Improving the Performance of Unitary Recurrent Neural Networks and Their Application in Real-life Tasks

Ivan Ivanov
under the guidance of Li Jing
Massachusetts Institute of Technology
ivan@vankata.tech

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 three common real-life problems: the automatic text understanding task, the speech recognition task, and cryptoanalysis, 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 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, speech recognition, and cryptoanalysis, and achieve higher performance than the current state-of-the-art model.

***Key words*:** neural network, recurrent, LSTM, unitary matrix, automatic text understaning, speech recognition, cryptography, vigenère

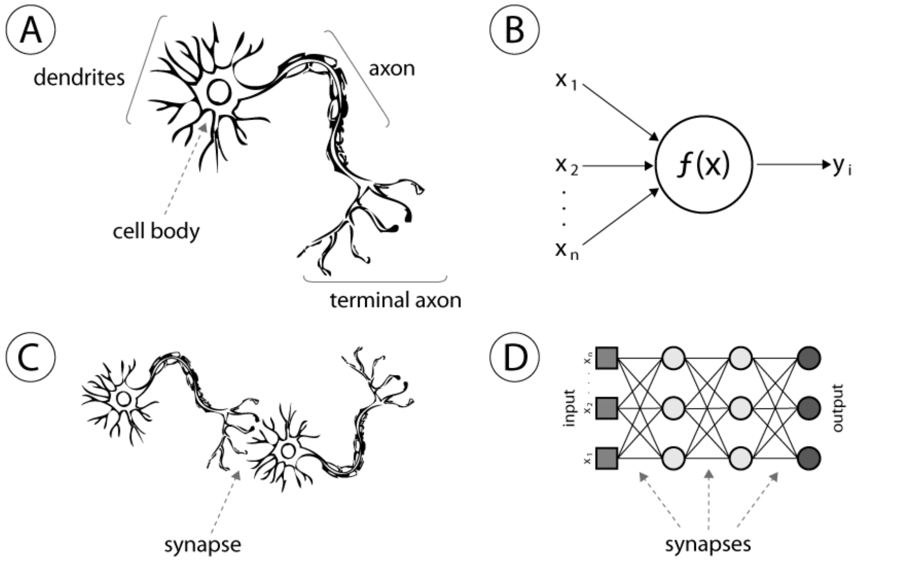
I. Introduction

Artificial neural networks are robust artificial intelligence devices capable of solving complex problems in computer vision, speech recognition, and natural language processing. They present one of the greatest and most important paradigms in programming [1]. Conventionally, a computer receives direct instructions what operations to perform, and complex problems are being 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 notion of neural networks was coined due to their resemblance of the human nervous system (Figure 1a). Their learning ability resides in the weighted connections between the different layers in their structure. After initialization, these weights are altered during the process of learning and generally improve the performance of the network.

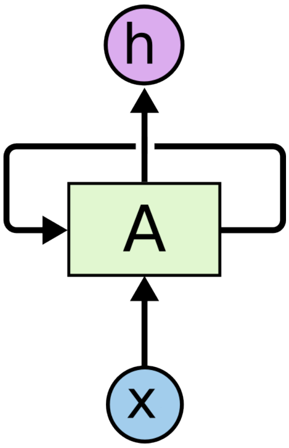
In the general case, to achieve sufficiently high accuracy on a particular task, a neural network must undergo many training iterations. For complex tasks such as speech recognition and natural language processing, the network should furthermore have a large number of weighted connections and ability to access past data. To provide the network with a *memory* ability, we introduce the recurrent connection which is a cyclic connection between the inside layers (Figure 3).

If used for a prolonged period of time, the first and last hidden layers further diverge from one another which makes their connection unstable and prevents the network from accurately connecting pieces of previous information, impairing its learning ability. This is known as the *long-term dependency problem*, also referred to in literature as the *exploding or vanishing gradient problem* [4]. Common attempts, aiming to solve the problem, have decreased the computational time of the recurrent neural network, but imposed serious limitations on its memory ability.

An approach proposed by Jing et al. in 2016 manages to avoid the problem, without affecting the memory ability, by keeping the recurrent connection weight matrix in a unitary state [5]. However, the implementation of Jing et al. does not meet the theoretically expected performance due to low degree of parallelism. This paper proposes changes to the algorithms by Jing et al. that provide an equivalent computation efficiency, but are highly parallelizable. Section II presents current common approaches dealing with the long-



(a) An analogy between the artificial neural network and the human nervous system [2]. Both A (a biological neural cell) and B (an artificial neuron) take several inputs and produce a single output based on a particular *activation* function. Additionally, C (a biological neural network) and D (an artificial neural network) illustrate the interneuronal connections, highlighting their similar structure (Maltarollo, 2013).



(b) A model of a simple recurrent neural network (RNN) [3]. The input layer is marked with x, the output layer is marked with h, the hidden layer (one in this case) is marked with A (Olah, 2015).

term dependencies problem, and explains in greater detail the unitary RNN model we are optimizing. In Section III, we elaborate on the performed optimizations. In Section IV, we provide standard benchmarks of our improved implementation. In Section V, we apply our improved implementation to common problems from the real world such as the automatic text understanding task, the speech recognition task, and the cryptoanalysis task, and analyze the achieved results. We publish our code under the GNU General Public License v3.0 on GitHub, accessible here.

