Karlsruhe Institute of Technology

Analysis and Advancement of Differentiable Neural Computers for Question Answering

Master's thesis
submitted by

Jörg Franke

at the Interactive Systems Lab
Institute for Anthropomatics and Robotics
Karlsruhe Institute of Technology (KIT)

Reviewer:	Prof. Dr. Alexander Waibel
Second reviewer:	Prof. Dr. Tamim Asfour
Advisor:	Dr. Jan Niehues

Process Period: 14. October 2017 – 13. April 2018

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 13. April 2018

Abstract

The differentiable neural computer (DNC), a memory-augmented neural network introduced by Graves et al. in 2016 [1], was designed as a general problem solver which could be used in a wide variety of tasks. It separates computation and memorization using a controller as well as a readable and writable memory matrix. This allows long-term storage ability and an enclosed representation and manipulation of complex data structures. Furthermore, a more precise model design is possible due to the separation of computation and memory components. The model is fully differentiable; therefore, it can be trained with data samples by backpropagation.

This work considers the DNC in solving question answering (QA) tasks. Herein, a model receives a context in natural language and is asked to answer a question. This is a current issue in research and merges the intersection of natural language processing, information retrieval and machine comprehension. A DNC seems to be predestined for QA tasks since they require the ability to store information over a longer time frame and exploits the memorization to create an answer. However, the DNC is currently only applied to small synthetic QA tasks.

The goal of this work is to successfully apply a DNC based model to common large-scale QA datasets. Hence, an extensive analysis of the DNC examines the functionality and the computational consumptions. The analysis identifies several possible improvements which keep the general character of this model intact but make it more robust and scalable. The results lead to an advanced DNC with respect to QA. This advanced DNC is easier to apply through a robust training focused on memory usage, a slim memory unit which reduces computational resources and a bidirectional architecture.

This allows an efficient training of the model on large-scale QA tasks in a word-by-word fashion without the need of additional sentence representations or an attention mechanism. According to the best of my knowledge, the advancements not only achieve new state-of-the-art performance on the bAbI task with zero failed tasks but also minimize the performance variance between different initializations. The goal of this work—an easier applicability of the DNC to new QA tasks—is reached with competitive results on the children book test and the CNN reading comprehension task.

Zusammenfassung

Der differenzierbare neuronale Computer (DNC) ist ein speicher-erweitertes neuronales Netzwerkmodell, welches von Graves et al. 2016 eingeführt wurde [1]. Das Modell ist als allgemeiner Problemlöser konzipiert, welcher für eine Vielzahl von Aufgaben eingesetzt werden kann. Es trennt die Berechnung und die Speicherung innerhalb des Netzwerkmodells mit Hilfe eines Controllers und einer les- und schreibbaren Speichermatrix. Dies ermöglicht eine langfristige Speicherung von Informationen in einer kompakten Repräsentation und die Möglichkeit Gespeichertes zu verändern. Darüber hinaus erlaubt die Trennung einen präziseren Modellentwurf. Durch die vollständige Differenzierbarkeit ist es überwacht trainierbar mit Beispieldaten und durch die Rückführung von Fehlern.

Diese Arbeit untersucht das DNC Modell in der Anwendung zur Beantwortung von Fragen. Dabei erhält das Modell einen Kontext in natürlicher Sprache und soll eine dazugehörige Frage beantworten. Diese Art von Anwendung ist aktueller Gegenstand der Forschung und befindet sich in der Schnittmenge von natürlicher Sprachverarbeitung, Informationsrückgewinnung und maschinellem Sprachverständnis. Eine DNC scheint für diese Art von Aufgabe prädestiniert zu sein, da diese die Fähigkeit erfordert Informationen über einen langen Zeitraum zu speichern und das Gespeicherte zu nutzen, um die Antwort zu bilden. Allerdings wird das DNC derzeit nur bei kleinen, synthetischen Question-Answering (QA) Datensätzen eingesetzt.

Das Ziel dieser Arbeit ist die erfolgreiche Anwendung eines DNC-basierten Modells auf gängigen, großen QA Datensätzen. Zu Beginn untersucht eine umfangreiche Analyse das DNC, dessen Funktionalität und die benötigten Rechenressourcen. Die Analyse identifiziert mehrere mögliche Verbesserungen, die den allgemeinen Charakter dieses Modells beibehalten, es aber robuster und skalierbarer machen. Die Ergebnisse führen zu einem verbesserten DNC für die Anwendung auf QA Datensätzen. Dieses verbesserte DNC ist einfacher anzuwenden, da ein robustes Training eine frühe Speichernutzung erzwingt, eine sparsame Speichereinheit den Bedarf an Rechenressourcen reduziert und eine bidirektionale Architektur den Kontext besser aufbereitet.

Diese Erweiterungen ermöglicht ein effizientes Training des DNC an großen QA Datensätzen ohne das zusätzliche Satzkompressionen oder ein Aufmerksamkeitsmechanismus erforderlich sind. Nach meinem besten Wissen erreichen die Weiterentwicklungen nicht nur ein neues State-of-the-Art-Ergebnis auf dem bAbI-Datensatz, sondern minimieren auch die Ergebnisvarianz zwischen verschiedenen Initialisierungen. Die praktikable und erfolgreiche Anwendbarkeit des DNC auf großen QA Datensätzen wird anhand des „Children Book Test" und der CNN-Leseverständnisaufgabe gezeigt.

Contents

1 Introduction **1**
- 1.1 Learnable computers . . . 1
- 1.2 Goals . . . 2
- 1.3 Structure of work . . . 4

2 Basics **5**
- 2.1 Artificial Neural Networks . . . 5
- 2.2 Recurrent Neural Networks . . . 8
- 2.3 Long Short-Term Memory . . . 10
- 2.4 Bidirectional RNN . . . 14
- 2.5 Differentiable Neural Computer . . . 14
  - 2.5.1 System overview . . . 15
  - 2.5.2 The memory unit . . . 16
    - 2.5.2.1 Memory unit overview . . . 16
    - 2.5.2.2 Generating control signals . . . 18
    - 2.5.2.3 Write mechanism . . . 19
    - 2.5.2.4 Memory update . . . 22
    - 2.5.2.5 Read mechanism . . . 22
  - 2.5.3 Summary . . . 26

3 Data **27**
- 3.1 Copy Task . . . 27
- 3.2 bAbI 20 Task . . . 28
- 3.3 Children Book Test . . . 30
- 3.4 CNN Reading Comprehension Task . . . 32

4 Related work **35**
- 4.1 Related models for QA . . . 35
- 4.2 Related enhancements . . . 37

5 Analysis of the DNC **39**
- 5.1 DNC Training . . . 39
- 5.2 DNC Functionality . . . 42
- 5.3 DNC Memory consumption . . . 49
- 5.4 DNC Computation time . . . 52
- 5.5 Analysis conclusion . . . 53

6 Advancements in the DNC **55**
- 6.1 Robust DNC training . . . 55

6.1.1 DNC Normalization 55
6.1.2 Bypass Dropout 57
6.2 Advanced Architecture 58
6.2.1 Bidirectional DNC 58
6.2.2 Atop RNN 59
6.3 Content-Based Memory Unit 60

7 Experiments **63**
7.1 Empirical methods evaluation 63
7.1.1 Training and architecture advancements 63
7.1.1.1 bAbI Task 1 evaluation 63
7.1.1.2 Copy Task evaluation 66
7.1.2 Content-Based Memory Unit 67
7.2 bAbI 20 Task 68
7.2.1 Task 16 augmentation 68
7.2.2 Training details 69
7.2.3 Results 70
7.3 Children Book Test 72
7.3.1 Training details 72
7.3.2 Results 73
7.4 CNN Reading Comprehension Task 73
7.4.1 Training details 73
7.4.2 Results 74
7.5 Results overview 75

8 Conclusion **77**
8.1 Summary 77
8.2 Discussion 78
8.3 Further work 79

List of Figures **93**

List of Tables **95**

List of Abbreviations

ADNC	Advanced Differential Neural Computer
ANN	Artificial Neural Network
BiADNC	Bidirectional Advanced Differential Neural Computer
BiDNC	Bidirectional Differentiable Neural Computer
BiLSTM	Bidirectional Long Short-Term Memory
BiRNN	Bidirectional Recurrent Neural Network
BD	Bypass Dropout
BN	Batch Normalization
BP	Backpropagation
BPTT	Backpropagation Through Time
CBA	Content-based addressing
CBMU	Content-Based Memory Unit
CBT	Children Book Test
CNN	Cable News Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DNC	Differential Neural Computer
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HMM	Hidden Markov Model
LM	Linkage Matrix
LN	Layer Normalization
LSTM	Long Short-Term Memory
MANN	Memory-Augmented Neural Network
MemNN	Memory Neural Network
MLP	Multilayer Perceptron
MU	Memory Unit
NLP	Natural Language Processing
NTM	Neural Turing Machine
QA	Question Answering
RAM	Random-Access Memory
RC	Reading Comprehension
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
TDNN	Time-Delay Neural Networks
VRAM	Video Random-Access Memory
WER	Word Error Rate

List of Notations

General Notations

X	Input vector size
Y	Output vector size
S	Sequence length
B	Batch size
$\boldsymbol{x}$	Input vector
$\boldsymbol{y}$	Predicted output vector
$\boldsymbol{z}$	Target output vector
θ	Parameters
W/U	Weights matrix
b	Bias scalar
$\boldsymbol{b}$	Bias vector
$\boldsymbol{h}$	Hidden states
$\sigma()$	Sigmuid activation function
tanh()	Hyperbolic tangent activation function
L()	Loss function
l	Loss
$\boldsymbol{x}_t$	Input vector at time t
$\boldsymbol{y}_t$	Predicted output vector at time t
$\boldsymbol{z}_t$	Target output vector at time t
$\boldsymbol{h}_t$	Hidden states at time t

LSTM Notations

C	Hidden units
L	Layers
f_t	Forget gate at time t
i_t	Input gate at time t
o_t	Output gate at time t
$\tilde{c}_t$	Cell input at time t
c_t	Cell state at time t
$\boldsymbol{h}_t$	Node output at time t

DNC Notations

C	Controller size
L	Controller layer
N	Memory matrix length / slots
W	Memory matrix width
R	Read heads
P	Memory unit output size (RW)
$\boldsymbol{h}_t$	Controller output vector at time t
$\boldsymbol{\mu}_t$	Memory unit output vector at time t
$\boldsymbol{\xi}_t$	Control vector at time t
M_t	Memory matrix at time t
L_t	Linkage matrix at time t
E_t	Erase matrix at time t
A_t	Add matrix at time t
$\boldsymbol{v}_t$	Write vector at time t
$\boldsymbol{e}_t$	Erase vector at time t
$\boldsymbol{k}_t^w$	Write key at time t
$\boldsymbol{k}_t^{r,i}$	Read key at time t of read head i
β_t^w	Write strength at time t
$\beta_t^{r,i}$	Read key at time t of read head i
$\boldsymbol{\pi}_t^i$	Read mode at time t of read head i
g_t^w	Write gate at time t
g_t^a	Allocation gate at time t
$g_t^{f,i}$	Free gate at time t of read head i
$\boldsymbol{\psi}_t$	Retention vector at time t
$\boldsymbol{u}_t$	Usage vector at time t
$\boldsymbol{a}_t$	Allocation weighting vector at time t
$\boldsymbol{\phi}_t$	Free list vector at time t
$\boldsymbol{p}_t$	Precedence weighting vector at time t
$\boldsymbol{c}_t^w$	Content-based write weighting vector at time t
$\boldsymbol{c}_t^{r,i}$	Content-based read weighting vector at time t of read head i
$\boldsymbol{w}_t^w$	Write weighting vector at time t
$\boldsymbol{f}_t^i$	Forward weighting vector at time t of read head i
$\boldsymbol{b}_t^i$	Backward weighting vector at time t of read head i
$\boldsymbol{w}_t^{r,i}$	Read weighting vector at time t of read head i
$\boldsymbol{r}_t^i$	Read vector at time t of read head i
C()	Cosine similarity function
CBA()	Content-based addressing function
Δ_N	Unit simplex

Advanced DNC Notations

LN()	Layer normalization function
$\boldsymbol{r}_t$	Bernoulli distributed random vector at time t
$\boldsymbol{h}_t^{fw}$	Forward controller output vector at time t
$\boldsymbol{h}_t^{bw}$	Backward controller output vector at time t

1. Introduction

”May I ask you a question?” is a very common phrase in human communication, but when it comes to interaction with a computer it does not work out. So it is not surprising that one of the main goals of artificial intelligence is to construct a machine or rather a program which allows a conversation in natural language. One may communicate to the machine or ask a question and the machine answers. This idea goes back to e.g. 1950 when Alan Turing proposed in ”Computing Machinery and Intelligence” to ask machines questions to test if they exhibit intelligent behaviour, also known as the Turing test [2].

Until today, the answering of questions or question answering (QA) is an open research topic but with a long history [3, 4]. It is a subfield of natural language processing (NLP) and information retrieval. QA also includes machine text comprehension since that not only the question but also the context and the answer are in natural language. It is one of the key technologies for dialogue modelling and nearly every NLP task is representable as a QA problem e.g. ”What is the Chinese translation?” [5].

The field of QA is dividable into open- and close-domain which refers to whether the questions could be about anything or about a special context. This work focuses on close-domain QA tasks which have either a synthetic or a natural origin. In both cases, a machine receives a context and is asked to answer a question about it.

A possible approach to such a task is to design an algorithm which answers questions given the context like the ELIZA or SHRDLU system in the 60s or 70s [6, 7]. Since it is too extensive to cover every possible context or question variation it is presumably a good idea to learn such an algorithm from sample data.

1.1 Learnable computers

A common approach for machine learning models is an artificial neural network model trained with backpropagation of errors [8]. In case of interdependent sequences like text, the model of choice are recurrent neural networks (RNN) [9, 10] or more sophisticated long short-term memories (LSTM) [11]. In theory, RNNs are

Turing complete but not in practice [12]. They also have several other issues [13]. For example, the perfect model size for a certain task is unknown. The internal capacity is influenced by the length of dependencies, the computational complexity and the structure of data. Another issue is that ”computation” and ”memory” is mixed in one neural network setup and it is hard to represent data structures over many time steps.

A promising approach that tackles some issues of classic recurrent models is the differentiable neural computer (DNC). A model introduced by Alex Graves et al. in late 2016 as a general neural network model with an external memory ”to solve complex, structured tasks” [1]. Unlike a vanilla neural network, it is able to separate computation and memorization similar to a conventional computer with CPU and random-access memory (RAM). This makes it easier to represent and manipulate complex data structures. Moreover, the DNC can be seen as a generic memory-augmentation framework with a controller and a memory unit which are independently modifiable. This allows an accurate model design and is easy to analyze. Due to its fully differentiable design, it can be learned from data samples in a supervised fashion like a recurrent neural network with backpropagation though time [14].

The separation of computation and memory has several benefits. The memory can represent complex data structures over long time periods, while the computing part of the network does not depend on the length of data dependencies. This reduces catastrophic forgetting [15] which means the uncontrollable reuse of already trained network areas. Furthermore, the independence of computation and memorization allows a more refined model design. The computing part, also called controller, learns to use the memory and is therefore able to emulate algorithms. This ability is shown in the original publication on three applications [1]. In graph experiments, the authors show that the DNC is able to assimilate a graph like the London underground network and answers queries like the shortest connection between two stations. The block puzzle experiment is inspired by the SHRDLU demonstration of a rule-based agent that executes instructions. The DNC is trained with reinforcement learning to plan changes of board states with respect to some goals. The third application is the bAbI task, a synthetic question answering data set with short stories for textual reasoning. The DNC has outperformed related work at publication time.

The DNC approach also works on some other applications like one-shot learning or meta-learning [16, 17]. The models are able to use the memory to quickly assimilate new data, store them over a long time period and retrieve them. Consequently, the model needs only a few examples which are represented in the memory to make accurate class predictions by internal memory reconciliation. Another application is the interaction with discrete interfaces with reinforcement learning [18]. These are examples where such a learnable computer can be used. But when applying this model to new, more realistic tasks like large-scale QA, often no satisfying result is achieved. The issues of large-scale QA are the huge vocabulary, the length of the contexts and the required model complexity to find a correct answer.

1.2 Goals

In a QA task, each word of a sentence or context updates the human internal representation of the related environment. To solve a QA task with machine learning,

the model requires such an internal updatable representation as well. A memory-augmented neural network like the DNC is able to build an internal representation. However, a memorization of vector representations based on whole sentences could lose information in contrast to a word level representation, due to the compression. To prevent this, a word-by-word operating model is crucial. Another essential issue to deal with is limited storage. For example, a dialogue agent which stores every word can not work anymore when the memory is full. Therefore, an overwrite or forget mechanism is another important aspect.

The DNC model possesses these characteristics and seems predestined for QA. However, it is currently only applied to the small synthetic QA bAbI tasks. This leads to the following goals of this work. First, the DNC becomes analyzed to understand the issues when porting it to new QA tasks. Thus, are four main challenges identified:

1. The high memory consumption makes it hard to train large model efficient.
2. The large variance in performance within different initializations makes its training inconsistent.
3. Slow and unstable convergence requires long training times.
4. A unidirectional architecture makes it hard to manage a input sequence with distribute query statements.

The second goal is to address these challenges. Thus, this work introduces advancements in several scopes while keeping the general character of the model. The underlying motivation is to keep the character of the DNC as an algorithm learner and not to adapt it to a specific dataset. The primary target is not to beat a benchmark but rather to show the usability of memory-augmentations and examine if there is a long-term perspective for them in NLP. We introduce a novel advanced DNC (ADNC) with the following contributions:

1. A sparse, memory efficient content-based memory unit for question answering tasks.
2. An enhanced training with a strong focus on early memory usage and robustness.
3. A bidirectional DNC architecture which allows a richer encoding of information from sequences.

The third goal is to evaluate these advancements and to validate the general improvements of the DNC in contrast to existing bAbI task experiments. Thereby the ADNC shows performance improvements by 80% compared to the DNC. These are new state-of-the-art mean results in a joint training. We also decrease the variance up to 90% between different random initializations. Additionally, with training-data augmentations on one specific task, our model solves all tasks and provides the best-recorded results according to the best of my knowledge. The evaluation should also test the portability. This is realized by applying the model on two common large-scale QA tasks: the Children Book Test and the CNN reading comprehension dataset. Therein, the ADNC achieves competitive results without any task-specific adaption.

1.3 Structure of work

The next chapter explains the basics of neural networks as well as details about recurrent neural networks, long short-term memories, and the differentiable neural computer. This is the foundation for all subsequent chapters. The third chapter describes the datasets which are used in this work. Chapter 4 gives an overview of the related works with respect to related models for QA and existing enhancements to the DNC. Afterwards, the two main contributions of this work are presented, the analysis and advancements of DNCs. Chapter 5 analyses the functionality of the DNC and gives an intuition of the memory consumption and computation time. Chapter 6 introduces the advancements to the DNC. Chapter 7 provides experiments which investigate the impact of these advancements and benchmarks the ADNC on large-scale QA datasets. Finally, in Chapter 8 the work is concluded, the results are discussed and some possible future work is described.

2. Basics

This chapter introduces vanilla and more complex artificial neural network models. These are the necessary principles to comprehend later chapters like related work, analysis or methods. At first, the basics about artificial neural networks and their history are presented. The second chapter introduces recurrent models and their drawbacks. Afterwards, two more sophisticated models are introduced, the long short-term memory and the differentiable neural computer.

2.1 Artificial Neural Networks

An artificial neural network (ANN) is a computing model which is used as a general learnable function approximation. More precisely an ANN has a specific structure with internal parameters θ:

$$\boldsymbol{y} = \mathrm{ANN}(\boldsymbol{x}, \boldsymbol{\theta}) \cdot \tag{2.1}$$

The parameters are learned by repeated adjustments with use of a data sample collection: 1. Feeding an arbitrary data sample $\boldsymbol{x}$ from the dataset to the ANN. 2. Compute the loss between the estimated output $\boldsymbol{y}$ from the ANN and the target output $\boldsymbol{z}$. 3. Adjusting the parameters θ hence the loss is low. 4. Repeating this with a huge amount of samples until the ANN generalizes, which means it approximates a target function $\boldsymbol{z} = f(\boldsymbol{x})$. This is called supervised learning and is a sub-domain of machine learning [8]. For example, a data sample could be an image of an object and the associated target could be the class of this object, then the ANN learns object detection [19]. The following describes the ANN internals as well as a brief history.

Artificial neural networks are inspired by the biological neural network like the human brain. More precise they are oriented structurally on the basic functionality of synapses and neurons. Those could simplify described as follows: Synapses are the connections between neurons and transmit electrical signals. The neurons get charged by signal potentials from other neurons via the synapses. When the membrane potential within a neuron reaches a threshold then the neuron emits an action potential and sends it into synapses to further neuron cells [20].

To recreate this simplified biological system input data $\boldsymbol{x}$ gets multiplied element-wise by scalar weights w and summed up. These represent the charge weighted input potentials. When this potential reaches a threshold, the recreated neuron emits an output signal and otherwise not

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{|\boldsymbol{x}|} x_i w_i > b \\ 0 & \text{otherwise} \end{cases} . \tag{2.2}$$

A neural network has multiple of these neuron nodes or also called perceptrons. These allow approximating more complex functions. To calculate them the weights can be rewritten as a weight matrix W. Furthermore, the threshold can be reformulated with use of the Hessian plain formulation

$$\boldsymbol{y} = \sigma(\boldsymbol{x}W + b) , \tag{2.3}$$

while $\sigma()$ represents the sign-function. By replacing the sign-function with a differentiable steadily function, also called the activation function, the whole ANN gets differentiable. This is a key requirement for ANNs. The parameters of the ANN are the weights and biases $\theta = \{W, b\}$. The functionality is illustrated in Figure 2.1.

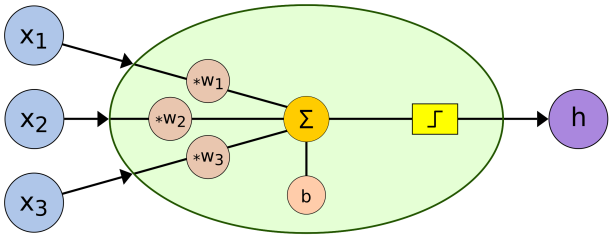


Figure 2.1: Basic perceptron functionality. The green oval in the figure symbolizes the artificial neuron node, the small circles are the parameters, the larger circles the outputs and inputs and the yellow rectangle the activation function.

The emergence of these idea starts in the early 1940s when Warren McCulloch and Walter Pitts introduced a model of a neurologically inspired network with threshold switches as neuron nodes [21]. They also showed that this kind of networks is able to calculate near any logic function and built thereby the foundation block of artificial neural networks. These neural nodes were later improved to a nonlinearly activated sum of weighted inputs by Frank Rosenblatt [22], see Equation 2.2. Later in 1960 Bernard Widrow and Marcian E. Hoff described the delta rule, which allows adaptive, gradient descent based network updates [23].

But it took until the mid-1970s when Paul Werbos introduced an algorithm called "dynamic feedback" which we know today as backpropagation and showed first applications on ANNs [24]. In the later 1980s David E. Rumelhart et. al. refined the backpropagation learning method as a generalization of the delta rule and increased the awareness [25]. This was the breakthrough for artificial neural networks. Afterwards a wide range of different applications got addressed for example phoneme

recognition with time-delay neural networks (TDNN) [26], handwriting digits recognition with a enhancement of TDNN to 2D [27] or the control of nonlinear systems [28]. An extensive history of neural networks can be found in Schmidhuber's review [11].

The vanilla ANN has multiple neuron nodes per layer and multiple layers per network. The output $\boldsymbol{h}$ of the first inner or hidden layer is the input of the second layer and so on:

$$\boldsymbol{h_1} = \sigma(\boldsymbol{x}W_1 + b_1) \tag{2.4}$$
$$\boldsymbol{h_2} = \sigma(\boldsymbol{h_1}W_2 + b_2) \tag{2.5}$$
$$\boldsymbol{y} = \sigma(\boldsymbol{h_2}W_3 + b_3) \cdot \tag{2.6}$$

The output of the last layer is then used as prediction $\boldsymbol{y}$. This model is called multilayer perceptron (MLP). If the number of layers increases it is named deep neural networks and the learning of this models is called deep learning [29].

A common technique to find ANN weight parameters θ is stochastic gradient descent (SGD) with use of backpropagation, described in detail in [8]. The weights are initialized randomly and a training routine adjusts them: First, a randomly picked sample is presented to the network and determines a prediction $\boldsymbol{y}$. A loss l is computed by a loss function L and the target output $\boldsymbol{z}$

$$l = L(\boldsymbol{y}, \boldsymbol{z}) \cdot \tag{2.7}$$

Because of the fully differentiable design of an ANN, the gradients of the parameters can be calculated with respect to the input. Now a parameter update adjusts the network to improve its prediction with respect to the loss l. Therefore the gradients are multiplied by a learning rate lr and added to the parameters. Instead of this simple optimization, a more sophisticated algorithm can be used like RMSprop [30]. A more detailed description of RMSprop and an overview of available optimization algorithms can be found in [31].

Another common variant is to process multiple examples in one batch. This reduced the training time due to fewer iterations are necessary to process the whole training dataset. Furthermore, it increases the generalization since the loss signal relies on multiple samples and is therefore more stable. This kind of training is called mini-batch SGD and can be used with any optimization algorithm.

The routine of feed-forwarding propagation, loss determination, backpropagation and weight update happens iteratively until the network converges to a mostly local optimum. In a possible over-adaption or overfitting to the training data, the network memorizes the whole data set rather than generalizing to the target function. To recognize this, a training and validation data split is essential. The validation set is not used for training and monitors only the progress on unseen samples. Furthermore, several regularizations and normalization techniques, e.g. dropout, weight decay or layer normalization, are available to reduce overfitting and achieve a better generalization [32].

2.2 Recurrent Neural Networks

Dealing with sequences is a difficulty for ANNs. Either the sequence is processed step-wise. This would imply that each step is independent of the previous. Or the sequence would be input at once to the network, then the input needs a fixed length. Both are unfavourable due to real-world sequences are mostly interdependent and arbitrarily long for example like time series, speech or language. Thereby the current step in a sequence depends on the previous one, illustrated in Figure 2.2

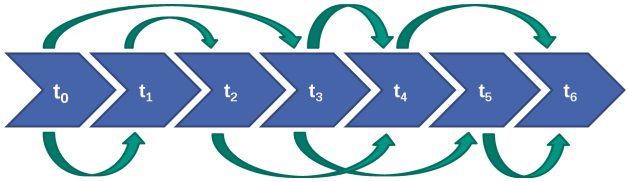


Figure 2.2: An illustration the an interdependent sequence. Each step depends on all previous ones.

In modelling dependencies between steps, recurrent neural networks (RNN) are a common generalization of ANN. The core idea is that they link the output from the previous step in addition to the input of the current step, see Figure 2.3. This allows to keep or link information from previous states to the current. Furthermore, the sequence can have an arbitrary length since the RNN is applied incrementally. They are widely used e.g. in speech recognition [33], phoneme boundary detection [34], machine translation [35], generating image description [36] or segmentation of DNA [37].

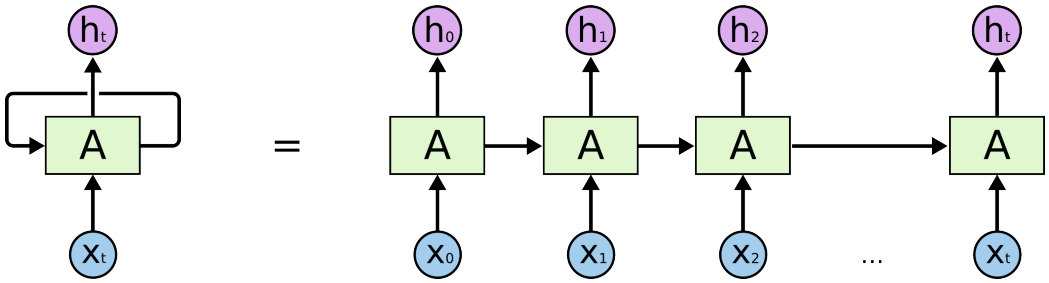


Figure 2.3: A recurrent neural network and the unfolding in time of the computation involved in its forward computation. Graphic from [38].

There are two common simple variants of RNNs. The first was introduced by Jordan in 1986 [39] and links the output back to the input of the next step. The second was introduced by Elman in 1990 [40] and creates the recurrence within the hidden state:

Elman Network:	Jordan Network:
$\boldsymbol{h}_t = \sigma_h(W_h\boldsymbol{x}_t + U_h\boldsymbol{h}_{t-1}) + b_h$	$\boldsymbol{h}_t = \sigma_h(W_h\boldsymbol{x}_t + U_y + b_h\boldsymbol{y}_{t-1})$
$\boldsymbol{y}_t = \sigma_y(W_y\boldsymbol{h}_t) + b_y$	$\boldsymbol{y}_t = \sigma_y(W_y\boldsymbol{h}_t) + b_y$

(2.8)

The additional weight matrix U weights the recurrent signals. In case of the Elman network the recurrent signal is the previous output of the hidden layer and in case of the Jordan network, it is the previous output of the whole network.

In a multilayer setup, the Elman network is more common. It has mostly a tanh or sigmoid activation function. Each layer node gets the output from the layer below and from the same layer one time step before. Figure 2.4 shows an unfolded Elman RNN layer over three time steps. The output of each hidden node goes to both, the layer output $\boldsymbol{h}_t$ and to the nodes in the next step. Most important, the weights for the computation within the nodes are the same in each time step.

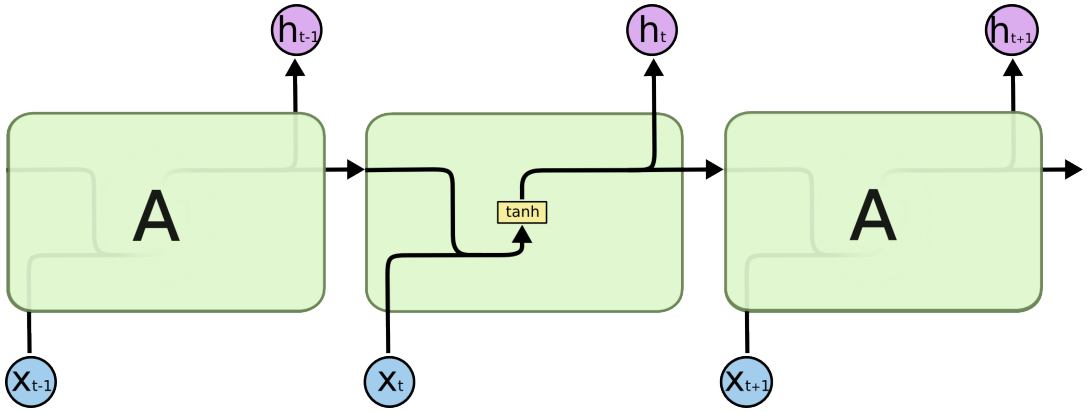


Figure 2.4: A RNN node with a tanh activation function. Black lines show the signal flow. $\boldsymbol{h}_t$ and $\boldsymbol{x}_t$ are the output and input signal in each time step t. Graphic from [38].

