Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 6
Prof. Dr.-Ing. Hermann Ney

Selected Topics in Human Language Technology and Pattern Recognition (WS 2016)

Neural Turing Machines and Related

Arne Nix

Matriculation number 331423

01/02/2017

Supervisor: Albert Zeyer

Contents

List of Tables

List of Figures

1 Introduction

Machine learning techniques and algorithms have become a major building block of modern computer systems. Especially artificial neural networks have experienced a renaissance in the last years. They have proven to outperform traditional models in various disciplines. Among these are classification, image recognition, speech recognition, machine translation and much more.

Additionally, it has been proven that recurrent neural networks (RNNs) are in theory equivalent to Turing machines [33]. Therefore, an RNN should in principle be able to express any computable function. However, the theoretical expressiveness does not carry over to the practice. This becomes evident by the fact that basic RNN architectures struggle to learn even very simple algorithms from examples or fail to generalize to longer inputs after being trained on inputs of a limited length. Inspired by this shortcoming of the traditional architectures, there have been several approaches to modify and extend the existing RNN models to reach the full potential the theoretical findings are promising. All these approaches have in common that they provide the network with additional memory or more advanced memory access methods respectively. These so-called memory augmented RNN approaches have been able to learn simple algorithms on many different tasks. For example, the most prominent approach is the neural Turing machine (NTM) [13] which is able to learn algorithms like copying, sorting and associative recall from input and output examples. This is achieved by extending a neural network with an additional memory with explicit read and write operations.

In this work, the NTM approach, as well as multiple different approaches, will be discussed and compared in the context of practical and theoretical computability. Additionally, their applicability to real-world tasks will be examined for the area of machine translation.

1.1 Outline

This work will first explain the basic principles of artificial neural networks (section 2) with a special focus on recurrent neural networks (section 2.1). This includes a more detailed description of the two most prominent RNN architectures with the long short-term memory (section 2.1.1) and gated recurrent unit (section 2.1.2).

Then the theoretical (section 2.2.1) and practical (section 2.2.2) capabilities of neural networks are discussed and compared. With this background knowledge as a starting point, the specific solutions to some of the shortcomings of RNNs are examined in section 3. Among these are attention-based approaches (section 3.1) which provide read-only memory to the network as well as extensions which introduce a stack data-structure (section 3.2) or random-access memory (section 3.3) to the network.

In section 4, the results that were reported in the literature for the presented approaches are examined and also compared if possible. This includes a short discussion of possible applications for the presented techniques in the field of machine translation. The findings from analyzing the literature are again briefly summarized in the conclusion which is given in the last section.

2 Artificial Neural Networks

Artificial neural networks and especially deep neural networks have become very popular over the last years [25]. Responsible for this phenomena are the recent successes that were achieved by applying deep neural networks to different areas, such as speech recognition [12], image recognition [23] and machine translation [1].

An artificial neural network is, similar to the human brain it was inspired by, composed of many small units that are all computing very simple functions, when regarded in isolation. One such unit computes the weighted sum of its inputs x_1^I with weight parameters w_{ji} and a bias b_j as an intermediate representation z_j. The final activation y_j of this unit is obtained by applying an activation function σ_j to z_j.

$$z_j = \sum_{i=1}^{I} w_{ji} \cdot x_i + b_j \tag{1}$$

$$y_j = \sigma_j(z_j) \tag{2}$$

A common choice for this activation functions σ_j are the logistic sigmoid σ_{sigmoid} and the hyperbolic tangent $\sigma_{\tanh}$.

$$\sigma_{\text{sigmoid}}(z) = \frac{1}{1+e^{-z}} \tag{3}$$

$$\sigma_{\tanh}(z) = \tanh(z) = \frac{e^{2z}-1}{e^{2z}+1} \tag{4}$$

These functions are necessary to introduce non-linearity into the network, which is crucial for most real-world problems.

Also very important for all but the most simple tasks is the introduction of a multi-layered architecture for the network. Otherwise, a network would be restricted to model the class of linear-separable problems. Therefore, the single units, as they are defined in equation (1) are summarized in a layer l, which is defined by the weight matrix $W^{(l)} \in \mathbb{R}^{N^{(l-1)} \times N^{(l)}}$ and a bias vector $b^{(l)} \in \mathbb{R}^{N^{(l)}}$. Each layers activation $y^{(l)}$ is therefore calculated by elementwise application of an activation function $\sigma^{(l)}$ to the product of $W^{(l)}$ and the output vector $y^{(l-1)}$ from the previous layer:

$$\begin{aligned} g^{(l)}_{\{W^{(l)},b^{(l)}\}} :& \mathbb{R}^{N^{(l-1)}} \to \mathbb{R}^{N^{(l)}}, \\ & y^{(l-1)} \mapsto y^{(l)} = g^{(l)}_{\{W^{(l)},b^{(l)}\}}(y^{(l-1)}) = \sigma^{(l)}(W^{(l)} y^{(l-1)} + b^{(l)}) \end{aligned} \tag{5}$$

Given this modular definition of each layer l by its parameters $\theta^{(l)} = \{W^{(l)}, b^{(l)}\}$, it becomes obvious that a neural network consisting of multiple layers is simply a composition of the layer functions $g_{\theta^{(l)}}$ for $l = 1, \dots, L$:

$$g_\theta : \mathbb{R}^{N^{(0)}} \to \mathbb{R}^{N^{(L)}} : x \mapsto g_\theta(x) = (g^{(L)}_{\theta^{(L)}} \circ \dots \circ g^{(1)}_{\theta^{(1)}})(x) = y^{(L)} \tag{6}$$

For many use-cases in which artificial neural networks are applied, it is necessary to have a normalized distribution as the final output of the network. A softmax function is usually applied to the output of layer L in order to obtain a normalized distribution over all units in layer L:

$$y_c = \frac{e^{y_c^{(L)}}}{\sum_{k=1}^{N^{(L)}} e^{y_k^{(L)}}} \quad \forall\, c \in \{1, \dots, N^{(L)}\} \tag{7}$$

When applied to a classification task, the softmax output can be interpreted as the posterior probability for each class c:

$$p_\theta(c|x) := y_c \tag{8}$$

With this interpretation, the loss function that is used to optimize the network is intuitively defined as the negative log probability, which gets computed for the whole training set $\mathcal{X}_{\text{train}} = \{(x_n, c_n)|\ n = 1, \dots, N\}$ or a subset (mini-batch) of the training data:

$$\mathcal{L}_{\text{CE}} := -\frac{1}{N}\sum_n \log p_\theta(c_n|x_n) \tag{9}$$

By substitution and reformulation, this loss function can be transformed into the cross-entropy between the true distribution $pr(c|x_n)$ and the distribution $p_\theta(c|x_n)$ that is modeled by the neural network.

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{c\in\mathcal{C}} pr(c|x_n)\log p_\theta(c|x_n) \tag{10}$$

To optimize the parameters of a neural network, one seeks to minimize the error that is specified by the loss function for any given training set $\mathcal{X}$. This optimization is usually done by exploiting the special compositional structure of the model, which makes the network differentiable with respect to its parameters and input. Using the backpropagation algorithm [26, 27], which effectively applies the chain rule to the composition of functions that is the neural network, it is possible to efficiently compute the gradient w.r.t. any specific parameter $w_{ji}^{(l)}$ and update the network using this information.

The update rule which should optimize the weight parameters is usually implemented by some variation of gradient descent. Gradient descent, intuitively changes the weights $w_{ji}^{(l),n+1}$ in the update step $n+1$ into the direction of the negative error gradient $\frac{\partial\mathcal{L}}{\partial w_{ji}^{(l),n}}$, which is the direction of the steepest descent.

$$w_{ji}^{(l),n+1} = w_{ji}^{(l),n} - \eta\frac{\partial\mathcal{L}}{\partial w_{ji}^{(l),n}} \tag{11}$$

To avoid local minima during optimization, the gradient is scaled by a learning rate factor η, which needs to be tuned separately.

2.1 Recurrent Neural Networks (RNNs)

Neural networks as they were described in the previous section are usually limited to fixed size inputs and outputs if they are applied without any further modification. This limits their usefulness for tasks that process sequential data.

