# Design and Implementation of a Neural Turing Machine with a Spiking Neural Network Controller

Bachelor Thesis

submitted on May 06, 2024


Faculty of Business and Health

Business Information Systems - Data Science

Course WWI2021F

by

Jonas Michel

Corporate Supervisor

Dr. Valeria Bragaglia
Staff Research Scientist
IBM Research Zurich

DHBW Stuttgart:

Professor Dr. Kai Holzweißig
Dean and Head of Studies
Course Director WWI2021F


Supervisor's Signature

# Acknowledgments

I want to express my gratitude to Dr. Valeria Bragaglia and Dr. Folkert Horst for their supervision and invaluable feedback throughout this project. Special thanks to Dr. Stanislaw Woźniak for enlightening discussions on Spiking Neural Networks and Peter Kersting for his insights on Neural Networks.

# Contents

# List of Abbreviations

| | |
|---|---|
| **ADAM** | Adaptive Moment Estimation |
| **ANN** | Artificial Neural Network |
| **DNC** | Differentiable Neural Computer |
| **DNN** | Deep Neural Network |
| **D-NTM** | Dynamic Neural Turing Machine |
| **DSR** | Design Science Research |
| **E-NTM** | Evolving Neural Turing Machine |
| **GRU** | Gated Recurrent Unit |
| **LIF** | Leaky Integrate-and-Fire |
| **LSTM** | Long Short-Term Memory |
| **MANN** | Memory Augmented Neural Network |
| **NTM** | Neural Turing Machine |
| **RMSProp** | Root Mean Square Propagation |
| **RNN** | Recurrent Neural Network |
| **R-NTM** | Reinforcement Neural Turing Machine |
| **SAM** | Sparse Access Memory |
| **SDR** | Systems Development Research |
| **SNN** | Spiking Neural Network |
| **SNTM** | Spiking Neural Turing Machine |
| **SNU** | Spiking Neural Unit |
| **SRN** | Simple Recurrent Network |

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Motivation

Classical Artificial Neural Network (ANN) architectures attempt to mimic the real world by emulating the brain,[1] but a more directly brain-inspired approach exists in the form of Spiking Neural Network (SNN) architectures.[2] Brain-inspired SNNs can be engineered to tackle cognitive tasks[3], yet this necessitates significant computational power, memory capabilities, and an aspiration towards Turing completeness.[4] Maass/Markram 2004 provided a theoretical proof that SNNs have computational capabilities on par with classical ANNs. Additionally, recent research indicates that SNNs can form complete computational frameworks.[5] SNN architectures draw inspiration from the brain, with neurons communicating through discrete signals known as spikes.[6] Various SNN models[7], starting with the foundational "integrate-and-fire" model by Lapicque 1907, have been established since the early 20th century. Despite their theoretical foundations being established quite some time ago, SNNs are gaining popularity today, thanks to advancements in specialized hardware and the ability to train via backpropagation.[8] While large ANNs require high-end graphic cards[9], SNNs offer the advantage of running on specialized, energy-efficient hardware.[10] As SNNs gain in popularity, an increasing number of simulation and abstraction tools are becoming available[11], such as Brain2[12], Nest[13], and SpykeTorch[14]

Currently, there are various applications for SNNs. Zheng, H. et al. 2024 developed an SNN-based framework for automatically learning timing factors, which finds application in fields such as speech recognition, visual recognition, and robot place recognition.[15] Furthermore, SNN architectures are utilized for decision-making[16], object recognition[17], and action understanding.[18] Also the challenge of real-time interaction with the environment,[19] crucial for applications such as autonomous driving[20] and in the IoT sector, can be effectively addressed by employing SNNs.[21]