II. Background

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

A. The Artificial Neural Network

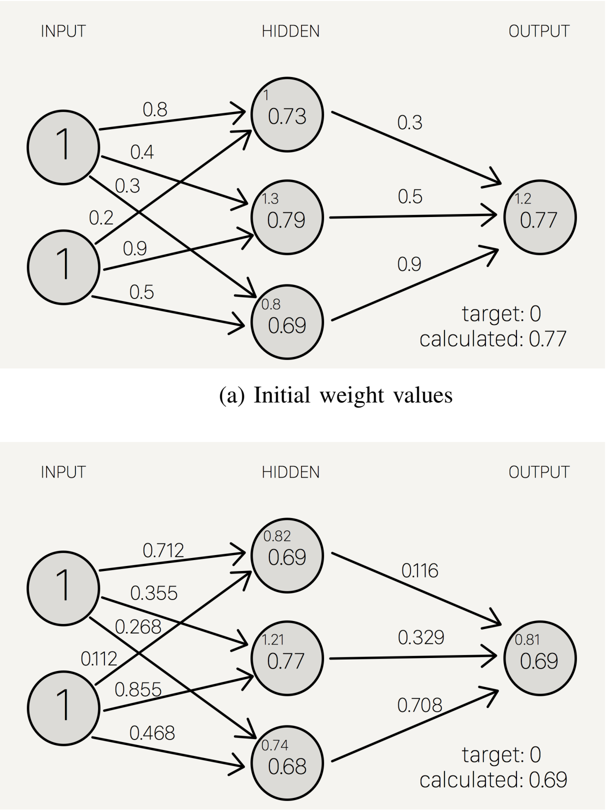
Artificial neural networks are organized in layers: the input, hidden, and output layer which respectively accept input data, process it, and store the produced result. For convenience, we present the interlayer connections in the form of matrices whose elements are the weight values. In a standard model 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. In each neuron cell, the pieces of data are combined with a certain *bias* value, stored in the cell, also undergoing adjustments, and an additional non-linear *activation* function is applied to them to avoid fitting to the input data. During *training*, the result produced by the neural network is compared to the expected one and the weight values are adjusted based on the discrepancy, so that the next time when 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. A sample step from the training process is presented on Figure 2.

B. The Concept of Recurrence

The recurrent neural network (RNN) is a specific type of artificial neural network described by a recurrent connection present in each of its hidden layers [7]. This connection allows for some of the output of the network to be conserved and then used as input during the next iteration. In this way, the functioning of the network becomes dependent on the previously processed data which creates the memory ability. This approach is especially useful when processing sequential data with dependencies, such as text and speech, as it makes the network able to consider these in its working process [8]. Typical use cases for this type of network include speech recognition and prediction, text analysis and translation. An unrolled model of a recurrent neural network is presented on Figure 3.

C. Backpropagation and Gradient Descent

One of most widely used models for training of a neural network is the *gradient descent* algorithm [1]. It aims to reduce the discrepancy between the produced and the expected answer, also defined as the *cost function*, by adjusting the specific weights of the connections and the biases in the neuron cells (Figure 4). To compute the exact change in each of these properties, it uses the *backpropagation* function



(a) Initial weight values

(b) Updated weight values

Fig. 2: A neural network learning mechanism [6]. 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 (Miller, 2015).

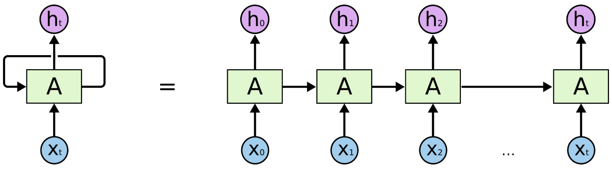


Fig. 3: A recurrent neural network (RNN) model and its representation in conventional means through time [3]. The input layer is marked with x, the output layer is marked with h, the hidden layer (only one in this case) is marked with A, and the moment in time is marked by t (Olah, 2015).

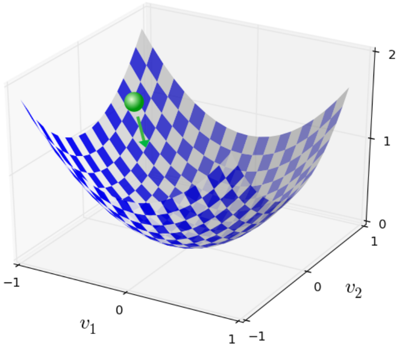


Fig. 4: The concept behind the *gradient descent* algorithm [1]. It aims to reduce the *cost function* (the green ball) by adjusting its parameters: the weights and the biases (Nielsen, 2017).

that determines how much each connection, or bias value has contributed to the final result and how much of that led to the difference in the final answers. Currently, many variations of the *gradient descent* algorithm exist such as Adam, RMSprop, etc. that adjust the network in different ways, based on the gradient of the cost function. However, in the base of all lies the *backpropagation* function calculating the gradient [9].