Recurrent neural networks (RNNs) are designed to solve this limitation by extending the feed forward network structure with recurrent connections in one or multiple hidden layers. The output $y^{(l)}(t)$ of such a recurrent layer l in timestep t depends not only on the output $y^{(l-1)}(t)$ of the previous layer from the same timestep but also on the output $y^{(l)}(t-1)$ of the same layer from the previous timestep.

$$y^{(l)}(t) = \sigma^{(l)}(W^{(l)}y^{(l-1)}(t) + U^{(l)}y^{(l)}(t-1)) \tag{12}$$

RNNs can in principle be trained similar to feed forward networks by gradient descent. Due to the exploding and vanishing gradient problem [16, 2], the unmodified variants of RNNs are not capable of preserving long term dependencies through the recurrent connections. Since long-term dependencies are very important for most tasks involving sequential data, there are several extensions to the standard RNN architecture that try to prevent this problem. Two approaches are explained briefly in the following sections.

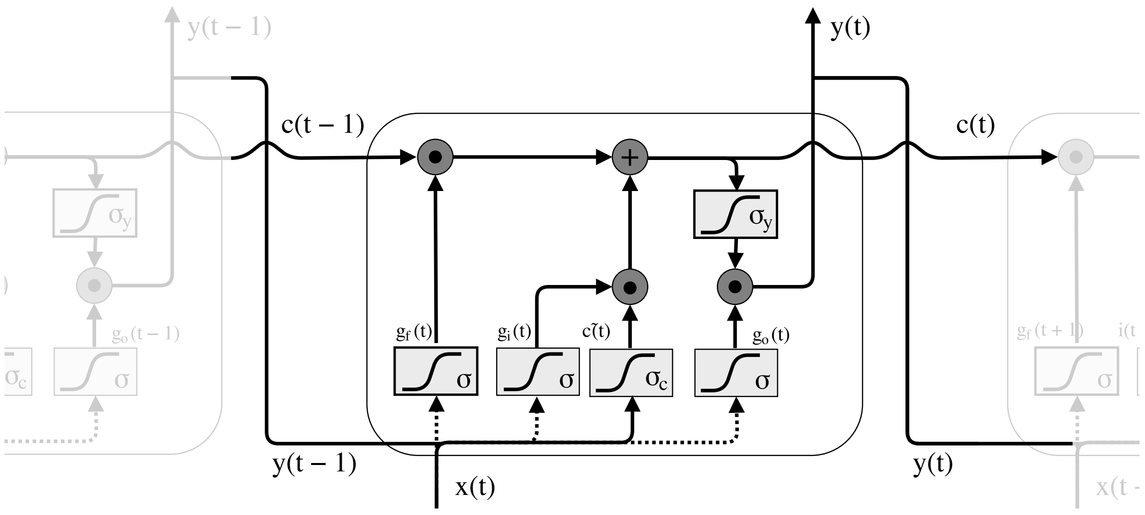

Figure 1: Architecture of an LSTM unfolded over time.[1]

2.1.1 Long Short-term Memory (LSTM)

The long short-term memory (LSTM) [17, 9] unit prevents the vanishing gradient problem by separating the internal information into a protected cell state $c(t)$. The information in $c(t)$ is never directly subject to any activation function when handed over in time, which can be seen as applying the identity function and therefore results in a derivative of 1. This prevents the gradient from approaching zero over longer time dependencies, but at the same time makes a gating mechanism necessary to modify the cell state.

For this purpose, the LSTM has the forget $(g_f(t))$, input $(g_i(t))$ and output gate $(g_o(t))$, which are vectors with values between zero and one that represent a closed or opened gate. The gate activations are computed by applying a single hidden layer to the input of the LSTM $x(t)$ and the output $y(t-1)$ of the same unit from the previous timestep:

$$g_\star(t) = \sigma(W_{x\star}x(t) + W_{y\star}y(t-1) + b_\star) \tag{13}$$

$$\tag{14}$$

The difference between these three gates are the different parameters $W_{x\star}, W_{y\star}$ for $\star \in \{i, f, o\}$. The update candidate $\tilde{c}(t)$ is also computed by a single hidden layer, with parameters W_{xc}, W_{yc} but different from the gate activations, the activation function for the update candidate σ_c is often implemented by a tanh.

$$\tilde{c}(t) = \sigma_c(W_{xc}x(t) + W_{yc}y(t-1) + b_c) \tag{15}$$

The update candidate $\tilde{c}(t)$, filtered by the input gate, and the old cell state $c(t-1)$ filtered by the forget gate are used to form the new cell state $c(t)$. The filtering is done by elementwise vector multiplication, which means that input gate decides which parts of the update candidate are added to the new cell state. Similarly the forget gate deletes elements in the cell state.

$$c(t) = i(t) \odot \tilde{c}(t) + f(t) \odot c(t-1) \tag{16}$$

[1]Graphic inspired by `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

The new cell state $c(t)$ is then handed over to the next timestep and is additionally used to compute the output $y(t)$ of the LSTM unit for the next layer.

$$y(t) = o(t) \odot \sigma_y(c(t)) \tag{17}$$

Here the activation function σ_y, usually a tanh, logisitc sigmoid or identity function, is applied to $c(t)$ and the output is finally filtered by the output gate $o(t)$.

2.1.2 Gated Recurrent Unit (GRU)

The gated recurrent unit (GRU) [3] is a modified version of the LSTM architecture, which applies several simplifications to the standard LSTM unit.

The most significant change compared to the LSTM is the fact that cell state and output are merged. This is possible since the non-linearity that gets applied to the output in the LSTM is dropped in the GRU. Additionally, the input and forget gate are replaced by a single update gate $g_u(t)$, which is used to compute the new hidden state $y(t)$ as an affine combination of the old state $y(t-1)$ and the update candidate $\tilde{y}(t)$:

$$y(t) = [1 - g_u(t)] \odot y(t-1) + g_u(t) \odot \tilde{y}(t) \tag{18}$$

The update candidate is computed similarly to $\tilde{c}(t)$ in the LSTM architecture, with the modification that the old hidden state filtered by a reset gate $g_r(t)$:

$$\tilde{y}(t) = \sigma_{\tanh}(W_{xy}x(t) + W_{yy}[g_r(t) \odot y(t-1)] + b_y) \tag{19}$$

The gate activations are again computed by a single hidden layer applied to the hidden state $y(t-1)$ and the input $x(t)$:

$$g_r(t) = \sigma_{\text{sigmoid}}(W_{xr}x(t) + W_{yr}y(t-1) + b_r) \tag{20}$$
$$g_u(t) = \sigma_{\text{sigmoid}}(W_{xu}x(t) + W_{yu}y(t-1) + b_u) \tag{21}$$

GRUs are essentially a simplified version of LSTMs and result in a reduction in the number of parameters. This means that a GRU has only two-thirds of the parameters an LSTM would have, while at the same time being equally powerful as LSTMs on sequence modeling tasks [6].

2.2 Computational Power of Neural Networks

2.2.1 Theoretical

A first theoretical exploration of neural networks by Cybenko et al. [7] resulted in the universal approximation theorem. This essentially says that a linear combination of neurons, i.e. an artificial neural network with at least one hidden layer, can approximate any continuous function f on compact subsets of $\mathbb{R}^n$.

Another theoretical analysis of neural networks and in particular recurrent neural networks by Sigelmann and Sontag [33] proved that recurrent neural networks are Turing complete. That means that recurrent neural networks are theoretically capable of simulating a universal Turing machine and therefore can compute any computable function. Siegelmann and Sontag show this by simulating a two-stack machine, which itself is also

Turing complete, with an RNN. In their proof, the contents of each stack is represented as a rational number between 0 and 1 of the form:

$$s = \sum_{i=1}^{n} \frac{a_i}{4^i} \tag{22}$$

In this special representation, the ith element on the stack is encoded as the ith element a_i on the right side of the decimal point in a finite expansion of s in base 4. With two stacks encoded in this way, the authors show how to setup the network to update the stack using push and pop operations and thereby simulate a push-down automaton with two binary stacks.

