell, but due to the compatibility of our implementation, we are able to replace the LSTM cell and conduct the tests with our model.

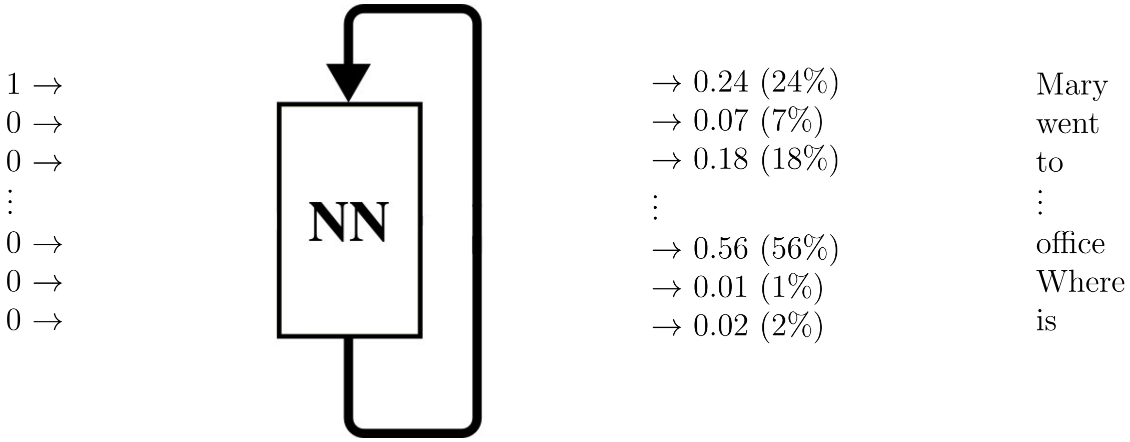


Figure 11: The one-hot vectors are fed consecutively into the RNN and at each iteration an output vector containing a probability distribution, quantifying the word index in the created dictionary is produced. When the input vector codes a question, the word with the biggest probability is returned.

5.3 Tests

Task	Our Approach	LSTM
1 - Single Supporting Fact	50.5%	**52.0%**
2 - Two Supporting Facts	**31.8%**	15.1%
3 - Three Supporting Facts	**25.4%**	19.1%
4 - Two Arg. Relations	71.2%	**73.5%**
5 - Three Arg. Relations	**67.1%**	34.4%
6 - Yes/No Questions	**52.9%**	50.5%
7 - Counting	**71.3%**	56.5%
8 - Lists/Sets	**68.2%**	38.8%
9 - Simple Negation	61.8%	**63.8%**
10 - Indefinite Knowledge	**46.0%**	45.1%
11 - Basic Coreference	72.3%	**74.1%**
12 - Conjunction	73.4%	**76.1%**
13 - Compound Coreference	**94.0%**	83.0%
14 - Time Reasoning	**36.4%**	18.6%
15 - Basic Deduction	**55.0%**	21.2%
16 - Basic Induction	**48.8%**	32.2%
17 - Positional Reasoning	48.4%	**50.6%**
18 - Size Reasoning	**89.5%**	89.2%
19 - Path Finding	**7.9%**	6.6%
20 - Agents Motivations	**95.5%**	90.6%
Mean Performance	**58.4%**	49.6%

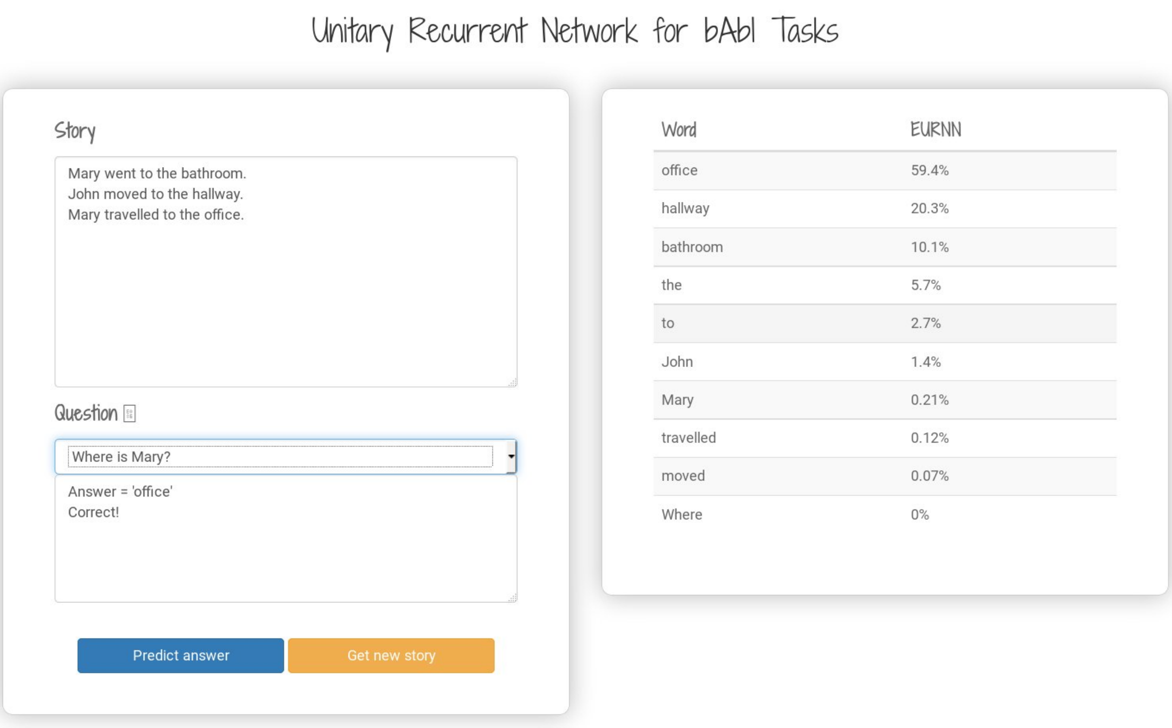
Table 1: Results of the LSTM implementation and our improved implementation on the bAbI dataset. The implementations are run with hidden layer size of 256 and our improved implementation uses the simple net decomposition model with 2 columns. As our goal is to use as little data as possible and achieve high performance, in the training procedure the implementations were presented only with the standard 10000 training samples.