D. The Long-Term Dependencies Problem and Long Short Term Memory (LSTM) Networks

During prolonged time execution, the unrolled RNN model consists of a significantly big number of hidden layers which makes it hard for the *backpropagation* function to properly compute the gradient values [10]. Thus, some parameters receive gradient closer to zero, while others a value close to infinity. This leads to two problems known as the *vanishing and exploding* gradient that together form the *long-term dependencies problem* which stands a limit for the performance of recurrent neural networks. The Long Short-Term Memory (LSTM) RNN model is one the current best solutions to the long-term dependencies problem, first proposed by Hochreiter et al. [11]. The approach introduces three additional structures in the RNN cells, called *gates* (Figure 5). They utilize the sigmoid function (producing values between 0 and 1) to filter the amount of data flowing through the cell and cut some of the connections between the hidden layers. However, this method also limits the memory ability of the network, which in turn lowers its accuracy on important computational tasks [3]. Therefore, LSTM networks present only a partial solution to the long-term dependencies problem because by trading computational intensity they lower their working accuracy.

E. 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. [12]. 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 this RNN model becomes finding an efficient method to retain unity throughout the training process. In the solution proposed by Arjovsky et

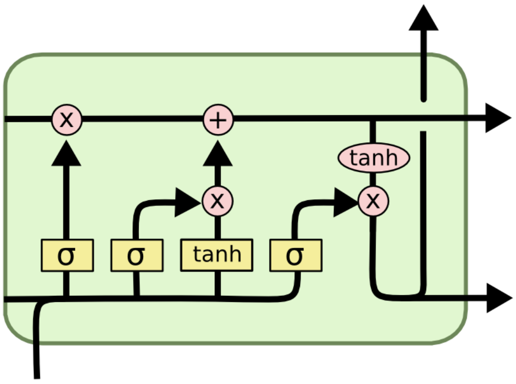


Fig. 5: Model of a LSTM network [3]. From left to right, the three rectangles inside the center cell marked by the σ signs show the input, forget, and output gates (Olah, 2015).

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 values for the recurrent connection matrix and limits the learning abilities of the network. In attempt to resolve this, a full-space coverage method is proposed by Wisdom et al. [13]. 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. [5]. 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 6). 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.

Even though Jing et al.'s model uses full-space coverage and operation optimization, its implementation is not efficiently parallelized. As a result, it 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.

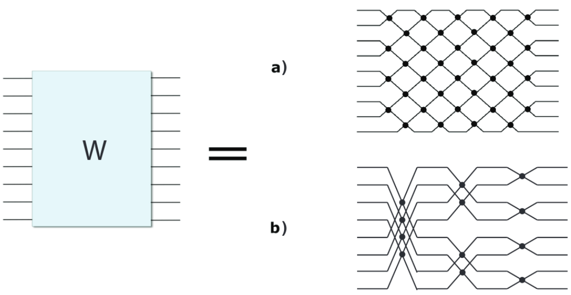


Fig. 6: Jing et al.'s unitary model consists of two full-space coverage decompositions of the unitary matrix in terms of 2×2 rotation matrices marked by the points of intersection [5]. The *simple net* model (a) presents a rectangular arrangement consisting of two repeating patterns of columns. The *lightweight* model (b) is an optimized version of the simple net model following the Fast Fourier Transformation (FFT) decomposition method (Jing et al., 2016).

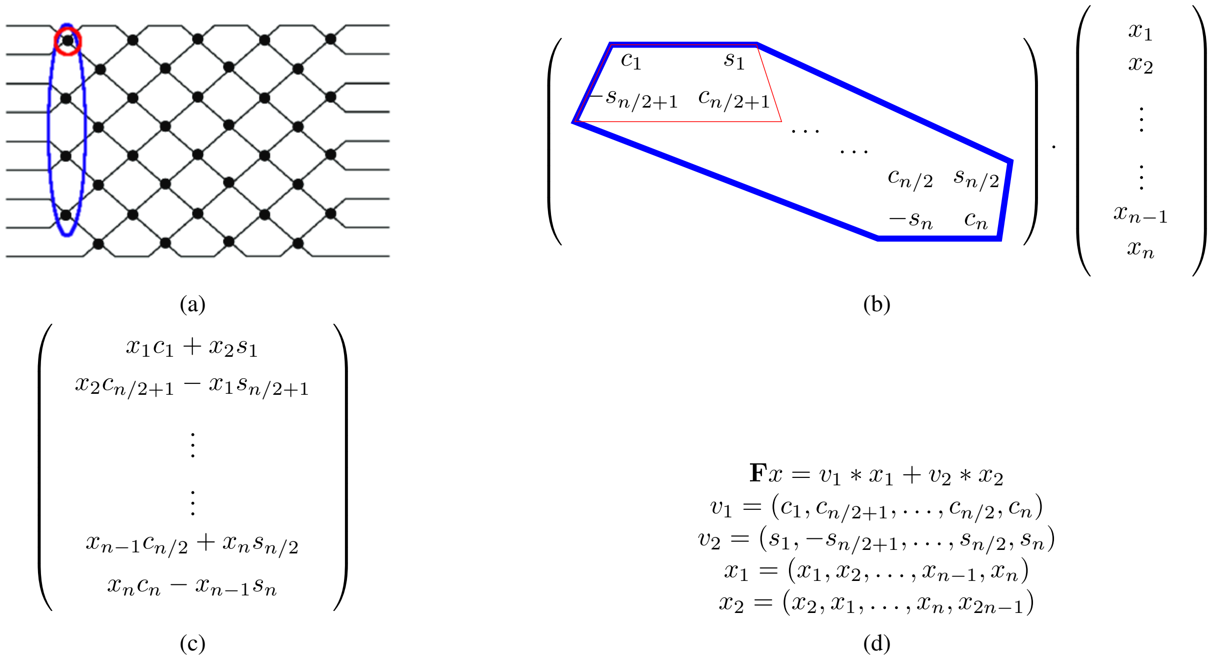
III. Optimizations

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.