2.2.2 Practical

The universal approximation theorem shows the theoretical effectiveness of artificial neural networks. In practice, however, this theorem does not help us in finding the parameters we would need to model a given function f. Although this problem has been successfully overcome in practice by state-of-the-art optimization methods, the universal approximation theorem still gives no statement about whether the approximation would be able to generalize from the approximated function f to different functions $\tilde{f}$. Additionally, the theorem only holds for inputs and outputs form a compact subset of $\mathbb{R}^n$ and thus can not make any statement on the large subset of computable functions that is defined on inputs and outputs of arbitrary length.

Something similar holds for the Turing completeness of RNNs. In theory, RNNs are able to express any Turing machine but in practice, the encoding used in the proof of Sigelmann and Sontag [33] to represent the stack would introduce considerable limitations. Due to the finite representation of rational numbers in practice, the stack would not only be finite but also very limited in capacity (e.g. 28 items for 64 bit floating point representation). This is a discrepancy between the computation power in theory and practice that has to be emphasized here. Results from practical experiments [13, 8, 15] with standard LSTM units show that they are hard to train on some algorithmic tasks or require a larger hidden state to solve those. Additionally, Kaiser et al. [20] show that even with an additional, fixed-size, read-only memory, the LSTM is not able to generalize well after training on binary addition.

With those results on the one hand and the competitive results RNNs showed e.g. in natural language processing [1, 12, 35] on the other hand, it is reasonable to assume that simple RNNs are in practice limited to perform an approximation for finite state machines. This hypothesis is supported by the theoretical finding that RNNs able to exactly simulate finite state machines [36] and the work by Kolen [22] that showed that it is even possible to extract finite state machines that were learned by RNNs.

Possible issues that limit the RNNs, including LSTMs and GRUs, to these tasks is the fixed size of the memory of the network, which usually just the hidden state. This restriction makes it impossible to learn many computable functions, since the memory fixed memory size will prevent these models to generalize to inputs of arbitrary length even if the network is able to learn the algorithm on the training data.

There have been numerous attempts to solve these problems and some of the most prominent and successful approaches will be presented in the next section. The common idea behind these methods is to augment the RNN with memory which can be increased without needing to retrain the model. This promises generalization to arbitrary length on the one hand but introduces the new challenge to access this memory of unknown length on the other hand. Each of the approaches in the following section is offering a distinct solution to this problem by introducing different data structures and corresponding access operations to augment RNNs with additional memory.

3 Augmenting RNNs with Memory

As described in the previous section, RNNs have a great potential in theory, but it has not yet been shown that this carries over to the practice for the standard architectures, including the already advanced approach of LSTMs.

It is widely believed [13, 15] that this is due to fixed capacity limiting the internal memory and the insufficent control over the provided memory. The approaches explained in this section are explicitly designed to tackle one, respectively both of these limitations.

The first approach that could be classified as an memory augmented RNN was the attention mechanism [10, 1]. This method gives the network the ability to access an arbitrarily long memory, which for these models only allows read access, with a trainable input-dependent focus on one position. The first augmented memory model that introduced an external memory with read and write heads that can be trained independent from the size of the memory, was the neural Turing machine (NTM) [13]. The NTM also represents most general structure of memory augmented RNNs, which is illustrated in Figure 5. This architecture consist of a controller network which can be an RNN or also a simple MLP, a memory matrix M, specific read and write operations.

Noteworthy here is the fact that the standard LSTM also fits into this structure, if the internal state $c(t)$ is seen as the memory matrix $M(t)$. The read operation is described in equation (17) and the write operation analogously in equation (16). This comparison also exposes the greatest weakness of the LSTM architecture, which is its limited memory capacity which is fixed to a single vector usually with a size similar to the input size and can not be changed without retraining. Therefore, it may be possible to successfully train an LSTM on an arbitrary algorithmic task, but it is not reasonable to expect it to generalize to inputs and outputs of arbitrary length.

3.1 Attention

A simple and common extension to RNNs, especially for sequence generation tasks, is the attention mechanism [10, 1]. This approach has a wide range of applications and a similar number of different implementations. The underlying concept, however, is always to use some learnable encoding of the input as an additional read-only memory M for the controller network, which is usually an RNN. The important difference to standard RNNs and LSTMs is the more advanced read operation, which operates in two steps. First, the attention mechanism computes the attention weights $\alpha_i(t)$, which are normalized by a softmax function and dependent on some input from the controller network $k_i(t)$:

$$\alpha_i(t) = \frac{\exp(f_{\text{att}}[k_i(t)])}{\sum_j \exp(f_{\text{att}}[k_j(t)])} \tag{23}$$

Then, these attention weights are used to perform the actual lookup on the memory matrix:

$$r(t) = \sum_i \alpha_i(t) M_i \tag{24}$$

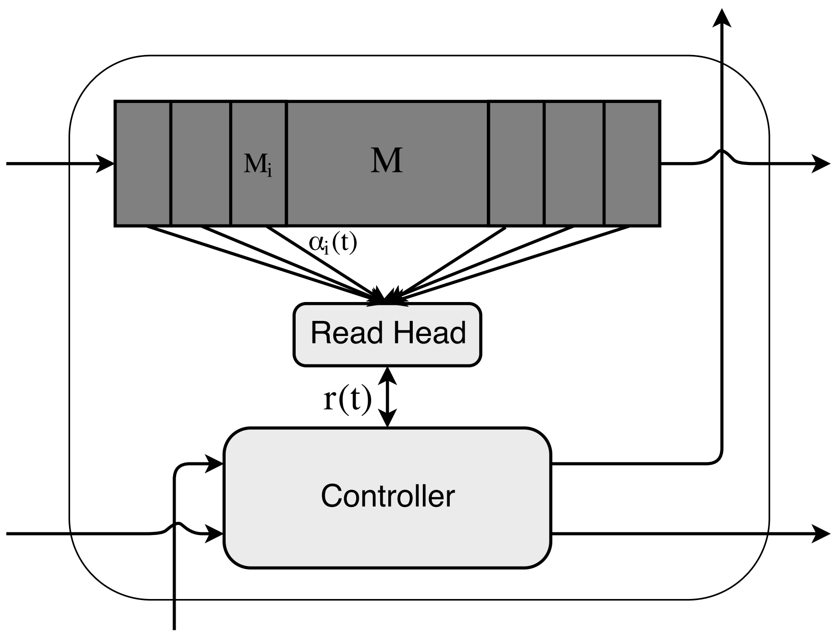

Figure 2: An abstract illustration of the attention-based approach.

The controller network can access the attention mechanism to compute lookup vectors generated by different weightings, i.e. by different addressing, of the possibly very large memory vectors in matrix M.

This approach has proven to be very effective for tasks like machine translation [1] and image caption generation [43] and showed some promising results in speech recognition [5].

3.1.1 End To End Memory Networks (MemN2N)

Memory networks [41] and especially end-to-end memory networks (MemN2N) [34] are a special type of attention-based models.

An important difference to the standard attention approach is the application of separate embeddings $m^{(1)}(t)$ and $m^{(2)}(t)$ for the memory matrix in the addressing and the lookup steps. The addressing is computed as the softmax of the match between the embedded memory $m^{(1)}(t) = E^{(1)}(t)[M]$ and input $x(t-1)$:

$$\alpha_i(t) = \frac{\exp([x(t-1)]^T m_i^{(1)}(t))}{\sum_j \exp([x(t-1)]^T m_j^{(1)}(t))} \tag{25}$$

With the attention weights as addressing and a second embedding $m^{(2)}(t) = E^{(2)}(t)[M]$ of the same memory M, the lookup vector is computed similarly to equation (24):

$$r(t) = \sum_i \alpha_i(t) m_i^{(2)}(t) \tag{26}$$

The embedding may change in every step, but the memory content itself is kept fixed since the MemN2N model does only provide read-only access to the memory through the