The training of RNNs is similar to common ANNs but regarding the recurrent connections, the gradient depends on all previous time steps. For example, the output of step $t + 1$ depends on all previous inputs and hidden states as shown in Figure 2.5.

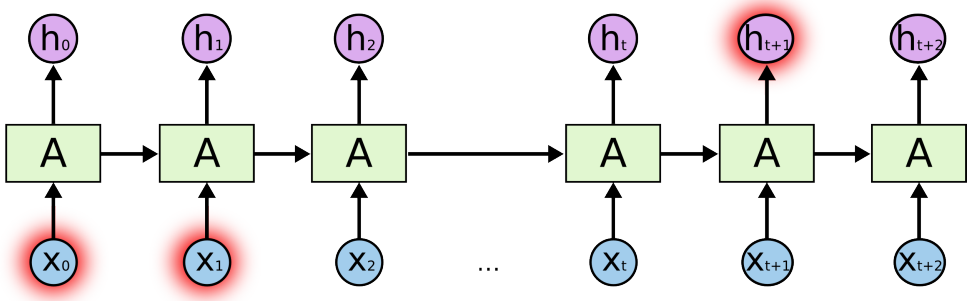


Figure 2.5: The long term dependencies in RNNs. Graphic from [38].

A common technique to compute the gradients is to unfold the model in time and calculate the gradients with respect to all previous inputs. This type of training is called backpropagation through time (BPTT) [41]. But this also leads to the issue that long-term dependencies get underrated since the differential calculation of the recurrent connections is done by the chain rule. In practice, this is a multiplication of the gradient of the activation function which is typically in the range $(0, 1)$. Since

the product of multiple small values vanishes, the influence of inputs in the past vanishes as well. This issue is called vanishing gradient problem and reported detail in [42] and later in [43]. This makes simple RNNs in practice weak learners and only usable for short-term relationships. However, when ignoring this issues, RNNs are theoretical Turing complete and able to solve any problem that can be solved by a conventional computer [12, 44].

A common solution, especially in speech or handwriting recognition, are artificial neural networks (ANN) combined with Hidden Markov Models (HMM) [45]. But HMMs themselves have several drawbacks. They need a lot of task-specific knowledge and explicit dependency assumptions. Furthermore, their training is generative [46]. So it is worthwhile to model the recognition solely with neural networks. They do not require prior knowledge and they train discriminatively, means they calculate directly the class probabilities [47]. RNNs also tend to be robust to temporal and spatial noise [46]. The long short-term memory (LSTM) in next section is a specially extended network node architecture which is able to solve the problem of vanishing gradients.

2.3 Long Short-Term Memory

In 1997 Hochreiter and Schmidhuber introduced long short-term memories (LSTM) for a more richer modelling of interdependencies and to overcome the problem of vanishing gradient [48]. They replace the ordinary activation function through a more sophisticated architecture. While in simple RNNs the hidden state gets computed every step, the LSTM stores information without activation over time. Therefore a hidden cell state is introduced. Every time step the cell state can forget information and add new information without passing an activation function.

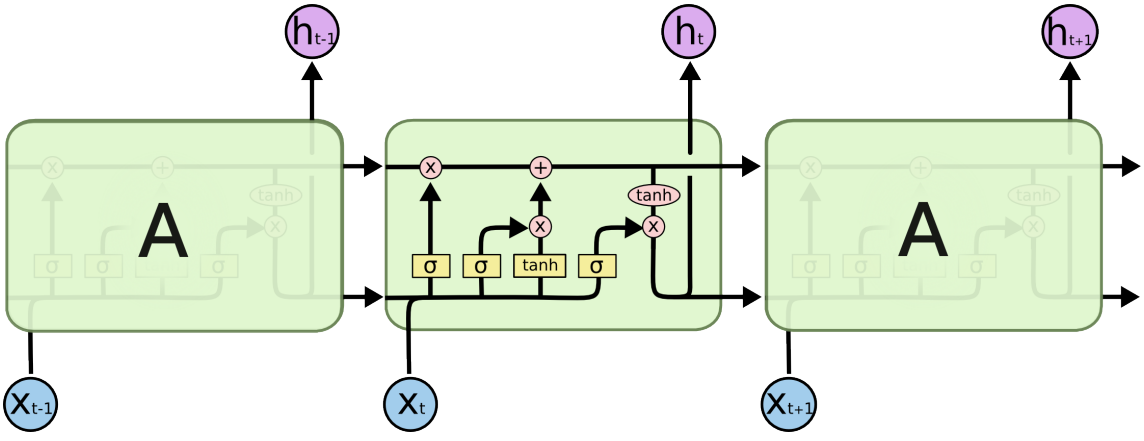


Figure 2.6: The inner function of a LSTM node. Circles with $\boldsymbol{x}$ and + are signal multiplication and aggregation. σ and tanh are activation functions. Black lines show the signal flow. $\boldsymbol{h}_t$ and $\boldsymbol{x}_t$ are the output and input signal in each time step t. Graphic from [38].

Figure 2.6 shows a LSTM node. The concatenation of the last layer output $\boldsymbol{h}_t - 1$ and the current input data $\boldsymbol{x}_t$ is the nodes incoming signal. In addition to this, it receives the internal cell state c_{t-1} from the previous LSTM node. In Figure 2.6 the upper horizontal line illustrated the internal cell state. It gets multiplied and

new signals are added but it has no activation function. This tackles the vanishing gradient problem and leads to a more stable long-term modelling. Furthermore, an LSTM node has three gates (forget gate, input gate and output gate) and activation functions for both the input and the output signal. The input of every nonlinear function is weighted by independent weights. The following section describes the functionality in detail step by step.

The forget gate f_t controls how much the internal cell state keeps from the previous step, see σ activation function in Figure 2.7. The input to the gate is a weighted concatenation of the input data $\boldsymbol{x}_t$ and the previous node output $\boldsymbol{h}_{t-1}$. A scalar bias value is added for thresholding the activation:

$$f_t = \sigma(W_f[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + b_f) \cdot \tag{2.9}$$

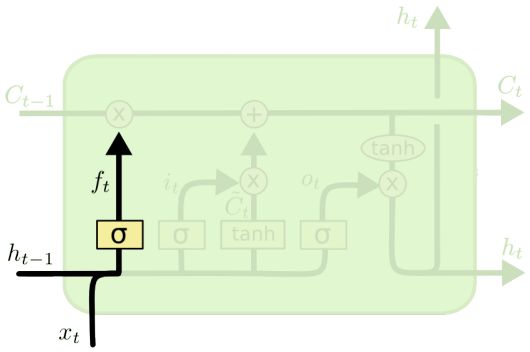


Figure 2.7: The forget gate of a LSTM. Graphic from [38].

The sigmoid function outputs a value between 0 and 1 which gets multiplied by the last cell state $c_t - 1$. The sigmoid activation works like a valve if the output is 1 the cell keeps everything from the last step if the outputs are 0 it forgets everything. The second σ activation function is the input gate

$$i_t = \sigma(W_i[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + b_i) \tag{2.10}$$

and controls how much information from the incoming signal affects the internal cell state, see Figure 2.8. The incoming signal, which is another weighted concatenation of input data $\boldsymbol{x}_t$ and the previous node output $\boldsymbol{h}_{t-1}$, passes an tanh activation function:

$$\tilde{c}_t = \tanh(W_c[h_{t-1}, \boldsymbol{x}_t] + b_c) \cdot \tag{2.11}$$

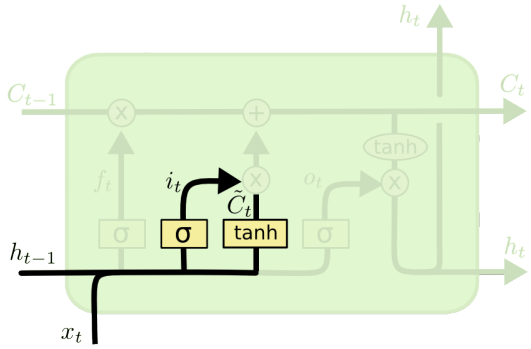


Figure 2.8: The input gate and input activation function. Graphic from [38].

Afterwards the internal cell state gets updated, see Figure 2.9. The previous cell sate c_{t-1} is multiplied with the forget gate f_t and added with a product of the activated incoming signal $\tilde{c}_t$ and the input gate i_t:

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t \cdot \tag{2.12}$$

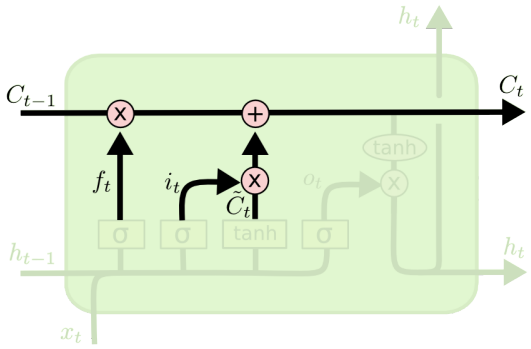


Figure 2.9: The internal cell state of a LSTM. Graphic from [38].

The updated internal cell state is directly forwarded to the next step c_t. It is also used to compute the output of the new node $\boldsymbol{h}_t$, see Figure 2.10. Therefore an output gate o_t gets computed with use of the third σ activation function and another weighted concatenation of input data $\boldsymbol{x}_t$ and the previous node output $\boldsymbol{h}_{t-1}$:

$$o_t = \sigma(W_o[h_{t-1}, \boldsymbol{x}_t] + b_o) \cdot \tag{2.13}$$

This output gate controls the strength of this activated output before it leaves the node $\boldsymbol{h}_t$. Thereby the new cell state c_t is passed through the second tanh activation function and gets multiplied with the output gate o_t:

$$\text{Output signal: } \boldsymbol{h}_t = o_t \tanh(c_t) \cdot \tag{2.14}$$

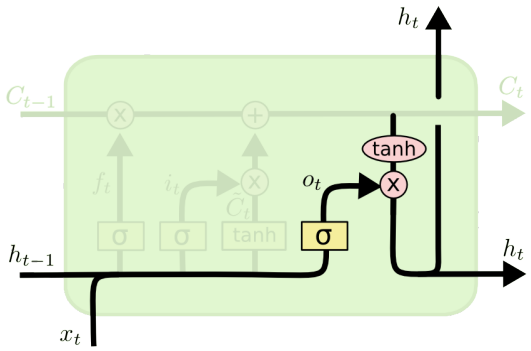


Figure 2.10: The output gate and the activation of the output stream of a LSTM. Graphic from [38].

This so generated output signal leaves the node in two directions. One leads to the next layer above and a second to all nodes in the same layer but in the next time step. A common implementation has multiple LSTM nodes in one layer. The output signal from the previous step gets shared between the nodes but the internal cell state not, see Figure 2.11.

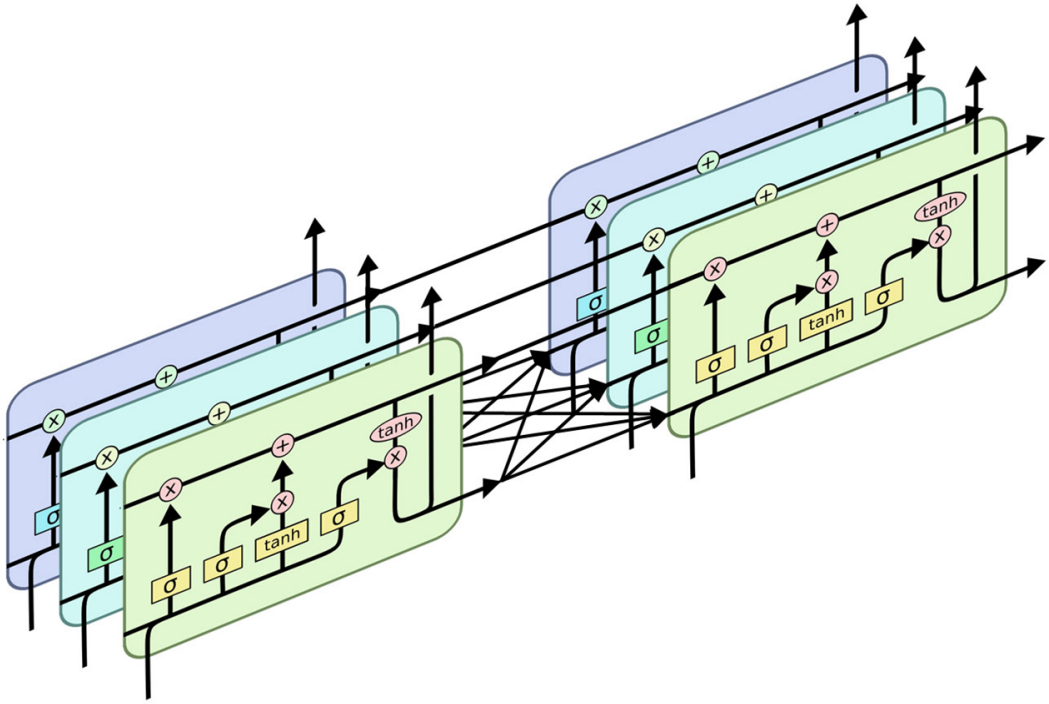


Figure 2.11: A multi-node LSTM over two time steps. The output signals of each node are shared between the time steps but the cell states not.

A mathematical based description of the LSTM and its derivation can be found in the Graves book [47]. There are plenty of different LSTM variants [49]. Another commonly used variant is the Gated Recurrent Unit (GRU) [50]. A GRU simplifies a regular LSTM by combining forget and input gates to a single update gate. Furthermore, it combines the internal cell state with the cell output, sees Figure 2.12. This modification needs fewer weights, hence faster training and outperforms LSTM in some specific tasks [51] but not in general [52].

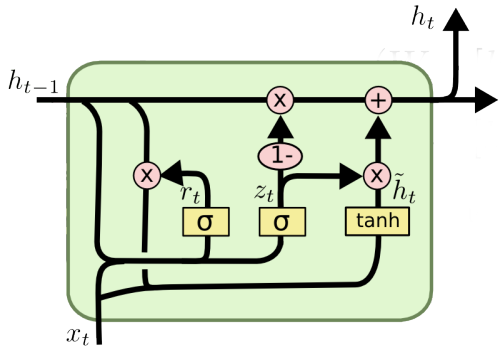


Figure 2.12: A Gated Recurrent Unit. Graphic from [38].

2.4 Bidirectional RNN

Another common enhancement is the bidirectional RNN (BiRNN). Thereby two RNNs are used in one layer but in different directions, one forward and one backward. This type of architecture allows the model to have complete, sequential information about the past and future time steps. This network architecture was introduced by Schuster et. al in 1997 [53] and is illustrated in Figure 2.13. Graves et. al introduced in 2005 an extension to bidirectional LSTMs (BiLSTM) and shows their advantage on phoneme classification [54]. These models are also unfoldable like RNNs and trainable with backpropagation through time.

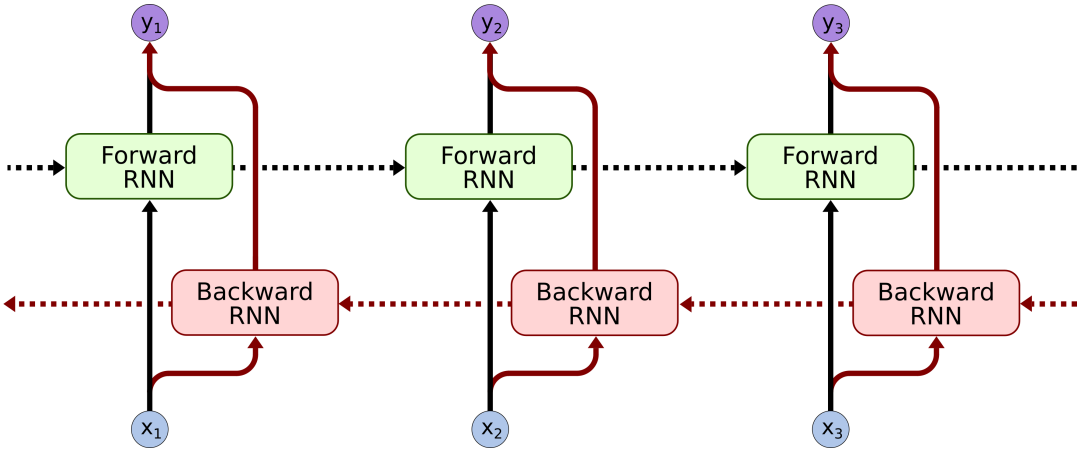


Figure 2.13: Bidirectional RNN architecture. The input signal gets doubled and forwarded to the forward and backward RNNs and concatenated afterwards to the output.

2.5 Differentiable Neural Computer

Even huge LSTM models struggle to process data over long timescales or to infer about seen data. A possible solution is an explicitly defined memory. An approach for such a memory-augmented neural network (MANN) is the differentiable neural computer (DNC). Besides many other models, see related work Chapter 4, it implements a read-, erase- and writeable memory as a matrix. A DNC is able to store, manipulate or infer complex data structures like a conventional computer but can

be trained like an RNN with stochastic gradient descent. It is fully differentiable and controls the memory only by weighted input signals.

The DNC was introduced in 2016 by Alex Graves [1] and is a successor of the Neural Turing Machine (NTM) from 2014 [55]. Both models have a similar architecture and discern in access mechanism to the memory. Already the NTM is able to repeat, reverse or sort a sequence but the DNC extend these possibilities. In the nature paper [1] it is described how it performs a graph search or how it treats the bAbI toy NLP task [56].The following section describes the functionality of the DNC and its mechanisms in detail based on [1].

2.5.1 System overview

The DNC model consists of two main parts, a controller and a memory unit (MU), see Figure 2.14. The controller is either a fully-connected neural network or an RNN respectively an LSTM. In the original paper as well as in this work it is an LSTM. The controller output serves the MU as input but also gets added to the model output via a bypass connection. The output signal of the whole DNC is a sum of the weighted controller output and the MU output. There are recurrent connections within the controller and within the memory. A third recurrent connection is from the MU output to the controller input.

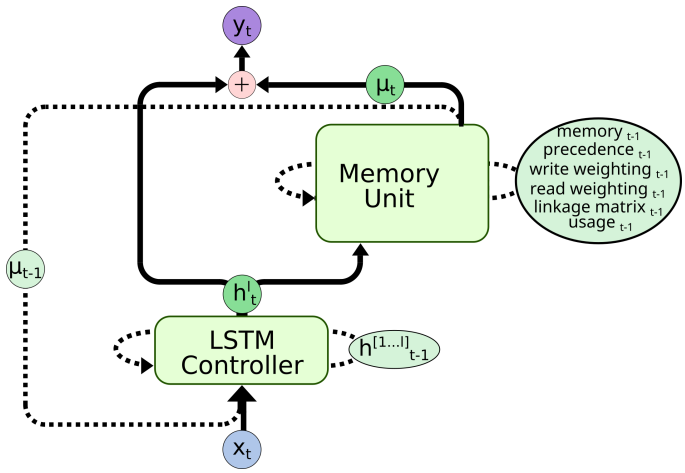


Figure 2.14: The Differential Neural Computer (DNC) with the signal transitions. The dotted lines illustrate recurrent connections.

The DNC receives at each time step a data vector $\boldsymbol{x}_t \in \mathbb{R}^X$ and emits an output vector $\boldsymbol{y}_t \in \mathbb{R}^Y$ which is a prediction of the target vector $\boldsymbol{z}_t \in \mathbb{R}^Y$. More specific, the controller input to a time step t is a concatenation of the input signal $\boldsymbol{x}_t$, the MU output from the last time step $\boldsymbol{\mu}_{t-1} \in \mathbb{R}^P$ and the controller output from the last time step $\boldsymbol{h}_{t-1} \in \mathbb{R}^C$. C is the controller output size and P the MU output size. If the controller network has multiple layers C_l then either the last layer can be used as controller output $C = C_l$ or a concatenation of all LSTM layers $C = [C_1 \dots C_l]$. The controller can be considered as a closed function with a set of trainable weights parameters θ_c:

$$\boldsymbol{h}_t = \text{Controller}([\boldsymbol{x}_t, h_{t-1}, \boldsymbol{\mu}_{t-1}], \theta_c) \cdot \tag{2.15}$$

The purpose of the controller is to manage the memory unit and additionally to help building the output signal via a weighted bypass connection. The MU gets as input the controller output $\boldsymbol{h}_t$ and contains a set of trainable weights parameters θ_μ as well:

$$\boldsymbol{\mu}_t = \text{MemoryUnit}(\boldsymbol{h}_t, \theta_\mu) \cdot \tag{2.16}$$

The output signal of the whole DNC $\boldsymbol{y}_t \in \mathbb{R}^Y$ is a sum of the weighted controller output and the MU output. Y is the size of the target vector $\boldsymbol{z}_t \in \mathbb{R}^Y$. The MU output $\boldsymbol{\mu}_t$ is weighted with matrix $W_\mu \in \mathbb{R}^{Y \times P}$ and the bypass connection from the controllers output $\boldsymbol{h}_t$ is weighted with $W \in \mathbb{R}^{Y \times C}$:

$$\boldsymbol{y}_t = W_h \boldsymbol{h}_t + W_\mu \boldsymbol{\mu}_t \cdot \tag{2.17}$$

The hyper-parameters of the controller network are the number of layers l and the nodes per layer C_l. The hyper-parameters of the MU are the width W of the internal memory matrix per slot and the number of slots N. There can be multiple read heads R which are extract information from the memory matrix. This leads to the MU output size $P = R \times W$.

2.5.2 The memory unit

After an overview of the memory unit, the functionality of it is described in detail.

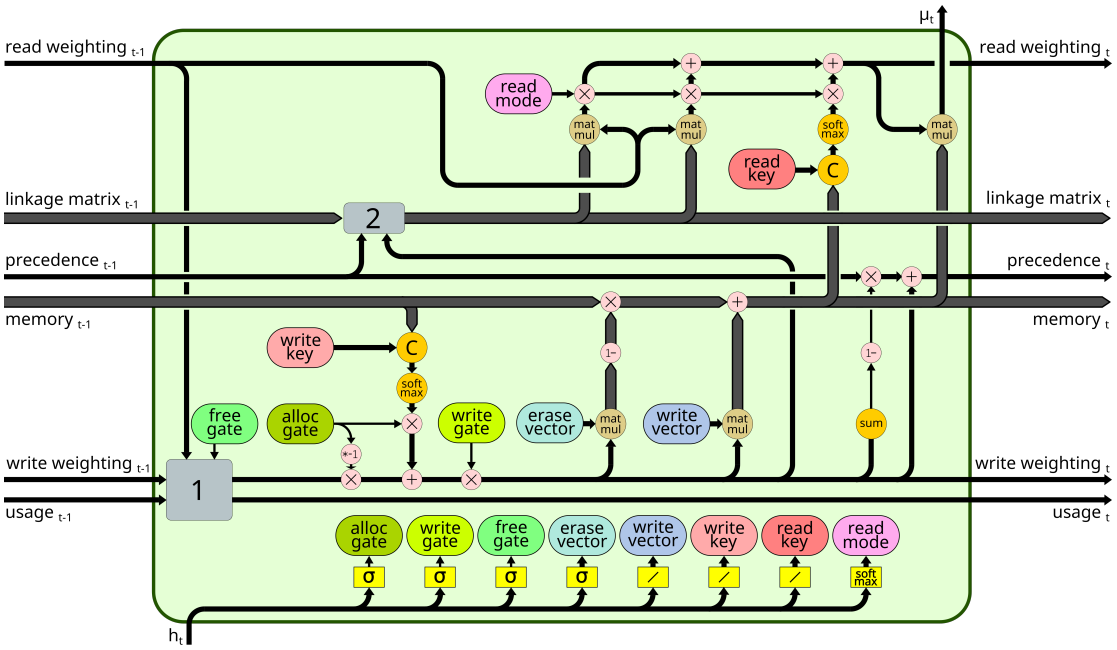
2.5.2.1 Memory unit overview

The memory unit (MU) contains a memory matrix $M \in \mathbb{R}^{N \times W}$. It stores information in form of real-valued vectors $v \in \mathbb{R}^W$ over multiple slots N. The input from the controller gets multiplied by the input weights and divided into different control gates, vectors and keys which are controlling four mechanisms. The four mechanisms are used for writing and reading the memory matrix: The write mechanism, the memory matrix update, a temporal memory linkage and a read mechanism.

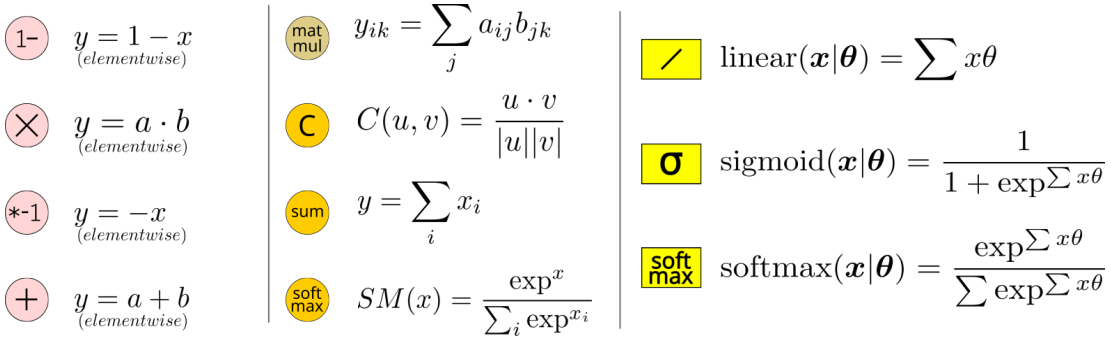
The write mechanism aims to find the location of the memory unit which should be updated with new content. The location can be found either by the least used memory location or by content-based addressing. Contend-based addressing means that given a key $\boldsymbol{k}_t \in \mathbb{R}^W$ the memory location $\boldsymbol{w}_t \in \mathbb{R}^N$ with the closes cosine distance to the key is used. This helps to add information to existing memories.

With use of the location form the write mechanism and a write and erase vector become the memory matrix updated by adding and erasing information. The temporal memory linkage contains a linkage matrix $L_t \in \mathbb{R}^{N \times N}$ which stores information about the order of memory location updates. It is used to restore sequences in forward or backward directions and finds temporal relationships in the memory.

The read mechanism creates the read weightings which determines the location from which should be read. If the DNC has multiple read heads R, this mechanism exists multiple times. The read weightings can be determined with use of the temporal linkage matrix to find forward and backward weightings but also though a content-based addressing. Similar to the write mechanism it finds the location which is



(a)



(b)

Figure 2.15: The memory unit (MU) of a DNC and a legend of the symbols. Slim arrow lines are scalar transitions, medium lines are vector transitions and wide lines are matrix transitions. The gray blocks are summarized functions. Horizontal outgoing lines are recurrent connections within the MU.

closest to a given read key. A read mode determines which of these weightings should be used.

At least the memory is read by a multiplication of the memory matrix and the read weightings. The memory unit output is a concatenation of all read heads readings $\boldsymbol{\mu}_t \in \mathbb{R}^{W \times R}$.

Figure 2.15 (a) shows an MU with one write head and one read head. The colored ovals are control signals, the yellow squares are weighted activation functions, the brown circles are matrix multiplications, the orange circles are either softmax operations or cosine weightings, the pink circles are element-wise vector operations, the gray boxes are summarized functions, the slim lines are scales tensors, the medium lines are vector and the wide lines are matrix tensors, see Figure 2.15 (b). The following sections describe these processes in detail.

2.5.2.2 Generating control signals

The control signals is created by the slice of an control vector $\boldsymbol{\xi}$. The control vector has a size of $(W \times R) + 3W + 5R + 3C$ and is the weighted controller output $\boldsymbol{h}_t$:

$$\boldsymbol{\xi}_t = \boldsymbol{h}_t W_\xi + \boldsymbol{b}_\xi \cdot \tag{2.18}$$

The parameters are the memory unit weight matrix $W_\xi \in \mathbb{R}^{C\times((W\times R)+3W+5R+3)}$ and a bias $\boldsymbol{b}_\xi \in \mathbb{R}^{(W\times R)+3W+5R+3}$. The resulting control vector $\boldsymbol{\xi}$ is sliced into ten different control signal types:

$$\boldsymbol{\xi} = [\boldsymbol{k}_t^w; \hat{\beta}_t^w; v_t; \hat{\boldsymbol{e}}_t; g_t^a; g_t^w; \hat{f}_t^1, \ldots, \hat{f}_t^R; k_t^{r,1}, \ldots, \boldsymbol{k}_t^{r,R}; \hat{\beta}_t^{r,1}, \ldots, \hat{\beta}_t^{r,R}; \hat{\boldsymbol{\pi}}_t^{r,1}, \ldots, \hat{\pi}_t^{r,R}] \cdot \tag{2.19}$$

The control signal is used by the write and read mechanisms to make use of the memory. Table 2.1 gives a overview in detail. Some of them get activated by a sigmoid or softmax activation function, see Figure 2.16.

Name	Symbol
write key	$\boldsymbol{k}_t^w \in \mathbb{R}^W$
write strength	$\beta_t^w = \text{oneplus}(\hat{\beta}_t^w) \in [1, \infty)$
write vector	$\boldsymbol{v}_t \in \mathbb{R}^W$
erase vector	$\boldsymbol{e}_t = \sigma(\hat{\boldsymbol{e}}_t) \in [0, 1]^W$
allocation gate	$g_t^a = \sigma(\hat{g}_t^a) \in [0, 1]$
write gate	$\sigma(\hat{g}_t^w) \in [0, 1]$
free gates	$\sigma(\hat{g}_t^{f,i}) \in [0, 1]; 1 \leq i \leq R\}$
read keys	$\{\boldsymbol{k}_t^{r,i} \in \mathbb{R}^W; 1 \leq i \leq R\}$
read strengths	$\{\beta_t^{r,i} = \text{oneplus}(\hat{\beta}_t^{r,i}) \in [1, \infty); 1 \leq i \leq R\}$
read modes	$\{\boldsymbol{\pi}_t^i = \text{softmax}(\hat{\boldsymbol{\pi}}_t^i) \in S_3; 1 \leq i \leq R\}$

Table 2.1: The control signals of the memory unit.