A. 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 [14]. As its main data type the library uses the tensor data structure which 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 7 and Figure 8), 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 that cannot be parallelized efficiently. For the simple net decomposition model, we generate the block diagonal matrices as 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 according to 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:



(a)

(b)

$$\begin{pmatrix} x_1c_1 + x_2s_1 \\ x_2c_{n/2+1} - x_1s_{n/2+1} \\ \vdots \\ \vdots \\ x_{n-1}c_{n/2} + x_ns_{n/2} \\ x_nc_n - x_{n-1}s_n \end{pmatrix}$$

(c)

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

(d)

Fig. 7: The simple net decomposition model of the unitary matrix [5]. 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.

$$\begin{gathered} \text{Data vector shuffle pattern:} \\ \text{(applied to } x_2\text{)} \\ (1,0,3,2,\dots,2k-3,2k-4,2k-1,2k-2) \\[1em] \text{Rotation matrices shuffle pattern:} \\ \text{(applied to both } v_1 \text{ and } v_2\text{)} \\ (0,k,1,k+1,\dots,k-2,2k-2,k-1,2k-1) \end{gathered} \tag{1}$$

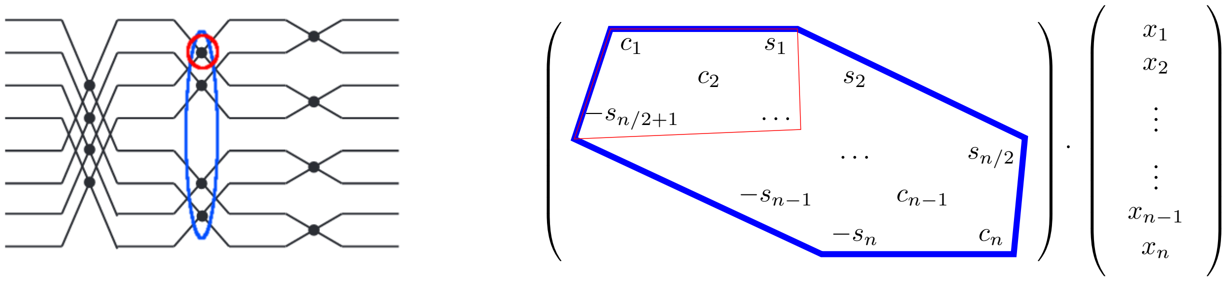
For efficient parallelization of the generation of the permutations, we propose a method replacing the hardly parallelizable `for` loops with reshaping, reversing, and transposing tensor operations that can be divided in independent parts and therefore easily parallelized. For the data vector permutation generation, we reshape the initial array into two columns, reverse the columns, and reshape the array back into a single row (2). For the rotation matrices permutation generation, we reshape the initial array into two rows, transpose the array, and reshape it back into a single row (3).

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

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

For the lightweight decomposition model, we generate the block diagonal matrices as presented on Figure 8b and perform the optimized multiplication method from the original approach as shown on Figure 8d and Figure 8c. Based on the visual representations of the matrices on Figure 8b and Figure 8c, we reorder the vectors as illustrated on Figure 8d. Therefore, based on the order of the vectors presented on Figure 8d, we can derive the general form of the required permutations as:

$$\begin{gathered} \text{Data vector shuffle pattern:} \\ \text{(applied to } x_2\text{)} \\ \begin{aligned} (k,\dots,2k-1,0,\dots,k-1) \text{ for } s=0 \\ (k/2,\dots,k,0,\dots,2k-1,k/2,\dots,3k/2-1) \text{ for } s=1 \\ \dots \\ (1,0,\dots,2k-1,2k-2) \text{ for } s=log_2s-1 \end{aligned} \\[1em] \text{Rotation matrices shuffle pattern:} \\ \text{(applied to } v_1 \text{ and } v_2\text{)} \\ \begin{aligned} (0,1,\dots,2k-2,2k-1) \text{ for } s=0 \\ (0,2,\dots,2k-2,1,3,\dots,2k-3,2k-1) \text{ for } s=1 \\ \dots \\ (0,k,\dots,k-1,2k-1) \text{ for } s=log_2s-1 \end{aligned} \end{gathered} \tag{4}$$



(a)

(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)

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

(d)

Fig. 8: The lightweight decomposition model of the unitary matrix [5]. 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.