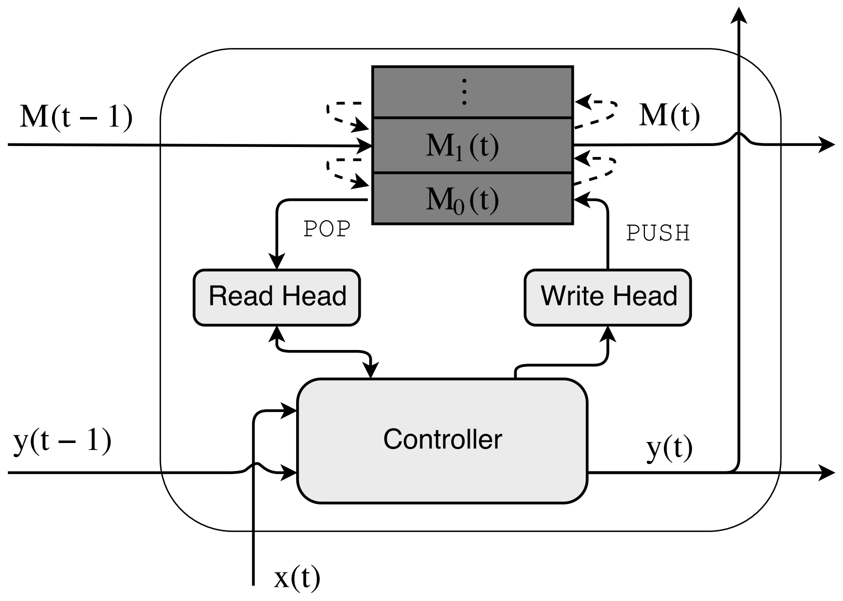

Figure 3: Stack augmented RNN.

attention mechanism. In contrast to the standard attention approach, the MemN2N does not necessarily give an output after every single memory access. Therefore, the network has the chance to perform reasoning in multiple steps, by setting the input of step $t+1$ to the output of step t combined with its input:

$$x(t) = x(t-1) + r(t) \tag{27}$$

Finally, in the last computation step T, the final output is computed:

$$y = \text{softmax}(W[r(T) + x(T-1)]) \tag{28}$$

To reduce the number of free parameters in the model, it is common to share the weights between different computation steps $t = 1, ..., T$. For a feed-forward structure, the weights of adjacent embeddings are shared, i.e. $E^{(2)}(t+1) = E^{(1)}(t)$. In this case, the number of computation steps T and thereby the depth of the network is fixed. However, further parameter sharing with $E^{(1)}(1) = ... = E^{(1)}(T)$ and $E^{(2)}(1) = ... = E^{(2)}(T)$ results in a recurrent network structure that can, in theory, become arbitrary deep.

When applied in practice, it is important to choose a suitable embedding for the inputs that are stored in the memory matrix and handed to the network as initial input $x(0)$. For the task of question answering, the question would usually be given as the initial input and the sentences providing the facts that lead to the answer are stored in the memory matrix and accessed in every computation step. If all sentences are properly encoded, e.g. as bag of words, the model can simulate a reasoning process in multiple steps that focuses on a different part of the text in every step.

3.2 Augmenting RNNs with Stacks

Even though the attention mechanism offers random access to the memory matrix, this extension only introduces read-only memory. Therefore, attention-based RNNs are still limited to the hidden state of the controller network to save information and carry it over to future computation steps.

An approach that is disregarding random memory access but introducing write operations for the memory matrix is introduced by Joulin et al. [19]. This extends a standard RNN by a stack data-structure. The stack is implemented by restricting the access to the first vector $M_0(t)$ of memory matrix $M(t)$, which is supposed to represent the top of the stack. Given this data-structure, the network has three options `PUSH`, `POP` and `NO-OP` to manipulate the stack $M(t)$. The controller network decides what operation to perform through the vector $\alpha(t) \in \mathbb{R}^3$, which is computed as:

$$\alpha(t) = \text{softmax}(W_\alpha y(t)) \tag{29}$$

Here, W_α is the trainable weight matrix that determines the operation depending on the current network state $y(t)$. The action decision $\alpha(t)$ is applied to the stack as a weight in an affine combination. Therefore, the network first updates top of the stack, with the options to set (`PUSH`) the top to the current network state $y(t)$ embedded by the weight matrix $W_{\texttt{PUSH}}$, replace (`POP`) the top of the stack with the second item $M_1(t-1)$ or keep (`NO-OP`) the old memory $M_0(t-1)$:

$$M_0(t) = \alpha_{\texttt{PUSH}}(t)\sigma[W_{\texttt{PUSH}}y(t)] + \alpha_{\texttt{POP}}(t)M_1(t-1) + \alpha_{\texttt{NO-OP}}(t)M_0(t-1) \tag{30}$$

Furthermore, the following levels in the stack have to be updated in a similar manner:

$$M_i(t) = \alpha_{\texttt{PUSH}}(t)M_{i-1}(t-1) + \alpha_{\texttt{POP}}(t)M_{i+1}(t-1) + \alpha_{\texttt{NO-OP}}(t)M_i(t-1) \tag{31}$$

The only difference here is the `PUSH` operation, which replaces the stack content at position i with the item from the level above. This soft decision execution is necessary in order to obtain a structure that is differentiable through all components.

The controller network in this case is just a standard RNN with a single layer consisting of parameters W_x, W_y and W_M, that has access to the network input $x(t)$, the hidden state $y(t-1)$ and the k top-most elements on the stack $M_{0:k}(t-1)$:

$$y(t) = \sigma(W_x x(t) + W_y y(t-1) + W_M M_{0:k}(t-1)) \tag{32}$$

Common modifications to this structure are the extension to multiple stacks and the renouncement of the `NO-OP` action.

3.3 Giving RNNs controllable RAM

The networks described in the first section were all limited to read-only memory. Following the hypothesis in section 2.2.2, these methods will still be restricted to the class of regular languages, i.e. they can only simulate finite state automata. The extension to stack augmented RNNs introduces a very restricted set of operations for read and write actions and should give the network the computational power of a push down automaton. Following this reasoning, an RNN extended with multiple stacks is supposed to be Turing complete. Nevertheless, the stack datastructure imposes a hard restriction on the possible operations to be performed on the memory. Such a restriction may very well be counterproductive in practice, since it can make it harder for the network to find the right solution from input-output data alone. To extend over this limitation, the methods described in this section represent three different approaches to give the controller network random read and write access to the memory.

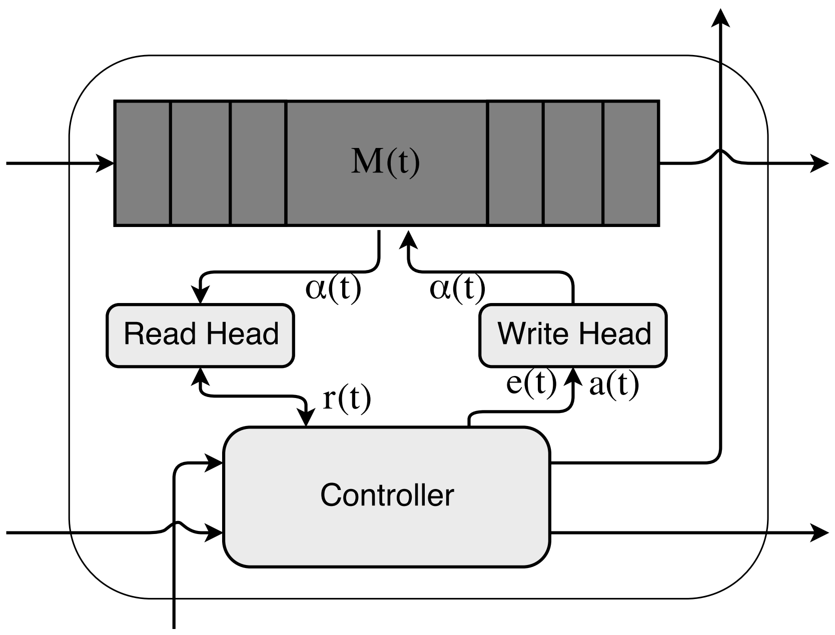

Figure 4: Illustration of a NTM.

3.3.1 Neural Turing Machines (NTMs)

Neural Turing machines (NTMs) as introduced by [13] are designed to follow the general architecture described in the beginning of this section. Therefore, an NMT is composed of a controller network which can be an RNN or feed forward network and multiple write and read heads to perform operations on the memory matrix $M(t)$ at each timestep t. The operations on the memory are always performed on the whole memory matrix after weighting each memory column i by the corresponding weight $\alpha_i(t)$. The weights which essentially represent an addressing method are computed by a special variant of the attention mechanism.