[1] Cf. McCulloch/Pitts 1943; Kussul/Baidyk/Wunsch 2010; Cox/Dean 2014
[2] Cf. Tavanaei et al. 2019; Yamazaki et al. 2022; Gorgan Mohammadi/Ganjtabesh 2024
[3] Cf. Gorgan Mohammadi/Ganjtabesh 2024
[4] Cf. Zhang, Y./Qu/Zheng, W. 2021, p. 671
[5] Cf. Zhang, Y./Qu/Zheng, W. 2021, p. 665
[6] Cf. Yamazaki et al. 2022, p. 1
[7] Cf. Hodgkin/Huxley, A. F. 1952; FitzHugh 1961; Hindmarsh/Rose/Huxley, Andrew Fielding 1984
[8] Cf. Lee/Delbruck/Pfeiffer 2016
[9] Cf. Tavanaei et al. 2019, p. 1
[10] Carrillo et al. 2012; Merolla et al. 2014; Zheng, H. et al. 2024
[11] Cf. Yamazaki et al. 2022, p. 25
[12] Cf. Stimberg/Brette/Goodman 2019
[13] Cf. Gewaltig/Diesmann 2007
[14] Cf. O'Connor et al. 2013a
[15] Cf. Zheng, H. et al. 2024, p. 2
[16] Cf. Héricé et al. 2016
[17] Cf. Kheradpisheh et al. 2018
[18] Cf. Kirkland et al. 2020
[19] Cf. Lee/Delbruck/Pfeiffer 2016, p. 1
[20] Cf. Liu, L. et al. 2021, p. 6480
[21] Cf. O'Connor et al. 2013b; Neil/Liu, S.-C. 2016; Xue/Chen, X./Li 2017

Gorgan Mohammadi/Ganjtabesh 2024 propose a framework that advances towards actual cognitive tasks[22], highlighting the significant impact and importance of SNNs in this domain.

The ability to store and access information is crucial when mimicking cognitive tasks with neural networks, as it allows the behavior to be modified based on the stored information.[23] Long Short-Term Memory (LSTM) architectures[24] deliver a solution to store information long-term.[25] Used in various industry applications, LSTM architectures are a popular choice for sequential tasks.[26] However, there are Memory Augmented Neural Network (MANN) architectures that outperform classical LSTM architectures on big sequential tasks.[27] Graves/Wayne/Danihelka 2014 proposed the Neural Turing Machine (NTM), an example of a MANN. The NTM operates with a controller, which is a neural network, to perform read and write operations on a memory matrix.[28] This architecture introduces a neural network that can be trained via backpropagation, featuring addressable memory to store and retrieve information.[29] This memory enrichment serves as an "analogy to Turing's enrichment of finite-state machines by an infinite memory tape."[30] With the advancements from Greve/Jacobsen/Risi 2016 and the contributions of Mark Collier/Beel 2018, the NTM has undergone further theoretical refinement. However, NTMs also find practical applications in industry, such as estimating the remaining useful life of machinery[31], in air pollution prediction[32], or when used as classifiers.[33]

## 1.2 Problem Statement

In contemporary research, NTMs build upon Recurrent Neural Network (RNN) frameworks, such as LSTM and Gated Recurrent Unit (GRU) cells as foundational controller elements.[34] While there are various open-source NTM implementations available,[35] a number of them are unstable during training.[36] Mark Collier/Beel 2018 address this training challenges and proposes a comprehensive implementation guideline for NTMs. However, this manual also focuses exclusively on the use of the LSTM cells as controllers. While the complexity of the controller is linked

[22] "Cognitive tasks are those undertakings that require a person to mentally process new information (i.e., acquire and organize knowledge/learn) and allow them to recall, retrieve that information from memory and to use that information at a later time in the same or similar situation (i.e., transfer)." Kester/Kirschner 2012, p. 619
[23] Cf. Greve/Jacobsen/Risi 2016, p. 117
[24] Cf. Hochreiter/Schmidhuber 1997
[25] Cf. Mark Collier/Beel 2018, p. 1
[26] Cf. Sutskever/Vinyals/Le 2014; Venugopalan et al. 2015; Vaswani et al. 2017
[27] Cf. Sukhbaatar et al. 2015; Wayne et al. 2016
[28] Cf. Graves/Wayne/Danihelka 2014, p. 5
[29] Cf. Graves/Wayne/Danihelka 2014, p. 22
[30] Graves/Wayne/Danihelka 2014, p. 1
[31] Cf. Falcon et al. 2022
[32] Cf. Asaei-Moamam et al. 2023
[33] Cf. Faradonbe/Safi-Esfahani 2020
[34] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 11
[35] See GitHub Implementations: Chiggum 2016; Camigord 2017; Loudinthecloud 2018; Snowkylin 2019
[36] Cf. Mark Collier/Beel 2018, p. 95

to the performance of the NTM,[37] currently, alternative network architectures have not been examined as controllers for NTMs.[38]