The logistic sigmoid function σ is constrained to $[0, 1]$ and 'oneplus' function

$$\text{oneplus}(x) = 1 + log(1 + e^x) \tag{2.20}$$

is constrained to $[1, \infty)$. The read modes are computed by a softmax function and constrained to a unit simplex:

$$S_N = \{\alpha \in \mathbb{R}^N : \alpha_i \in [0, 1], \sum_{i=1}^{N} \alpha_i = 1\} \tag{2.21}$$

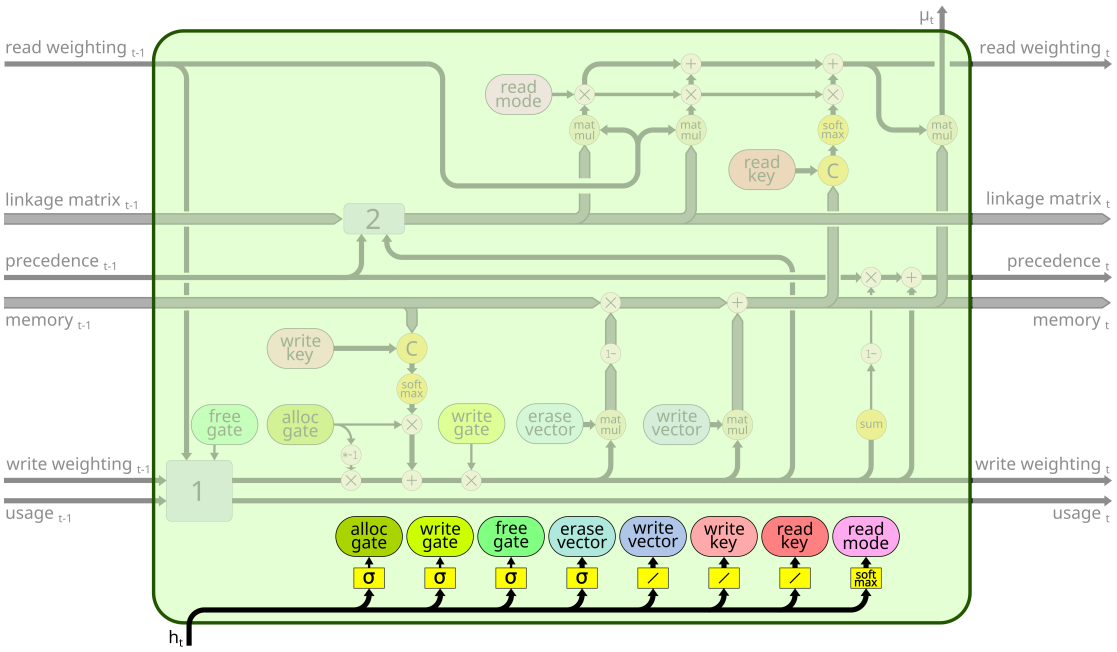


Figure 2.16: The control signals used in the memory unit of a DNC.

2.5.2.3 Write mechanism

The most central aspect of the MU is the memory matrix $M_t \in \mathbb{R}^{N \times W}$, illustrated as the wide line in the middle of Figure 2.15. In each time step, the whole memory will first be manipulated by a write head and then read by a read head. Because of the necessary differentiability of the model are only operations to the whole memory matrix possible. To define the impact, to each memory location which should be manipulated or read, are weightings for writing $w_t^w \in \Delta_N$ and reading $w_t^{r,R} \in \Delta_N$ necessary. The weightings are vectors with the length of the memory N and constraint by

$$\Delta_N = \{\alpha \in \mathbb{R}^N : \alpha_i \in [0,1], \sum_{i=1}^{N} \alpha_i \leq 1\} \ , \tag{2.22}$$

a variant of the unit simplex. The next step is to compute the write weightings w_t^w, shown in Figure 2.17. It's a part of the write head and necessary to determine which memory locations in which intensity should be erased and written to in this time step. Thereby are two ways to compute them: by a content-based addressing and by dynamic memory allocation. Both methods used jointly and influence the write weightings.

Content-based addressing

The idea behind the content-based addressing (CBA) is to find the memory location which is close to a given key $k \in \mathbb{R}^W$. This enables to add information to an existing location or to erase a specific part. Because of the fully differentiable design this can not be an unique location but a distribution over all memory locations N.

As distance metric between the key k and each memory location in the memory matrix M a cosine similarity

$$\mathrm{C}(\boldsymbol{k}, M[i,:]) = \frac{\boldsymbol{k} \cdot M[i,:]}{|\boldsymbol{k}||M[i,:]|} \tag{2.23}$$

is used. To compute the content-based weighting which are the distribution of the similarity between the key k and the memory matrix locations a softmax

$$\mathrm{CBA}(M, \boldsymbol{k}, \beta)[i] = \frac{exp(\mathrm{C}(\boldsymbol{k}, M[i,:])\beta)}{\sum_j exp(\mathrm{C}(\boldsymbol{k}, M[j,:])\beta)} \cdot \tag{2.24}$$

normalizes the cosine similarity. A strength β sharpens the location distribution. In the write head, the content-based weighting $c_t^w \in S_N$ is constructed with the use of the write key k_t^w and write strength β_t^w:

$$\boldsymbol{c}_t^w = \mathrm{CBA}(M_{t-1}, \boldsymbol{k}_t^w, \beta_t^w) \cdot \tag{2.25}$$

In the Figure 2.17 this mechanism is illustrated with the red write key and the two orange circles which represents the cosine similarity (C) and the softmax function. The write strength is for a better reading not shown in the illustration.

Dynamic memory allocation

The dynamic memory allocation mechanism aims to find the at least used memory location. A memory location is used, when some content is written to or if it is read in the past. At first, with use of the free gates $g_t^{f,i}$ and the previous read weightings $\boldsymbol{w}_{t-1}^{r,i}$ the most recent read location is found. The free gate determines how strong the need for allocating a memory location is. The memory retention vector $\boldsymbol{\psi}_t \in [0,1]^N$ represents how much each location will not be freed by the free gates of each read head:

$$\boldsymbol{\psi}_t = \prod_{i=1}^{R} (\mathbf{1} - g_t^{f,i} \boldsymbol{w}_{t-1}^{r,i}) \cdot \tag{2.26}$$

Afterwards an usage vector $\boldsymbol{u}_t \in [0,1]^N$ with $\boldsymbol{u}_0 = \mathbf{0}$ is determined. Memory location which has been retained by the memory retention vector ψ_t or either were already in use or has been written to have a higher value up to 1:

$$\boldsymbol{u}_t = (\boldsymbol{u}_{t-1} + \boldsymbol{w}_{t-1}^w - \boldsymbol{u}_{t-1} \circ \boldsymbol{w}_{t-1}^w) \circ \boldsymbol{\psi}_t \cdot \tag{2.27}$$

Every use of a location increases the usage vector value. Only using the free gates can decrease it to 0. Let $\circ$ be an element-wise multiplication.

A free list $\boldsymbol{\phi}_t \in \mathbb{Z}^N$ is computed with the use of the usage vector $\boldsymbol{u}_t$ by sorting the indices of the memory locations in ascending order of usage. Thereby, the least used

location is $\boldsymbol{\phi}_t[1]$. Now an allocation weighting $\boldsymbol{a}_t \in \Delta_N$ can be computed to provide an unused location for writing new content:

$$\boldsymbol{a}_t[\psi_t[j]] = (1 - \boldsymbol{u}_t[\boldsymbol{\psi}_t[j]]) \prod_{i=1}^{j-1} \boldsymbol{u}_t[\boldsymbol{\psi}_t[i]] \cdot \tag{2.28}$$

If the usage vector is very high then the allocation weightings are close to 0 and the free gates need to release some locations. The sort operation is not continuous but this is ignored when calculating the gradient for training the DNC.

Write weightings

The write weighting $\boldsymbol{w}_t^w$ is a weighted sum of the content-based weighting $\boldsymbol{c}_t^w$ and allocation weighting $\boldsymbol{a}_t$. The allocation gate g_t^a determines the influence of the two mechanisms. The write gate g_t^w determines the intensity of the write weightings

$$\boldsymbol{w}_t^w = g_t^w(g_t^a \boldsymbol{a}_t + (1 - g_t^a)\boldsymbol{c}_t^w) \cdot \tag{2.29}$$

If the write gate is 0, nothing is written and the memory is saved from unnecessary manipulation. This is similar to an attention mechanism. The whole computation of the allocation weighting and the update of the usage vector is condensed in Block 1 in Figure 2.17. The gating mechanisms are illustrated with the two oval gate symbols and the element-wise operations.

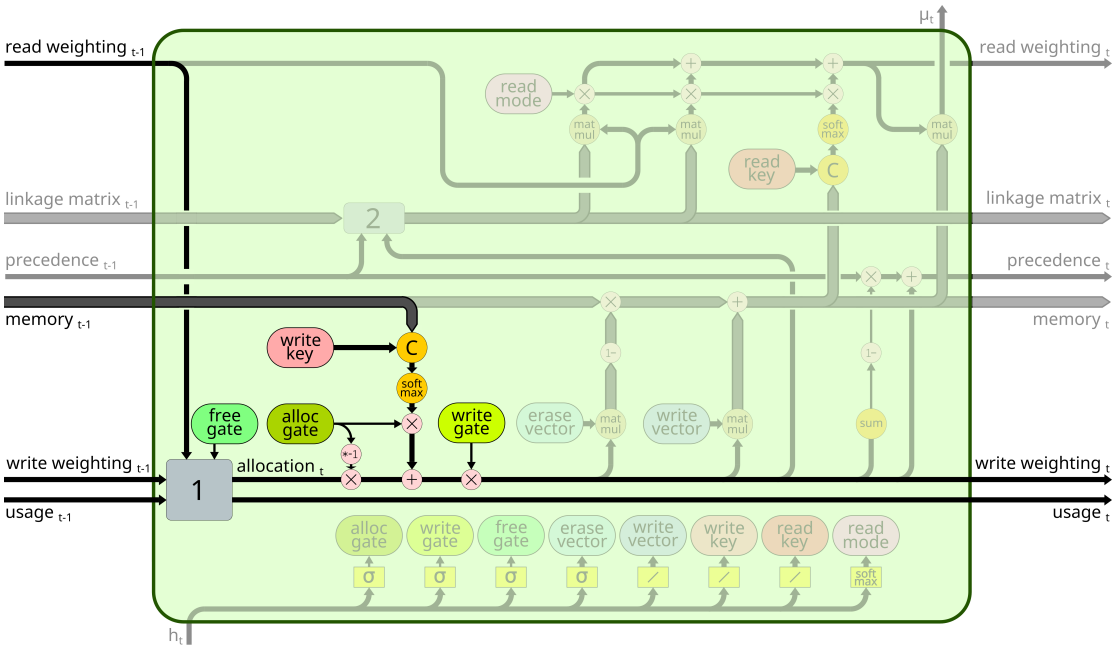


Figure 2.17: The figure shows the mechanism to compute the write weightings. It describes the location in the memory matrix where to add or erase information. The write weightings are computed based on the content-based addressing (write key and the previous memory matrix) and the dynamic memory allocation (the previous usage, read and write weighings and the free gate)

2.5.2.4 Memory update

With use of the write weightings a memory update is possible, as illustrated in Figure 2.18. The memory update contains an erase and an add operation. To erase content an erase matrix E_t is computed by building a matrix product with use of the erase vector $\boldsymbol{e}_t$ and the write weightings $\boldsymbol{w}_t^w$:

$$E_t = \mathbf{1} - \boldsymbol{w}_t^w \boldsymbol{e}_t^\top \cdot \tag{2.30}$$

For adding information a add matrix A_t is computed with use of the write vector $\boldsymbol{v}_t$ and the write weightings $\boldsymbol{w}_t^w$:

$$A_t = \boldsymbol{w}_t^w \boldsymbol{v}_t^\top \cdot \tag{2.31}$$

The last operation of the write head is an update of the previous memory matrix M_{t-1} with use of the erase matrix and the write matrix:

$$M_t = M_{t-1} \circ E_t + A_t \cdot \tag{2.32}$$

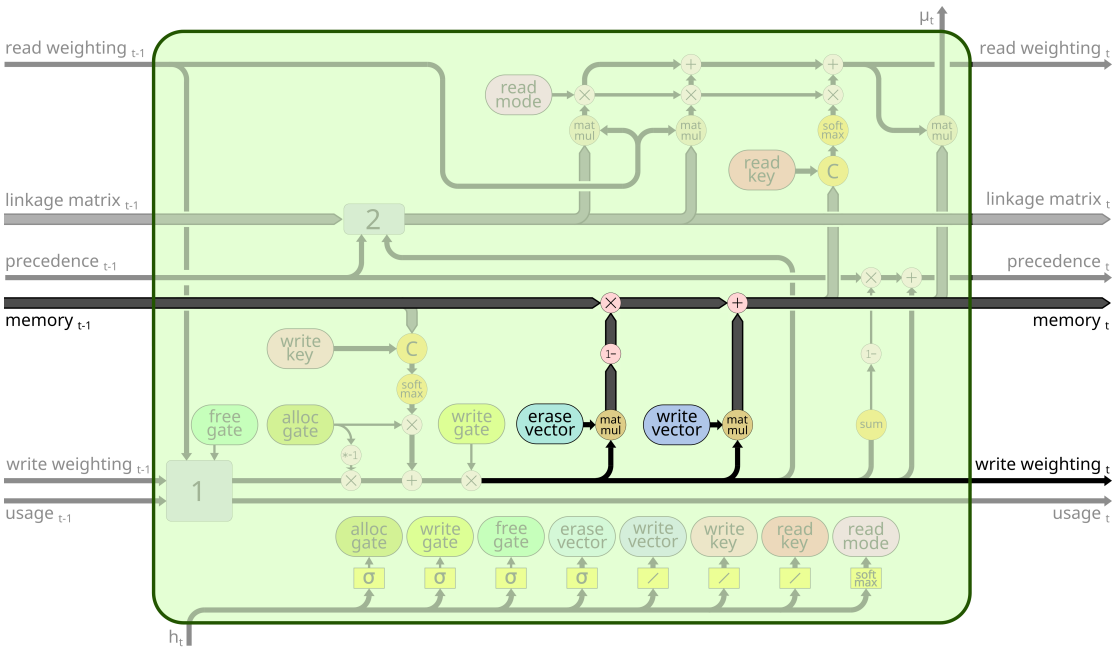


Figure 2.18: The memory matrix update mechanism, every time step the whole matrix gets updated.

2.5.2.5 Read mechanism

A read head is used to read stored information from the memory matrix. A memory unit can have multiple read heads to read on different locations at the same time.

Each read head has two capabilities to find the read weightings which determine the location in the memory matrix to read from. A content-based weighting with use of a key similar to the content-based addressing in the write head and a temporal memory

linkage mechanism. The temporal memory linkage uses a precedence weighting and linkage matrix to find the order, in which sequential information was written to the memory. This allows reproducing coherent sequences.

The operations can be segmented into four parts: A precedence weighting update, a linkage matrix update, a read weighting update and the actual memory reading.

Precedence weighting update

The precedence weighting $\boldsymbol{p}_t \in \Delta_N$ represents to which location was written in the past. Figure 2.19 illustrated the computation of it in the memory unit. It is computed with use of the precedence weighting form the previous time step $\boldsymbol{p}_{t-1}$ and the current write weighings $\boldsymbol{w}_t^w$:

$$\boldsymbol{p}_t = \big(1 - \sum_i \boldsymbol{w}_t^w[i]\big)\boldsymbol{p}_{t-1} + \boldsymbol{w}_t^w \ . \tag{2.33}$$

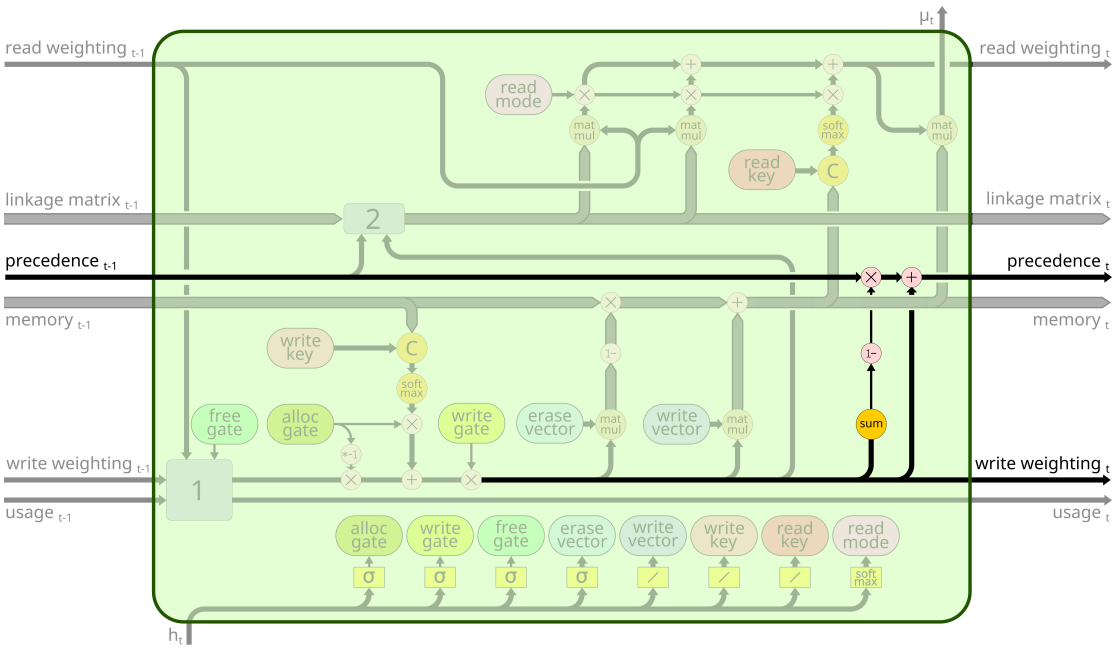


Figure 2.19: The illustration of the precedence weightings update.

Linkage matrix update

With use of the previous precedence weighting $\boldsymbol{p}_{t-1}$ and the current write weightings $\boldsymbol{w}_t^w$ the temporal linkage matrix $L_t \in [0,1]^{N \times N}$ gets updated with $L_0[i,j] = 0 \forall i,j$:

$$L_t[i,j] = (1 - \boldsymbol{w}_t^w[i] - \boldsymbol{w}_t^w[j])L_{t-1}[i,j] + \boldsymbol{w}_t^w[i]\boldsymbol{p}_{t-1}[j] \ . \tag{2.34}$$

The aim of the temporal linkage matrix is to store the transitions from the last written location to this written location in the memory. More specific, $L_t[i,j]$ describes the degree of writing location i after location j was written to, with the constraints $L_t[i,:] \in \Delta^N \forall i$ and $L_t[:,j] \in \Delta^N \forall j$. Self links are excluded $L_t[i,i] = 0 \forall i$. The process is illustrated in Figure 2.20. The original paper describes a sparse temporal

link matrix version as well but it is currently not implementable with use of common deep learning frameworks and therefore not closer described here.

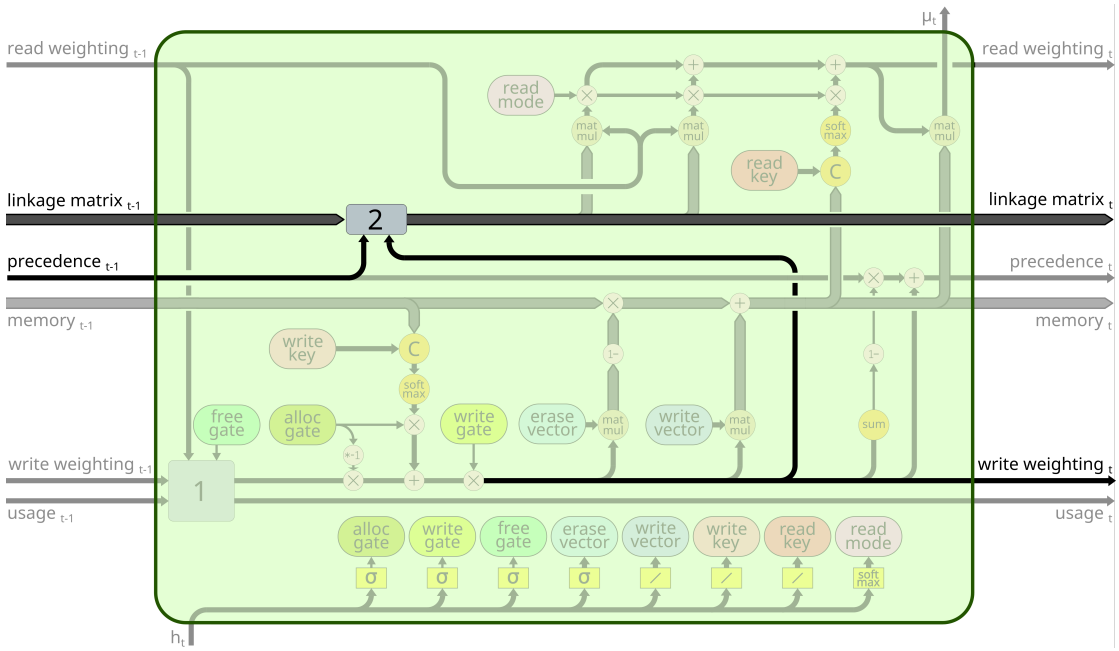


Figure 2.20: The linkage matrix mechanism, which is used to store the order of writings in the memory.

Read weightings update

With an updated temporal link matrix L_t two kinds of memory access are possible. A forward weighting $\boldsymbol{f}_t^i$ and a backward weighting $\boldsymbol{b}_t^i$ at time t of read head i. They use the temporal link matrix to find the memory location which was written before or after the last read operation. This allows to readout in a recorded sequence in both directions when start reading from the middle. The forward weighting is a matrix-vector product with the previous read weightings $\boldsymbol{w}_{t-1}^{r,i}$ and normal linkage matrix

$$\boldsymbol{f}_t^i = L_t \boldsymbol{w}_{t-1}^{r,i} \tag{2.35}$$

and the backward weighting with the transposed linkage matrix

$$\boldsymbol{b}_t^i = L_t^\top \boldsymbol{w}_{t-1}^{r,i} \ . \tag{2.36}$$

The upper index i describes the read head. Each read head has its own weightings but they shares the linkage matrix and precedence weighting. Following the content-based addressing in the write head, the read head uses the same operation. The read content-based weighing $\boldsymbol{c}_t^{r,i} \in \Delta_N$ computes the closest memory location to a given read key $\boldsymbol{k}_t^{r,i}$:

$$\boldsymbol{c}_t^{r,i} = C(M_t, \boldsymbol{k}_t^{r,i}, \beta_t^{r,i}) \ . \tag{2.37}$$

The read strength $\beta_t^{r,i}$ determines the sharpness of the softmax function. To find the new read weightings each read head has a read mode $\boldsymbol{\pi}_t^i \in S_3$ which determines the influence of the forward, backward or content-based weighting in the final read weighting $\boldsymbol{w}_t^{r,i} \in \Delta_N$:

$$\boldsymbol{w}_t^{r,i} = \boldsymbol{\pi}_t^i[1]\boldsymbol{b}_t^i + \boldsymbol{\pi}_t^i[2]\boldsymbol{c}_t^{r,i} + \boldsymbol{\pi}_t^i[3]\boldsymbol{f}_t^i \cdot \tag{2.38}$$

The whole process of finding the new read weighings for one read head is illustrated in Figure 2.21.

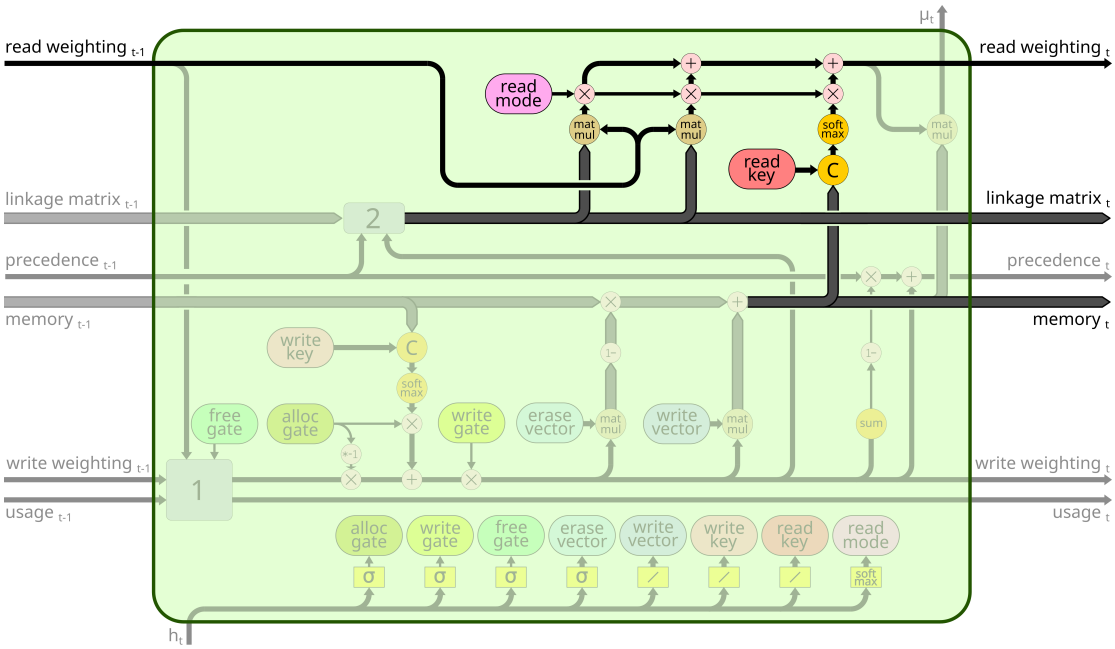


Figure 2.21: The read head, the mechanism to compute the read weightings, the location in the memory from where to read information.

Memory reading

The final reading of the memory matrix with a specific read head is a matrix product of the read weightings $\boldsymbol{w}_t^{r,i}$ and the current memory matrix M_t, see Figure 2.22. The result is a read vector $\boldsymbol{r}_t^i \in \mathbb{R}^W$ for each read head:

$$\boldsymbol{r}_t^i = M_t^\top \boldsymbol{w}_t^{r,i} \cdot \tag{2.39}$$

The whole MU output $\boldsymbol{\mu}_t \in \mathbb{R}^P$ is a concatenation of all read vectors

$$\boldsymbol{\mu}_t = [\boldsymbol{r}_t^1 \dots \boldsymbol{r}_t^R] \tag{2.40}$$

and the MU output size is the memory width times the read head $P = R \times W$.

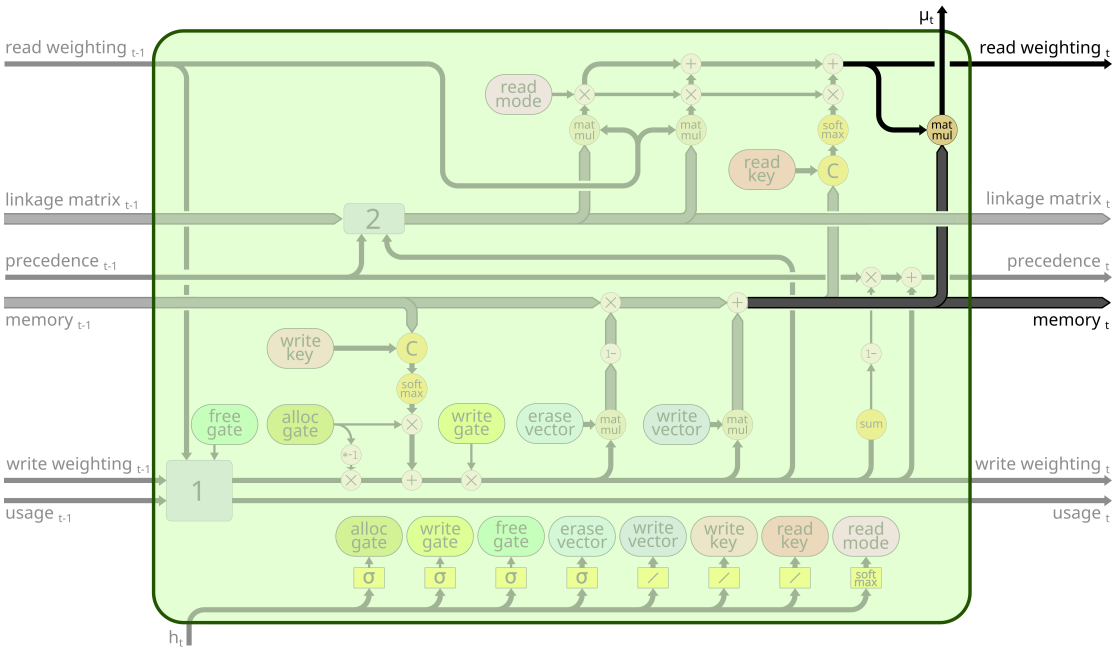


Figure 2.22: The illustration of the memory reading of one read head.

2.5.3 Summary

The MU output serves the output of the whole system and as an input to the controller in the next time step. Figure 2.23 shows the DNC system in an unfolded manner. The recurrent connection of the LSTM is the hidden cell state $\boldsymbol{h}_t$. The recurrent connection from the MU is much richer. It contains the usage vector, the write weightings, the memory matrix, the precedence weightings, the linkage matrix and the read weightings.