Consequently, the reading operation is very similar to the attention-based approach and the MemN2N approach described in section 3.1:

$$r(t) = \sum_{i=1}^{N} \alpha_i(t) M_i(t) \tag{33}$$

The additional writing operation is performed in two steps. First, the information is erased by multiplying the complement of the erase vector $e(t)$ weighted by $\alpha_i(t)$ with the old memory matrix at column i:

$$\tilde{M}_i(t) = M_i(t-1)[1 - \alpha_i(t) e(t)] \tag{34}$$

That means that each position j in $M(t-1)$ for that $e_j(t)$ is one or close to one gets deleted. In the second step of each writing operation, the cleaned memory $\tilde{M}(t)$ gets updated by summation with the add vector $a(t)$.

$$M_i(t) = \tilde{M}_i(t) + \alpha_i(t) a(t) \tag{35}$$

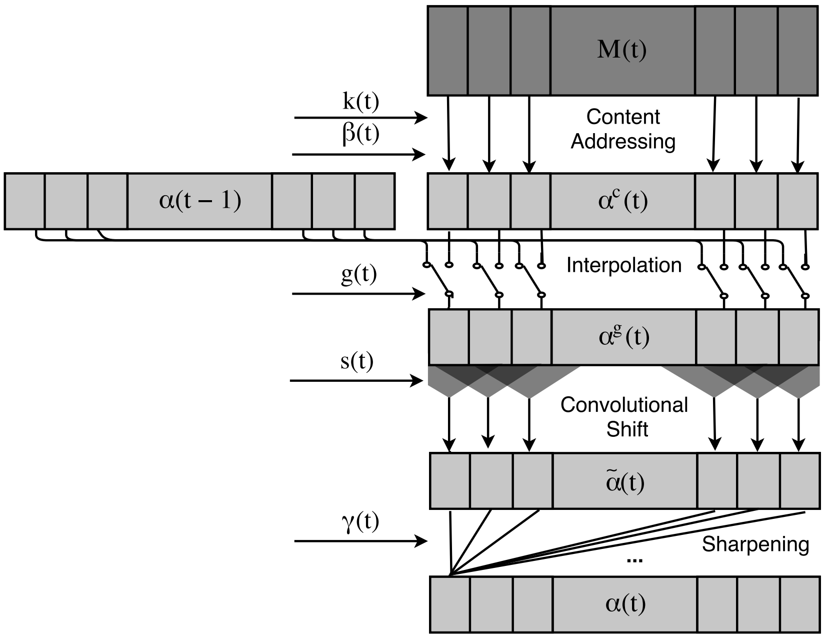

Figure 5: Illustration of the location and content based address computation for the NTM.

The attention weights for both the read and the write heads is computed by combining two addressing mechanisms with complementary facilities. The first addressing mechanism implements a content-based attention mechanism similar to the addressing described in section 3.1. In this case, the attention energies $\tilde{\alpha}_i^c(t)$ are computed as the cosine similarity ($K[u, v] = \frac{u \cdot v}{||u|| \cdot ||v||}$) between a key vector $k(t)$, generated by the controller network, and the memory column $M_i(t)$:

$$\tilde{\alpha}_i^c(t) = \beta(t) K[k(t), M_i(t)] \tag{36}$$

The additional key strength factor $\beta(t)$ in this equation is also emitted by the controller network and determines the precision of the focus for the resulting attention weights. The attention energies are then normalized by a softmax function, similar to equation (23), to obtain the content-based attention weights $\alpha_i^c(t)$.

Then, the second addressing mechanism computes the final attention-weights, based on these content-based attention weights, w.r.t. positional information. The location-based approach again works in several steps. First, the positional information is included by interpolating the newly computed content-based attention $\alpha^c(t)$ with the old attention weights $\alpha(t)$ from the previous timestep:

$$\alpha^g(t) = g(t)\alpha^c(t) + [1 - g(t)]\alpha(t-1) \tag{37}$$

The interpolation is weighted by the interpolation gate $g(t)$ which takes values in the range from zero to one. In the next step, the result of the gated interpolation is convolved with a shift vector $s(t)$:

$$\tilde{\alpha}_i(t) = \sum_{j=0}^{N-1} \alpha_j^g(t) s_{i-j}(t) \tag{38}$$

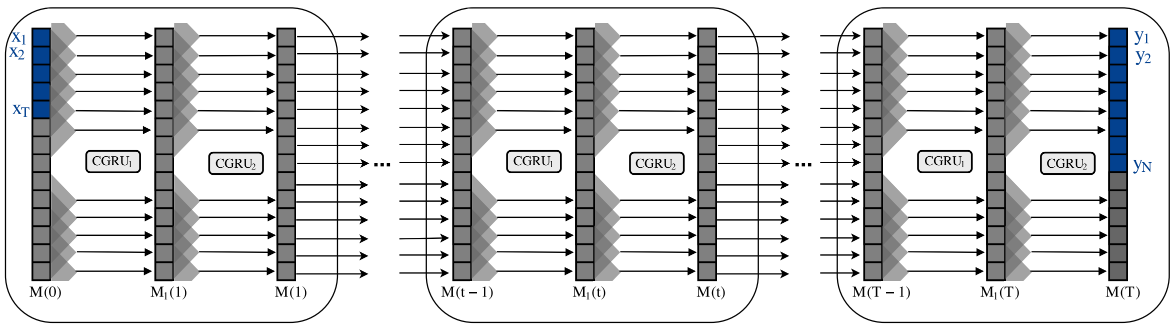

Figure 6: Neural GPU for one timestep with 2 layers and $w = 3$.

Here, the shift vector $s(t)$ represents a distribution over all allowed integer shifts, which can be emitted e.g. by a softmax function. This one dimensional convolution shifts the information in $\alpha^g(t)$ according to the distribution given by $s(t)$ and can therefore be especially useful to access neighboring cells as it is needed in many algorithms.

The interpolation and especially the shift operation can lead to a blurred distribution in $\tilde{\alpha}(t)$. Therefore, the last step to obtain the final attention weights $\alpha_i(t)$ performs a sharpening with the parameter $\gamma(t) \geq 1$:

$$\alpha_i(t) = \frac{\tilde{\alpha}_i(t)^{\gamma(t)}}{\sum_{j=1}^{N} \tilde{\alpha}_j(t)^{\gamma(t)}} \tag{39}$$

An extension to this approach is proposed by Gulcehre et al. [15], who introduce the dynamic neural Turing machine (D-NTM) as a generalization of the NTM. For this extension, they introduce two important new concepts into the model. First, a learnable addressing scheme that gives each memory cell in $M_i(t)$ a specific address value a_i that is a parameter which can be optimized jointly with the whole network. Second, they propose a discrete, hard, addressing to replace the soft attention-based addressing. The explicit addressing scheme allows the D-NTM to perform highly nonlinear location-based addressing. Additionally, the hard addressing, which essentially computes a one-hot vector, enables a more precise memory control and reduces the computation time in practice. The downside to this discrete addressing is clearly that it is not differentiable and needs to be trained with reinforcement learning instead of a gradient based training method.

3.3.2 Neural GPUs

Attention-based approaches, including the NTM model, are designed to focus the read and write actions on specific parts of the memory. This access method is very inefficient since the whole memory needs to be touched while only one value is significantly changed afterwards. There are several different approaches to solve this for the existing NTM model by introducing sparse read and write operations [32] or using hard attention [15]. These improvements of the standard NTM architecture accelerate the memory operations but at the same time prevent the network from accessing the whole memory in each step.

The active memory approach [20] offers an alternative to the focused access method of the attention mechanism. Active memory methods accesses the whole memory uniformly in each write and read operation, e.g. by applying a convolutional operation, such that the computation is performed over the entire memory. In contrast to the attention approach, this method is highly efficient due to the fact that all operations can be performed in parallel.