Exploring new controller architectures, like different SNN architectures, could lead to solving more complex cognitive tasks.[39] This would be an integration of current state-of-the-art methodologies, which could catalyze enhancements within the NTM architecture.[40] Different Neural Network architectures could also enhance the NTM controller, leading to improved memory access.[41]

## 1.3 Research Objective

As highlighted, SNN architectures draw inspiration from the brain by utilizing spikes for information processing, while the NTM emulates brain-like functionality through the incorporation of memory mechanisms. To address the identified challenge of testing the NTM framework with various network architectures as controllers and considering the stated potential of SNN architectures for this role, this study is designed to explore the following questions:

**Is it feasible to construct an NTM architecture with an SNN controller?** The objective is to design, implement, and evaluate a prototype of an NTM using an SNN controller, introduced as Spiking Neural Turing Machine (SNTM). The SNN controller will be simulated using an Spiking Neural Unit (SNU) architecture, which can be trained via backpropagation and embedded into the classical TensorFlow workflow.[42]

The second research question aims to further elaborate on the potential of an SNTM by asking: **How does the resulting SNTM perform compared to classical NTM and LSTM architectures?** The performance of the SNTM will be benchmarked against the NTM and LSTM architectures on a set of standard tasks for NTM architectures. The evaluation will be based on three simple sequential tasks derived from the literature: copy, reverse, and repeat copy. Through benchmarking, potential advantages and limitations of the SNTM architecture will be identified and discussed. The goal here is to lay a foundation for further research directions by embedding the SNTM into the current literature.

## 1.4 Research Methodology and Thesis Structure

To address the specified research objective, this study will follow the Design Science Research (DSR) paradigm, which approaches research questions through the creation of innovative arti-

[37] Cf. Zaremba/Sutskever 2015, p. 8
[38] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 11
[39] Cf. Greve/Jacobsen/Risi 2016, p. 123
[40] Cf. Falcon et al. 2022, p. 10
[41] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 20
[42] Discussed in Subsection 2.3.3

facts.[43] These artifacts are either artificial objects, such as models and instantiations, or processes, including methods and software.[44] In this thesis, the developed artifact is an SNTM implemented in Python utilizing the TensorFlow framework. The DSR paradigm consists of three cycles that contribute to the development of the artifact: *Relevance*, *Design*, and *Rigor*.[45] While the Relevance Cycle focuses on the contextual environment of the research, the Rigor Cycle provides the scientific background from the literature.[46] These two cycles feed information into the Design Cycle, enabling the artifact to be developed and evaluated iteratively.[47]

The thesis is structured according to the DSR publication schema proposed by Gregor/Hevner, A. 2013, detailed in Appendix 1/1. An introduction to the topic, along with the motivation, problem statement, and research objectives, is presented in Chapter 1. Chapter 2 provides a literature review on the current state of the art in NTM and SNN architectures, as described by Webster/Watson 2002. The research methodology and design are explained in detail in Chapter 3, which also further elaborates on the research objectives. The Design Cycle of the artifact is enhanced with the Systems Development Research (SDR) methodology, as proposed by Nunamaker/Chen, M. 1990, to guide the design and development of the SNTM artifact. This design process is outlined in Chapter 4. Subsequently, the artifact is evaluated using scientific machine learning benchmarks, as proposed by Thiyagalingam et al. 2022. The evaluation and discussion of the artifact are combined in Chapter 5. The thesis concludes with a critical reflection and a future outlook in Chapter 6.

[43] Cf. Hevner, A. R./Chatterjee 2010, p. 5
[44] Cf. Gregor/Hevner, A. 2013, p. 341
[45] Cf. Hevner, A. 2007, p. 87
[46] Cf. Hevner, A. 2007, p. 88 et seqq.
[47] Cf. Hevner, A. 2007, p. 90 et seq.

# 2 Review of the Current State of Research

Building on the defined problem statement and research objective, the following chapter conducts a detailed literature review. Initially, the methodology for identifying relevant articles is outlined. This is followed by an explanation of fundamental concepts and a discussion of the current state of research. The chapter concludes by presenting and evaluating current NTM artifacts.

## 2.1 Literature Review Methodology for Identifying Artifacts

The literature review was conducted according to the methodology proposed by Webster/Watson 2002. The objective is to identify and examine current NTM artifacts, analyze their architecture, and explore the integration of SNNs as NTM controllers.