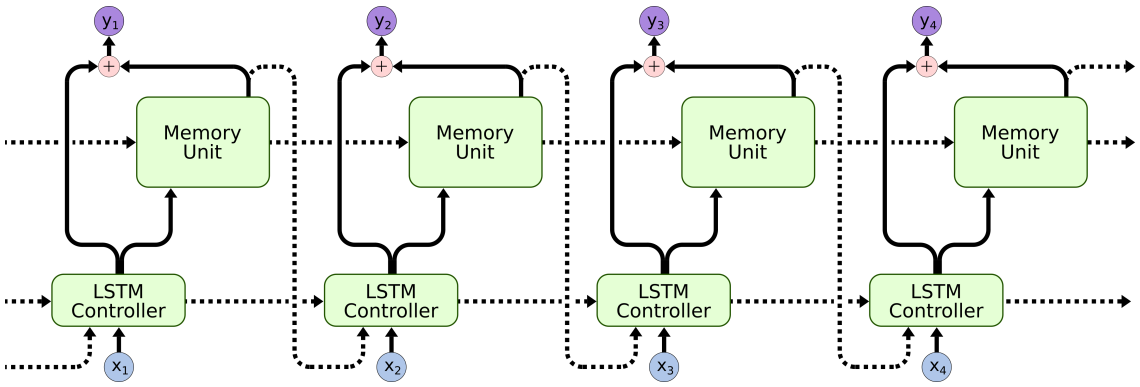


Figure 2.23: An overview of the DNC parts and its signals in unfolded view. The dotted lines illustrate recurrent connections.

3. Data

This work uses four data sets. A synthetic copy task and the synthetic bAbI QA task are already applied to the NTM or DNC papers [1, 55] and are used in the analysis Chapter 5. In the experiments Chapter 7 novel models are evaluated on the bAbI task as well as at two large-scale QA task. The two datasets are the "Children Book Test" and the CNN reading comprehension (RC) task. They are commonly used, freely available and often benchmarked QA tasks. The following sections describe the data sets in detail.

3.1 Copy Task

With the use of an external memory is algorithmic operations like copy, recall or sort is possible. In the copy task, the model is asked to store and retrieve a data sequence. The task aims to test if the model is able to learn a simple algorithm and is able to use the memory correctly. It is presented in the Neural Turing Machine paper [55].

In this work, the copy task is used for analysis the DNC and for evaluation of the different DNC advancements. The task in this work is larger then the original task and the validation set used longer sequences than the training set. This allows testing the generalization ability of the model. Thereby the task addresses both reading mechanisms. The linkage matrix to repeat the sequence in the correct order and content-based addressing to find the first number of the sequence.

	Training Set	Validation Set
Sample amount	6000	600
Min. sequence length	20	50
Max. sequence length	50	100
Feature width	100	100

Table 3.1: Parametrization of the copy task in this work.

In this setup, the model receives a sequence of random numbers in a specific range called feature width, needs to store it and is asked to repeat the whole sequence

after a delimiter symbol which requests the repetition, see Figure 3.1. The actual sample has a fixed length which has 2× the feature width and +1 for the delimiter symbol. Because of the fixed length, no end of sequence symbol is necessary.

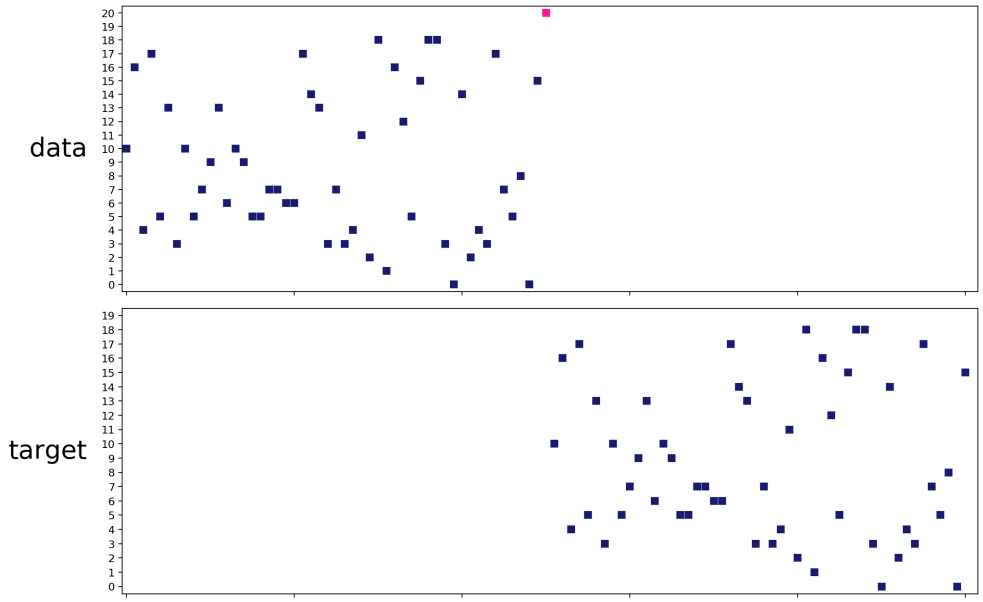


Figure 3.1: An example of the copy task with a feature width of 20 and a sequence length of 50. The red dot is the delimiter symbol which requests the repetition.

The numbers are presented as one-hot vectors with the size of feature width plus one for the delimiter symbol. When a number is expected the input vector is zero. The target numbers are represented as one-hot vectors with the size of feature width. The length of the sequences can be arbitrary. An additional mask is used to mask the output of the model in the phase when no output is requested. Thus only when a number is asked in the target sequence a learning signal is computed. The copy task used in this work is parametrized as in Table 3.1.

3.2 bAbI 20 Task

The bAbI 20 task is a set of 20 synthetic question answering tasks for testing text understanding and logical reasoning. Each sample contains context, questions and corresponding answers, all in natural language. Humans are able to achieve 100% correct answers and no external information is required to answer the questions. Each task of the bAbI set has another question type and needs a different solution strategy.

The bAbI 20 task is introduced in Weston et. al. [56] and the source code to generated the data set is open source under BSD license [1]. Additionally, a set of pre-generated samples is freely to download [2]. The pre-generated sets contain all training, validation and test sets. There are samples in English and in Hindu as well as different large collections available. There are training sets available with 1000 answers per task and with 10.000 answers per task. The validation set and the test

[1] https://github.com/facebook/bAbI-tasks (08.01.2018)
[2] https://research.fb.com/downloads/babi/ (08.01.2018)

set are ten times smaller than the training set. This work uses the pre-generated English version with 10.000 answers per task (en-10k).

Each task contains short stories with one or more questions. Each story is treated as an independent sample. For example task 1 requires one fact to answer the question, the location of the person, see Sample 3.1. Another example, task 7 asks for simple counting exercise, see Sample 3.2. At least in task 17, the model needs to reason about positions, see Sample 3.3. A full description of all tasks can be found in the original paper [56].

bAbI 20 Task - en-10k							
Task	Name	Samples	Avg. Quest.	Vocab. Size	Min. Length	Mean Length	Max. Length
1	Single Supporting Fact	2000	5	22	85	87	93
2	Two Supporting Facts	2000	5	36	89	164	452
3	Three Supporting Facts	2000	5	37	130	486	1920
4	Two Arg. Relations	10000	1	17	24	24	24
5	Three Arg. Relations	2000	5	42	94	208	820
6	Yes/No Questions	2000	5	38	93	98	191
7	Counting	2000	5	46	97	121	361
8	Lists/Sets	2000	5.343	38	87	112	346
9	Simple Negation	2000	5	26	95	100	109
10	Indefinite Knowledge	2000	5	27	95	106	124
11	Basic Coreference	2000	5	29	90	94	100
12	Conjunction	2000	5	23	105	107	112
13	Compound Coref.	2000	5	29	100	104	111
14	Time Reasoning	2000	5	28	116	132	156
15	Basic Deduction	2500	3.241	20	72	72	72
16	Basic Induction	10000	1	20	47	47	47
17	Positional Reasoning	1250	7.034	21	92	109	128
18	Size Reasoning	1978	4.092	21	80	100	249
19	Path Finding	10000	1	26	53	53	53
20	Agent's Motivations	933	10.718	38	45	139	161
all		62661		159	24	91	1920

Table 3.2: Statistics of the bAbI 20 task in the English 10k version.

Each story or sample is pre-processed by removing numbers, lower all words and split the sequences into word tokens. The whole set contains 156 unique words and three symbols, '?', '!', and '-'. The '-' symbol in the input sequence symbolizes that an answer is requested and is used in the target sequence as fill symbol between the correct answers. The statistics in Table 3.2 show that the provided dataset contains some inconsistencies. Some tasks have fewer samples or a different number of questions than expected. Sometimes a question is asked twice direct afterwards, this is detected and deleted.

For training, each sample is encoded as an input and output sequence. Each word is encoded with a one-hot vector with the size of the vocabulary, in the full bAbI task, its 159. An additional sequence mask is used to generate only training signals when an answer is requested, so the outputs of all other time steps are ignored during training. Furthermore, no additional information is provided e.g. what is the current task. The output itself, a vector of the vocabulary size, is activated

with a softmax function. For training, the cross-entropy loss between the prediction vector and the target one-hot vector is minimized. In case of mini-batch training, the different long sequences get padded and masked out with use of a mask. For a more efficient GPU usage, the maximum sequence length during training is limited to 800. This allows larger mini-batches. The loss metric for the bAbI task is the word error rate (WER), the fraction of incorrectly answered words to all requested words.

Task 1: Single Supporting Fact - Sample 3.1

John travelled to the hallway. Mary journeyed to the bathroom. Where is John? ***hallway*** *Daniel went back to the bathroom. John moved to the bedroom. Where is Mary?* ***bathroom*** *John went to the hallway. Sandra journeyed to the kitchen. Where is Sandra?* ***kitchen*** *Sandra traveled to the hallway. John went to the garden. Where is Sandra?* ***hallway*** *Sandra went back to the bathroom. Sandra moved to the kitchen. Where is Sandra?* ***kitchen***

Task 7: Counting - Sample 3.2

Sandra went to the bedroom. Mary went to the office. Mary took the apple there. Mary put down the apple. How many objects is Mary carrying? ***none*** *Mary went back to the bedroom. Mary travelled to the bathroom. How many objects is Mary carrying?* ***none*** *John went to the hallway. John got the football there. How many objects is John carrying?* ***one*** *Mary moved to the hallway. John went to the office. How many objects is John carrying?* ***one*** *Mary went back to the garden. John grabbed the apple there. How many objects is John carrying?* ***two***

Task 17: Positional Reasoning - Sample 3.3

The triangle is above the pink rectangle. The blue square is to the left of the triangle. Is the pink rectangle to the right of the blue square? ***yes*** *Is the blue square below the pink rectangle?* ***no*** *Is the blue square to the right of the pink rectangle?* ***no*** *Is the blue square below the pink rectangle?* ***no*** *Is the blue square below the pink rectangle?* ***no*** *Is the pink rectangle to the left of the blue square?* ***no*** *Is the blue square to the left of the pink rectangle?* ***yes*** *Is the pink rectangle to the right of the blue square?* ***yes***

3.3 Children Book Test

The children book test (CBT) is designed to measure directly how well a model can exploit wider linguistic context. In contrast to the bAbI task, it is built on a collection of children book stories from freely available books through the Project Gutenberg. It has a natural vocabulary size and sample length which are closer

to real-world tasks. The CBT was introduced by Hill et. al in [57]. The data set is under BSD license and freely to download [3]. Each sample in the CBT is constructed by taking 20 consecutive sentences from a book text which build the context and the 21st sentence as a query statement. In the query statement, one word is blanked with a placeholder and the task is to find this word. The dataset is split on book level into training, validation and test set. Additionally, four sub-datasets exists regarding the word category of the blanked word. The four categories are named entities (NE), common nouns (CN), verbs (V) and preposition (P). The word category was found by the Stanford Core NLP tool. While the aim of this work is reading comprehension to exploit the general structure of knowledge rather than correct language and due ot limited computational resources for experiments only the sub-dataset with common nouns is used. An example of the CN dataset is in Sample 3.4.

As a target, each sample contains the correct answer word and a set of ten possible words. The possible words are including the correct word and nine words from the same word category. All possible words have to be present in the previous 20 sentence. The samples are not disjunct. Sometimes the same context and query statement is used with a different blanked word or sometimes the cut between the samples overlaps a few sentences.

In the pre-process each context and query statement gets converted in lower case, numbers get normalized (234rd -> 234) and special symbols get uninformed. Further, the word tokenizer from the NLTK toolkit is used to tokenize the samples. Table 3.3 shows the statistic of the CN CBT dataset.

Children Book Test - Common Nouns Dataset			
	Training Set	Validation Set	Test Set
Books	98	5	5
Samples	120,760	2,000	2,500
Min. sample length	77	500	1,400
Mean sample length	191	476	845
Max. sample length	169	491	1,069
Vocabulary size	53,139		
Possible word vocabulary size	10,696		

Table 3.3: Statistics of the CN CBT

For the training, context and the query statement are concatenated and an input and a target sequence are generated. The input sequence represents the context and query statement word by word as the index of a one-hot vector with the size of the vocabulary. The target vector has non-tokens except on the position of the blanked word, there the true answer is encoded as the index of a one-hot vector with the size of all possible words in the dataset. An additional sequence mask is used to generate only training signals when the answer word is requested, so the outputs of all other time steps are ignored during training.

[3] https://research.fb.com/downloads/babi/ (08.01.2018)

Children Book Test - Sample 3.4

Context:*'What is it?' answered he. 'The ogre is coming after us. I saw him.' 'But where is he? I don't see him.' 'Over there. He only looks about as tall as a needle. 'Then they both began to run as fast as they could, while the ogre and his dog kept drawing always nearer. A few more steps, and he would have been by their side, when Dschemila threw the darning needle behind her. In a moment it became an iron mountain between them and their enemy. 'We will break it down, my dog and I, 'cried the ogre in a rage, and they dashed at the mountain till they had forced a path through, and came ever nearer and nearer. 'Cousin !' said Dschemila suddenly . 'What is it ? ' ' The ogre is coming after us with his dog . ' ' You go on in front then, 'answered he; and they both ran on as fast as they could, while the ogre and the dog drew always nearer and nearer. 'They are close upon us! 'cried the maiden , glancing behind, 'you must throw the pin. 'So Dschemil took the pin from his cloak and threw it behind him , and a dense thicket of thorns sprang up round them, which the ogre and his dog could not pass through. 'I will get through it somehow, if I burrow underground, 'cried he, and very soon he and the* ***XXXXX*** *were on the other side.*
Answer: *dog*
Possible Words: *cousin, cloak, dog, maiden, mountain, needle, path, pin, side, steps*

Furthermore, a possible-words mask is created which masks out all words in the output vector which are no possible words. Each time step, the model outputs a vector with the size of all possible answer words. It is multiplied element-wise by the possible words mask. A softmax function activates the output vector. During training, the cross-entropy loss between the predicted and masked vector and the target one-hot vector is minimized. In case of mini-batch training, the different long sequences get padded and masked out with use of the mask. The loss metric for the CBT is the accuracy, the fraction of all samples with correct words to the number of samples in total.

3.4 CNN Reading Comprehension Task

Similar to the CBT, the CNN reading comprehension (RC) task has a natural source. It is based on crawling the online news articles from the CNN website[4] collected from April 2007 to April 2015. This large-scale training dataset was introduced by Hermann et. al [58] and is freely downloadable as raw collection[5] under Apache2 license or pre-processed [6]. The dataset contains news articles as context and short article summaries as query statement. Each article is the source for four queries on average and each query is the replacement of a word by a placeholder in the summary. The articles and summaries anonymized all name entities with tokens and each missing word is such a name entity, see Sample 3.5. The articles contain rarely name entities like celebrities and the task aims to exploit general relationships between anonymized entities rather than common knowledge.

[4]http://cnn.com/ (08.01.2018)
[5]https://github.com/deepmind/rc-data (08.01.2018)
[6]https://cs.nyu.edu/ kcho/DMQA/ (08.01.2018)

CNN RC Dataset			
	Training Set	Validation Set	Test Set
Month	95	1	1
Articles	90,266	1,220	1,093
Samples	380,298	3,924	3,198
Min. sample length	16	107	96
Mean sample length	775	777	731
Max. sample length	2,018	2,007	2,009
Vocabulary size	118,497 (50,429)		
Tokens	408		

Table 3.4: Statistics of the CNN RC Task.

The pre-processed dataset, which is used in this work, has all words in lower case and reduces some inconsistencies. For the training, the query (summary with the placeholder) and context are concatenated (the query first) and an input sequence is generated. The input sequence represents the summary with the placeholder and the article word by word as the index of a one-hot vector with the size of the vocabulary. In contrast to the CBT dataset, the target is only the correct word represented as the index of an one-hot vector with the size of all possible name entity tokens. An additional candidate mask is created which masks out all name entity tokens which are not present in the sample.

CNN RC Task - Sample 3.5

Query: *@entity2 took a photo next to tourists looking at the "**@placeholder**" home.*
Context: *(@entity0) what happens when @entity2 shows up outside the "@entity5" home that made him a star? nothing, apparently. for reasons unknown to us, @entity2 was recently lurking outside the popular @entity10 attraction when a group of tourists showed up. it appears they were too busy gawking at the house to realize "@entity16" was standing next to them. it appears that @entity2 was amused and posted an @entity19 of him standing next to them. the caption? "boy, these youngsters have 0.0 idea what they're missing. @entity26 turnaround. "hopefully, he found some consolation in the 46,000 likes and thousands of comments from swooning fans. otherwise, @entity2 is keeping busy with guest roles on "@entity36" and "@entity37" amid rumors of a possible "@entity5" reunion.*
Answer: *@entity5*

During training, the last model output in the sequence predicts the word. It has the size of all possible name entity tokens and is activated by a softmax. The candidate mask is multiplied element-wise to this output. So the output for all entity tokens which are not present in the current sample is ignored. During training, the cross-entropy loss between the prediction output vector and the target one-hot vector is minimized. In case of mini-batch training, the different long sequences get padded from begin of the sequence that the last sequence step stays the same. The loss

metric for the CNN is the accuracy, the fraction of all samples with correct words to the number of samples in total.

4. Related work

This chapter contains two sections. The first one is concerned with related models to the DNC in the context of solving NLP problems. It illuminates alternative approaches, memory-augmented neural networks as well as non-memory approaches and shows some advanced applications. The second part addresses existing enhancements for NTMs and DNCs.

4.1 Related models for QA

At the same time as the NTM paper has been published a work by Weston et al. was released which introduced an alternative family of memory-augmented neural networks, the memory neural networks (MemNNs) [59]. It is a concept of an external memory component for QA and exists of input feature map, generalization, output feature map and response components. The approach based on a sentence representation and requires supporting facts during training like the supporting words in the bAbI task to find the answer. It is successfully applied to the bAbI task [56] as well as to the large simple QA task KB-Freebase [60]. In contrast, the DNC does not require supporting facts, is trainable end-to-end and is more interactive with the memory due to the read and write heads.

The follow-up work of Sukhbaatar et al. [61] extends the memory network with a soft attention mechanism, needs less supervision and is trainable end-to-end. In contrast to DNC, it stores information explicitly in memory without the opportunity to erase or modify them during an episode. The memory network has no addressing mechanism and the memory is written sequentially. This makes the model simpler and more focused on the task itself. It is mainly used as a knowledge database with a very limited interaction between computation and storage. There are several successors of memory networks in a QA context: The hierarchical memory networks tackle the issue of extremely large memories with a hierarchical approach [62]. The key-value memory network splits the memory matrix in two, one for key embeddings, one for value embeddings [63]. The long-term memory network uses an LSTM to generate arbitrary long answer sentence [64]. The gated memory network introduces a regulation mechanism inspired by the connection short-cutting principle [65]. The

dynamic memory networks extend the model with an episodic memory module to provide information about the question on each state for a better attention [5]. The dynamic memory network (DMN) and its successor the DMN+ have a similar concept but with an episodic memory module [5, 66]. It is a sequential model and the DMN + requires no labelled supporting facts during training.

The same approach also finds applications besides QA like in dialogue modelling [67], machine translation to deal with out-of-vocabulary words [68] or visual question answering [69]. Especially in video QA, where a model is asked to answer questions related to a video, memory-augmentation is particularly successful and achieve here state-of-the-art results [70]. The memory-augmentation uses a LSTM controlled fixed-size memory matrix [71] or builds a dynamic memory matrix embedding on the fly [72, 73].

The relation memory network (RMN) embeds sentences into a memory object and applies attention with use of the question. An update step renews the memory and a second attention is applied to find the answer [74]. In contrast to the DNC, the RMN is fitted to QA tasks and uses sentence representation as memory slots similar to the MemNN.

Another related model is the recurrent entity network (EntNet) which "can be viewed as a set of separate recurrent models whose hidden stated store the memory slots" [75]. The memory slots exist of a key vector and a content vector and have an own gated RNN as a controller. In contrast, our model has one memory matrix with no distinction between key and content.

Some other memory-augmented approaches are worth to mention besides the memory networks. Grefenstette et al. introduce a logically unbounded memory with push and pop operations [76]. The "neural semantic encoder" from Munkhdalai and Yu has three separate controller modules for reading, update or write operations [77]. It has a sophisticated addressing mechanism to avoid information collision in the memory and is applied to machine translation as well as QA [78]. Henaff et al. introduced the "recurrent entity network" which has simple parallel mechanisms to update several memory locations simultaneously similar the hidden state in an LSTM network but with a content-based addressing [75, 79]. The RelNet from Bansal et al. models entities as abstract memory slots and is able to reason about them [80].

The following models are more focused on large-scale QA tasks. Hermann et al. [58] uses a Deep LSTM Reader which performs a forward pass over the context and the question to find the answer. They also introduce the attentive and impatient readers which build a document representation by direct attention respectively incrementally attention. In contrast to the DNC, it needs a separation between document and query.

[81] introduces the Stanford Attentive Reader which enhances the attentive reader and introduces a bilinear term to compute the attention between the document and the query. The Attention-Sum (AS) Reader from [82] uses also separate encoding for the document and the query. Its successor, the Attention-over-Attention Reader, applies a two-way attention mechanism to find the answer [83]. Some models use a multi-run reasoning over a hidden representation of the document. [57] applied the MemNN with use of a fixed-length windows representations of text surrounding each

candidate answer. Then an attention mechanism is used multiple times to find the answer. The Iterative Attention Reader and the ReasoNet use iterative reasoning as well [84, 85]. The Gated-Attention Reader from [86] also uses multiple hops over the document to build an attention over the candidates to select the answer token. [87] introduces the EpiReader which uses two neural networks, one extracts candidates using the AS Reader and the other re-ranks them conditioned on the query and the document. These readers are all conceptual adapted to the QA tasks they solve. Our solution is manifold usable due to a more flexible and universal design. This allows an easier handling of new tasks.

Besides the memory-augmented neural networks, attention-based models are a promising approach. Either through a question representation based attention [88] or with use of a 2D attention embedding for example with a similarity matrix [89], an attention embedding matrix [90] or a matching matrix [91]. Another common proceeding is to transfer the context and/or question with the use of neural networks in a utile representation and use a neural network to predict an answer or answer span. This can be done with bidirectional LSTMs [92] or a Match-LSTM and answer pointer [93]. The dynamic co-attention network fuses representations of the document and the question and predicts spans with the use of a pointer [66]. Some works using a combination of attention and matching layer [94]. This also can be done with a combination of word-level and character-level representations [95] or with use of a convolution neural network [96]. The DNC does not require an attention mechanism or this reformulation of the problem.

4.2 Related enhancements

The dynamic neural Turing machine from Gulcehre et al. 2016 splits the memory of the NTM into a content and address vector for better location-based addressing [97]. They also implement a soft differentiable and a hard non-differentiable read/write mechanism. Similar to a work from Khadka et al. a GRU is used as controller [98]. Khadka et al. also uses gates to determine if the content is written, updated or read. Another approach follows Gulcehre et al. 2017 with a more simpler discrete addressing for write/read operations to learn so-called wormhole connections, which store the previously hidden states of an RNN in the memory that later provide a flashback in the history of the RNN [99].

Zhang et al. introduce three extending variants of an NTM, an additional hidden memory for smoothing, an additional hierarchically memory and a multi-layer implementation [100]. They showed alleviating of over-fitting and increased accuracy. A binary tree with leaves corresponding to memory cells is built by Andrychowicz and Kurach a hierarchic attentive memory based on the NTM model [101]. This leads to faster memory access and enables learning of algorithms like merging, sorting or binary searching.

A greatly technical enhancement provide Rae and Hunt et al. with the introduction of sparse read and write operations [102]. They address the problem that the memory consumption depends on the memory matrix size since all memory slots get manipulated at once. To solve this issue a threshold determines a sparse subset of interest memory slots which get modifications. This reduces computation time and memory consumption but unfortunately, it is currently not implementable with

common frameworks like TensorFlow or PyTorch due to the lack of differentiable sparse tensors [103, 104].

5. Analysis of the DNC

The analysis examines the functionality and computations resources of a classic DNC to get a better understanding of this model. The analysis is split into four sections and a main contribution of this work. The first section investigates the DNC training and the second its functionality. The third section examines the memory consumption and the last section the computation time. This chapter ends with drawing a conclusion from the DNC analysis.

5.1 DNC Training

The training behaviour is investigated with two data sets, the copy task and the bAbI task 1 en-10k dataset, see Sections 3.1 and 3.2. Both tasks are small enough to train them within a day on one GPU. The performance on the copy task is an indicator for the DNC as an algorithmic solver and the bAbI task 1 represents the performance on QA tasks. With both datasets, five training runs are recorded. These have the same hyper-parameters but different seeds. The hyper-parameters are listed in Table 5.1, both tasks have the same values except for the batch size. The models are trained with use of the RMSprop optimization algorithm [30].

Part	Paramter	Symbol	Value
Controller	LSTM layers	L	1
	LSTM units	C	64
Memory Unit	Memory length	N	128
	Memory width	W	32
	Read heads	R	2
Training	Learning rate	lr	1e-4
	Momentum	m	0.9
	Batch size	B	16(copy task)/32(bAbI task)

Table 5.1: The hyper-paramter of DNC training analysis.

To get comparable results the following two tasks are evaluated with a DNC and a vanilla LSTM. The LSTM on the copy task does not achieve a better performance than a wrong number rate of 0.97 and it also overfits slightly. This means that the LSTM fails on the copy task completely. The bAbI task 1 is reported in the original paper with a mean word error rate of 0.28 by using an LSTM with one hidden layer and 512 hidden units[1].

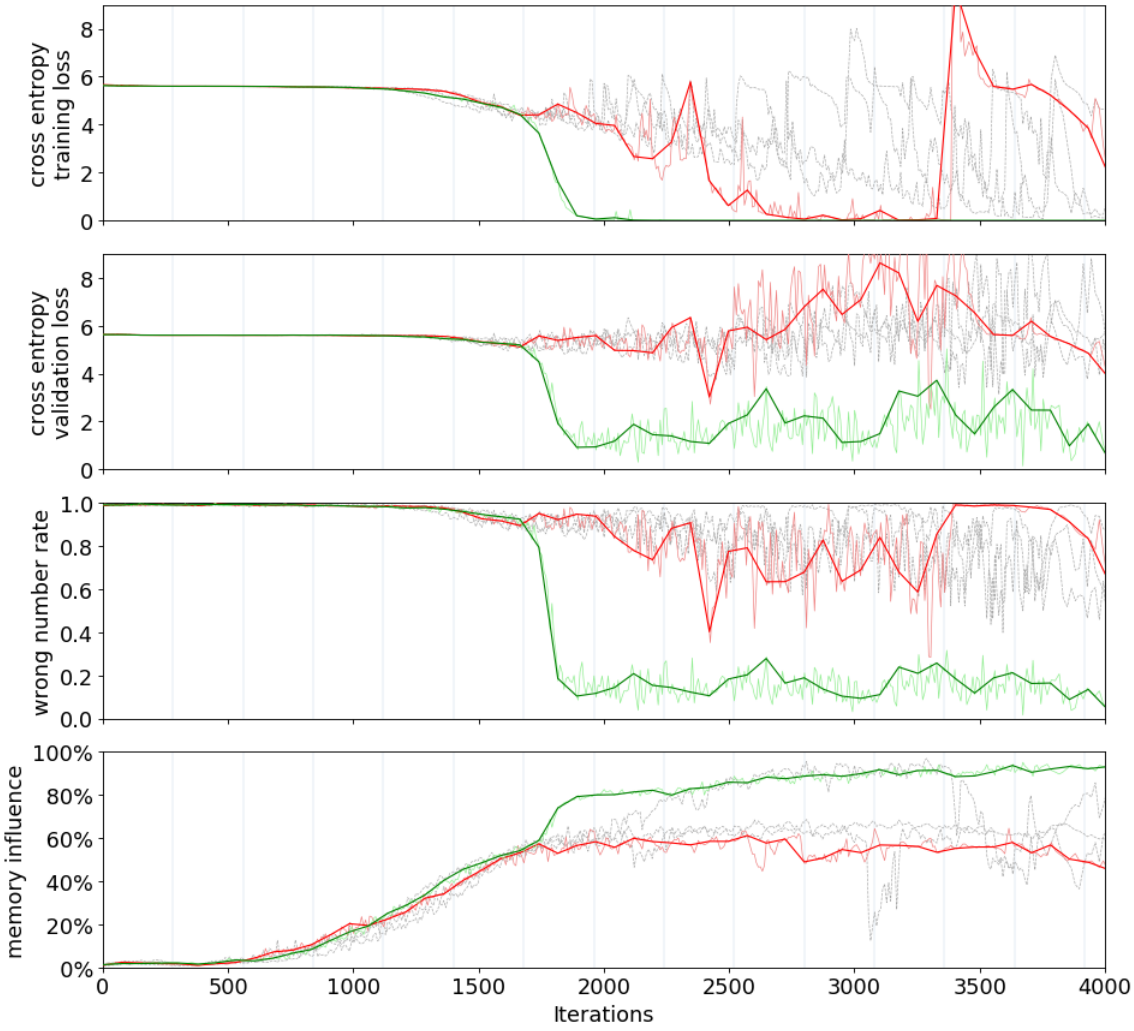


Figure 5.1: The training progress of a DNC with the copy task on five runs. The red line is the worst run and green line the best run. The grey lines are the other runs, full lines are smoothed, dashed are raw.

Figure 5.1 shows the training progresses of the copy task. The plot shows five runs while the green line is the best run with respect to the rate of wrong numbers and the red line the worst. The full lines are smoothed and the dashed lines are the true training progress. The first plot shows the training loss of the five runs. The second plot shows the validation loss. The third shows the wrong number rate. The validation loss and the wrong number rate were calculated after every tenth training updates. The bottom plot shows the influence of the memory read output to the system output signal. It is computed by the consideration of the change in system output by setting the memory read output to zero. It is normalized with use of setting the bypass output to zero. A memory influence of zero means only

the controller via the bypass influences the system output and a memory influence of one means only the memory read output affect the output signal.