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 (5). 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 (6).

$$\begin{gathered} [a_0, a_1, \dots, a_{2k-2}, a_{2k-1}] \mapsto \\ \mapsto \begin{bmatrix} [a_0, \dots, a_{x-1}] & [a_x, \dots, a_{2x}] \\ \vdots & \vdots \\ [a_{2k-3x-1}, \dots, a_{2k-2x-1}] & [a_{2k-2x}, \dots, a_{2k-1}] \end{bmatrix} \mapsto \\ \mapsto \begin{bmatrix} [a_x, \dots, a_{2x}] & [a_0, \dots, a_{x-1}] \\ \vdots & \vdots \\ [a_{2k-2x}, \dots, a_{2k-1}] & [a_{2k-3x-1}, \dots, a_{2k-2x-1}] \end{bmatrix} \mapsto \\ \mapsto [a_x, \dots, a_{2x}, a_0 \dots, a_{x-1}, \dots, a_{2k-3x-1}, \dots, a_{2k-2x-1}] \end{gathered} \tag{5}$$

$$\begin{gathered} [a_0, a_1, \dots, a_{2k-2}, a_{2k-1}] \mapsto \\ \mapsto \begin{bmatrix} a_0 & a_1 & \dots & a_{d-2} & a_{d-1} \\ a_d & & \dots & & a_{2d-1} \\ \vdots & & & & \vdots \\ a_{2k-2d-1} & & \dots & & a_{2k-d-2} \\ a_{2k-d-1} & a_{2k-d} & \dots & a_{2k-2} & a_{2k-1} \end{bmatrix} \mapsto \\ \mapsto \begin{bmatrix} a_0 & a_d & \dots & a_{2k-2d-1} & a_{2k-d-1} \\ a_1 & & \dots & & a_{2k-d} \\ \vdots & & & & \vdots \\ a_{d-2} & & \dots & & a_{2k-2} \\ a_{d-1} & a_{2d-1} & \dots & a_{2k-2-d} & a_{2k-1} \end{bmatrix} \mapsto \\ \mapsto [a_0, a_d, \dots, a_{2k-d-1}, a_1, a_{d+1} \dots, a_{d-1}, a_{2d-1}, \dots, a_{2k-1}] \end{gathered} \tag{6}$$

With this, we have transformed the inefficient serial generation of permutations, necessary for the approach, into highly parallelizable TensorFlow operations and have accomplished our goal of implementing a parallelization architecture. A comparison between the original implementation and our improved implementation is presented on Figure 9.

B. *Reducing Operations Complexity*

Even though we optimize 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 the rotation matrices and data vectors permutation process which is executed multiple

Algorithm 1 Original implementation for the simple net model permutations [5]

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 2 Suggested 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 3 Original implementation for the lightweight model permutations [5]

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 ← $\{2^t$,...,$2^{t+1}-1$,0,...,$2^t-1\}$
        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 4 Suggested 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
```

Fig. 9: A comparison of the permutation generation functions in the original and the suggested implementation

times for each RNN cell during a single iteration. In the original implementation, the permutation is accomplished through the use of memory and time consuming functions such as the *gather* function from the TensorFlow library, 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 5.

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 6. Therefore by introducing this method we accomplish our

Algorithm 5 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 6 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
```

goal of reducing the complexity of used operations.

C. 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 is defined only for even number of columns and even hidden layer size and the lightweight decomposition model is only applicable for hidden layer sizes which are powers of two. These are important architecture problems that limit the possible configurations of the RNN cell and our 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 [15]. 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 are added to the rotation matrices values in the other block diagonal matrices and produce the final product.

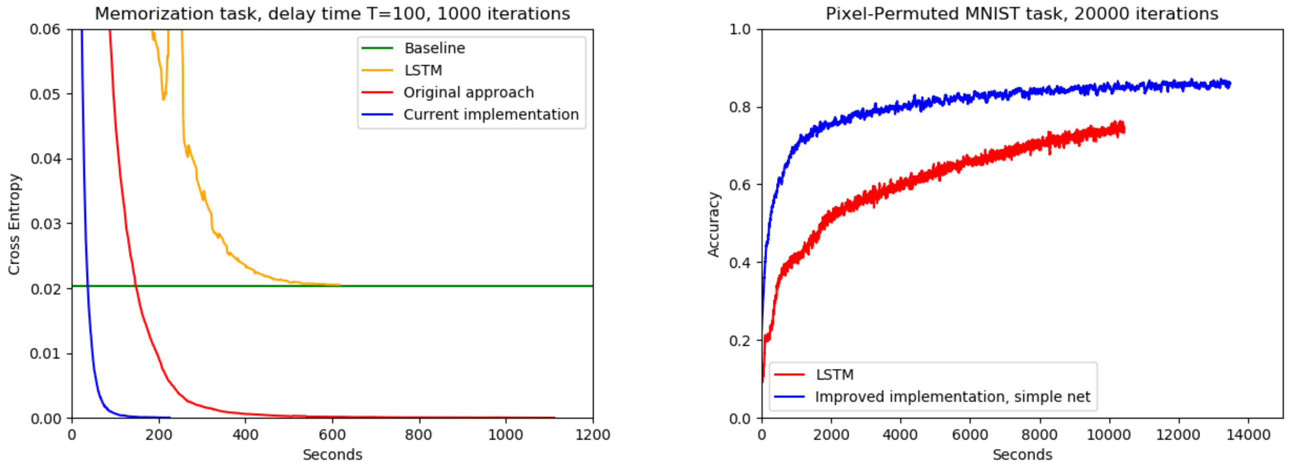
However, with these improvements, expanding the hyperparameter range of the lightweight decomposition, we have increased the computational time of the RNN model. This is a problem we would like to address in the future by finding a more effective way for expanding the hyperparameter range.

IV. Benchmark and Analysis

In this section, we test our 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.