Therefore, the following key concepts are identified:

- Neuromorphic Computing (NC)
- Spiking Neural Networks (SNN)
- Neural Turing Machines (NTM)
- Neural Networks (NN)
- Memory Augmented Neural Networks (MANN)

The methodology involved utilizing specific search terms, listed in Appendix 2, to locate relevant literature in several databases: Google Scholar[48], ScienceDirect[49], Nature[50], arXiv[51], IEEE Xplore[52], and SpringerLink[53]. The first 75 results from each database are reviewed and filtered based on relevance to the topic of this work, their recency, and the journal in which they are published. Only articles that have been cited in other works were included for further consideration.

Upon the initial identification of relevant literature, a comprehensive backward and forward search was executed.[54] This entailed a thorough examination of the references cited within the identified articles and the citations these articles received, aiming to uncover further relevant studies. To achieve this, the websites ConnectedPapers[55] and ResearchRabbit[56] were utilized,

[48] https://scholar.google.de/
[49] https://www.sciencedirect.com/
[50] https://www.nature.com/
[51] https://arxiv.org/
[52] https://ieeexplore.ieee.org/
[53] https://link.springer.com/
[54] Cf. Webster/Watson 2002, p. XVI
[55] https://www.connectedpapers.com/
[56] https://www.researchrabbit.ai/

enabling a streamlined review of both citations and references. This approach guaranteed the identification of literature closely aligned with the research topic.

In Section 2.5, Review of Neural Turing Machine Artifacts, a concept matrix is constructed to examine similar current artifacts.[57]

## 2.2 Classic Artificial Neural Networks

The classical Artificial Neural Network (ANN) takes a high-level brain-inspired approach, comprising neurons interconnected with synapses, where their neural dynamics represent an input-output transformation through a nonlinear activation function.[58]

### 2.2.1 Basic Machine Learning Definitions

Machine learning aims to identify patterns within data to facilitate predictions grounded in probabilistic analysis.[59] The primary objective of machine learning is to develop models that can generalize from the data they have been trained on to make predictions about new, unseen data.[60] To achieve this, simple ANNs take a high-level inspiration from components of the human brain, detailed in Appendix 3/1.

The basis of first-generation ANN architectures, perceptrons, also called neurons, was introduced by Rosenblatt 1962 and can be mathematically described as follows:[61]

$$y = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases} \tag{2.1}$$

Where the output of the network is either 0 or 1, depending on the weighted ($w_j$) sum of the inputs ($x_j$) and a given threshold.[62]

The second-generation ANN architectures can be expressed as follows:[63]

$$y = \varphi(w^t \cdot x + b) \tag{2.2}$$

Where $y$ is the output, $\varphi$ is the activation function, $w$ is the weight vector, $x$ is the input vector, and $b$ is the bias.[64] These perceptrons can assume values within the set of real numbers $\mathbb{R}$, contingent on the utilized activation function.[65]

[57] Cf. Webster/Watson 2002, p. XVII
[58] Cf. Woźniak et al. 2020, p. 325
[59] Cf. Janiesch/Zschech/Heinrich 2021, p. 685
[60] Cf. Jo 2023, p. 22
[61] Cf. Nielsen 2015; Wang/Lin/Dang 2020
[62] Cf. Nielsen 2015, p. 3
[63] Cf. Wang/Lin/Dang 2020; García Cabello 2022
[64] Cf. García Cabello 2022, p. 4
[65] Cf. Nielsen 2015, p. 7 et seqq.

In a classical ANN architecture, the perceptrons are clustered into multiple layers.[66]

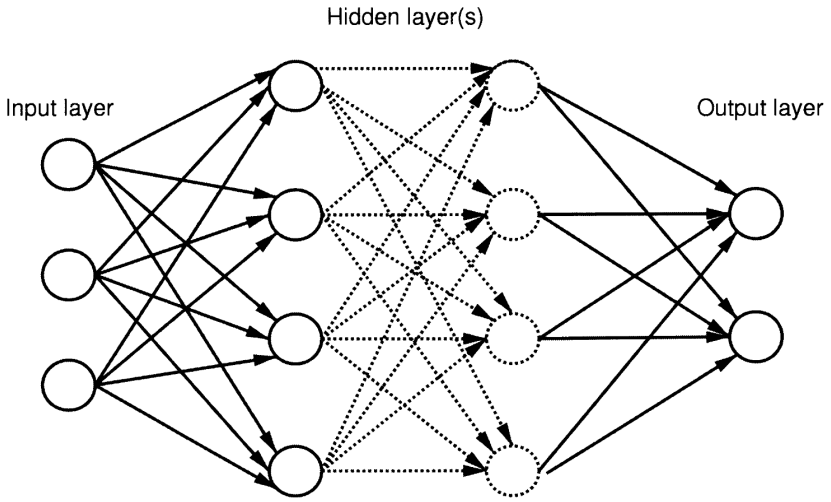



Fig. 1: Fully Connected Feed-Forward Network[67]