Only one of the five runs converges after a long time of nearly no progress in any loss. This could indicate that the model needs time to learn how to use the memory. The run in which training loss converges to zero has the best validation performance and a larger memory influence in the output signal. This also indicates a more intensive or efficient usage of the memory unit. Only if the model learns to use the memory unit it is able to generalize to the validation task. The spike in the worst run, in the plot the red line, shows how unstable the training of the DNC is. The model massively overfits in the first 3000 iterations and then suddenly a few training updates let collapse the overfitting and the training loss explodes.

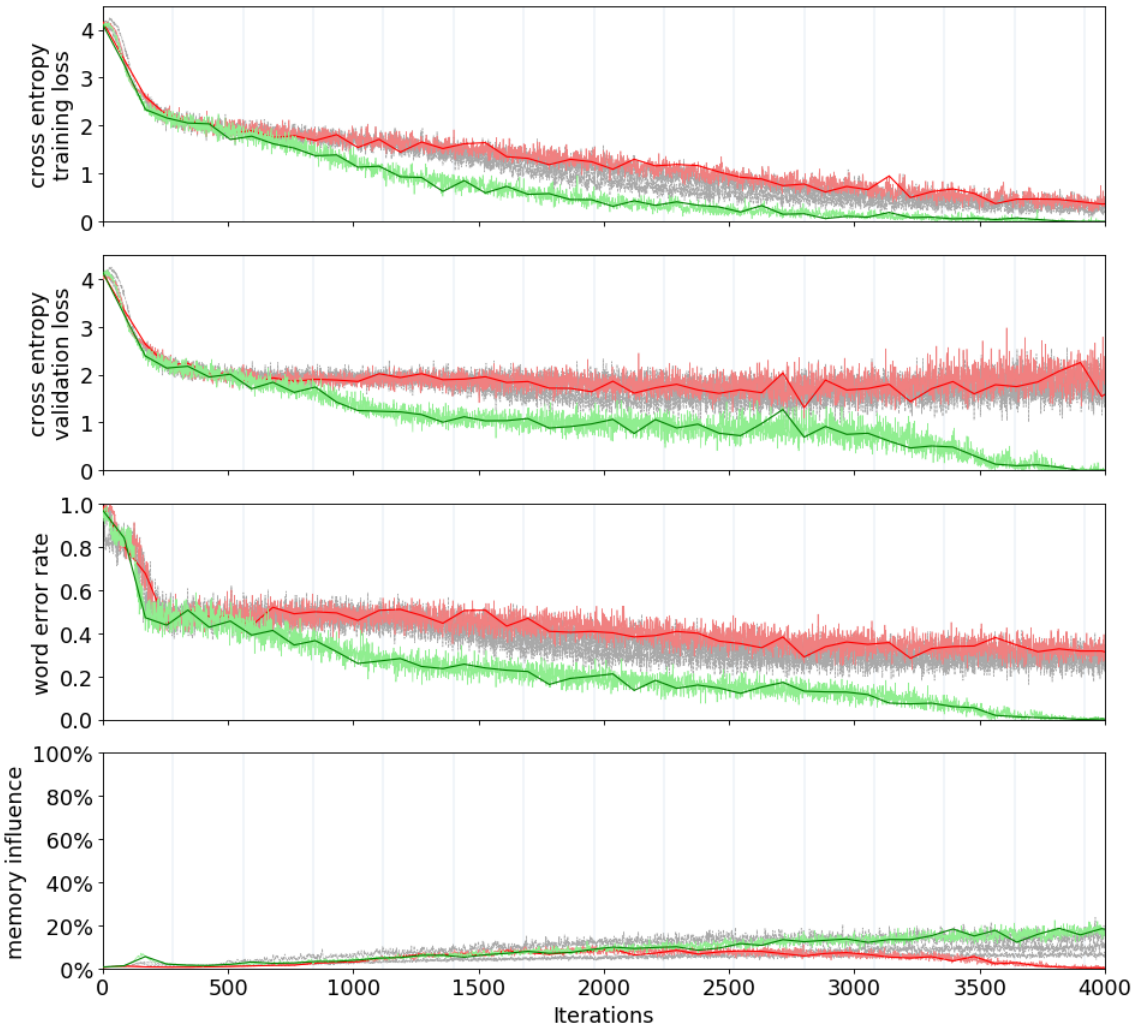


Figure 5.2: The training progress of a DNC with the bAbI task 1. 5 runs, red line is the worst and green line the best run.

The performance on the bAbI task 1 has the same relation in terms of training progress, see Figure 5.2. The plot is recorded and plotted in the same manner as in the copy task analysis. The model with the best convergence uses the memory most. But the memory influence is below 20% compared to above 80% in the copy task training. The convergence is more smooth perhaps since the task is more easy to handle. The passable performance of the LSTM could imply that the correct memory

usage is not mandatory for an incremental performance increase. In contrast to the copy task, which requires a working memory mechanism for good performance. But again, with the stronger use of the memory, the task is solved completely. Overfitting is recognizable in both models but when the training loss converges completely to zero and the memory is used then the validation loss converges as well.

Both experiments show that the model performance strongly depends on the memory usage. If the DNC learns to use the memory mechanisms, the tasks are fully solvable with it. Additionally, the model is unstable during training and the performance depends on the initialization. Sometimes it convergences and the DNC learns to use the memory mechanisms and sometimes not. The DNC does not tend to overfit since the training loss, the validation loss and the performance rate (wrong numbers or word error) are strongly correlated when the memory mechanism is in use. This indicates out that the memory usage is strongly correlated with the model performance and only if the DNC learns to use the memory unit, the performance rises.

5.2 DNC Functionality

Similar to the previous section the copy task and the bAbI task 1 is used to analyze the functionality. Two fully trained and converged models, the best runs from the previous section with the same parameters, are used to investigate the functionality of a DNC. The analysis consists of four plots which show gate usage, weightings and influences of the read heads. The first two plots show the general DNC functionality in the copy task and bAbI 1 task. The third plot provides insights in the write mechanism exemplary with the bAbI task 1 and the fourth plot shows details of the memory handling and read head exemplary with the copy task.

Name	Description
Questions	The questions plot shows the model input, the grey pads symbolize that an answer is expected. The model first receives the numbers which should be repeated after the delimiter symbol.
Predictions	The prediction plot shows the model output, green pads corresponds to correct numbers and red to false numbers.
Free Gates	This plot shows the free gates of the two read heads.
Alloc Gate	This plot shows the allocation gate for determining where to write new content. Yellow corresponds to the usage of the ”at least used memory slots” and blue pads to the content-based addressing.
Write Gate	This plot shows the write gate, only the input sequence is written.
Read Modes 1	This plot shows the read mechanism of the first read head.
Read Modes 2	This plot shows the read mechanism of the second read head.
Head Influences	This plot shows the influence of the read heads on the output of the memory unit. It is similarly computed as the memory influence in Section 5.1.
Output Influences	The bottom plot shows the influence of the memory unit and controller to the output signal.

Table 5.2: The description of DNC functionality subplots in Figure 5.3.

Each plot is described in detail in the following. The first plot in Figure 5.3 gives an overview of the DNC functionality in the copy task. The nine subplots show one short sample of the copy task and are described in Table 5.2.

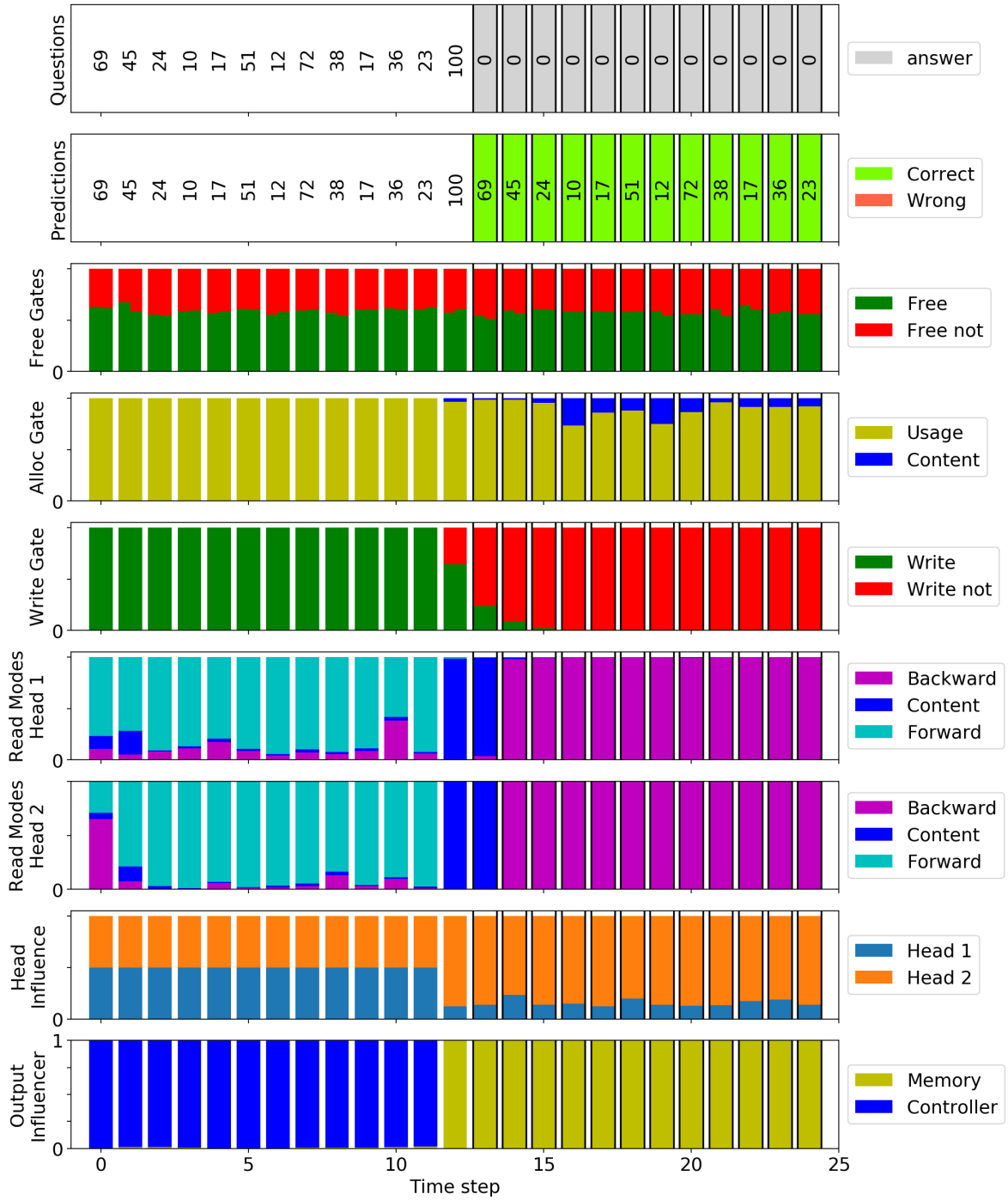


Figure 5.3: The functionality of the DNC gates and influences in the copy task. The activity of the gates and read modes are the actual values between 0 and 1. Then influences are calculated and normalized.

The free gates seem not to be used, probably since the memory is larger than the task is long and there is no need to free memory slots. Because the numbers in the copy task are independent only the dynamic memory allocation mechanism is used to compute the write weighting. During the number input sequence, the write

gate is close to one because content needs to be written into the memory. But it is close to zero during the response since manipulating the memory may harm the memorization.

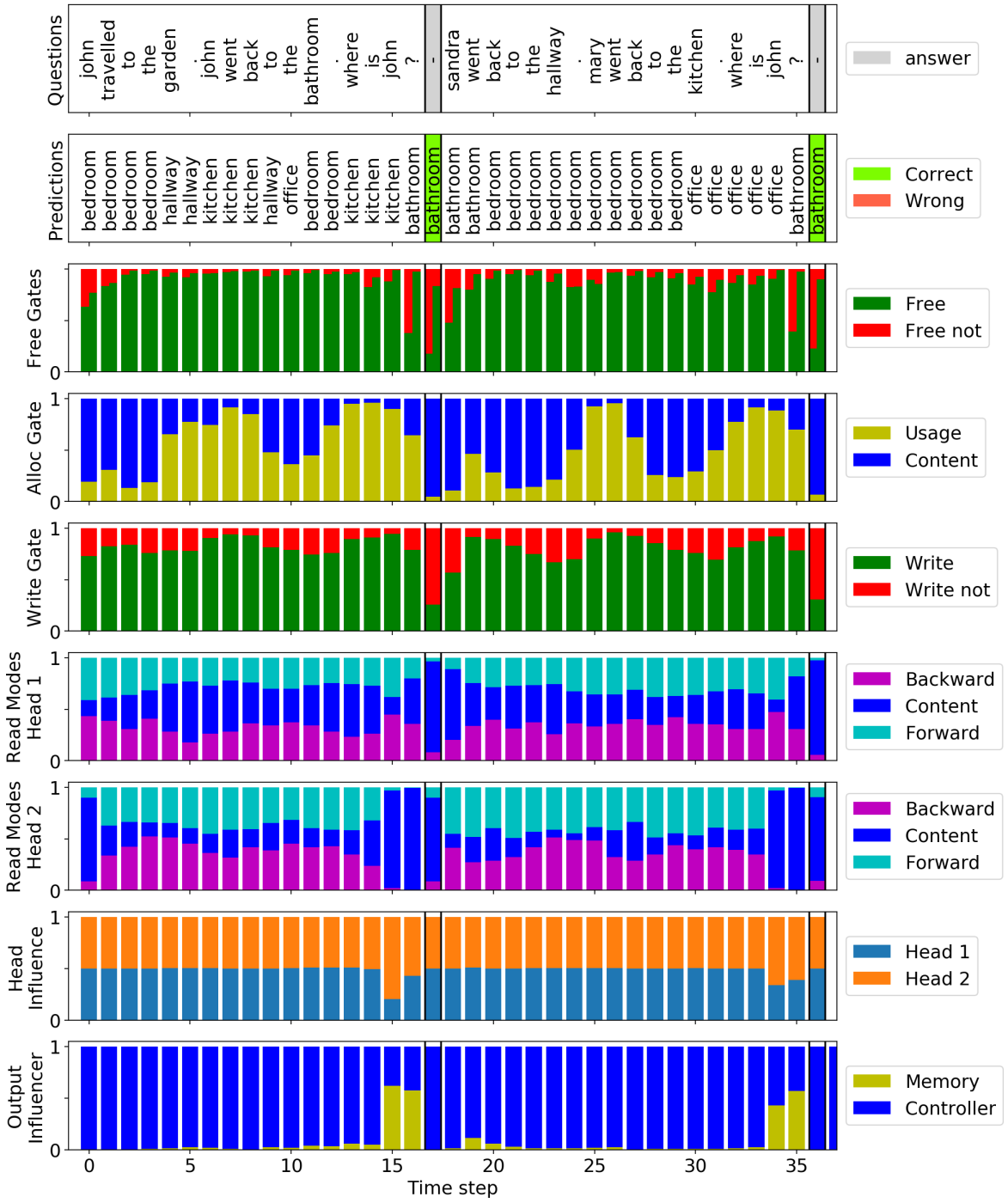


Figure 5.4: The functionality of the DNC gates and the influences on base of the bAbI task 1. The activity of the gates and read modes are the actual values between 0 and 1. Then influences are calculated and normalized.

Both read heads use the content-based weighting to find the first number in the memory and afterwards only backward weightings since the input sequence needs to be repeated. The larger influence of the second head suggests that the second read head performs better. During the number input, the read mode is forward direction

probably since it helps to allocate unwritten memory locations. The dynamic memory allocation mechanism uses these previous read weightings, see Section 2.5.2.3. In the response sequence, only the memory influences the output signal, which shows the successful usage of the memory unit.

The same type of plot shows in Figure 5.4 the DNC functions with a shortened sample from the bAbI task 1. The output of the model is masked out when no answer is requested. So the model's output is ignored in these phases. This could be the reason why it outputs arbitrary words when no answer is requested. In this scenario, the model uses more the content-based addressing and permanently frees space because some sample-sequences are longer than the memory. The write gates show that permanently content is written to the memory matrix except when an answer is requested. Probably the model reacts to the last word of the question or the question mark rather than the question token '-'. This could explain why the memory influence is high on the steps before the answer is requested. The model likely finds the answer with use of the content-based reading. This explains the read modes in the second read head right before the answer is requested. In that case, the influence of the second read head is stronger. Then it holds the correct answer until it is requested. Conversely, the forward and backward weighting mechanisms are not used in this QA scenario. This makes sense since a sequence repeat is not necessary to find the correct word in the memory.

Name	Description
Questions	The questions plot shows the model input, the grey pads symbolize that an answer is expected. The model first receives the numbers which should be repeated after the delimiter symbol (100).
Predictions	The prediction plot shows the model output, green pads correspond to correct numbers and red to false numbers.
Free Gates	This plot shows the free gates of the two read heads.
Allocation Weightings	This plot shows the allocation weighting which determines slot positions which are not or sparsely in use so far.
Usage Weightings	The usage vector in this plot memorizes which slot is in use.
Content Weightings	This plot shows the content weighting based on the write key from the controller.
Allocation Gate	This plot shows the allocation gate which determines whether the allocation or content-based weightings are used for the new write weightings.
Write Weightings	This plot shows the updated write weighting.
Write Gate	This plot shows the write gate which determines the intensity of writing.

Table 5.3: The description of DNC write functionallity shown in subplots in Figure 5.5.

The following two plots consider the write, memory update and read mechanism of the DNC in detail to get a deeper understanding of the functionality. Figure 5.5 shows the process to find the write weightings. The gates are similar to the previous

plots and the weightings show the intensity of the locations. They have the length of the memory matrix and the red bars show the location weighing from 1 in red to 0 in white in a logarithmic scale.

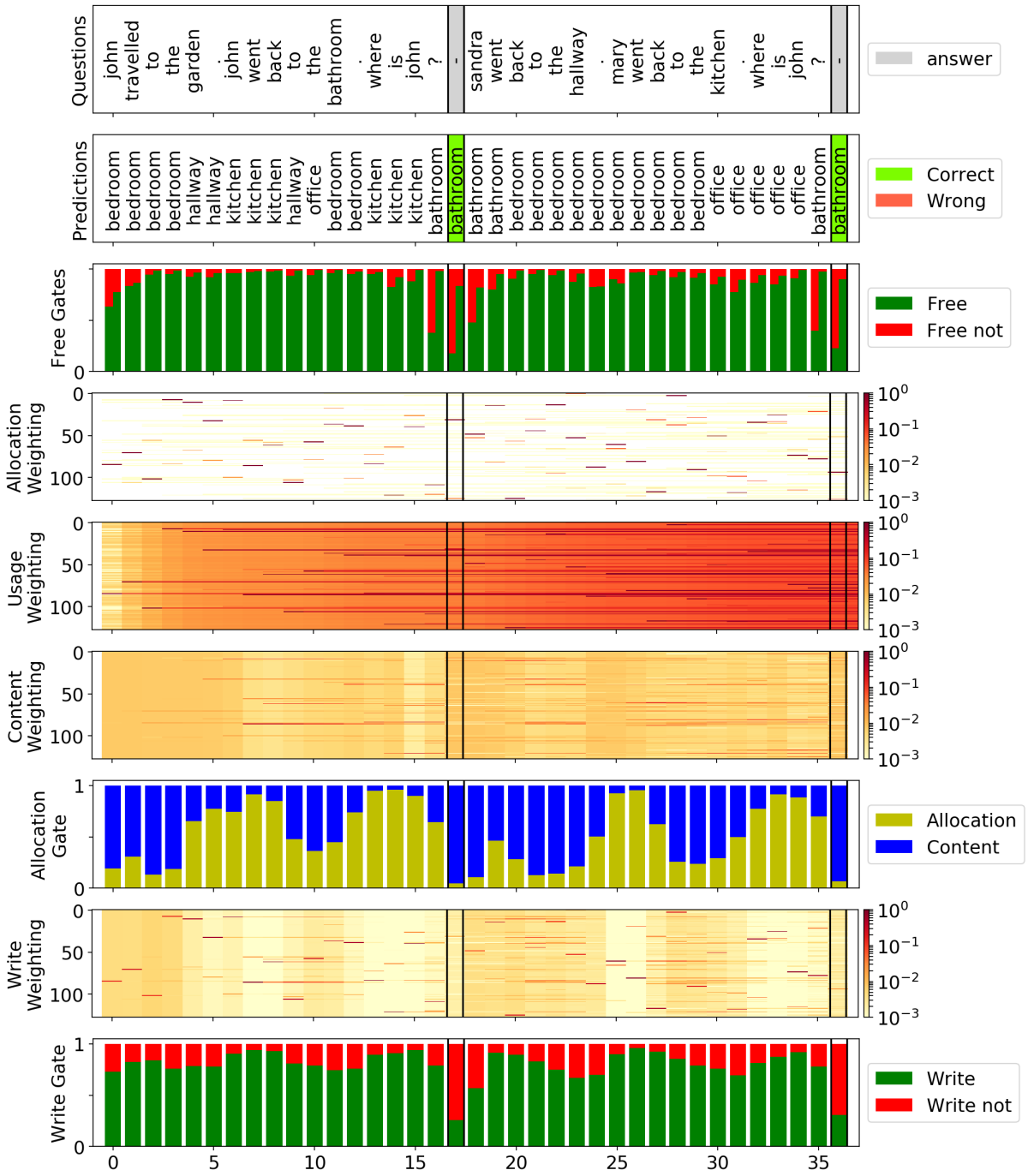


Figure 5.5: The function of the write mechanisms in the memory unit during the bAbI 1 task. The activity of the gates are the actual values between 0 and 1. The weightings have the length of the memory matrix and the red bars show the location intensity.

The nine subplots show all gates and weightings of the write mechanism of one short sample of the bAbI task 1 and are described in Table 5.3.

The allocation weighing provides in each time step a different unwritten memory location due to the empty memory and the sort based design. The actual location depends on the random initialization. The usage vector stores the used memory location and is filled over time. In the first steps, no content-based addressing is possible since the memory is quite empty. But couple steps before an answer is requested the content weightings are more exact and provide the locations which help to find the answer. The allocation gate determines the influence of the allocation and content weighting to build the actual write weighting.

Another detailed analysis provides Figure 5.6. It shows the memory handling step by step in a memory unit during the copy task and is described in Table 5.4.

Name	Description
Task	The task plot shows the model input in the grey pads and the model output in the colored pads. Green pads imply correct number and red false numbers.
Old Memory	This plot illustrates the memory over the time steps. The memory slot content is depicted with respect to the color bar on the right.
Write Weightings	This plot shows the write weightings from the write mechanism.
Write Vector	The write vector in this plot came directly from the input signals. It is the weighted controller output.
Add Matrix	This plot shows the add matrix created with the write vector and the write weightings.
Erase Vector	The erase vector in this plot came also directly from the input signals.
Erase Matrix	This plot shows the erase matrix, which is created with the use of the erase vector and the write weightings.
New Memory	This plot shows the updated memory matrix with use of the add and erase matrices.
Read Weightings 1	This plot shows the read weightings of the first read head.
Read Vector 1	This plot shows the read vector of the first read head.
Read Weightings 2	This plot shows the read weightings of the second read head.
Read Vector 2	This plot shows the read vector of the second read head.

Table 5.4: The description of DNC memory update functionallity shown in subplots in Figure 5.6.

This plot shows how the memory gets full filled with new content provided by the add matrix. The erase mechanism provides weak weightings, probably since erasing is in this task not important. Afterwards, the memory gets written through the two read heads. Read head 2 has more clear weightings and produces better read vectors. This also explains why read head 2 is more used in Figure 5.3.

The analyze plots point out the importance of the content based addressing in the read head for the bAbI 1 task. Especial in QA the forward and backward weighting makes less sense since it is not impotent to repeat a sequence rather than to find a specific information in the memory matrix. The write mechanism plot and the

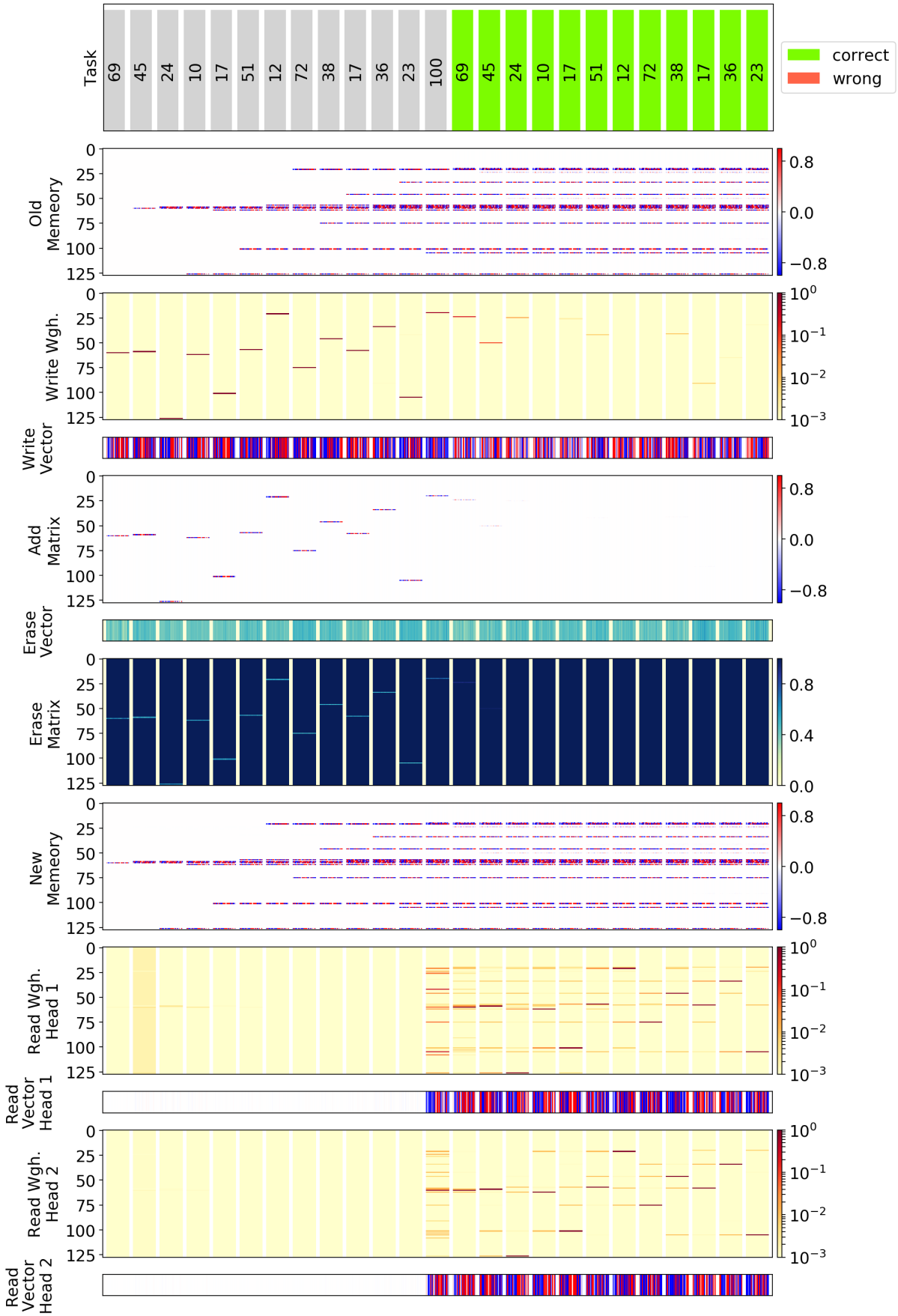


Figure 5.6: The memory handling in a DNC during the copy task. The colored bars are the content in a location. Either in a vector or in a matrix.

memory update plot show that the DNC mechanisms work as expected and provide a deeper imagination of the functionality.

5.3 DNC Memory consumption

This section considers the memory consumption of a DNC relating to the main causer and compares to a standard LSTM. The memory consumption of an RNN, in general, depends on the four parts: library overhead, parameters, operations footprint and recurrent connections. The library overhead is empirically determined and is approximately 150 MB with use of the TensorFlow 1.3 library [103]. It does not depend on batch size or sequence length. The following calculation makes use of the notation from Chapter 2. Table 5.5 provides an overview. For a more simpler calculation, no multilayer LSTM/Controller networks are considered.

Model type	Parameter	Symbol
	Input size	X
	Output size	Y
	Sequence length	S
	Batch size	B
LSTM	Layers	L
	Hidden units	H
DNC	Controller size	C
	Memory length	N
	Memory width	W
	Read heads	R

Table 5.5: The notation of DNC and LSTM parameters.

Based on the software implementation of the DNC the parameters of an LSTM and a DNC can be calculated as follows:

$$\begin{aligned} &\text{LSTM Parameters:} \\ &\text{HiddenNodesParameters} \equiv (X + H + 1) \cdot 4 \cdot H \\ &\text{OutputParameters} \equiv (H + 1) \cdot Y \end{aligned} \tag{5.1}$$

$$\begin{aligned} &\text{DNC Parameters:} \\ &\text{ControllerParameters} = (X + C + R \cdot W + 1) \cdot 4 \cdot C \\ &\text{MemoryParameters} = (C + 1) \cdot (3 + 5R + 3W + R \cdot W) \\ &\text{OutputParameters} = (R \cdot W + C + 1) \cdot Y \end{aligned} \tag{5.2}$$

Each operation produces a memory footprint to calculate the gradients for the back-propagation step. The footprint of an operation has the size of the operations outcome tensor. The overall footprint strongly depends on the implementation of the

model. The source code of this work is available online[1]. The operations footprint for one step in an LSTM and a DNC can be calculated as follows:

LSTM Operations Footprint:	
LSTMOperations	$= 16 \cdot 8H$
OutputOperations	$= Y$

(5.3)

DNC Operations Footprint:	
ControllerOperations	$= 16C$
MemoryUnitWritting	$= 3 + R + 2W + 10N + W \cdot N$
LinkageMatrixOperations	$= 4N + 3N \cdot N$
MemoryUnitReading	$= 5R + R \cdot W + 6RN$
OutputOperations	$= Y$

(5.4)

The recurrent connections generate an additional footprint for the back-propagation step. In the LSTM setting, it is the hidden states and the output size of the nodes. In the DNC setting it is the read weighings, the linkage matrix, precedence weightings, memory matrix, write weightings and the usage vector. The following calculations are for the recurrent connections footprint for one-time step.