The first benchmarking task we use is the memorization task, defined by [11], [12], and [8]. 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.

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 10a). In particular cases during the process of testing, the time performance improvement reached up to 12 times better than the original implementation. In these, the original implementation used a large portion of the GPU *cache* memory, probably due to the *gather* function, which was absent in our implementation. In the general case, our enhanced implementation performs at least 20% faster than the original implementation for all test cases. This could be due to low utilization level of the GPU eventually lowering the amount of computational cores that can be used for parallelization.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 also explained by the replacement of



Lower is better

Higher is better

(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. For viewing the cost function, we use the cross entropy distribution; the lower the cross entropy value, 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.

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 in its initial 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.

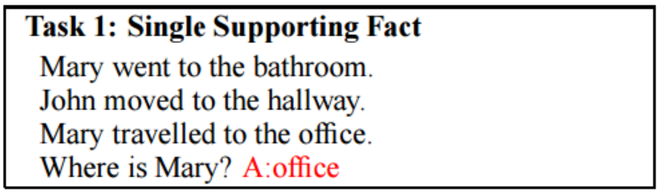
V. Tests on Problems from Real Life

This section evaluates the new implementation with several tasks from the real world assessing the network's appliance in reality. Due to content and time constraints, this paper presents only the most significant tests and graphics, conducted before the date of submission. The full test data and the most recent results are published in the GitHub Wiki of the code repository, accessible here.

A. Reading Comprehension: 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 [16]. It consists of 150 words used in 20 different tasks for automatic text understanding and reasoning each testing different capabilities of the RNN model.

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 11 and 12).


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 [16] (Weston, 2015).

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

(b) Dictionary of the vocabulary used.

$$\textit{Where is Mary?} \longrightarrow \textit{(10, 11, 0)}$$
$$\downarrow$$
$$\textit{([0,0,...,1,0], [0,0,...,0,1], [1,0,...,0,0])}$$

(c) Mapping the question into the one-hot vector data type that is inserted into the network.

Fig. 11: 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) [17]. 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 necessary. The implementation originally uses the standard

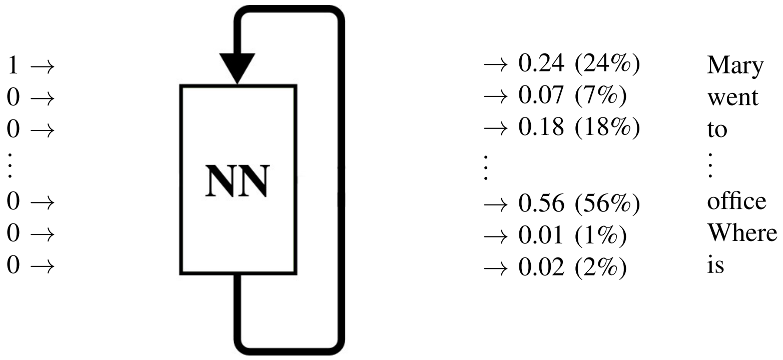


Fig. 12: The one-hot vectors are fed consecutively into the RNN and at each iteration an output vector containing a probability distribution, quantifying the word index in the created dictionary, is produced. When the input vector codes a question, the word with the highest probability is returned.

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.

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%

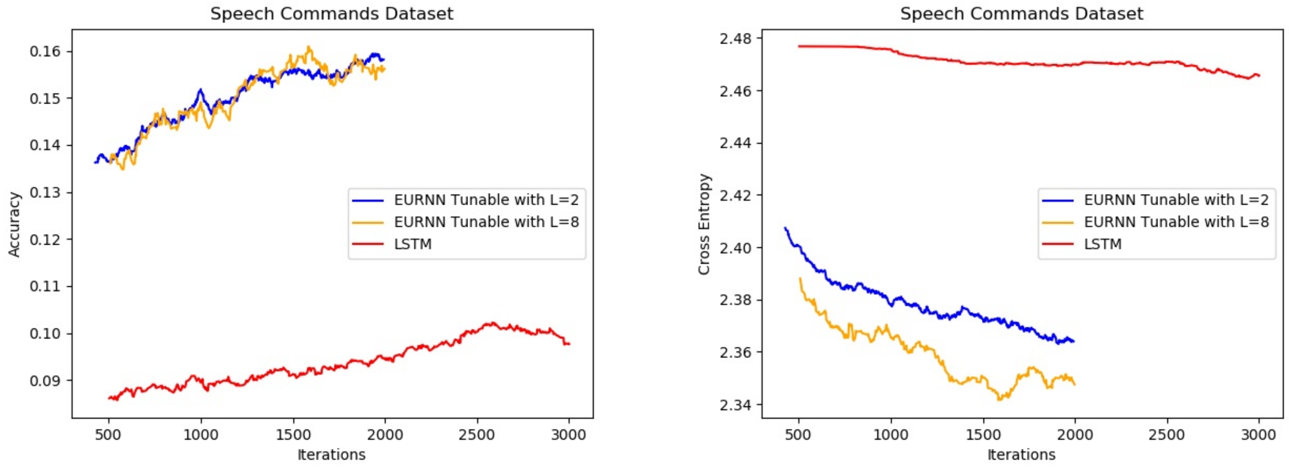
TABLE I: Results of the LSTM implementation and our improved implementation on the bAbI dataset. The implementations are run with hidden layer size of 256 and our improved implementation uses the simple net decomposition model with 2 columns. As our goal is to use as little data as possible and achieve 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 I: hidden layer size of 256 and simple net decomposition model with 2 columns. In all configurations, our unitary model implementation achieves accuracy between 50% and 58.4% which is higher than the LSTM.