The results show that our improved implementation performs better than the LSTM, achieving mean accuracy of over 50%. It reaches up to 2 times greater accuracy compared to the LSTM implementation on tasks requiring long-range pattern detection such as **Two/Three Supporting Facts, Three Arg. Relations, Basic Deduction, and Basic Induction** highlighting its enhanced long-term dependency management. Additionally, our unitary model implementation also outperforms the LSTM model implementation on the real-life search problem included in the **Path finding** task. On the remaining tasks, the two implementations share similar performance with the LSTM doing slightly better on tasks with low memory dependency and more factors affecting the answer.

The hyperparameters in the current environment consist of the batch size, the learning rate, the decay rate, the hidden layer size, the type of representation used by the unitary model, and the number of columns in the net if the simple net decomposition model is used, but due to the usage of the DNC architecture, only the last three affect the RNN cell. Additional tests with small adjustments in these three hyperparameters were performed and uncovered that the best performing configuration is the one presented in Table 1: hidden layer size of 256 and simple net decomposition model with 2 columns. In all configurations, our unitary model implementation achieves higher mean accuracy than the LSTM and for the unitary implementations the mean accuracy varies between 50% and 58.4%.

5.4 Web Application

For better visualization of the work of the RNN on the bAbI task, we have developed a web application solving data samples in real time. The application is implemented in Python and in addition to the answer of the task, presents the probability distribution as returned by the RNN, see Figure 12.



Unitary Recurrent Network for bAbI Tasks

Story

Mary went to the bathroom.
John moved to the hallway.
Mary travelled to the office.

Question

Where is Mary?

Answer = 'office'
Correct!

Word	EURNN
office	59.4%
hallway	20.3%
bathroom	10.1%
the	5.7%
to	2.7%
John	1.4%
Mary	0.21%
travelled	0.12%
moved	0.07%
Where	0%

Figure 12: A snapshot of the web application assessing the neural network model on the sample test case from Task #1 of the bAbI dataset.

6 Conclusion

We presented the optimizations conducted in the implementation of a novel recurrent network model and provided results of its application to the real-life task of automatic text understanding. Based on the results on the standard benchmarks, our implementation improves over the time performance of the original by at least 20%, reaching its best of 15 times better, reduces the runtime memory usage, and retains the accuracy of Jing et al.'s model. Based on the bAbI task results, the improved implementation achieves maximum mean accuracy of around 60% and in all of its hyperparameter configurations beats the commonly used LSTM model. This means that the unitary model can successfully be introduced to more real-life problems and replace some of the currently used RNN models.

As future work we plan on improving the structure of the lightweight decomposition model avoiding the increase in computational time, reducing even more the memory and time usage of TensorFlow operations by replacing them with ones from lower-rank, and applying our improved implementation to the problem of speech recognition which uses similar deep learning technique as automatic text understanding.

7 Acknowledgments

I would like to thank my mentor Li Jing for the guidance and support during the development of the project. I am also thankful to Rumen Dangovski, colleague of Li Jing, who also contributed with ideas for the research. I thank as well my tutor, Dr. Jenny Sendova, for revising my papers and giving me very important advice during the research program. I would like to thank Charles Tam, Andrew Jin, Stanislav Atanasov, Yusu Liu, Hristo Stoyanov, William McInroy, Harshal Sheth, and Kristian Georgiev for the edits and recommendations they gave me when reading my paper. I also express my gratitude to Milen Ferev for giving me computational resources. I am specially thankful to the RSI staff, CEE, and MIT for making RSI possible and supporting my research. Last, but not least, I would like to acknowledge Ms. Natalia Miteva, ”America for Bulgaria Foundation,” and Mr. Michael Tachev and Professor Dimo Platikanov, International Foundation ”Sts. Cyril & Methodius” , for their continuous support of the Bulgarian participation in RSI.