Figure 1 illustrates a fully connected feed-forward network, a basic form of a simple ANN.[68] Each circle represents one perceptron, with the lines representing the connections. The network consists of three components: the input layer, the hidden layer(s), and the output layer.[69] Fully connected feed-forward means that each perceptron in one layer is connected to each perceptron in the next layer, and no connections are made in the same layer.[70] Like synapses in the brain, each connection between the perceptrons has a weight, which is adjusted during the learning process.[71]

Deep Neural Network (DNN) architectures are a type of ANN which consists of more than one hidden layer, organized in deeply nested network architectures with more advanced types of neurons (e.g., RNN).[72]

Nearly endless options of algorithms, architectures, and hyperparameters exist, each adjustable to fine-tune a model for a particular task.[73] This customization, however, introduces a triangular trade-off among accuracy, complexity, and computational cost.[74] Accurate comparisons between different models are feasible only when variations are made to one of the three edges of the triangle at a time, with consistent metrics reported.[75]

Bräunl 2003, Guresen/Kayakutlu 2011, García Cabello 2022, and Nielsen 2015, provide additional insights into ANN theory.

[66]Cf. Fiesler 1994, p. 233
[67]Included in: Bräunl 2003, p. 275
[68]Cf. Bräunl 2003, p. 275
[69]Cf. Nielsen 2015, p. 11
[70]Cf. Bräunl 2003, p. 274
[71]Cf. Janiesch/Zschech/Heinrich 2021, p. 687
[72]Cf. Janiesch/Zschech/Heinrich 2021, p. 678
[73]Cf. Janiesch/Zschech/Heinrich 2021, p. 691
[74]Cf. Nielsen 2015, p. 12
[75]Cf. Janiesch/Zschech/Heinrich 2021, p. 691

### 2.2.2 Recurrent Neural Networks

Recurrent Neural Network (RNN) architectures are a type of deep neural networks that can process sequences of data.[76] The connection between the neurons in RNN architectures generate a direct graph, which allows the network to maintain an updated internal state with each new input.[77] Thus, the previous complex inputs are remembered for long periods and used to influence the current output.[78]

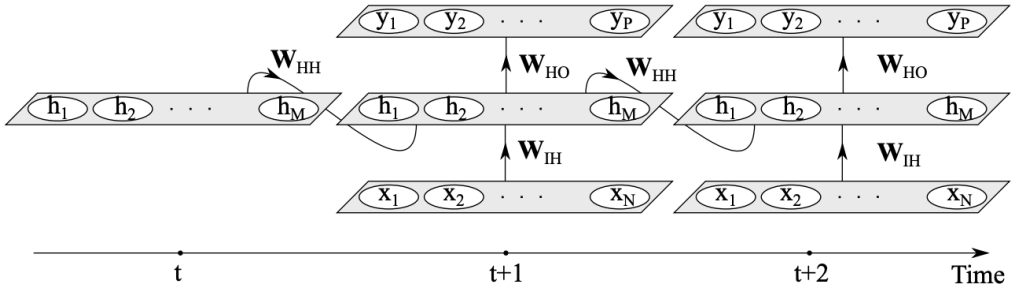



Fig. 2: Unfolded Recurrent Neural Network[79]

Figure 2 depicts a Basic RNN with a single hidden layer.[80] The input layer is at the bottom, the hidden layer is in the middle, and the output layer is at the top. The utilized units in the hidden layer are Simple Recurrent Network (SRN) units, which are connected to each other in a chain-like structure. For the SRN unit, the mathematical function is delineated by the following two equations:[81]

$$\mathbf{h}_t = \tanh(\mathbf{W}_{IH}\mathbf{x}_t + \mathbf{W}_{HH}\mathbf{h}_{t-1} + \mathbf{b}_h) \tag{2.3}$$