B. Speech Recognition

The second famous sequential data real-life task on which we test our implementation is speech recognition, whose underlying logic is similar to the one of text understanding. As proof of concept, we have selected the newly released *Google Speech Commands Dataset* consisting of 65,000 WAVE audio files of people saying thirty different words [18]. The dataset is small which allows us to train our models more, but at the same time assesses all the necessary elements for speech recognition. It has been originally designed for basic feedforward and convolutional neural networks, but as it works with the standard TensorFlow Neural Network model, we are able to replace it with our implementation and the standard LSTM one. The results from the conducted tests are presented on Figure 13.



Higher is better

(a)

Lower is better

(b)

Fig. 13: Results of two variations of the unitary implementation (EURNN) and the LSTM model on the speech recognition task performed with a decay rate of 0.9 and learning rate of 0.0001. The unitary configuration used hidden layer size of 40 and the complex domain for weights. The LSTM configuration used hidden layer size of 64.

The results clearly outline the verdict from our previous task: the unitary model performs much better than the LSTM one, achieving higher accuracy for less iterations. In this case, it even does not fall below on a single instance, despite the LSTM using larger hidden layer size which gives it multiple times the parameters of the unitary model. This once again shows the superiority of the unitary model to the conventional LSTM one as a better RNN model for practical tasks.

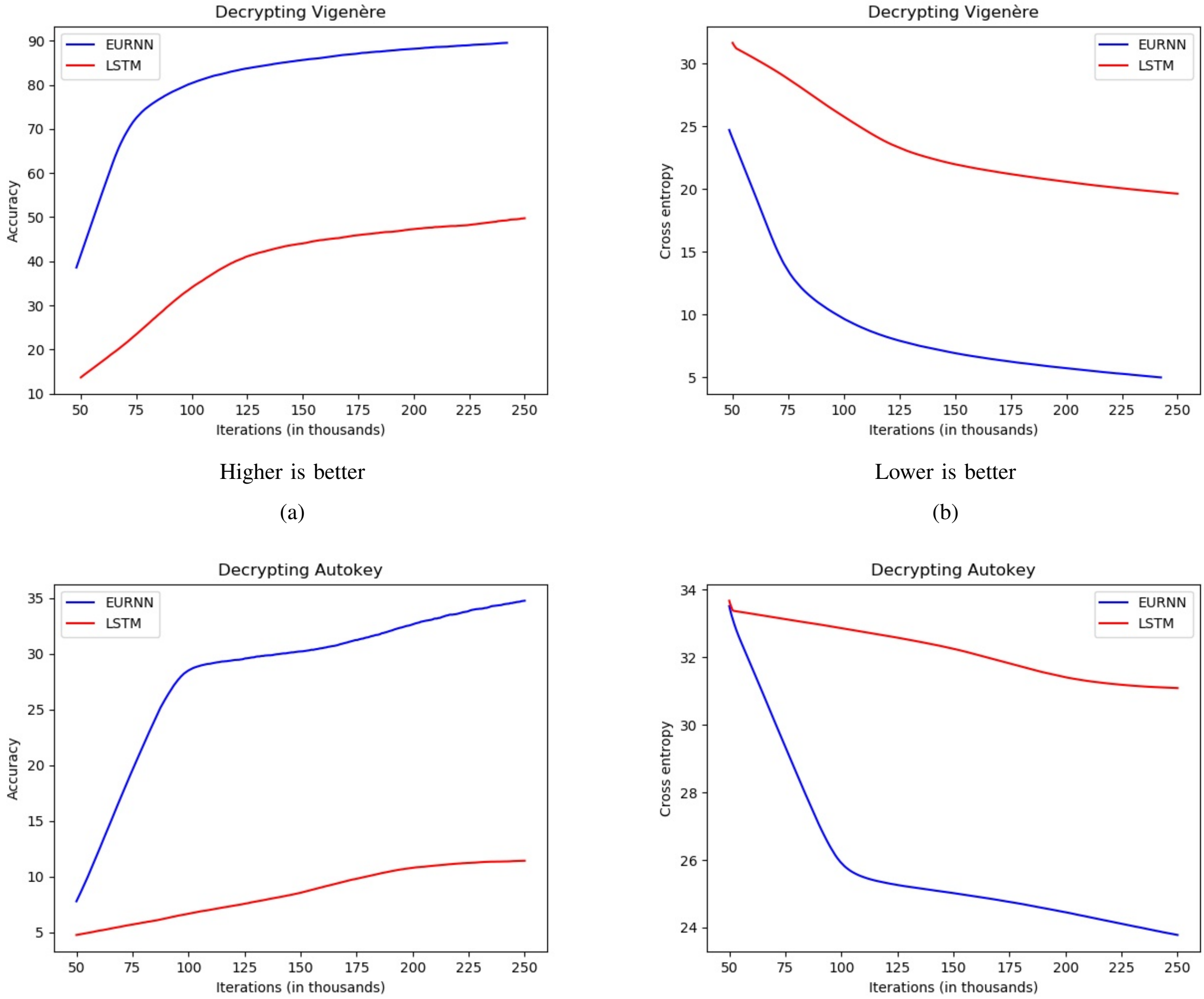
However, the fact that the unitary model performs better than the LSTM, does not mean it copes well with the task. As we can see, the achieved accuracy of both tested configurations is less than 20% which shows that both models are quite ineffective for such a task. We clearly see that the model with bigger depth performs slightly better, giving lower cross entropy values, which means that with even more parameters and depth it may give the desired accuracy. However, this would imply much more time and memory resources and would make the process ineffective. This means that despite the unitary model's superiority, it appears not to be as usable for speech recognition as for text understanding in practice.