LSTM Recurrent Connections Footprint:	
HiddenStates	$= H$
RecurrentOutput	$= H$

(5.5)

DNC Recurrent Connections Footprint:	
ControllerRecurrentConnections	$= 2C$
MemoryWrittingRecurrentConnections	$= 2N + NW$
LinkageMatrixRecurrentConnections	$= N + NN$
MemoryReadingRecurrentConnections	$= RN$

(5.6)

The memory requirement of the parameters stay the same for each step and each batch size. But the operations footprint, the data footprint and the recurrent connections footprint depends on the sequence length and batch size. It is a product of the footprint per step, the sequence length and the batch size. The resulting overall footprint is a sum of all four parts, the overheads, the footprints, the parameters and the actual training data. Additionally, the optimizer can have a footprint due to cache values and the sample input data and the corresponding target data. In case of a back-propagation setup the memory consumption of the operations and recurrent connections footprint is doubled, one forward and one backward [105]. The

[1] https://github.com/joergfranke/ADNC (10.04.2018)

following calculation is for the memory consumption of the whole model during a back-propagation training setting:

$$
\begin{aligned}
&\text{Library Overhead} \\
&+ \text{Parameters} \\
&+ \text{Optimizer Overhead} \\
&+ \text{S B (Input Data + TargetData)} \\
&+ \text{2 S B (Operations Footprint)} \\
&+ \text{2 S B (RecurrentConnections Footprint)} \\
&= \text{Total Footprint}
\end{aligned}
\tag{5.7}
$$

To make a statement about the memory impact of the single components in a DNC model and to identify the main causer a parametrization is necessary. The following example with the hyper-parameters of the DNC model from the original paper in the bAbI experiment [1], see Table 5.6. In this example, the RMSprop optimizer is used which leads to an optimizer footprint in size of the parameter amount.

Part	Parameter	Symbol	Size
	Input size	X	159
	Output size	Y	159
	Sequence length	S	100
	Batch size	B	1
Controller	LSTM layers	L	1
	LSTM units	H	512
Memory Unit	Controller size	C	256
	Memory length	N	256
	Memory width	W	64
	Read heads	R	4

Table 5.6: The parametrization of the DNC in original paper for bAbi task [1].

All values are stored in 32bit floating-point format. This leads to the following memory consumptions:

	Values per step	Factor	Total Values	Total Size
Library Overhead				150 MB
Parameters			1,457,823	5.83 MB
Optimizer			1,457,823	5.83 MB
Operations Footprint	8351	200	1,670,200	6.68 MB
RecurrentConnections Footprint	512	200	102,400	0.41 MB
Input Data	159	100	15,900	0.06 MB
Target Data	159	100	15,900	0.06 MB
LSTM Training				**168.87MB**

	Values per step	Factor	Total Values	Total Size
Library Overhead				150 MB
Parameters			890,742	3.56 MB
Optimizer			890,742	3.56 MB
Operations Footprint	227,386	200	45,477,200	181.91 MB
RecurrentConnections Footprint	84,224	200	16,844,800	67.38 MB
Input Data	159	100	15,900	0.06 MB
Target Data	159	100	15,900	0.06 MB
DNC Training				406.54 MB

In this example, compared to an LSTM the DNC has more than the double memory footprint in training. The main memory consumption is due to the operations and recurrent connections footprint. This grows with sequence length and batch size double. The main consumer of memory is the dynamic temporal linkage mechanism with about 210 MB. This is over half of the total memory need. If, for example, the sequence length grow up to 1000 steps then memory of 2.6 GB would necessary. Thereby the linkage matrix consumes 2.1 GB or about 80% of the total memory amount.

This example shows that the dynamic temporal linkage is the main consumer of memory. This is due to the huge linkage matrix with size $N \times N$. In large-scale QA tasks, this can be a massive problem since the sequence length is often longer than 1000 words per sample. This slows the training not only due to computational effort but also due to the smaller batch size due to limited GPU memory.

5.4 DNC Computation time

To get an intuition of the computation time of a DNC and a comparing to an LSTM model the hyper-parameters from the original paper [1] are used, see Table 5.6. All experiments accomplished on a workstation containing an Intel Core i7 and one single Nvidia GeForce GTX 1080 GPU with use of CUDA 8.0, cuDNN 6.0 and TensorFlow 1.3. As data set the copy task with a sequence length of 20 is used. The resulting time measurement in Table 5.7is the mean of 3 runs measured with the TensorFlow internal program TFprofiler [103].

	LSTM	DNC
Inference per Step	1 ms	4 ms
Inference per Sequence	24 ms	79 ms
Back-propagation per Sequence	87 ms	267 ms

Table 5.7: Time mean comparison between a DNC and LSTM with the reference parametrization from the original paper [1].

The duration difference between the two models for inference and back-propagation is nearly the same. So the DNC takes 3 times longer for training and 3.3 times more time for inference. If the sequence length gets increased to 1000 steps then the

back-propagation time grows up to 8.2 s per sequence. This results in training times per epoch on a dataset with $100k$ samples and a batch size of 16 in 14.2 h.

This shows that the additional mechanisms of the DNC results in longer training times compare to vanilla LSTMs. This can be a serious constraint for the usability of the model especially if the batch size is small due to the huge memory requirements. A model with training times of weeks is harder to develop and a hyper-parameter tuning is expensive. This can limit the application of the DNC.

5.5 Analysis conclusion

This analysis shows that the better results in the two considered tasks, the copy task and the bAbI task both reported in [1, 55], have their price. The computation time, as well as the memory consumption, are significantly larger comparing to a standard LSTM model. The performance distinctions are examined in more detail in Chapter 7. Furthermore, this analysis contains a couple of important insights. First, Section 5.1 shows that learning to use the memory unit is crucial to the convergence success. The miscarrying runs in Figure 5.1 and 5.2 seems to affiliate to an unused memory unit or an excessive use of the controller LSTM via the bypass. It also takes a very long time, until the model learns to use the memory unit.

A second insight is a large amount of memory consumption and long computing times. With the same hyper-parameters as in Section 5.3 but a sequence length up to 1000 steps, which is normal length for large QA tasks, would result in a required memory amount of 2.61 GB for one sample. In a mini-batch training setting a batch size of 5 would result in over 131 GB which does not fit in a common GPU VRAM. Also, a training time of 8.2 s per sample in a large QA dataset scenario with 100.000 samples would take 228 h per epoch or in case of mini-batch training with 4 samples per batch 57 h.

Another important finding is the exclusive use of the content-based read weighting mechanism in the bAbI QA task seen in section 5.2. Potentially the content-based addressing is sufficient for most QA tasks. It would only be important to the question asks for a sequence or a span of words in the memory. In any other case, the answer depends more on a content-based query. This is important since the mechanism for the forward and backward weightings are the most memory expensive part in the DNC model, seen in Section 5.3. The next Chapter 6 provides several approaches which address issues of high computing resources as well as an earlier memory usage.

6. Advancements in the DNC

The previous Chapter 5 shows several drawbacks of the DNC model in terms of training performance or usability for large QA tasks. This chapter tackles some of this issues and presents advancements to the DNC setup. The first section improves the training and makes it faster and more robust. The second section introduces some more sophisticated architectures which are novel in the DNC setting and enhance the performance as well. The third section introduces a memory unit without a linkage matrix mechanism for large-scale content focused tasks with a greatly reduced memory footprint.

6.1 Robust DNC training

The goal of improving the training is to get a more robust convergence behaviour and a faster usage of the memory unit. An enhanced training should lower the large variance in training performance within different initializations and the slow and unstable convergence behaviour. This makes the training repeatable and reduces the training time. There are two common methods to enhance training: normalization and regularization. This section applies normalization for a more consistent convergence and bypass dropout as regularization for a precision force to memory usage. Both are novel in conjunction with DNCs.

6.1.1 DNC Normalization

The DNC shows in present experiments on the bAbI task a high variance in performance between different runs. We approach this issue with normalization techniques to enforce a robust and a smoother convergence behaviour. In recent years two normalization techniques for neural networks are popular: batch normalization [106] and layer normalization [107]. Both have the same goals, reducing over-fitting, increase convergence velocity and better generalization.

Batch normalization (BN) normalizes the input of an activation function to zero mean and unit variance over the mini batch. So the mean and variance are calculated for each input feature element-wise across the samples in the mini-batch. The BN is only applied in training and before each activation function. In an RNN setting

the recurrent inputs from the previous time step must be treated separately from the input of the current time step to keep the recurrent functionality [108].

However, the paper where LN is introduced and a recent work show that LN outperforms BN [107, 109]. Furthermore, the efficiency of BN depends on the batch size. Several papers show good results but only with batch sizes larger than 24 samples [108, 110]. Because of the huge memory consumption of the DNC, see analysis in Section 5.3, the batch size is often below 24 samples. Therefore and due to the better results in the literature, this work applies LN to the DNC model.

In the DNC setup, it can be applied to the controller as well as in the memory unit. Let $\mu_t^{(l)}$ be the mean of a vector x_t

$$\mu_t^{(l)} = \frac{1}{H}\sum_{h=1}^{H} x_{h,t}^{(l)} \tag{6.1}$$

and $\sigma_t^{2(l)}$ the variance of it

$$\sigma_t^{2(l)} = \frac{1}{H}\sum_{h=1}^{H}(x_{h,t}^{(l)} - \mu_t^{(l)})^2 \ . \tag{6.2}$$

The normalization is calculated over the inputs of activation function $x_t \in \mathbb{R}^{B\times H}$ in each iteration and for each layer l. This is done for each sample independent in the mini-batch. Each layer use trainable variables, called bias $\boldsymbol{b}^{(l)}$ and gain $\boldsymbol{g}^{(l)}$. They scale the normalization which is applied to the weighted layer signals $\boldsymbol{x}$ before the activation:

$$\mathrm{LN}(\boldsymbol{x}_t^{(l)}) = \boldsymbol{g^{(l)}} \circ \frac{\boldsymbol{x}_t^{(l)} - \mu_t^{(l)}}{\sqrt{\sigma_t^{2(l)} + \epsilon}} + \boldsymbol{b}^{(l)} \ . \tag{6.3}$$

Let $\circ$ be an element-wise multiplication. The bias variable normally added to the weighted input can be omitted. It is applied in training and test times. In an RNN setting the recurrent and current input signals can be computed jointly.

To stabilize the state dynamics in the DNC LN is applied to the input signals of the memory unit. This is called "DNC normalization". It can be applied to the gates, the vectors and the keys separately but this does not show performance increase compared to a joint normalization of all signals. So it is applied after the weighting of the controller output and before the vector is split into the different signals. This leads to changes in Equation 2.18 which describe the use of the controller output as memory unit input.

$$\xi_t = \mathrm{LN}(\boldsymbol{h}_t W_\xi) \tag{6.4}$$

The drawback is more computation time and more memory need due to the momentum calculation and the additional gain and bias variables.

6.1.2 Bypass Dropout

By observing the convergence behaviour of the DNC a strong correlation between the model performance and the usage of the memory unit is apparent. Only if the memory unit mainly impacts the system output, the model achieves good performance, see analysis in Section 5.1. This insight demands the explicit force of memory unit usage during the training to reach a faster convergence and a better performance.

To force the memory units influence on the output, the impact of the controller to the output via the bypass connection can be diminished. This can be reached by different approaches. One solution cloud be impairing the controller network. For example by reducing the controller's network size or a strong regularization, but this would also affect the control functionality of the memory unit.

A better and novel approach is to diminish the direct influence of the controller network to the output. Therefore the connectivity between the controller network and the output gets reduced by adding dropout to the bypass connection called bypass dropout (BD), see Figure 6.1.

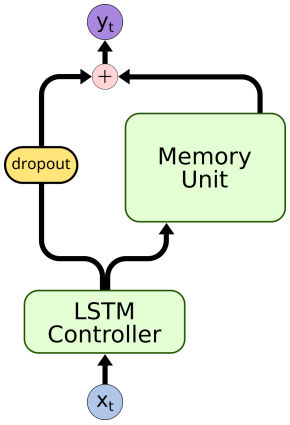


Figure 6.1: The DNC system with bypass dropout between the LSTM Controller and the system's output.

Dropout is a regularization technique introduced 2014 by Srivastava et al. to prevent neural networks from overfitting [111]. In this usage, dropout allows an adjustable reduction of the bypass connectivity during training and helps to enforce a faster usage of the memory unit. With dropout the input $\boldsymbol{x}_t$ of a layer l becomes multiplied with an iteratively drawn, Bernoulli distributed, random variable $\boldsymbol{r}_t$:

$$\begin{aligned} \boldsymbol{r}_t^{(l)} &\sim \text{Bernoulli}(p) \\ \widetilde{\boldsymbol{x}}_t^{(l)} &= \boldsymbol{x}_t^{(l)} \circ \boldsymbol{r}_t^{(l)} \\ \boldsymbol{y}_t^{(l)} &= \sigma(\widetilde{\boldsymbol{x}}_t^{(l)} \boldsymbol{W}^{(l)} + \boldsymbol{b}^{(l)}) \end{aligned} \tag{6.5}$$

This causes that a random subset of the parameters of an ANN is frozen in each iteration. This increases the robustness of the parameter weights and reduces the possibility of full adaption to the dataset. Dropout is only applied during training and not during the testing or inference.

In this work, dropout allows an exact regularization of the signal flow from the controller to the output without declining the controller functionality itself. Only the controller influence to the output signal is constrained. It's only applied during training to force an earlier memory usage. BD is easy to apply in the Equation 6.6 of the DNC setup

$$\boldsymbol{y}_t = W_h(\boldsymbol{h}_t \circ \boldsymbol{r}_t) + W_\mu \boldsymbol{\mu}_t \cdot \tag{6.6}$$

with the element-wise multiplication of a Bernoulli vector $\boldsymbol{r}_t$ to the controllers output $\boldsymbol{h}_t$. It does nearly use no more computation power or memory usage. The strength of the regularization can be adjusted with the probability destitution p of the multiplied Bernoulli variable $\boldsymbol{r}_t \sim \text{Bernoulli}(p)$.

6.2 Advanced Architecture

Architecture advancements is another way to improve the DNC performance. This section introduces two novel enhancements, the bidirectional DNC and an atop neural network layer instead of a single output layer.

6.2.1 Bidirectional DNC

The unidirectional architecture of the DNC makes it hard to handle variable input where for example the requested answer word can appear in the middle of a text. It also prevents a rich information extraction in forward and backward direction in any sequential task. In order to provide a complete availability of the input sequence to the model, this work introduces a bidirectional DNC setup. There is no more distinction between context and question necessary. In the bidirectional DNC (BiDNC) an addition RNN in backward linkage provides a sequential comprehension in both directions and provides a view into the future like described in section 2.4.

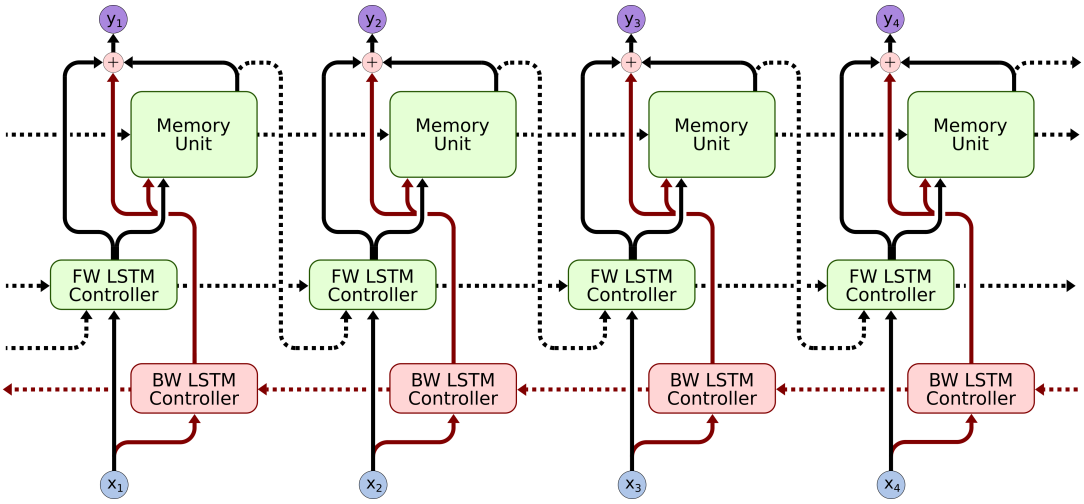


Figure 6.2: The bidirectional controller architecture in the BiDNC setup.

In the unidirectional DNC setup, the output of the memory unit feeds the input of the controller in the next time step. Because of this recurrent connection, an encapsulated bidirectional controller is not possible. The backward controller can

not receive an input feed from the previous memory unit output because this would depend on the future output of the backward controller. Such a model wouldn't be unfoldable in time. The solution introduced in this work at first applies a backward directed recurrent controller network without input from the memory unit. This backward controller provides an additional input signal to the memory unit and output layer, see Figure 6.2. Therefore, the BiDNC has two controllers in the setup, a forward controller

$$\boldsymbol{h}_t^{fw} = \text{ForwardController}([\boldsymbol{x}_t, \boldsymbol{h}_{t-1}^{fw}, \boldsymbol{\mu}_{t-1}], \theta_{c^{fw}}) \tag{6.7}$$

and a backward controller

$$\boldsymbol{h}_t^{bw} = \text{BackwardController}([\boldsymbol{x}_t, \boldsymbol{h}_{t+1}^{bw}], \theta_{c^{bw}}) \tag{6.8}$$

with independent weights θ and recurrent connections $\boldsymbol{h}_{t+1}$. The memory unit receives as input a concatenation of the two controller outputs

$$\boldsymbol{\mu}_t = \text{MemoryUnit}([\boldsymbol{h}_t^{fw}, \boldsymbol{h}_t^{bw}], \theta_{mu}) \cdot \tag{6.9}$$

The output of the BiDNC system is a sum of the weighted memory output, the weighted backward controller output and the weighted forward controller output:

$$\boldsymbol{y}_t = W_{\mu}\boldsymbol{\mu}_t + W_{fwh}\boldsymbol{h}_t^{fw} + W_{bwh}\boldsymbol{h}_t^{bw} \cdot \tag{6.10}$$

This architecture firstly allows an independent unfolding of the backward controller and secondly an unfolding of the forward controller and memory unit.

6.2.2 Atop RNN

The fundamental idea of an atop RNN instead of a vanilla output layer is an increase of computation complexity to generate the output signal. Some models in related work have such an output structure and shown a benefit of more complexity in the output layer [66, 89]. In the DNC setting, it would enable a time-related rendition of the memory output to solve complex interdependent sequential tasks.

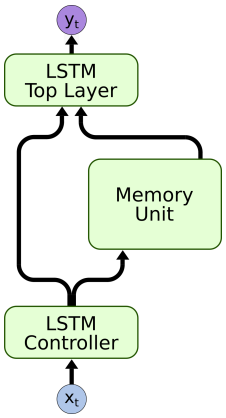


Figure 6.3: A DNC setup with an atop RNN.

The architecture change is simple. The add aggregation of the controller output and memory unit output is replaced by an additional RNN, in this work a LSTM network, see Figure 6.3. The inputs to this atop RNN are a concatenation of the controller output and the memory unit output. The outputs of the atop RNN are the new residuals and replaces the Equation 6.6:

$$\boldsymbol{y}_t = \text{AtopRNN}([\boldsymbol{h}_t, \boldsymbol{\mu}_t], \theta_{top}) \cdot \tag{6.11}$$

The output RNN extends the unidirectional setup with an additional computation complexity. This could help to enrich the interpretation of the memory read results.

6.3 Content-Based Memory Unit

For an efficient usage of the model and the scalability to large-scale tasks is less memory consumption mandatory. A lower memory requirement allows dealing with larger sequences, bigger batch sizes and faster iterations for a feasible hyper-parameter tuning.

But the memory consumption of the DNC is very high due to many hidden states in form of vectors, weightings and matrices. They need to be kept in memory for the backpropagation for every single step. The analysis in Section 5.3 shows the temporal linkage mechanism is particularly expensive since the linkage matrix has the size $N \times N$.

However, the analysis of the DNC functionality in Section 5.2 shows that the DNC mainly uses the content-based addressing to find the read weightings in the bAbI task. This makes sense since restoring of sequences is barely necessary for finding the correct answer in a QA task. Therefore is no need for forward and backward memory queries and the linage matrix (LM) is no more in use.

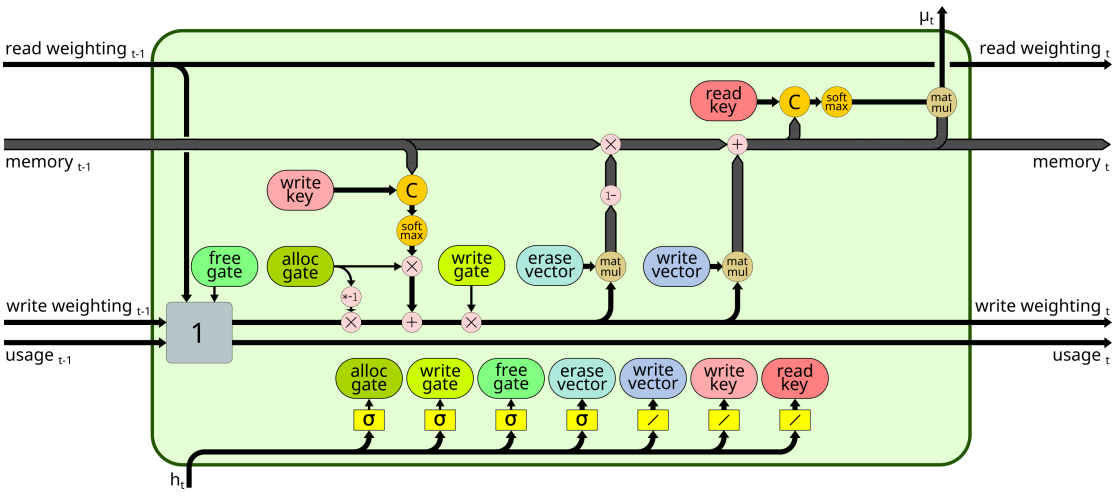


Figure 6.4: The schematic of an content-based memory unit.

To allow a more efficient usage in QA a novel, slim and only content-based memory unit (CBMU) is introduce. The CBMU has the same features as the DNC's MU but without the temporal memory linkage mechanism, see Figure 6.4. Consequently, following components are removed: the precedence weightings, the linkage matrix

including its updating mechanism, the forward and backward weightings and its computation and the read mode for finding the read weightings. The read weightings are only based on the content-based addressing. The write mechanism, memory update and the actual memory reading stay the same. The content-based weightings

$$\boldsymbol{c}_t^{r,i} = C(M_t, \boldsymbol{k}_t^{r,i}, \beta_t^{r,i}) \tag{6.12}$$

are the only read weightings and no more read modes are required to compute the memory unit's output:

$$\boldsymbol{\mu}_t = [\boldsymbol{c}_t^1 \dots \boldsymbol{c}_t^R] \cdot \tag{6.13}$$

This leads to a drop of the linkage matrix operations, the read operations and the recurrent connections of these:

$$\begin{array}{ll} \text{Dropped Parts:} & \\ \hline \text{LMOperations} & = 4N + 3NN \\ \text{ReadOperations} & = 5R + 5RN \\ \text{RecurrentConnections} & = N + NN \end{array} \tag{6.14}$$

The reduction of memory consumption in training depends on the sequence length S and the batch size B:

$$\begin{array}{l} \quad 2 \text{ S B LMOperations} \\ + \text{ 2 S B ReadOperations} \\ + \text{ 2 S B RecurrentConnections} \\ \hline\hline = \text{Memory Savings} \end{array} \tag{6.15}$$

The factor of 2 results from the forward propagation and backpropagation of the signal in the network during training. In the standard bAbI setting with parameters equal to Section 5.3, the reduction is from 406.5 MB to 191.5 MB which is a reduction of 52.9%.

Besides the memory consumption reduction, the computation effort drops as well. This speeds up the training but the most impotent advantage is that such a slim model allows sizable mini-batches. The drop in memory consumption depends on the hyper-parameter and the sequence length but in this work, it is between 30% and 70%. The computation time is reduced by 10% to 50%. Accordingly, the training time can be reduced from weeks to days. Additionally, the inference time gets reduced what is important for practical applications.

With the loose of the dynamic temporal linkage mechanisms, the model gets closer to related models like MemNN or DMN since they also have no linkage mechanism. But the DNC is still more flexible due to opportunity to manipulate the memory at each step. It keeps its dynamic addressing mechanisms for writing and reading and is still a very flexible model but the CBMU allows to train larger QA tasks.

7. Experiments

This chapter experientially evaluates the hypothesis from the methods Chapter 6. At first, each training improvement and architecture advancements get evaluated in detail with smaller tasks due to limited computing resources. Afterwards, the advancements in the DNC are evaluated with the full bAbI 20 task. Because of the memory consumption reduction and decreased computation time due to the content-based memory unit a DNC training with a large-scale dataset is possible. Therefore it gets evaluated on the children book test and the CNN reading comprehension task. A least the results get analyzed and summarized. All used datasets are introduced in Chapter 3 and all experiments are implemented in TensorFlow [103].

7.1 Empirical methods evaluation

Two smaller datasets are used to evaluate the methods introduced in Chapter 6. They are chosen to reach training times from one or two days instead of weeks concerning the limited computation resources. The first subsection evaluates the training improvements and architecture advancements described in Sections 6.1 and 6.2. The second subsection examines the content-based memory unit comparing to a vanilla DNC memory unit.

7.1.1 Training and architecture advancements

To compare the enhancements the evaluation focus is more on training convergence rather than best/mean result. Therefore the DNC's validation loss of the different advancements is compared. For a consistent comparison, each advancement uses the same basic model and is recorded on five runs. The initialization seeds variates between the runs but the different seeds are the same in all advancements experiments.

7.1.1.1 bAbI Task 1 evaluation

The first smaller task is the bAbI task 1 whereby the hyper-parameters of the model are found by a small grid search and are as follows: The controller is an LSTM with one hidden layer and a size of 64. The memory unit has a length of 128 and

a width of 32 with one write and two read heads. The model's input and output is a one-hot word vector with size 32 due to the vocabulary size 32. For training, mini-batch SDG with a batch size of 32 is used. As optimizer RMSprop is used with a constant learning rate of 0.0001 and a momentum of 0.9. The loss is the cross-entropy between a softmax output and a one-hot target vector of the correct word.

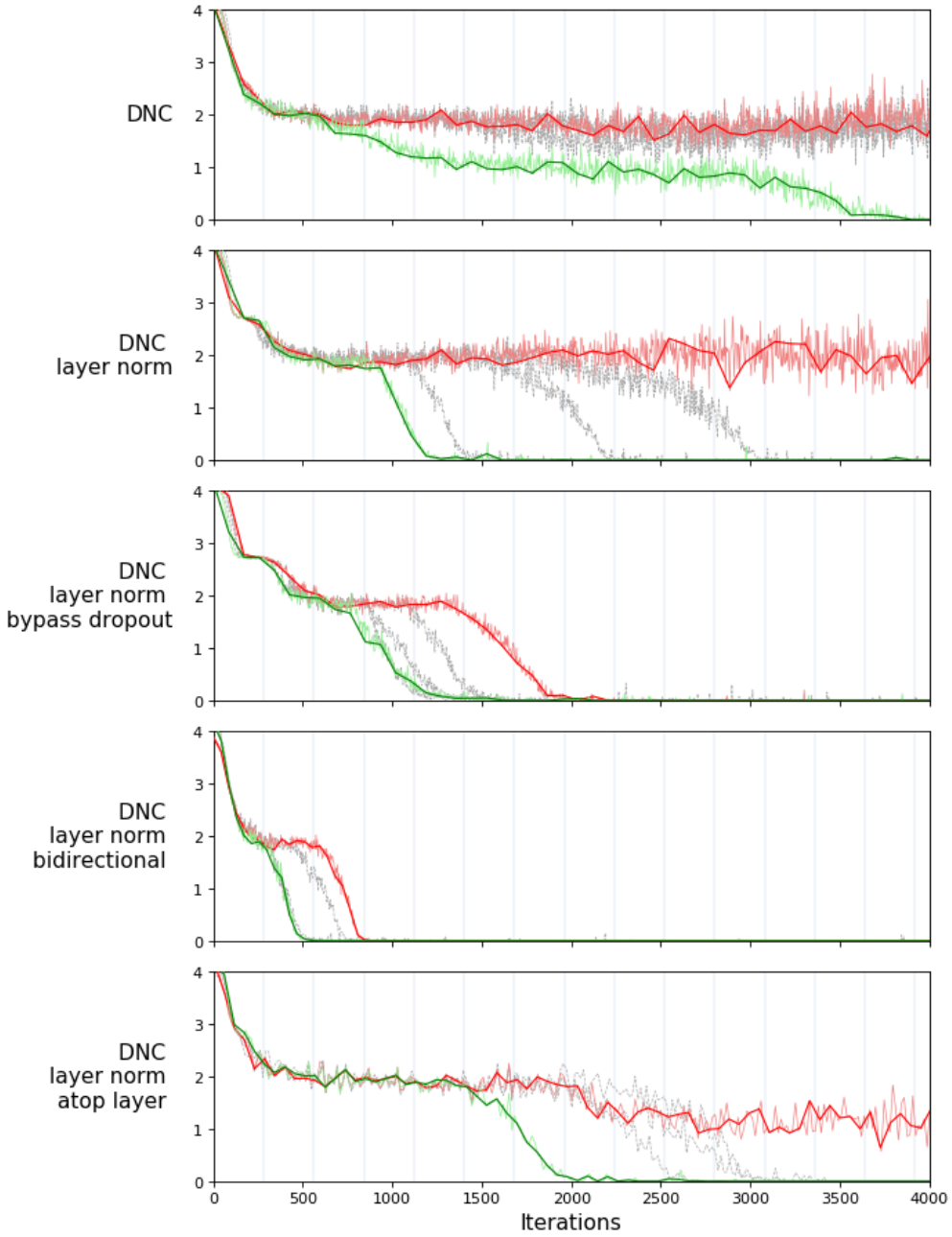


Figure 7.1: Comparison of the cross entropy loss of the validation set between the different DNC advancements on the bAbI task 1.

Figure 7.1 shows the comparison of the different advancements. Each plot shows the continuously recorded validation loss of five runs. The green line is the best run and the red line the worst. The dashed lines are the true loss and the full lines are the smoothed loss.