The first approach that follows the concept of active memory is the neural GPU [21]. The neural GPU uses a convolutional gated recurrent unit (CGRU), which is a modified version of the standard GRU as it was described in section 2.1.2. The CGRU has an expanded memory and changed interaction with the memory to convolutional operations. The hidden state $y(t)$ of the traditional GRU is replaced by a memory tensor $M(t) \in \mathbb{R}^{w \times h \times m}$, which contains a $w \times h$ vectors of size m. Apart from the new dimensions of the hidden state, the update operation remains unchanged from equation (18) and is therefore given by:

$$M(t) = g_u(t) \odot \tilde{M}(t) + [1 - g_u(t)] \odot M(t-1) \tag{40}$$

The important change compared to the standard GRU is the computation of the update candidate $\tilde{M}(t)$, which no-longer gets any direct input from the outside, but instead has to work solely on the Memory. It does this by convolving the weight matrix W_M over the memory tensor after filtering it through the reset gate $g_r(t)$.

$$\tilde{M}(t) = \sigma_{\tanh}(W_M * [g_r(t) \odot M(t)] + B_M) \tag{41}$$

The computation of the gate activations is also obtained through convolutional operations:

$$g_u(t) = \sigma(W_u * M(t) + B_u) \tag{42}$$
$$g_r(t) = \sigma(W_r * M(t) + B_r) \tag{43}$$

Where the convolution of a kernel bank $W \in \mathbb{R}^{k_w \times k_h \times m \times m}$ of width k_w and height k_h with the memory tensor is defined as:

$$W * M_{x,y,i} = \sum_{u=\lfloor -k_w/2 \rfloor}^{\lfloor k_w/2 \rfloor} \sum_{v=\lfloor -k_h/2 \rfloor}^{\lfloor k_h/2 \rfloor} \sum_{c=1}^{m} M_{x+u,y+v,c} \cdot W_{u,v,c,i} \tag{44}$$

Applying such a CGRU to an input $i_1, \ldots, i_N$ written in $M(0)$ results in highly parallelized computations. The neural GPU exploits this structure and applies L different CGRUs in succession for N steps on an input of length N. This results in $N \cdot L$ applications of CGRUs after which the output can be obtained from the memory as:

$$y = \max_k \{W_y M_{0,k,:}(N)\} \tag{45}$$

For the training, the maximization has to be handled by a softmax function.

3.3.3 Associative LSTM

The associative LSTM [8] is another approach to extend a neural network with additional RAM. But different to the explicit addressing in the attention-based read and write heads of the NTM, the associative LSTM integrates the memory directly into the controller network.

Therefore, the associative LSTM approach modifies the standard LSTM, as it was explained in section 2.1.1, such that it allows for key-value storage in the cell state of

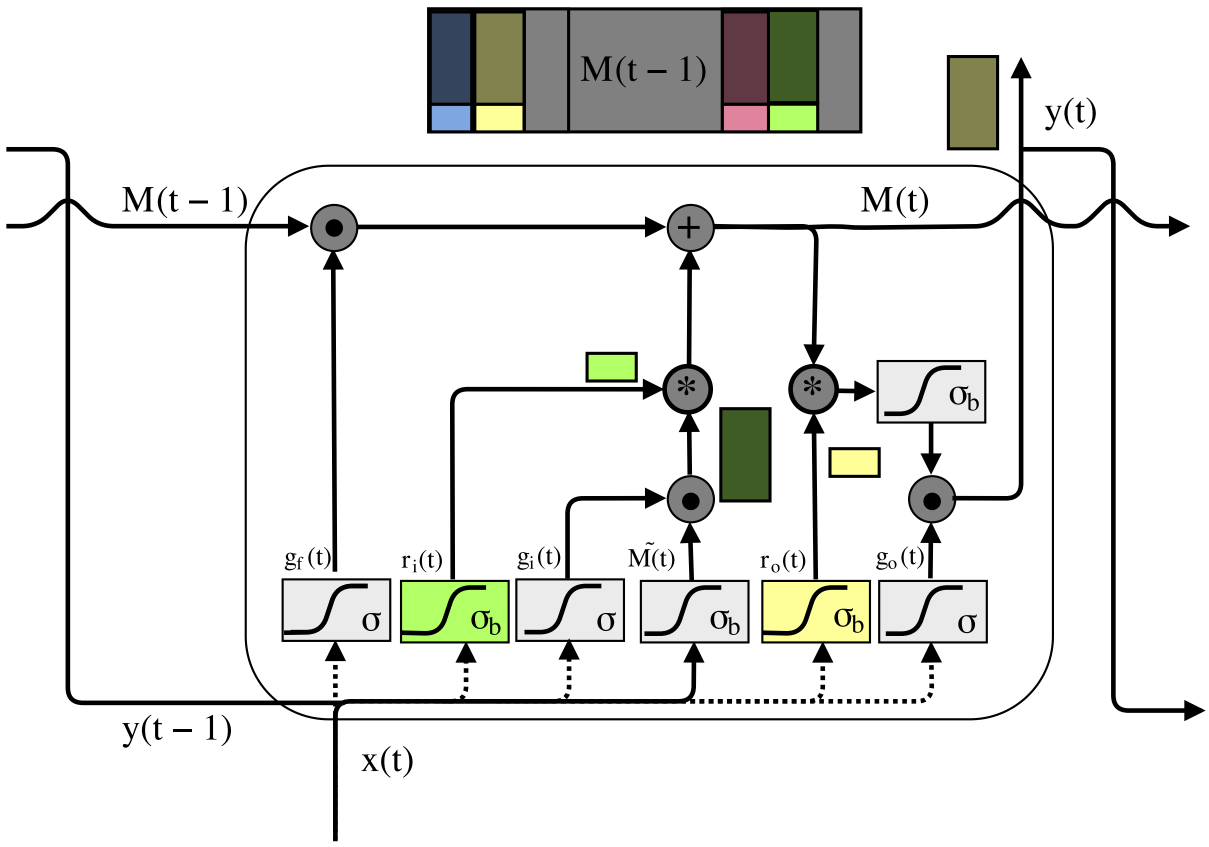

Figure 7: Associative LSTM

the LSTM. The approach combines holographic reduced representations [30] which are performed using complex vectors $z = h_{\text{real}} + ih_{\text{imaginary}}$ with the LSTM architecture to achieve this. Such complex vectors are stored with the following convention:

$$h = \begin{bmatrix} h_{\text{real}} \\ h_{\text{imaginary}} \end{bmatrix} \tag{46}$$

With this convention, the gate activations are simply computed as:

$$g_\star(t) = \begin{bmatrix} \sigma[\hat{g}_\star(t)] \\ \sigma[\hat{g}_\star(t)] \end{bmatrix} \text{ for } \star \in \{f, i, o\} \tag{47}$$

Where $\hat{g}_\star(t)$ is defined as for the standard LSTM, by:

$$\hat{g}_\star(t) = W_{x\star}x(t) + W_{y\star}y(t-1) + b_\star \text{ for } \star \in \{f, i, o\} \tag{48}$$

Additional associative keys $r_\star$, which are used to control the operations on the map, are computed in the same way as the gate activations:

$$\hat{r}_\star(t) = W_{x\star}x(t) + W_{y\star}y(t-1) + b_\star \text{ for } \star \in \{i, o\} \tag{49}$$

$$r_\star(t) = \sigma_{\text{bound}}[\hat{r}_\star(t)] \text{ for } \star \in \{i, o\} \tag{50}$$

The main difference here is the activation function σ_{bound}, which was defined to restrict the modulus of a complex number to the range of zero to one, by elementwise division of the real and imaginary part with $d = \max(1, \sqrt{h_{\text{real}} \odot h_{\text{real}} + h_{\text{imaginary}} \odot h_{\text{imaginary}}})$:

$$\sigma_{\text{bound}}(h) = \begin{bmatrix} h_{\text{real}} \oslash d \\ h_{\text{imaginary}} \oslash d \end{bmatrix} \tag{51}$$

As mentioned above, the associative LSTM approach uses the cell-state as the memory. Consequently, the notation used for the standard LSTM in section 2.1.1, is modified to reflect this, by referring to the cell-state $c(t)$ as memory $M(t)$ in the following. Therefore, the update candidate for the memory is defined analogously to equation (15). The difference here is that the update candidate is getting complex vectors as input and is hence also applying σ_{bound} as activation function:

$$\tilde{M}(t) = \sigma_{\text{bound}}(W_{xM}x(t) + W_{yM}y(t-1) + b_M) \tag{52}$$