Where $\mathbf{W}_{IH}$ is the weight matrix for the input layer, $\mathbf{W}_{HH}$ is the weight matrix for the hidden layer, $\mathbf{b}_h$ is the bias for the hidden layer, $\mathbf{x}_t$ is the input at time step $t$, $\mathbf{h}_{t-1}$ is the hidden state at time step $t-1$ and $\mathbf{h}_t$ is the hidden state at time step $t$.[82] And:

$$\mathbf{y}_t = f_O(\mathbf{W}_{HO}\mathbf{h}_t + \mathbf{b}_o) \tag{2.4}$$

Where $f_O$ is the activation function for the output layer, $\mathbf{W}_{HO}$ is the weight matrix for the output layer, $\mathbf{b}_o$ is the bias for the output layer, and $\mathbf{y}_t$ is the output at time step $t$.[83]

A schematic view of the SRN unit is shown in Figure 3 on the left side. These SRN units suffer from the "vanishing gradient" and the "exploding gradient" problem.[84] To overcome these optimization challenges, the LSTM was introduced by Hochreiter/Schmidhuber 1997.[85] In Figure 3

[76] Cf. Janiesch/Zschech/Heinrich 2021, p. 690
[77] Cf. Onan 2022, p. 2103
[78] Cf. Salehinejad et al. 2018, p. 1
[79] Included in: Salehinejad et al. 2018, p. 2
[80] Cf. Jo 2023, p. 257
[81] Cf. Salehinejad et al. 2018, p. 2
[82] Cf. Onan 2022, p. 2103
[83] Cf. Onan 2022, p. 2103
[84] Cf. Janiesch/Zschech/Heinrich 2021, p. 690
[85] Cf. Greff et al. 2017; Van Houdt/Mosquera/Nápoles 2020; Janiesch/Zschech/Heinrich 2021

on the right side, the LSTM block is depicted, which can be used in the hidden layers of an RNN.[86]

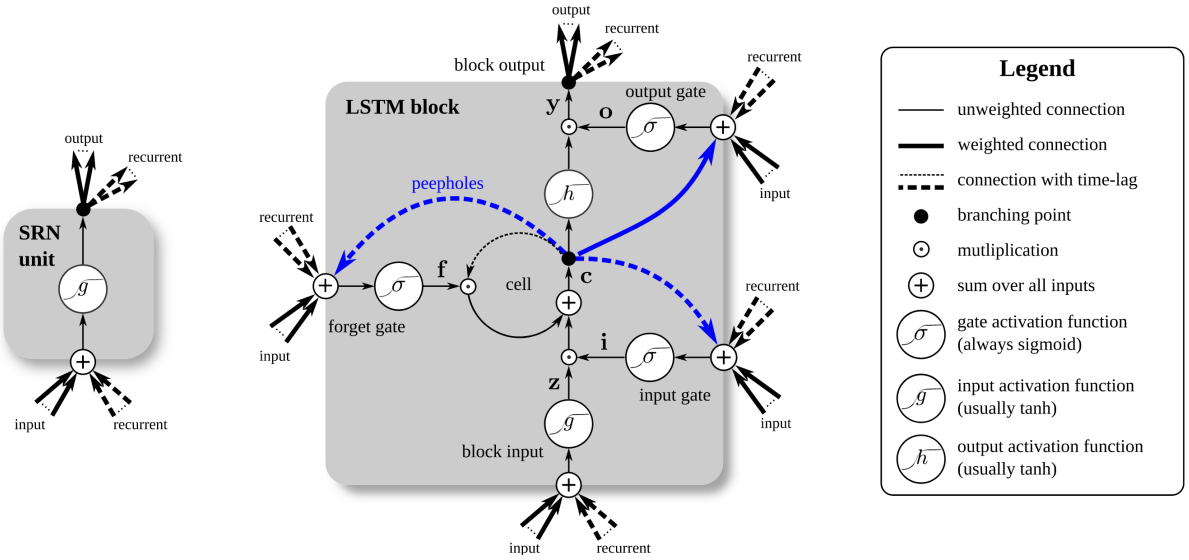



Fig. 3: Detailed Schematic of the SRN Unit (Left) and an LSTM Block (Right) as Used in the Hidden Layers of an RNN[87]