The first plot shows the vanilla DNC and the second plot the DNC with DNC normalization (layer norm). The third plot shows a model with bypass dropout (20% dropout rate) and the fourth plot the bidirectional architecture. At the bidirectional setup, each controller LSTM has a size of 32 units. Hence the parameter amount gets reduced from 53725 (uni/64) to 37341 (bi/32-32). The bottom plot is a model with an atop LSTM layer with 32 hidden units.

The benefits of LN and bypass dropout (BD) are discernible. The LN speeds up the convergence and makes the training more stable. Four of five runs solve the task while without LN it is only one. Because of this smoothing of the training progress, LN is used in all other experiments. The BD improves the convergence behaviour again. The anticipated faster convergence due to the dropout is clearly recognizable. All runs solve the task in less than 2000 training iterations.

The bidirectional setup has only about two-thirds of the parameter amount compared to a uni-directional model but increases the convergence behaviour once again. All runs solve the task in less than 1000 training iterations. It is faster, more consistent and solves all runs. A possible reason for this is the access to the question due to the backward path. Thereby, the model is able to attend to the important parts of the context. This complete availability of the input sequence to the model leads to a robust and fast training with fewer parameters.

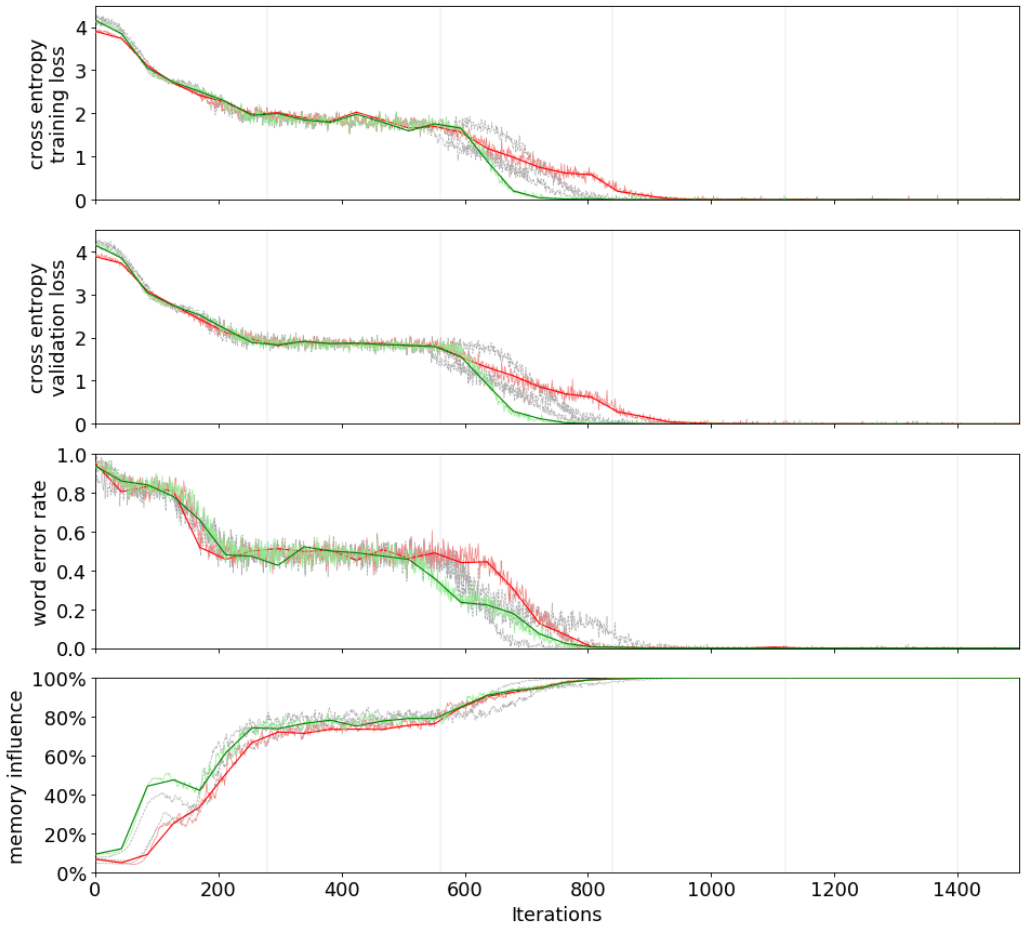


Figure 7.2: The convergence behavior of a DNC with LN and BD on the bAbI task 1 with respect to the memory influence.

The atop LSMT layer has a slight benefit since it reduces the loss of the run which is not solving the task. This could imply an increased model complexity but compare to the BD or bidirectional setup it is less successful and uses more parameters. In the following evaluation only LN, BD and the bidirectional setup are considered but not the atop LSTM due to the little improvement.

Figure 7.2 shows the best DNC with LN and BD on bAbI task 1 in the same plot-style as above. This plot shows the comparison between training loss, validation loss, word error rate and memory influence. Indifference to Figure 5.2, from the DNC analysis in Section 5.1 which shows the vanilla DNC, a faster memory influence is noticeable. It shows a strong correlation between train loss, validation loss, the word error rate and the memory influence. With every increase of the memory influence, the loss decreases. This underlines the impact of memory usage on the performance of a model in such a QA setting and shows that the advancements perform properly.

7.1.1.2 Copy Task evaluation

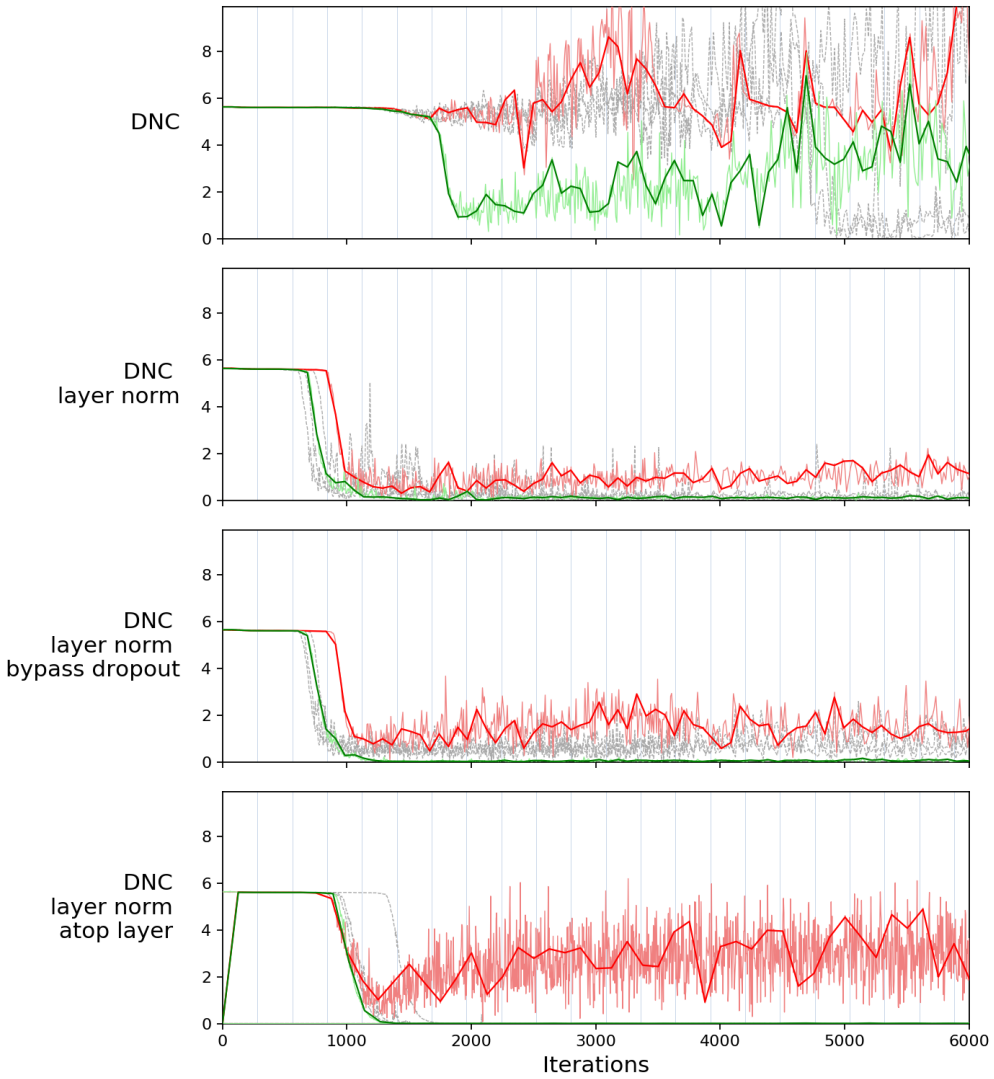


Figure 7.3: Comparison of the cross entropy loss of the validation set between the different DNC advancements on the copy task.

The second task to evaluate the enhancements is the copy task. It is used to evaluate the impact of LN, BD and atop RNN layer. In this setting, a bidirectional architecture does not make sense since the task is to store and repeat a sequence. The hyper-parameters and optimizer stay the same as in the bAbI task 1 but with a batch size of 16. The input and output to the model is a one-hot vector with size 100. The loss function is a cross entropy computed with a softmax output layer and a target one-hot vector.

The first plot in Figure 7.3 shows the convergence of the five runs with a vanilla DNC. Again the same plot-style is used as above. The second with LN, the third with LN and BD and the last with an atop RNN layer (LSTM, 32 units).

As in the bAbI task 1 experiment, a clear improvement in convergence is visible. The LN leads to a significantly more stable and faster convergence during training. The additional BD leads to a more smoother descend in the loss. Again the atop RNN layer shows benefits but compared to the other advancements it is too much effort for a worse result.

7.1.2 Content-Based Memory Unit

The evaluation of the content-based memory unit (CBMU) uses the bAbI task 1 as well. Again, the focus is on convergence rather than performance. The hyper-parameters are the same as in the previous section and the output is a softmax with the cross-entropy loss. The parameter amount of the model in the unidirectional setting is only decreased from 53725 with the vanilla memory unit (MU) to 53323 with the CBMU but the memory footprint of the bAbI task 1 training drops from 662 MB to 293 MB (mean sequence length 93, batch size 16).

Figure 7.4 compares the DNC with vanilla MU and the CBMU with different advancements. The first plot shows the DNC with the vanilla MU and LN. The second plot shows the DNC with CBMU and LN. The third and fourth plot shows it additional with bypass dropout and a bidirectional controller.

Between the DNC with vanilla MU and CBMU no clear degeneration is recognizable through the reduction of the linkage matrix mechanism. This could imply that the temporal linkage mechanism has no important impact on the performance in a QA task. The other plots show similar improvements through the advancements as in the previous Figure 7.1 with the vanilla DNC. This indicated that the DNC performs with the CBMU similar compared to the vanilla MU.

Furthermore, the inference and training time decrease significantly. Table 7.1 shows the computation time comparison between the DNC with vanilla MU and CBMU with the same reference parametrization and task as in Section 5.4.

Model	DNC	CB-DNC
Inference per Step	3.5 ms	2.7 ms
Inference per Sequence	79.1 ms	69.2 ms
Back-propagation per Sequence	267.1 ms	238.1 ms

Table 7.1: Time comparison between a DNC with MU and CBMU with the reference parametrization.

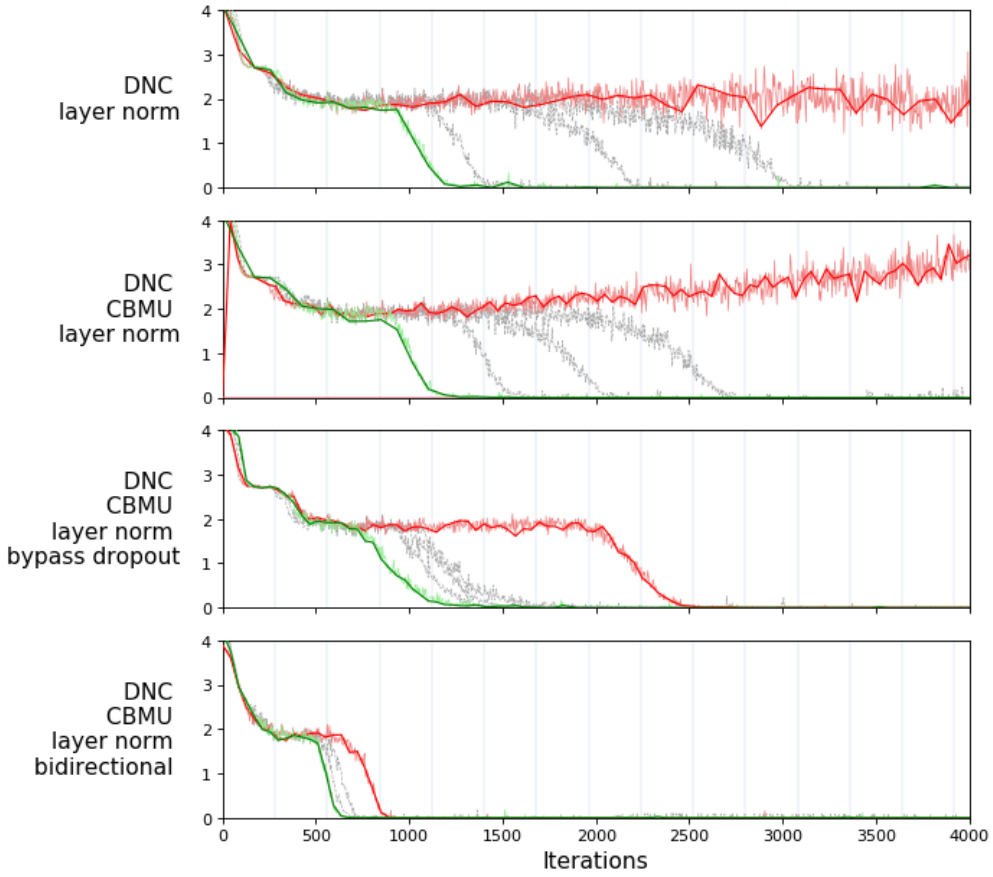


Figure 7.4: Comparison of the cross entropy loss of the validation set between the DNC with CBMU and different advancements on the bAbI task 1.

This experiment shows that the usage of the CBMU leads to a significant reduction in memory consumption. It also decreases the computation time but has no evident performance loss. The DNC with LN and an advancement have nearly the same performance improvements due to the advancements independent from the used memory unit. This could indicate that the temporal linkage mechanism has no significant benefit in a QA setting.

7.2 bAbI 20 Task

The DNC is evaluated in the original paper on the bAbI 20 task [1]. To measure the impact of the advancements in this work three advanced models are evaluated on the full bAbI 20 task. The task is described in detail in Section 3.2.

The models are the DNC with layer normalization, bypass dropout and the CBMU (ADNC), the bidirectional ADNC (BiADNC) and a BiDNC with data augmentation (BiADNC +aug16). All tasks are trained jointly on all 20 bAbI tasks and there is no information provided about the actual task. In the following, the augmentation of task 16 is described and afterwards the training details and the results.

7.2.1 Task 16 augmentation

Many related models struggle with task 16 in the bAbI set and only achieve word error rates about 50% [1, 61, 66]. The task is to find a colour regarding a name and

other name-animal-colour constellations. In the dataset are four different colours, four animals and five names but not equally distributed in each sample. In many samples one or more colours or animals are multiple times present, see Sample 7.1:

Task 16: Basic Induction - Sample 7.1

Greg is a lion. Julius is a swan. Julius is yellow. Greg is yellow. Brian is a swan. Bernhard is a frog. Brian is white. Lily is a frog. Bernhard is yellow. What color is Lily? ***yellow***

In the training set are 10k samples but in merely 4181 cases appears the correct word only once in the context. In all other cases, the correct word appears two, three or even four times in the context. When the correct word appears two times, the other two possible colours are equal in the 1370 samples and different in 1427. If the model learns only to count the colour words and answer the colour who appears multiple time, it would be in 3449 cases correct and in 1370 with a 50/50 probability. In 4181 cases it is able to guess and has a 1/4 probability to be correct. This leads to an overall probability for a correct answer by guessing and counting words of about 52%. This is a strong local optimum and makes it hard for the model to find a better solution strategy.

We provide an augmentation of task 16 that each sample contains different colours and different animals. Only the training data is modified and after convergence, it gets refined. In the refinement, each second batch is from the task 16 without any augmentation to learn to original distribution of data.

7.2.2 Training details

For a direct comparison, the hyper-parameters are orientated on the original paper. The unidirectional controllers have a one LSTM layer and 256 hidden units, in the bidirectional setup, the forward and backward LSTMs have 172 hidden units each. Thus both models have nearly the same amount of parameters, about 891k weights. The memory unit of all three models has 192 slots, a width of 64 and 4 read heads.

Each word of the bAbI task is encoded as a one-hot vector with the size of the vocabulary (159). An additional sequence mask is used to only generate training signals when an answer is requested, so the outputs of all other time steps are ignored. The output is a vector of the vocabulary size and normalized with a softmax function. For training, the cross-entropy loss between the prediction vector and the target one-hot vector is minimized. The loss metric for the bAbI task is the word error rate (WER), the fraction of incorrectly answered words to all requested words.

Each training uses mini-batches with a batch size of 32. The different long sequences get padded. The maximum sequence length during training is limited to 800 words. The optimizer was found by a grid search, the underlined options are used: optimization algorithm [SGD, Adam, RMSprop] with a learning rate of [1e-05, 3e-05, 1e-04], momentum of 0.9 and bypass dropout rate [10%, 20%, 30%], [30, 112].

Each model runs five experiments with different initializations. Each model runs 50 epochs and the epoch with the best WER on the validation set is used for the evaluation. No early stopping, no learning rate decay and no gradient noise are used.

	DNC	ADNC	BiADNC
Parameters	894,244	891,136	891,232
Avg. Time per Epoch	2.6h	1.3h	1.6h
Avg. Memory Consumption	15.5GB	4.3GB	4.3GB

Table 7.2: The resources of the different models evaluated on the bAbI 20 task with a batch size of 32.

The models are trained on a Nvidia K80 GPU. The parameter amount, average training time per epoch and average memory consumption can be found in Table 7.2. This shows that the memory consumption savings of the CBMU are over 70% in comparison with a vanilla MU. It also saves half of the computing time. The bidirectional setup needs about 25% more computing time per epoch than the unidirectional model. A model converges after 30-40 epochs and a training time of 1-3 days.

7.2.3 Results

Table 7.3 shows the mean word error rate and variance on the test set from all models of this work in comparison to the original paper (DNC) [1], the EntNet and the SDNC, the best model with reported mean results as far as I know [102]. The bottom row shows how many tasks failed, which means they have a WER above 5%.

Task	DNC	EntNet	SDNC	ADNC	BiADNC	BiADNC +aug 16
1: 1 supporting fact	9.0 ± 12.6	0.0 ± 0.1	**0.0 ± 0.0**	0.1 ± 0.0	0.1 ± 0.1	0.1 ± 0.0
2: 2 supporting facts	39.2 ± 20.5	15.3 ± 15.7	7.1 ± 14.6	0.8 ± 0.5	0.8 ± 0.2	**0.5 ± 0.2**
3: 3 supporting facts	39.6 ± 16.4	29.3 ± 26.3	9.4 ± 16.7	6.5 ± 4.6	2.4 ± 0.6	**1.6 ± 0.8**
4: 2 argument relations	0.4 ± 0.7	0.1 ± 0.1	0.1 ± 0.1	0.0 ± 0.0	**0.0 ± 0.0**	**0.0 ± 0.0**
5: 3 argument relations	1.5 ± 1.0	**0.4 ± 0.3**	0.9 ± 0.3	1.0 ± 0.4	0.7 ± 0.1	0.8 ± 0.4
6: yes/no questions	6.9 ± 7.5	0.6 ± 0.8	0.1 ± 0.2	0.0 ± 0.1	**0.0 ± 0.0**	**0.0 ± 0.0**
7: counting	9.8 ± 7.0	1.8 ± 1.1	1.6 ± 0.9	**1.0 ± 0.7**	1.0 ± 0.5	1.0 ± 0.7
8: lists/sets	5.5 ± 5.9	1.5 ± 1.2	0.5 ± 0.4	**0.2 ± 0.2**	0.5 ± 0.3	0.6 ± 0.3
9: simple negation	7.7 ± 8.3	0.0 ± 0.1	0.0 ± 0.1	**0.0 ± 0.0**	0.1 ± 0.2	**0.0 ± 0.0**
10: indefinite knowledge	9.6 ± 11.4	0.1 ± 0.2	0.3 ± 0.2	0.1 ± 0.2	**0.0 ± 0.0**	0.0 ± 0.1
11: basic coreference	3.3 ± 5.7	0.2 ± 0.2	**0.0 ± 0.0**	**0.0 ± 0.0**	**0.0 ± 0.0**	**0.0 ± 0.0**
12: conjunction	5 ± 6.3	**0.0 ± 0.0**	0.2 ± 0.3	**0.0 ± 0.0**	0.0 ± 0.1	**0.0 ± 0.0**
13: compound coreference	3.1 ± 3.6	0.0 ± 0.1	0.1 ± 0.1	**0.0 ± 0.0**	**0.0 ± 0.0**	**0.0 ± 0.0**
14: time reasoning	11 ± 7.5	7.3 ± 4.5	5.6 ± 2.9	**0.2 ± 0.1**	0.8 ± 0.7	0.3 ± 0.1
15: basic deduction	27.2 ± 20.1	3.6 ± 8.1	3.6 ± 10.3	**0.1 ± 0.1**	**0.1 ± 0.1**	**0.1 ± 0.1**
16: basic induction	53.6 ± 1.9	53.3 ± 1.2	53.0 ± 1.3	52.1 ± 0.9	52.6 ± 1.6	**0.0 ± 0.0**
17: positional reasoning	32.4 ± 8	8.8 ± 3.8	12.4 ± 5.9	18.5 ± 8.8	4.8 ± 4.8	**1.5 ± 1.8**
18: size reasoning	4.2 ± 1.8	1.3 ± 0.9	1.6 ± 1.1	1.1 ± 0.5	**0.4 ± 0.4**	0.9 ± 0.5
19: path finding	64.6 ± 37.4	70.4 ± 6.1	30.8 ± 24.2	43.3 ± 36.7	**0.0 ± 0.0**	0.1 ± 0.1
20: agent's motivation	0.0 ± 0.1	**0.0 ± 0.0**	**0.0 ± 0.0**	0.1 ± 0.1	0.1 ± 0.1	0.1 ± 0.1
Mean Error:	16.7 ± 7.6	9.7 ± 2.6	6.4 ± 2.5	6.3 ± 2.7	3.2 ± 0.5	**0.4 ± 0.3**
Failed Tasks (>5%):	11.2 ± 5.4	5.0 ± 1.2	4.1 ± 1.6	3.2 ± 0.8	1.4 ± 0.5	**0.0 ± 0.0**

Table 7.3: The mean word error rate of the 20 bAbI tasks of the different models. All models are trained jointly on all 20 bAbI tasks at once without information about the actual task. Best results in bold.

The ADNC outperforms the DNC from the original paper as well as all other joint trained models. This shows the impact of the normalization and the bypass dropout

which improves the model performance even without the temporal memory linkage mechanism. This could imply that this mechanism is not really important for QA tasks. It also leads to a significant drop in variance which could mean that our models are more robust.

The additional model complexity through the bidirectional design shows clear improvements without more parameters. It outperforms the DNC in terms of mean error as well as variance. The lower variance indicated a very robust model for different random initializations. But through the bidirectional controller, the model has access to information about the question during reading the context similar to the RMN or DMN+. Particularly the performance on task 3, 17 and 19 reaches a new quality in the mean results without any failed tasks.

The BiADNC with augmentation of task 16 leads to the best-reported overall results as far as I know. Trough the modifications the task is learned correctly and is completely solvable. Even when ignoring the task 16 in the results, the performance of this model is better. This could indicate that the correct task has a positive effect on the learning of the other tasks. The BiADNC has also a quiet lower variance due to a more stable and consistent convergence behaviour as reported in Section 7.1.2.

Task	DNC	EntNet	EntNet †	DMN+ †	SDNC	RMN	ADNC	BiADNC	BiADNC +aug 16
1: 1 supporting fact	**0.0**	0.1	**0.0**	**0.0**	**0.0**	**0.0**	0.1	0.1	0.1
2: 2 supporting facts	0.4	2.8	**0.1**	0.3	0.6	0.5	0.8	0.5	0.6
3: 3 supporting facts	1.8	10.6	4.1	1.1	**0.7**	14.7	2.5	2.5	1.6
4: 2 argument relations	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
5: 3 argument relations	0.8	0.4	**0.3**	0.5	**0.3**	0.4	1.6	0.7	0.4
6: yes/no questions	**0.0**	0.3	0.2	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
7: counting	0.6	0.8	**0.0**	2.4	0.2	0.5	1.5	0.3	0.6
8: lists/sets	0.3	0.1	0.5	**0.0**	0.2	0.3	0.1	0.4	0.6
9: simple negation	0.2	**0.0**	0.1	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
10: indefinite knowledge	0.2	**0.0**	0.6	**0.0**	0.2	**0.0**	**0.0**	**0.0**	**0.0**
11: basic coreference	**0.0**	**0.0**	0.3	**0.0**	**0.0**	0.5	**0.0**	**0.0**	**0.0**
12: conjunction	**0.0**	**0.0**	**0.0**	**0.0**	0.1	**0.0**	**0.0**	**0.0**	**0.0**
13: compound coreference	**0.0**	**0.0**	1.3	**0.0**	0.1	**0.0**	**0.0**	**0.0**	**0.0**
14: time reasoning	0.4	3.6	**0.0**	0.2	0.1	**0.0**	0.1	0.1	0.5
15: basic deduction	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	0.1	0.2	**0.0**
16: basic induction	55.1	52.1	0.2	45.3	54.1	0.9	52.0	49.9	**0.0**
17: positional reasoning	12.0	11.7	0.5	4.2	0.3	0.3	11.1	0.8	**0.2**
18: size reasoning	0.8	2.1	0.3	2.1	**0.1**	2.3	1.6	1.0	0.9
19: path finding	3.9	63.0	2.3	0.0	1.2	2.9	0.8	**0.0**	0.3
20: agent's motivation	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	0.2	0.1	**0.0**
Mean Error:	3.8	7.4	0.5	2.8	2.9	1.2	3.6	2.8	**0.3**
Failed Tasks (>5%):	2	4	0	1	1	1	2	1	**0**

Table 7.4: The word error rate of the best runs on the bAbI 20 task. Best results per row in bold. Models tagged with † are trained on each task individually, the others are trained on all tasks jointly.

Table 7.4 shows the WER of the best runs with additional comparison to the RMN, the best-reported model in the literature as far as I know. The table also contains an individual comparison to the EntNet and DMN+ which are trained on each task individually and therefore reach the best overall reported result. The bottom row shows how many tasks failed, which means they have a word error rate above 5%. The results of the best models per task are bold in the tables. Here the ADNC outperform the DNC as well. Particularly the performance on task 3, 17 and 19

reaches a new quality. The BiADNC does not outperform the RMN due to the task 16 issue. But with the augmentation of task 16, the BiADNC reaches a new state-of-the-art result and outperforms even the individually trained models.

7.3 Children Book Test

To assess the DNC performance on a large-scale QA task it is evaluated on the Children Book Task (CBT). The task is described in detail in Section 3.3 and the training details and results are in the following sections. One model is evaluated, a bidirectional DNC with DNC normalization, bypass dropout and a CBMU (Bi-ADNC).

7.3.1 Training details

All hyper-parameters are chosen based on the experience with the bAbI 20 task and not by grid search or similar due to limited computational resources.

The input sequence contains the context and the query which is the 21st sentence. It is fed word for word into the model and each word is represented as an one-hot vector. The missing word is represented as a question token. The CBMU model has a preceding word vector with an embedding size of 100. The word vector is initialized with a 6B GloVe word representation. The GloVe vector itself is pre-trained on a vocabulary of 400,000 words from Wikipedia 2014 and the Gigaword 5 Corpus [113]. Pre-trained GloVe word vectors are freely available[1]. There are 16,627 out-of-the-vocabulary words which are initialized with a uniform distributed vector within the interval $[-1, 1)$. During training, the word representation is a trainable parameter and gets optimized.

The model itself has two LSTM controllers, each with one hidden layer, a size of 172 and layer normalization. The BiADNC has a memory matrix with a length of 192, a width of 128 and four read heads are used. Additionally, bypass dropout is applied with a dropout rate of 10%, similar to the bAbI experiments.

The model's output is a one-hot word vector with a size of 10696. This represents all possible answer words. The knowledge of the ten candidate words is used to mask all impossible words in the output vector. The masked output vector is activated with a softmax activation function. Therefore, only the relevant words get updated and all others are ignored. A second mask ignores all outputs which are not requested answers. This results in a sparse but computing efficient gradient update. During training, the cross-entropy loss between the predicted answer vector and the target one-hot vector is minimized. The model is trained with mini-batch SGD with a batch size of 48. The model is optimized with RMSprop, a fixed learning rate of 3e-05 and a momentum of 0.9. During training, sequences longer than 800 are skipped but not during validation and testing. The experiments use early stopping regarding the cross-entropy validation loss.

The model is trained on a Nvidia Tesla P100. It has a training time of about 6h per epoch and requires 15.5GB of memory on the GPU. It converges after 15 epochs and an overall training time of 4 days. A model with the same hyper-parameters but a classic DNC memory unit would require over 60% more memory.

[1] https://nlp.stanford.edu/projects/glove/ (28.02.2018)

7.3.2 Results

The results of the CBT experiment are reported in Table 7.5. It shows the accuracy of the BiADNC in comparison to humans, an LSTM and the MemNN reported in [57] and the best model to my knowledge in literature so far, the GA reader [86]. The ADNC outperforms the LSTM but not the MemNN nor the GA Reader. The reported LSTM has 512 hidden units. Thus, the memory augmentation causes a performance gain. The difference to the MemNN could either be a conceptual drawback or caused by the lack of hyper-parameter tuning.

Children Book Test - Accuracy on Test Set					
CBT Test Set	Humans	LSTM	MemNN	GA Reader	BiADNC
Common Nouns	81.6	56.0	63.0	**70.7**	57.3
Named Entities	81.6	41.8	66.6	**74.9**	59.0

Table 7.5: The resulting accuracy on the children book test with the model of this work in comparison with models in literature.

The results demonstrate that the model is able to outperform the LSTM without any task-specific adoptions and is trainable in a feasible time due to the CBMU .