The implementation of map storage and lookup in $M(t)$ works by exploiting holographic reduced representations through complex multiplications. For this purpose, the output key r_o and the storage key r_i are introduced and the update equation (16) is extended to store the new information $g_i(t) \odot \tilde{M}(t)$ under the corresponding permuted input key $r_{i,s}(t)$:

$$M_s(t) = g_f(t) \odot M_s(t-1) + r_{i,s}(t) \circledast [g_i(t) \odot \tilde{M}(t)] \tag{53}$$

The challenging property of holographic reduced representations is the large amount of noise that gets introduced in the process. To counteract this property, the memory state is saved in $s = 1, \ldots, N_{\text{copies}}$ redundant copies $M_s(t)$, which only differ in the permutation introduced for the permuted input and output keys by the constant random permutation matrix P_s:

$$r_{\star,s}(t) = \begin{bmatrix} P_s & 0 \\ 0 & P_s \end{bmatrix} r_\star(t) \text{ for } \star \in \{i, o\} \tag{54}$$

Note that the mapping to the input key $r_{i,s}(t)$ is performed by elementwise complex multiplication $\circledast$ defined as:

$$a \circledast b = \begin{bmatrix} a_{\text{real}} \odot b_{\text{real}} - a_{\text{imaginary}} \odot b_{\text{imaginary}} \\ a_{\text{real}} \odot b_{\text{imaginary}} + a_{\text{imaginary}} \odot b_{\text{real}} \end{bmatrix} \tag{55}$$

Finally, the output of each timestep is given by an average over the lookup results for each redundant copy:

$$y(t) = g_o(t) \odot \sigma_{\text{bound}} \left(\frac{1}{N_{\text{copies}}} \sum_{s=1}^{N_{\text{copies}}} r_{o,s}(t) \circledast M_s(t) \right) \tag{56}$$

4 Results and Discussion

4.1 Results

The methods presented above are in theory all applicable to the same tasks without any larger modifications. Unfortunately, there is no universal benchmarking task to evaluate the computational power of neural networks. Therefore, the authors who proposed these approaches in the first place used many different tasks to demonstrate the abilities of their methods instead.

However, some papers perform comparisons with a selection of different models. In the paper that introduces the neural GPU [21], the authors perform a comparison with the stack augmented RNN model [19] and an LSTM extended by the attention mechanism

[38]. This comparison was done on the task of binary addition and binary multiplication after being trained on examples with an input length of up to 20 digits. For this task, the neural GPU showed perfect scores not only for the input lengths it was trained on but also for much longer input with a length of up to 2000 digits. On the binary addition task, the stack augmented RNN, as well as the LSTM with attention, achieves a perfect score for length 20 but fails to generalize to longer inputs. The authors are not providing any results for the stack augmented RNN on the task of binary multiplication but the results for the LSTM with attention show that this approach is not able to learn this algorithm and even fails to fit the training data. These results show the strong limitations of read-only memory and stack structured memory and further support the hypothesis from section 2.2.2 that standard RNNs can only approximate finite state machines in practice. Similar experiments on an arithmetic task, that requires the network to add and subtract two long numbers, were performed with the standard and associative LSTM by Danihelka et al. [8]. They show that an associative LSTM is able to solve this task. A standard LSTM does not achieve the same performance, however, if the hidden state of the LSTM is increased, the LSTM is able to learn this task, although requiring 60% more parameters. However, the authors do not state anything about the generalization ability and performance on longer inputs.

The only task that can be seen as a consistent benchmark amongst the described methods is the Facebook bAbI QA task [40], which has been used to compare LSTM, MemN2N, NTM and D-NTM models [15]. This task is composed of a set of twenty different sub-tasks that are designed to determine whether a system is able to answer questions via chaining facts, simple induction, deduction and much more. The results show that MemN2N models perform best on this task with an average error rate of 4.24% and only 3 sub-tasks that show an error rate of more than 5%. Whereas NTM and D-NTM approaches lie far behind the MemN2N with average error rates of 32.76% (NTM) and 21.79% (D-NTM) respectively. Nevertheless, both approaches still show a large improvement over the LSTM baseline, which has an average error rate of 36.41%.

Apart from these comparisons, the individual results for each of the presented approaches clearly show that the memory extensions will be superior to standard RNNs and LSTMs on most tasks. This is supported by the fact that all approaches can report multiple experiments in which they outperform the standard RNN and even LSTM models.

4.1.1 Application to Machine Translation

Although the results presented above are very impressive, especially in comparison with traditional architectures, the fact that most of these tasks are only toy-tasks designed to benchmark these architectures makes the interpretation of the results very difficult. In particular, the performance on real-world tasks determines the impact of an approach in the long run.

One example for a relevant real-world task would be the area of machine translation. This field is particularly interesting since there has already been a large impact by attention-based models, which represent today's state-of-the-art [1] and outperform traditional methods on most translation tasks.

There are several approaches that build on the attention-based method by Bahdanau et al. and integrate techniques from NTMs in the decoder of the network. Wang et al. [39] and similarly Meng et al. [28] replace the standard content-based read operation of the attention mechanism by the read and write operations of NTMs, which are performed on the encoded input in every decoder timestep. Meng et al. [29] further extend this approach by using multiple layers of read and write operations that should enable the decoder to perform complex reasoning steps before generating an output word. All three approaches show significant gains in translation quality over the state-of-the-art attention-based approaches. Therefore, it is reasonable to conclude that improved memory control can be beneficial for models on translation tasks and may also help on related sequence-to-sequence tasks.

A different approach is proposed by Kaiser et al. [20], who try to replace the attention-based architecture by Bahdanau et al. [1] with a neural GPU. The concept simply writes the input sentence in the initial memory $M(0)$ and trains the neural GPU to compute the translated sentence as output after the last computation step. However, this naive application of neural GPUs to the problem did not succeed for translation. Instead, a modification called the extended neural GPU achieves competitive performance. For this modification, the neural GPU gets extended by additional layers of a CGRU-based decoder. This decoder is applied after the final computation step and performs its computations on the memory provided by the final state of the encoder CGRUs, while predicting one output word in each step. This modified version of the neural GPU achieves state-of-the-art performance on a translation task and outperforms the attention-based approach especially on long sentences.

4.2 Discussion

To discuss the results that were described above and compare the approaches that were presented in section 3, one has to clearly distinguish two viewpoints on the topic. On the one hand the theoretical categorization in the computational hierarchy [2] and on the other hand, the computational power that was shown in practical applications.

The attention-based methods described in section 3.1, for example, should, in theory, be no more powerful than a standard RNN, since they only offer a more advanced read access on the input. However, the practical results of attention-based NMT systems, as well as the successful application of the MemN2N approach to the bAbI task, demonstrate that attention-based extensions to the RNN can be very beneficial in practice. Both results are especially noteworthy since they involve natural language processing, while natural language is widely believed to be no finite-state language [4].

To also increase the theoretical power of RNNs, the logical first step would be to add an additional stack datastructure to the network. This should, in theory, result in a model that can simulate push-down automata, which are equivalently the extension of a finite-state machine with a stack datastructure. Following the findings in automata theory[18], adding a second stack to the RNN should give it an equivalent computation power to a two-stack machine and would, therefore, result in a Turing-complete model. The results reported by Joulin and Mikolov [19] clearly show that this idea also carries over to practical experiments, e.g. the binary addition task for which the RNN augmented with multiple stacks clearly outperforms traditional RNNs. The NTM approach is also inspired by the architecture of a theoretical model, namely that of the Turing machine [37]. Additionally, the neural GPU shows many similarities to the concept of a 2-dimensional cellular automaton, which is a very simple model of computation but nevertheless Turing

[2]i.e. the level in the computational hierarchy a model should be placed if standard RNNs are assumed to be equivalent to finite state machines

complete [18]. The experiments in the corresponding papers all show promising results which demonstrate that the models are performing well on tasks that seem impossible for traditional RNN models.