The vanilla LSTM architecture is composed of four main components in the LSTM architecture: the input gate, the forget gate, the output gate, and the cell state, and can be expressed by the following equations:[88]

$$z_t = \tanh(\mathbf{W}_z \mathbf{x}_t + \mathbf{R}_z \mathbf{y}_{t-1} + \mathbf{b}_z) \quad \text{block input} \quad (2.5)$$

$$i_t = \sigma(\mathbf{W}_i \mathbf{x}_t + \mathbf{R}_i \mathbf{h}_{t-1} + \mathbf{p}_i \odot \mathbf{c}_{t-1} + \mathbf{b}_i) \quad \text{input gate} \quad (2.6)$$

$$f_t = \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{R}_f \mathbf{y}_{t-1} + \mathbf{p}_f \odot \mathbf{c}_{t-1} + \mathbf{b}_f) \quad \text{forget gate} \quad (2.7)$$

$$\mathbf{c}_t = z_t \odot i_t + \mathbf{c}_{t-1} \odot f_t \quad \text{cell state update} \quad (2.8)$$

$$o_t = \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{R}_o \mathbf{y}_{t-1} + \mathbf{p}_o \odot \mathbf{c}_t + \mathbf{b}_o) \quad \text{output gate} \quad (2.9)$$

$$\mathbf{y}_t = \tanh(\mathbf{c}_t) \odot o_t \quad \text{block output} \quad (2.10)$$

Where $\mathbf{W}$, $\mathbf{R}$, and $\mathbf{p}$ denote the weights associated with $\mathbf{x}_t$, $\mathbf{y}_{t-1}$, and $\mathbf{c}$, respectively, and $\mathbf{b}$ represents the bias weight vector.[89]

The first calculation combines the current input $\mathbf{x}_t$ and the output from the LSTM unit at the previous timestep $\mathbf{y}_{t-1}$ to compute the `block input` $z_t$.[90] The `input gate` merges the current input $\mathbf{x}_t$, the output from the LSTM unit at the previous timestep $\mathbf{y}_{t-1}$, and the cell state from the prior iteration $\mathbf{c}_{t-1}$.[91] Through the `forget gate`, the LSTM block determines which information in the cell state is to be retained or discarded, thus eliminating irrelevant information

[86] Cf. Greff et al. 2017, p. 2223
[87] Included in: Greff et al. 2017, p. 2223
[88] Cf. Greff et al. 2017, p. 2223
[89] Cf. Van Houdt/Mosquera/Nápoles 2020, p. 5931 et seqq.
[90] Cf. Van Houdt/Mosquera/Nápoles 2020, p. 5931
[91] Cf. Van Houdt/Mosquera/Nápoles 2020, p. 5932

from the cell state.[92] During the `cell state update`, the inputs $z_t$ and $i_t$ are combined with the previous iteration's cell state $\mathbf{c}_{t-1}$, which is adjusted by the forget value $f_t$, to refresh the cell state $\mathbf{c}_t$.[93] The `output gate` entails the integration of the current input $\mathbf{x}_t$, the output from the LSTM unit at the previous timestep $\mathbf{y}_{t-1}$, and the current cell state $\mathbf{c}_t$, to determine the output $o_t$.[94] Finally, the `block output` is calculated by combining the cell state $\mathbf{c}_t$ and the output gate $o_t$ to produce the output $\mathbf{y}_t$.[95]

Appendix 3/2 presents an alternative illustration of the LSTM block architecture compared to that in Figure 2. Despite these variations in presentation, the underlying data flow is consistent.[96]

### 2.2.3 Model Training

One of the most common training methods for machine learning models is backpropagation.[97] Backpropagation involves a two-step process where the network's error is first calculated and then propagated backward through the network, updating the weights of connections to minimize this error.[98] Through this iterative adaptation, the network can learn from the discrepancies between its output and the actual desired output, thereby enhancing its accuracy over time.[99]

Numerous optimization algorithms have been developed to adjust the weights of a network to enhance the efficiency of the training process.[100] The foundational technique among these is Gradient Descent, a first-order iterative optimization algorithm designed to minimize the loss function.[101] It operates by computing the gradient of the loss function with respect to the weights, subsequently adjusting the weights in the direction opposite to the gradient.[102] Momentum can further refine this algorithm, enabling it to circumvent local minima by incorporating a momentum term with the gradient descent.[103] This addition allows movement maintenance towards the optimal direction, even in instances with small gradients, by partially adding the previous gradient to the new one.[104] Advanced optimizers, such as Root Mean Square Propagation (RMSProp), enhance the optimization process by accumulating past gradients to compute the next step.[105] This method aggregates gradients into an exponentially weighted average, effectively discarding old gradients not solely focusing on current gradient calculation.[106] RMSProp outperforms other