7.4 CNN Reading Comprehension Task

A second assessment of the DNC's performance on large-scale QA task is based on the CNN reading comprehension test. The task is described in detail in Section 3.4 and the training details and results are presented in the following sections. Two models are evaluated, an advanced DNC (ADNC) and a bidirectional ADNC (BiADNC).

7.4.1 Training details

The query and the article are concatenated (the query first) as an input sequence. The model is fed the sequence word by word and each word is represented as a word vector with size 100 and GloVe initialization [113]. The word vector is optimized during training. The target is the correct word represented as the index of a one-hot vector with the size of all name entity tokens. A candidate mask is created which masks out all name entity tokens that are not present in the sample. The last model output predicts the word and is normalized with a softmax function. The cross-entropy loss between the prediction output vector and the target one-hot vector is minimized.

We use mini-batches with a batch size of 32; the different long sequences get padded from begin of the sequence. Two models are evaluated on this task, the ADNC and a BiADNC. All hyper-parameters are chosen based on related work. The controller is a LSTM with one hidden layer and a hidden unit size of 512 in the unidirectional model. In the bidirectional model, both controllers have a size of 384 each. Both models use DNC normalization and have a memory matrix with 256 slots, a width of 128 and four read heads. Additionally, bypass dropout is applied with a dropout

rate of 10%. The maximum sequence length during training is limited to 1400 words. The model is optimized with RMSprop with use of a fixed learning rate of 3e-05 and a momentum of 0.9. The loss metric for the CNN is the accuracy; the fraction of all samples with correct words compared to the number of samples in total.

The model is trained on a Nvidia Tesla P100. It has a training time of about 18h per epoch and requires 12.4GB of memory on the GPU.It converges after 12 epochs and an overall training time of 9 days.

7.4.2 Results

The results on the CNN RC task are shown in Table 7.6. The table contains the accurary of the ADNC and BiADNC models in comparison to the Deep LSTM Reader [58], the Attentive Reader [58], the MemNN [57], the AS reader [82], the Stanford AR [81], the Iterative Attention Reader [84], the EpiReader [87], the AoA Reader [83], the ReasoNet [85] and the best model in literature, the GA Reader [86].

Both models are used without any adaption to the task, hyper-parameter tuning nor any other optimization competitive results. The bidirectional model outperforms the unidirectional ADNC, but only on the test set. In contrast to the CBT task, the ADNC outperforms the LSTM and the MemNN. Again, the difference between better models could be due to a weak model structure or the lack of hyper-parameter tuning.

CNN RC Dataset - Accuracy		
Model	valid	test
Deep LSTM Reader	55.0	57.0
Attentive Reader	61.6	63.0
MemNets	63.4	66.8
ADNC	67.5	69.0
AS Reader	68.6	69.5
BiADNC	67.1	69.8
Stanford AR	72.2	72.4
Iterative Attention Reader	72.6	73.3
EpiReader	73.4	74.0
AoA Reader	73.1	74.4
ReasoNet	72.9	74.7
GA Reader	77.9	77.9

Table 7.6: The validation and test accuracy (%) of the ADNC/BiADNC and other models from literature on CNN RC task.

This experiment supports the results of the CBT experiment. The ADNC is able to provide competitive results on a large-scale QA task without any task-specific adoptions, optimization or hyper-parameter tuning. This makes the ADNC an interesting candidate for a wide range of further applications. Its easy adaptability to other tasks and the memorization mechanisms seem flexible enough for a variety of use cases.

7.5 Results overview

The previous experiments show that the advancements to the DNC introduced in Chapter 6 improve its performance. Table 7.7 provides an overview of the three evaluated datasets and the performance of models reported in literature compared to the model in this work, the BiADNC.

Results Summary			
Task	bAbI 20 Task*	CBT CN	CNN RC Dataset
Human	0.0[0]	81.6[3]	-
LSTM	27.3[1]	56.0[3]	57.0[5]
DNC	16.7[1]	-	-
Best in Literature	6.4[2]	**70.7**[4]	**77.9**[4]
This work / BiADNC	**0.4**	57.3	69.8
Metric	WER ↓	Accuracy ↑	Accuracy ↑

Table 7.7: Summary of the results on different datasets. *The models are jointly trained on all bAbI 20 tasks and the used metric is the mean word error rate (WER). Results marked with [0] are taken from [56], [1] are taken from [1], [2] are taken from [102], [3] are taken from [57], [4] are taken from [86], and [5] are taken from [58].

In the bAbI 20 task, the BiADNC clearly outperforms all reported works so far and provides new state-of-the-art results. This is mainly driven by the DNC normalization and the bidirectional architecture. The augmentation of task 16 overcomes the local minima of this task and allows the model to solve all tasks successfully.

The results of the two experiments on the large-scale datasets (CBT and CNN RC task) are quite different. On the one side, the performance on the CNN RC task is competitive without any hyper-parameter tuning. On the other side, there is only a slight improvement on the CBT dataset compared to the LSTM. This could have multiple reasons. It may be that the model is weak but this stands in a strong contrast to the CNN RC results. More likely is a poor experiment setup. While many related works use sentence representation, word windows or similar techniques to deal with the huge vocabulary or the long context length, the ADNC uses only a word vector representation and a word-wise input feed. Furthermore, this experiment lacks on hyper-parameter tuning too. All parameters in the large-scale QA tasks are inspired by related works and chosen buy best guess due to the lack of computational resources.

In further work, a more sophisticated CBT experiment could be designed or different word representations techniques could be evaluated. The next chapter concludes this work and provides additional ideas for further works.

8. Conclusion

In the conclusion, the achievements of this work are summarized and the gained insights are discussed. The last section points out possible further works.

8.1 Summary

The aim of this work is to analyze and advance the DNC, a memory-augmented neural network, to apply it to large-scale QA tasks. After an overview of the basics, the used data and the related work, this work analyzes the DNC in Chapter 5, advance it in Chapter 6 and evaluates the advancements in Chapter 7.

The analysis of the training in Section 5.1 shows that the model performance strongly depends on the weights initialization. Additionally, the training success depends on how fast the model learns to use the internal memory matrix. The functional analysis in Section 5.2 shows that in a synthetic bAbI QA task the temporal linkage mechanism has only a minimal influence on the predicted answer. However, the analysis of the computing resources in Section 5.3 shows that the temporal linkage mechanism is the main driver of the memory consumption. This could be the key issue why vanilla DNCs are not applied to large-scale QA tasks. These tasks contain much longer sequences and have a huge vocabulary compared to synthetic datasets. The DNC would need a larger memory matrix. This and the longer sequences massively increase the memory consumption and the training on a common GPU would take weeks or even months.

The methods in Chapter 6 describe several advancements to speed up training, increase robustness of the model and, therefore, gain performance. For a more stable training, DNC normalization is introduced based on layer normalization. For an earlier memory usage, bypass dropout is introduced. It is a regularization mechanism to decrease the controller influence on the output signal to force faster memory usage. Both advancements are empirically evaluated in Section 7.1, which shows that they improve the convergence behaviour, lead to a consistent training process and force an earlier memory usage.

To increase the DNC performance in general two novel model setups are introduced, a bidirectional controller and an additional RNN at the model's output. While

the additional output RNN does not provide a significant benefit, the bidirectional controller does. It leads to the best performance during evaluation.

Additionally, Section 6.3 introduces a novel memory unit. It tackles the issue of high memory consumption and long training times in large-scale QA tasks. The new memory unit removes the temporal linkage mechanism based on the circumstance that the mechanism is not in use in a QA application. The exclusive usage of the content-based memory addressing provides still a good performance although it only requires a fraction of memory and needs less computing time as shown in Section 7.1.

In the benchmarking in Chapter 7, on the synthetic bAbI QA task, a model with the bidirectional controller, DNC normalization, bypass dropout and the novel content-based memory unit, named bidirectional advanced DNC (BiADNC), outperforms the best-reported model in the literature to the best of my knowledge. With an additional task-specific training augmentation the bAbI 20 task is solved completely. Therefore, this work is the first to solve all tasks in a joint training.

The portability to large-scales QA tasks is evaluated on two datasets, the "Children Book Test" (CBT) and the CNN reading comprehension (RC) task. Without hyper-parameter tuning and further task-specific adoptions, the model outperforms an LSTM-baseline. In case of the CNN RC task, the ADNC reaches competitive results compared to recent specialized works. These experiments show that the ADNC is able to solve large-scale QA tasks without task-specific adaption.

8.2 Discussion

In contrast to an RNN or LSTM, the memory-augmentation of a DNC allows the explicit storage of complex data structures over a long time-scale. Furthermore, it allows to manipulate the stored information and to read from it. This is achieved by a division of computation and storage with the use of a controller and a memory matrix. This has several benefits.

The separation allows to store information directly and not in an implicit way like in an ANN or RNN. This prohibits catastrophic forgetting and allows the explicit storage of word embeddings in the memory in case of NLP applications. If the model stores information from multiple sentences in memory, it can reason about a whole paragraph easily since it is represented in its memory. Hence it allows a better handling of long-term dependencies. Furthermore, the separation allows a more detailed design of the model's hyper-parameters. The controller and the memory unit can be independently improved and adapted.

The DNC setup moves away from the idea of an artificial neural network as a counterpart of a natural neural network. It is less biologically plausible and explores new avenues with more engineering. The original DNC paper has a rough biological motivation but this work comes from a more technical perspective.

The advancements in this work increase the usability of the DNC. The new ADNC is training more stable, more robust to apply to complex tasks and less computationally costly. Therefore, it is easier to apply and can benefit a wider range of applications. This allows to apply memory-augmentation without major effort.

With the use of the more lightweight content-based memory unit, the ADNC can be adjusted to large-scale QA tasks in order to reach convenient training times. This makes the model applicable to a wider variety of large applications by reducing the amount of required computing resources. Another solution could be a sparse DNC implementation as described in [102], which also saves memory resources and speeds up the training progress.

The good performance of the bidirectional model on the bAbI task presumably arises from the fact that the model has information about the question right from the beginning of the sample sequence. This allows a better attention to content which should be stored. Additionally, the question can be in any position and refer to the past and future content due to the bidirectional setup. A unidirectional model is more general in real-time applications since it can be applied in a continuous manner without the information of the whole input sequence. This is important in a dialogue setup or when such a model should be used on the fly. But even in a uniform setup the ADNC outperforms related models and is a promising architecture for further work.

In contrast to recent related models described in Chapter 4, the ADNC does not require task-specific adaption, satisfies the requirement to a general model and processes a text word-by-word. This is shown with the successful application of the ADNC to large-scale QA tasks.

In the ADNC setup, the memory unit can be interpreted as an interdependent layer similar to an additional network layer. This makes it easy to add it to an existing model and needs no complete alignment of the model to get a memory extension. This ability of easy adaption makes the ADNC interesting for further NLP applications like machine translation.

8.3 Further work

There are several possibilities to improve the performance of the introduced models. First of all, each task needs its own hyper-parameter tuning. Either with a grid search or with use of a more sophisticated method. Promising approaches are Bayesian optimization [114] or the learning to learn method [115].

The amount of computing resources could be reduced via a lightweight controller with use of gated recurrent units (GRU) or with use of a sparse memory unit as introduced in [102].

A way to improve the training might be curriculum learning [116]. This may accelerate learning and force faster memory usage. Another feature of the ADNC's memory unit is that the trainable parameters do not depend on memory size. This allows the memory to grow with the task during training or usage. Possible application scenarios are lifelong learning or continuous learning [117].

Due to the modular usage of the memory unit an architecture with multiple memories or a hierarchical structure could be possible. This would allow the storage of large data in an organized manner. But not only large memories have advantages. A small memory matrix could cheaply increase the memorization performance in many models for example in machine translation in order to store context information. Furthermore, a pre-trained memory unit could be used to adapted to new tasks.

In general, the ADNC could be applied to more NLP tasks, for example with the use of the ParlAI platform for evaluating systems on multiple datasets [118]. Of special interest is the field of dialogue modelling where memory is important to remember earlier statements or provide knowledge-bases on pre-learned memory units. This requires an extension of the fixed sequence learning model from the QA tasks to a sequence to sequence model. Another interesting application field is meta-learning where a content-based memory unit could also reduce training effort and make DNCs more usable.

All of these ideas show that the ADNC could play an important role in the future. This work makes the DNC more practical, more robust and therefore easier to apply. This allows completely novel applications in which a large amount of information needs to be stored over a long time period or an explicit data representation is essential.

Bibliography

[1] 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*, vol. 538, no. 7626, pp. 471–476, 2016.

[2] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, no. 236, pp. 433–460, 1950.

[3] R. F. Simmons, "Natural language question-answering systems: 1969," *Communications of the ACM*, vol. 13, no. 1, pp. 15–30, 1970.

[4] A. Mishra and S. K. Jain, "A survey on question answering systems with classification," *Journal of King Saud University-Computer and Information Sciences*, vol. 28, no. 3, pp. 345–361, 2016.

[5] A. Kumar, O. Irsoy, P. Ondruska, M. Iyyer, J. Bradbury, I. Gulrajani, V. Zhong, R. Paulus, and R. Socher, "Ask me anything: Dynamic memory networks for natural language processing," in *International Conference on Machine Learning*, pp. 1378–1387, 2016.

[6] J. Weizenbaum, "Eliza—a computer program for the study of natural language communication between man and machine," *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, 1966.

[7] T. Winograd, "Procedures as a representation for data in a computer program for understanding natural language," tech. rep., Massachusetts Inst. of Tech. Cambridge Project MAC, 1971.

[8] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern classification*. John Wiley & Sons, 2012.

[9] D. P. Mandic, J. A. Chambers, *et al.*, *Recurrent neural networks for prediction: learning algorithms, architectures and stability*. Wiley Online Library, 2001.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[12] H. T. Siegelmann and E. D. Sontag, "On the computational power of neural nets," *Journal of computer and system sciences*, vol. 50, no. 1, pp. 132–150, 1995.

[13] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," *arXiv preprint arXiv:1506.00019*, 2015.

[14] P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural networks*, vol. 1, no. 4, pp. 339–356, 1988.

[15] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in cognitive sciences*, vol. 3, no. 4, pp. 128–135, 1999.

[16] B. Lüders, M. Schläger, A. Korach, and S. Risi, "Continual and one-shot learning through neural networks with dynamic external memory," in *European Conference on the Applications of Evolutionary Computation*, pp. 886–901, Springer, 2017.

[17] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap, "Meta-learning with memory-augmented neural networks," in *International conference on machine learning*, pp. 1842–1850, 2016.

[18] W. Zaremba and I. Sutskever, "Reinforcement learning neural turing machines," *arXiv preprint arXiv:1505.00521*, vol. 419, 2015.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[20] D. Kriesel, *A Brief Introduction to Neural Networks*. 2007.

[21] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[22] F. Rosenblatt, "Principles of neurodynamics. perceptrons and the theory of brain mechanisms," tech. rep., Cornell Aeronautical Lab INC Buffalo NY, 1961.

[23] B. Widrow and M. E. Hoff, "Adaptive switching circuits," tech. rep., Stanford Univ CA Stanford Electroninics Labs, 1960.

[24] P. Werbos and P. J. (Paul John, "Beyond regression : new tools for prediction and analysis in the behavioral sciences /," 01 1974.

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[26] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang, "Phoneme recognition using time-delay neural networks," *IEEE transactions on acoustics, speech, and signal processing*, vol. 37, no. 3, pp. 328–339, 1989.

[27] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[28] K. S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Transactions on neural networks*, vol. 1, no. 1, pp. 4–27, 1990.

[29] Y. Bengio, I. J. Goodfellow, and A. Courville, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[30] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[31] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[32] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning," *Genetic Programming and Evolvable Machines*, Oct 2017.

[33] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks," in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 1764–1772, 2014.

[34] J. Franke, M. Mueller, F. Hamlaoui, S. Stueker, and A. Waibel, "Phoneme boundary detection using deep bidirectional lstms," in *Speech Communication; 12. ITG Symposium*, pp. 1–5, Oct 2016.

[35] S. Liu, N. Yang, M. Li, and M. Zhou, "A recursive recurrent neural network for statistical machine translation," in *Proceedings of ACL*, pp. 1491–1500, 2014.

[36] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," *ArXiv Prepr. ArXiv14122306*, 2014.

[37] W.-C. Cheng, J.-C. Huang, and C.-Y. Liou, "Segmentation of DNA using simple recurrent neural network," *Knowledge-Based Systems*, vol. 26, pp. 271–280, Feb. 2012.

[38] C. Olah, "Understanding LSTM Networks." http://colah.github.io/posts/2015-08-Understanding-LSTMs/, Dec. 2015. Available at: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[39] M. I. Jordan, "Attractor dynamics and parallellism in a connectionist sequential machine," 1986.

[40] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, pp. 179–211, Apr. 1990.

[41] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proc. IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

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

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

[44] H. T. Siegelmann and E. D. Sontag, "Turing computability with neural nets," *Applied Mathematics Letters*, vol. 4, no. 6, pp. 77–80, 1991.

[45] Y. Bengio, "Markovian models for sequential data," *Neural Comput. Surv.*, vol. 2, pp. 129–162, 1999.

[46] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd international conference on Machine learning*, pp. 369–376, ACM, 2006.

[47] A. Graves and others, *Supervised sequence labelling with recurrent neural networks*, vol. 385. Springer, 2012.

[48] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[49] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *ArXiv150304069 Cs*, Mar. 2015.

[50] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *ArXiv Prepr. ArXiv14061078*, 2014.

[51] W. Zaremba, "An empirical exploration of recurrent network architectures," 2015.

[52] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 2342–2350, 2015.

[53] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[54] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural Networks*, vol. 18, no. 5, pp. 602–610, 2005.

[55] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[56] 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," *ICLR*, 2016.

[57] F. Hill, A. Bordes, S. Chopra, and J. Weston, "The goldilocks principle: Reading children's books with explicit memory representations," *ICLR*, 2016.

[58] K. M. Hermann, T. Kocisky, E. Grefenstette, L. Espeholt, W. Kay, M. Suleyman, and P. Blunsom, "Teaching machines to read and comprehend," in *Advances in Neural Information Processing Systems*, pp. 1693–1701, 2015.

[59] J. Weston, S. Chopra, and A. Bordes, "Memory networks," *ICLR*, 2015.

[60] A. Bordes, N. Usunier, S. Chopra, and J. Weston, "Large-scale simple question answering with memory networks," *arXiv preprint arXiv:1506.02075*, 2015.

[61] S. Sukhbaatar, J. Weston, R. Fergus, *et al.*, "End-to-end memory networks," in *Advances in neural information processing systems*, pp. 2440–2448, 2015.

[62] S. Chandar, S. Ahn, H. Larochelle, P. Vincent, G. Tesauro, and Y. Bengio, "Hierarchical memory networks," *arXiv preprint arXiv:1605.07427*, 2016.

[63] A. Miller, A. Fisch, J. Dodge, A.-H. Karimi, A. Bordes, and J. Weston, "Key-value memory networks for directly reading documents," *arXiv preprint arXiv:1606.03126*, 2016.

[64] F. Ma, R. Chitta, S. Kataria, J. Zhou, P. Ramesh, T. Sun, and J. Gao, "Long-term memory networks for question answering," *arXiv preprint arXiv:1707.01961*, 2017.

[65] J. Perez and F. Liu, "Gated end-to-end memory networks," *arXiv preprint arXiv:1610.04211*, 2016.

[66] C. Xiong, S. Merity, and R. Socher, "Dynamic memory networks for visual and textual question answering," in *International Conference on Machine Learning*, pp. 2397–2406, 2016.

[67] A. Bordes and J. Weston, "Learning end-to-end goal-oriented dialog," *arXiv preprint arXiv:1605.07683*, 2016.

[68] Y. Feng, S. Zhang, A. Zhang, D. Wang, and A. Abel, "Memory-augmented neural machine translation," *arXiv preprint arXiv:1708.02005*, 2017.

[69] C. Xiong, S. Merity, and R. Socher, "Dynamic memory networks for visual and textual question answering," *arXiv*, vol. 1603, 2016.

[70] M. Tapaswi, Y. Zhu, R. Stiefelhagen, A. Torralba, R. Urtasun, and S. Fidler, "Movieqa: Understanding stories in movies through question-answering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4631–4640, 2016.

[71] C. Ma, C. Shen, A. Dick, and A. v. d. Hengel, "Visual question answering with memory-augmented networks," *arXiv preprint arXiv:1707.04968*, 2017.

[72] S. Na, S. Lee, J. Kim, and G. Kim, "A read-write memory network for movie story understanding," *arXiv preprint arXiv:1709.09345*, 2017.

[73] K.-M. Kim, M.-O. Heo, S.-H. Choi, and B.-T. Zhang, "Deepstory: video story qa by deep embedded memory networks," *arXiv preprint arXiv:1707.00836*, 2017.

[74] H. Yang, S. Cho, *et al.*, "Finding remo (related memory object): A simple neural architecture for text based reasoning," *arXiv preprint arXiv:1801.08459*, 2018.

[75] M. Henaff, J. Weston, A. Szlam, A. Bordes, and Y. LeCun, "Tracking the world state with recurrent entity networks," *ICLR*, 2017.

[76] E. Grefenstette, K. M. Hermann, M. Suleyman, and P. Blunsom, "Learning to transduce with unbounded memory," in *Advances in Neural Information Processing Systems*, pp. 1828–1836, 2015.

[77] T. Munkhdalai and H. Yu, "Neural semantic encoders," in *Proceedings of the conference. Association for Computational Linguistics. Meeting*, vol. 1, p. 397, NIH Public Access, 2017.

[78] T. Munkhdalai and H. Yu, "Reasoning with memory augmented neural networks for language comprehension," *arXiv preprint arXiv:1610.06454*, 2016.

[79] A. Madotto and G. Attardi, "Question dependent recurrent entity network for question answering," *arXiv preprint arXiv:1707.07922*, 2017.

[80] T. Bansal, A. Neelakantan, and A. McCallum, "Relnet: End-to-end modeling of entities & relations," *arXiv preprint arXiv:1706.07179*, 2017.

[81] D. Chen, J. Bolton, and C. D. Manning, "A thorough examination of the cnn/daily mail reading comprehension task," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, pp. 2358–2367, 2016.

[82] R. Kadlec, M. Schmid, O. Bajgar, and J. Kleindienst, "Text understanding with the attention sum reader network," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, pp. 908–918, 2016.

[83] Y. Cui, Z. Chen, S. Wei, S. Wang, T. Liu, and G. Hu, "Attention-over-attention neural networks for reading comprehension," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, pp. 593–602, 2017.

[84] A. Sordoni, P. Bachman, and Y. Bengio, "Iterative alternating neural attention for machine reading," *CoRR*, vol. abs/1606.02245, 2016.

[85] Y. Shen, P.-S. Huang, J. Gao, and W. Chen, "Reasonet: Learning to stop reading in machine comprehension," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1047–1055, ACM, 2017.

[86] B. Dhingra, H. Liu, Z. Yang, W. Cohen, and R. Salakhutdinov, "Gated-attention readers for text comprehension," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, pp. 1832–1846, 2017.

[87] A. Trischler, Z. Ye, X. Yuan, P. Bachman, A. Sordoni, and K. Suleman, "Natural language comprehension with the epireader," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 128–137, 2016.

[88] Y. Yu, W. Zhang, K. Hasan, M. Yu, B. Xiang, and B. Zhou, "End-to-end answer chunk extraction and ranking for reading comprehension," *arXiv preprint arXiv:1610.09996*, 2016.

[89] M. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi, "Bidirectional attention flow for machine comprehension," *arXiv preprint arXiv:1611.01603*, 2016.

[90] Y. Cui, Z. Chen, S. Wei, S. Wang, T. Liu, and G. Hu, "Attention-over-attention neural networks for reading comprehension," *arXiv preprint arXiv:1607.04423*, 2016.

[91] L. Yang, Q. Ai, J. Guo, and W. B. Croft, "anmm: Ranking short answer texts with attention-based neural matching model," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pp. 287–296, ACM, 2016.

[92] K. Lee, T. Kwiatkowski, A. Parikh, and D. Das, "Learning recurrent span representations for extractive question answering," *arXiv preprint arXiv:1611.01436*, 2016.

[93] S. Wang and J. Jiang, "Machine comprehension using match-lstm and answer pointer," *arXiv preprint arXiv:1608.07905*, 2016.

[94] W. Wang, N. Yang, F. Wei, B. Chang, and M. Zhou, "Gated self-matching networks for reading comprehension and question answering," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, pp. 189–198, 2017.

[95] Z. Yang, B. Dhingra, Y. Yuan, J. Hu, W. W. Cohen, and R. Salakhutdinov, "Words or characters? fine-grained gating for reading comprehension," *arXiv preprint arXiv:1611.01724*, 2016.

[96] H. He, K. Gimpel, and J. J. Lin, "Multi-perspective sentence similarity modeling with convolutional neural networks.," in *EMNLP*, pp. 1576–1586, 2015.

[97] C. Gulcehre, S. Chandar, K. Cho, and Y. Bengio, "Dynamic neural turing machine with soft and hard addressing schemes," *arXiv preprint arXiv:1607.00036*, 2016.

[98] S. Khadka, J. J. Chung, and K. Tumer, "Evolving memory-augmented neural architecture for deep memory problems," 2017.

[99] C. Gulcehre, S. Chandar, and Y. Bengio, "Memory augmented neural networks with wormhole connections," *arXiv preprint arXiv:1701.08718*, 2017.

[100] W. Zhang, Y. Yu, and B. Zhou, "Structured memory for neural turing machines," *arXiv preprint arXiv:1510.03931*, 2015.

[101] M. Andrychowicz and K. Kurach, "Learning efficient algorithms with hierarchical attentive memory," *arXiv preprint arXiv:1602.03218*, 2016.

[102] J. Rae, J. J. Hunt, I. Danihelka, T. Harley, A. W. Senior, G. Wayne, A. Graves, and T. Lillicrap, "Scaling memory-augmented neural networks with sparse reads and writes," in *Advances In Neural Information Processing Systems*, pp. 3621–3629, 2016.

[103] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[104] N. Ketkar, "Introduction to pytorch," in *Deep Learning with Python*, pp. 195–208, Springer, 2017.

[105] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, "Memory-efficient backpropagation through time," in *Advances in Neural Information Processing Systems*, pp. 4125–4133, 2016.

[106] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, pp. 448–456, 2015.

[107] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *CoRR*, vol. abs/1607.06450, 2016.

[108] T. Cooijmans, N. Ballas, C. Laurent, Ç. Gülçehre, and A. Courville, "Recurrent batch normalization," *arXiv preprint arXiv:1603.09025*, 2016.

[109] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," in *Advances in Neural Information Processing Systems*, pp. 972–981, 2017.

[110] C. Laurent, G. Pereyra, P. Brakel, Y. Zhang, and Y. Bengio, "Batch normalized recurrent neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pp. 2657–2661, IEEE, 2016.

[111] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[112] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations*, 2015.

[113] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, 2014.

[114] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast bayesian optimization of machine learning hyperparameters on large datasets," *arXiv preprint arXiv:1605.07079*, 2016.

[115] Y. Chen, M. W. Hoffman, S. G. Colmenarejo, M. Denil, T. P. Lillicrap, M. Botvinick, and N. Freitas, "Learning to learn without gradient descent by gradient descent," in *International Conference on Machine Learning*, pp. 748–756, 2017.

[116] A. Graves, M. G. Bellemare, J. Menick, R. Munos, and K. Kavukcuoglu, "Automated curriculum learning for neural networks," *arXiv preprint arXiv:1704.03003*, 2017.

[117] M. Pickett, R. Al-Rfou, L. Shao, and C. Tar, "A growing long-term episodic & semantic memory," *arXiv preprint arXiv:1610.06402*, 2016.

[118] A. H. Miller, W. Feng, A. Fisch, J. Lu, D. Batra, A. Bordes, D. Parikh, and J. Weston, "Parlai: A dialog research software platform," *arXiv preprint arXiv:1705.06476*, 2017.

List of Figures

2.1 Basic perceptron functionality 6
2.2 Interdependent sequence 8
2.3 Unfolded recurrent neural network 8
2.4 Simple recurrent neural network node 9
2.5 The long term dependencies 9
2.6 LSTM node overview 10
2.7 LSTM forget gate 11
2.8 LSTM input gate and input activation 12
2.9 LSTM internal cell state 12
2.10 LSTM output gate and output activation 13
2.11 LSTM in a mulit-node view 13
2.12 Gated Recurrent Unit 14
2.13 Bidirectional RNN 14
2.14 DNC system overview 15
2.15 DNC memory unit and symbols legend 17
2.16 DNC Memory unit control signals 19
2.17 DNC write mechanism 21
2.18 DNC memory update mechanism 22
2.19 DNC precedence weightings update 23
2.20 DNC linkage matrix mechanism 24
2.21 DNC read mechanism 25
2.22 DNC memory reading 26
2.23 DNC unfolded system overview 26

3.1 Copy task example 28

5.1 DNC training progress with copy task 40
5.2 DNC training progress with bAbI 1 task 41
5.3 DNC general functionallity in the copy task 43
5.4 DNC general functionallity in the bAbI 1 task 44
5.5 DNC write functionality in the bAbI 1 task 46
5.6 DNC memory handling in the copy task 48

6.1 DNC bypass dropout 57
6.2 DNC bidirectional controller 58
6.3 DNC with atop RNN 59
6.4 Content-based memory unit 60

7.1 DNC advancements comparison in the bAbI 1 task 64
7.2 DNC with LN and BD on the bAbI task 1 65
7.3 DNC advancements comparison in the copy task 66
7.4 Comparison of the MU and CBMU 68

List of Tables

2.1 DNC Memory unit control signals 18

3.1 Parametrization of the copy task in this work. 27
3.2 Statistics of the bAbI 20 task in the English 10k version. 29
3.3 Statistics of the CN CBT 31
3.4 Statistics of the CNN RC Task. 33

5.1 Hyper-paramter of DNC training analysis 39
5.2 Description of DNC functionality plot 5.3 42
5.3 Description of DNC functionallity plot 5.5 45
5.4 Description of DNC functionallity plot 5.6 47
5.5 Notations of the DNC and LSTM 49
5.6 DNC parametrization of bAbI task 51
5.7 DNC and LSTM computing time 52

7.1 Time comparison between MU and CBMU 67
7.2 Computing resources of the bAbI 20 experiments 70
7.3 bAbI 20 task mean results 70
7.4 bAbI 20 task best results 71
7.5 CBT results 73
7.6 CNN RC results 74
7.7 Summary of the results 75