Although the models described above are very powerful and are also indicating Turing completeness in practice on some tasks, the most important attribute for these models is supposed to be the generalization ability. The common approach for these models to improve the generalization ability is the extension with additional memory. The additional memory is accessed in a way such that it is independent of the size of the memory so that the memory can be expanded without retraining. This is commonly solved by using dynamic addressing using the attention mechanism as it is done in MemN2N, NTM and stack augmented RNNs. Neural GPUs achieve the same by using convolutional operations for accessing the memory, which makes it applicable to a memory of arbitrary size due to the parameter sharing performed by the convolutional operation. The associative LSTM is a special case since here the cell-state of the LSTM has a fixed size, but the holographic reduced representation which is used to store the information can contain an arbitrary number of inputs. However, the actual number of elements that can be contained in the cell-state is limited by the limited precision of the floating point representation that is used in practice. Additionally, storing more information in the cell-state leads to increased noise on the retrieved data. This effect can be mitigated by adding more permuted copies of the memory to the network.

Even though the approaches of memory augmented RNNs are in principle completely flexible w.r.t. memory size, there are still two hyperparameters that need to be determined in advance by the user for each application. These hyperparameters are the size of the memory and more importantly the number of computation steps the network is supposed to perform before the final output. The latter can be integrated into the RNN by applying the adaptive computation time (ACT) algorithm [11]. In this extension that can be easily applied to any type of RNN and in particular to each of the models described above, an additional halting unit is introduced.

$$h^n(t) = \sigma(W_h s^n(t) + b_h) \tag{57}$$

This halting unit is used to compute a probability that determines in each intermediate step $n = 1, \dots, N(t)$ for each time step t the halting probability which determines that the timestep t is ended in this intermediate step n:

$$p^n(t) = \begin{cases} R(t) & \text{if } n = N(t) \\ h^n(t) & \text{otherwise} \end{cases} \tag{58}$$

where $N(t) = \min\{n' : \sum_{n=1}^{n'} h^n(t) \geq 1 - \epsilon\}$ and $R(t) = 1 - \sum_{n=1}^{N(t)-1} h^n(t)$.

The hidden state s and the output y are also computed for every intermediate step:

$$s^n(t) = \begin{cases} f_{\text{hidden}}(s(t-1), x^1(t)) & \text{if } n = 1 \\ f_{\text{hidden}}(s^{n-1}(t), x^n(t)) & \text{otherwise} \end{cases} \tag{59}$$

$$y^n(t) = W_y s^n(t) + b_y \tag{60}$$

with $x^n(t) = x(t) + \delta_{n,1}$.

These intermediate hidden states $s^n(t)$ and outputs $y^n(t)$ are used to compute the final update for state and output in time step t, by computing the weighted average with the probailities $p^n(t)$:

$$s(t) = \sum_{n=1}^{N(t)} p^n(t) s^n(t) \qquad\qquad y(t) = \sum_{n=1}^{N(t)} p^n(t) y^n(t) \tag{61}$$

Using this approach, Graves successfully showed that the network is able to learn the optimal number of computation steps it needs to produce each output. The only parameter that is still optimized by hand is the size of the additional memory for the models described above. Although an optimized number of computation steps might be more important for the overall runtime, the size of the memory is also very important since a larger memory than necessary would slow down all access operations while a memory chosen too small would make it harder for the network to reach the correct solution. There is no solution for this problem yet, however it might be possible to develop a solution similar to ACT for this issue.

5 Conclusion

This work deals with the subject of memory extensions for recurrent neural networks and the corresponding computation power that can be achieved with RNNs. There are theoretical results which state that recurrent neural networks are able to express any computable function, i.e. that they are Turing complete. But, as shown in section 2.2.2 and as can be shown in experiments (section 4), this proof does not hold in practice. Furthermore, it is reasonable to assume from the experimental results in the literature that RNNs can in practice only learn to approximate finite state machines. This limitation would prevent RNNs from computing algorithms for many interesting problems. Therefore, in the last two years, many methods have been proposed that try to solve this problem by extending the RNN with additional memory and offering more advanced control operations to modify this memory.

This work summarizes the most prominent approaches to this subject by determining the similarities and differences and defining an abstract common structure that fits for all of them. The different models are classified as random access read-only memory [1, 34], stack augmented [19] and random access read-and-writeable memory [13, 15, 21, 8] networks.

The results that are discussed in section 4 show clearly that more advanced algorithmic tasks require a more advanced memory extension. However, for the question answering benchmark bAbI, the most successful approach uses a limited read-only memory. Apart from the toy-tasks, there have already been several promising attempts to apply the concept of advanced memory control to real-world problems.

Also worth mentioning is the fact that the models covered in this report are only the initial execution of the corresponding approach. There have already been several publications in the last 2 years that extend MemN2N [42, 24], NTM [15, 14] and neural GPU [31, 20] models and report further improvements. Therefore it is safe to assume that this particular area of research will see even more attention in the upcoming years.

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

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

[3] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734. Association for Computational Linguistics, October 2014.

[4] Noam Chomsky, Morris Halle, and Fred Lukoff. On accent and juncture in english. *For Roman Jakobson*, pages 65–80, 1956.

[5] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 577–585. Curran Associates, Inc., 2015.

[6] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, December 2014.

[7] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[8] Ivo Danihelka, Greg Wayne, Benigno Uria, Nal Kalchbrenner, and Alex Graves. Associative long short-term memory. In *The 33rd International Conference on Machine Learning*, page pp. 19861994, 2016.

[9] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[10] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, August 2013.

[11] Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

[12] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[13] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[14] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[15] Caglar Gulcehre, Sarath Chandar, Kyunghyun Cho, and Yoshua Bengio. Dynamic neural turing machine with soft and hard addressing schemes. *arXiv preprint arXiv:1607.00036*, 2016.

[16] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, page 91, 1991.

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

[18] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Automata theory, languages, and computation. *International Edition*, 24, 2006.

[19] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems*, pages 190–198, 2015.

[20] Łukasz Kaiser and Samy Bengio. Can active memory replace attention? In *Advances In Neural Information Processing Systems*, pages 3774–3782, 2016.

[21] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[22] John F Kolen. Fool's gold: Extracting finite state machines from recurrent network dynamics. *Advances in neural information processing systems*, pages 501–501, 1994.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[24] Ankit Kumar, Ozan Irsoy, Jonathan Su, James Bradbury, Robert English, Brian Pierce, Peter Ondruska, Ishaan Gulrajani, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. *arXiv preprint arXiv:1506.07285*, 2015.

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[26] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[27] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. *Efficient BackProp*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[28] Fandong Meng, Zhengdong Lu, Hang Li, and Qun Liu. Interactive attention for neural machine translation. *arXiv preprint arXiv:1610.05011*, 2016.

[29] Fandong Meng, Zhengdong Lu, Zhaopeng Tu, Hang Li, and Qun Liu. A deep memory-based architecture for sequence-to-sequence learning. *arXiv preprint arXiv:1506.06442*, 2015.

[30] Tony A Plate. Holographic reduced representations. *IEEE Transactions on Neural Networks*, 6(3):623–641, 1995.

[31] Eric Price, Wojciech Zaremba, and Ilya Sutskever. Extensions and limitations of the neural gpu. *arXiv preprint arXiv:1611.00736*, 2016.

[32] Jack Rae, Jonathan J Hunt, Ivo Danihelka, Timothy Harley, Andrew W Senior, Gregory Wayne, Alex Graves, and Tim Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes. In *Advances In Neural Information Processing Systems*, pages 3621–3629, 2016.

[33] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449. ACM, 1992.

[34] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in neural information processing systems*, pages 2440–2448, 2015.

[35] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Interspeech*, pages 194–197, 2012.

[36] Peter Tino, Bill G Horne, and C Lee Giles. Finite state machines and recurrent neural networks–automata and dynamical systems approaches. 1998.

[37] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[38] Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *Advances in Neural Information Processing Systems*, pages 2773–2781, 2015.

[39] Mingxuan Wang, Zhengdong Lu, Hang Li, and Qun Liu. Memory-enhanced decoder for neural machine translation. *arXiv preprint arXiv:1606.02003*, 2016.

[40] Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015.

[41] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

[42] Caiming Xiong, Stephen Merity, and Richard Socher. Dynamic memory networks for visual and textual question answering. *arXiv preprint arXiv:1603.01417*, 2016.

[43] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard S Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2(3):5, 2015.