C. *Cryptoanalysis*

Seeing the limitations of the unitary model on the speech recognition task and its benefits on the text recognition task, we assess its performance on the quickly developing and highly important for our society topic of cryptoanalysis. Neural networks are increasingly used for such tasks because of the non-linearity that they employ and which stays in the basis of many cryptographic ciphers. Additionally, neural networks are able to *learn* and model functions which also makes them able to encrypt, or decrypt messages. While the description above applies to all types of artificial neural networks, RNNs are specifically able to learn better some patterns due to their memorizing ability. Therefore, they are expected to learn faster and produce higher results in the cryptographic operations. In our use case, we have taken two relatively simple ciphers, *Vigenère* and *Autokey*, and we have trained an LSTM network and our implementation to decrypt messages, encrypted with them. For testing, we have used Sam Greydanus' implementation which utilizes an LSTM network cell [19]. It again follows the standard TensorFlow neural network model, so we are able to quickly configure it for our own implementation. The results from the test are presented on Figure 14.

The results again show the better performance of the unitary model compared to LSTM. It is notable that the unitary model manages to reach an accuracy of 92% for the given number of training iterations on the Vigenère cipher while the LSTM hardly manages to reach half of it, 50%. This means that the unitary model will have much better application on real-life cryptographic tasks compared to the LSTM one. However, it is important to mention that despite its higher accuracy, the unitary model took much more time and memory resources for its work than the LSTM. This outlines a possible limitation for its use in devices with lower computational power as it will function ineffectively on them.

On the second cipher, the Autokey, which is a little more complex than Vigenère, both networks struggle to get an accuracy over 35% which is quite low for the task. Therefore, they may have problems working with even more complex ciphers such as the ones used in modern communications. This may be a limitation of the recurrent neural network model, as it learns slowlier than the standard feedforward one and therefore is not able to undergo many training



Higher is better

(a)

Lower is better

(b)

Higher is better

(c)

Lower is better

(d)

Fig. 14: Results of the unitary implementation (EURNN) and the LSTM model on decrypting Vigenère and Autokey, performed with a learning rate of 0.0005. The unitary configuration used hidden layer size of 128, depth of 8, and the complex domain for weights. The LSTM configuration used hidden layer size of 40 with the same number of parameters.

procedures. Still, based on the performance of the unitary model, we have discovered an additional use of the concept, aside from sequential data, in the field of cryptoanalysis. Although, it might not be able to beat plain feedforward networks, the unitary approach will still turn into the most functional RNN model for the task.

VI. Conclusion

We presented the optimizations conducted in the implementation of a novel recurrent network model and provided results of its application to three real-life tasks: of automatic text understanding, of speech recognition, and of cryptoanalysis. 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 real-life task results, the improved implementation achieves maximum mean accuracy of around 60% in all of its hyperparameter configurations, sufficiently beating the commonly used LSTM model on the automatic text understanding task. On the speech recognition task, both RNN models struggle to achieve an accuracy over 20% which is quite low for the use case, and presents that they are not effective solutions for such a problem. However, still the unitary model is able to show some learning behavior and completely outperform the LSTM one. Last, on the cryptoanalysis test, again the unitary model performs with higher accuracy than the LSTM, reaching 92% on Vigenère decryption and 35% on Autokey decryption. This shows it performs better than the LSTM on these types of tasks, but it requires much more time and memory resources than conventional networks which becomes one of its limitations.

Overall, the conducted tests show 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, further decreasing time and memory consumption by using lighter operations, and applying the concept of the unitary matrix in Recurrent Highway Neural Networks known for their higher capabilities in dealing with more complex tasks.

VII. Acknowledgments

I would like to thank my research mentor, Li Jing, the *Research Science Institute* 2017 staff and participants, the *High School Student Institute of Mathematics and Informatics*, and *America for Bulgaria Foundation* for making the development of this project possible.

The current research was initiated during the summer research program *Research Science Institute* hosted in the *Massachusetts Institute of Technology*. It was performed in the Department of Physics in cooperation with Prof. Marin Soljačic's Deep Learning research group and was supported with their computational resources.

References

[1] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 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] C. Olah. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015.

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

[5] 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.

[6] 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.

[7] D. Britz. Recurrent neural networks tutorial, part 1 introduction to rnns. `http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/`, 2015.

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

[9] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[10] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

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

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

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

[14] 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.

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

[16] 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.

[17] 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.

[18] P. Warden. launching the speech commands dataset. *Google Research Blog*, 2017.

[19] S. Greydanus. Learning the enigma with recurrent neural networks. *arXiv preprint arXiv:1708.07576*, 2017.