References

[1] S. R. Granter, A. H. Beck, and D. J. Papke Jr. Alphago, deep learning, and the future of the human microscopist. *Archives of Pathology & Laboratory Medicine*, 141(5):619–621, 2017.

[2] V. G. Maltarollo, K. M. Honório, and A. B. F. da Silva. Applications of artificial neural networks in chemical problems. In *Artificial neural networks-architectures and applications*. InTech, 2013.

[3] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2017.

[4] S. Miller. Mind: How to build a neural network (part one). `https://stevenmiller888.github.io/mind-how-to-build-a-neural-network/`. Accessed: 2017-07-28.

[5] C. Olah. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015.

[6] S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91, 1991.

[7] L. Jing, Y. Shen, T. Dubček, J. Peurifoy, S. Skirlo, M. Tegmark, and M. Soljačić. Tunable efficient unitary neural networks (eunn) and their application to rnn. *arXiv preprint arXiv:1612.05231*, 2016.

[8] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[9] M. Arjovsky, A. Shah, and Y. Bengio. Unitary evolution recurrent neural networks. pages 1120–1128, 2016.

[10] S. Wisdom, T. Powers, J. Hershey, J. Le Roux, and L. Atlas. Full-capacity unitary recurrent neural networks. pages 4880–4888, 2016.

[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, and C. C. et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[12] A. Genz. Methods for generating random orthogonal matrices. *Monte Carlo and Quasi-Monte Carlo Methods*, pages 199–213, 1998.

[13] M. Henaff, A. Szlam, and Y. LeCun. Orthogonal rnns and long-memory tasks. *arXiv preprint arXiv:1602.06662*, 2016.

[14] J. Weston, A. Bordes, S. Chopra, A. M. Rush, B. van Merriënboer, A. Joulin, and T. Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015.

[15] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

Appendix A Original Implementation vs. New Implementation

Algorithm 3 Original implementation for the simple net model permutations[7]

Input: hidden layer size H

Output: data vector shuffle pattern $ind1$, rotation matrices shuffle pattern $ind2$

```
  function genPermutation($H$)
      ind1 ← $\{0, \ldots, H-1\}$
      for i from 0 to $H-1$ do
          for even $i$, ind1[i]$-=$ 1
          for odd $i$, ind1[i]$+=$ 1
      end for
      ind2 ← $\{\}$
      for i from 0 to $H/2$ do
          ind2 $+= \{i, i+H/2\}$
      end for
      return ind1,ind2
  end function
```

Algorithm 4 New implementation for the simple net model permutations

Input: hidden layer size H

Output: data vector shuffle pattern $ind1$, rotation matrices shuffle pattern $ind2$

```
  function genPermutation($H$)
      ind1 ← $\{0, \ldots, H-1\}$
      ind1 ← reshape(ind1,[-1,2])
      ind1 ← reverse(ind1,[1])
      ind1 ← reshape(ind1,[-1])
      ind2 ← $\{0, \ldots, H-1\}$
      ind2 ← reshape(ind2,[2,-1])
      ind2 ← transpose(ind2,0)
      ind2 ← reshape(ind2,[-1])
      return ind1,ind2
  end function
```

Algorithm 5 Original implementation for the lightweight model permutations [7]

```
Input: column number, $s$, hidden layer size $H$
Output: data vector shuffle pattern $ind1$
  function dataPattern($t$)
      if t == 0 then: return [1,0]
      else
          ind1 ← $\{2^t,\ldots,2^{t+1}-1,0,\ldots,2^t-1\}$
          list1 ← dataPattern($t-1$,$H$)
          for i from 0 to t do:
              ind1 += $\{list1[i], list1[i]+2^t\}$
          end for
          return ind1
      end if
  end function
Input: column number $s$, hidden layer size $H$
Output: rotation matrices shuffle pattern $ind2$
  function matricesPattern($s$,$H$)
      for j from 0 to $2^s$ do
          ind2 += $\{j, j+2^s, \ldots, H\}$
      end for
      return ind2
  end function
```

Algorithm 6 New implementation for the lightweight model permutations

```
Input: column number $s$
Output: update vector shuffle pattern $ind1$, rotation matrices shuffle pattern $ind2$
  function genPermutation($s$)
      ind1 ← $\{0,\ldots,s-1\}$
      ind1 ← reshape(ind1,[-1,2,$N/(2^{s+1})$])
      ind1 ← reverse(ind1,[1])
      ind1 ← reshape(ind1,[-1])
      ind2 ← $\{0,\ldots,s-1\}$
      ind2 ← reshape(ind2,[$2^p$,-1])
      ind2 ← transpose(ind2,0)
      ind2 ← reshape(ind2,[-1])
      return ind1, ind2
  end function
```