[92] Cf. Hochreiter/Schmidhuber 1997, p. 1745
[93] Cf. Greff et al. 2017, p. 2223
[94] Cf. Van Houdt/Mosquera/Nápoles 2020, p. 5932
[95] Cf. Hochreiter/Schmidhuber 1997, p. 1744
[96] Cf. Van Houdt/Mosquera/Nápoles 2020, p. 5931 et seqq.
[97] Cf. Werbos 1990; Schaul/Antonoglou/Silver 2014; Haji/Abdulazeez 2021
[98] Cf. Werbos 1990, p. 1550 et seqq.
[99] Cf. Schmidhuber 2015, p. 90 et seq.
[100] Cf. Mukkamala/Hein 2017; Zou et al. 2019
[101] Cf. Haji/Abdulazeez 2021, p. 2716
[102] Cf. Botev/Lever/Barber 2017, p. 1899
[103] Cf. Haji/Abdulazeez 2021, p. 2719
[104] Cf. Botev/Lever/Barber 2017, p. 1899
[105] Cf. Mukkamala/Hein 2017; Zou et al. 2019
[106] Cf. Haji/Abdulazeez 2021, p. 2720

optimization algorithms in a large number of benchmark tests.[107] Despite the RMSProp algorithm, initially proposed by Hinton/Nitish/Kevin 2012, not being published in a formal paper, the algorithm is one of the most used optimization methods in the field of neural networks.[108] Its theoretical foundation and application have been further explored through mathematical descriptions[109] and practical implementations.[110] Adaptive Moment Estimation (ADAM), a method for stochastic gradient descent optimization, merges the methodologies of RMSProp and Momentum proposed by Kingma/Ba 2015. This approach is distinguished by its calculation of adaptable learning rates for individual parameters.[111] As one of the most commonly used optimization algorithms in the field of neural networks,[112] ADAM enhances the optimization process by considering the smooth gradient variation[113] and incorporating a bias correction mechanism.[114] ADAM reduces computational costs and requires less memory for execution.[115] A comprehensive mathematical description of ADAM is available in Kingma/Ba 2015, p. 2 et seqq. and Yi/Ahn/Ji 2020, p. 4. Different comparisons between optimization algorithms show that selecting the best-suited optimizer depends on the dataset and the model architecture.[116]

A loss function is essential for the computation of gradients, which are subsequently used for updating the weights.[117] Based on the loss function, gradients are calculated, and weights are updated through the various optimization algorithms,[118] so that the loss is minimized and the model's performance is optimized.[119] In the field of NTM architectures, the cross-entropy loss function is the most frequently utilized loss function.[120] Specifically, the sigmoid cross-entropy with logits function is notably effective for binary classification tasks.[121] The sigmoid cross-entropy with logits function is defined as follows within Tensorflow:[122]

$$f(x, z) = x - x \cdot z + \log(1 + e^{-x}) \qquad \text{for } x \geq 0 \tag{2.11}$$

Where $x$ are the predicted logits and $z$ are the target labels.[123] This variant allows for flexibility, enhancing its applicability by handling both hard binary labels and soft labels (represented by probabilities between 0 and 1).[124]

[107] Cf. Schaul/Antonoglou/Silver 2014
[108] Cf. Graves/Wayne/Danihelka 2014; Schmidhuber 2015; Mukkamala/Hein 2017; Mark Collier/Beel 2018
[109] Cf. Hinton/Nitish/Kevin 2012; Mukkamala/Hein 2017; Zou et al. 2019
[110] Cf. Keras 2024
[111] Cf. Kingma/Ba 2015, p. 2
[112] Cf. Yi/Ahn/Ji 2020, p. 4
[113] Cf. Haji/Abdulazeez 2021, p. 2720
[114] Cf. Kingma/Ba 2015, p. 3
[115] Cf. Fei et al. 2020, p. 1
[116] Cf. Chakrabarti/Chopra 2021; Haji/Abdulazeez 2021
[117] Cf. Haji/Abdulazeez 2021, p. 2716
[118] Cf. Yi/Ahn/Ji 2020, p. 3
[119] Cf. Haji/Abdulazeez 2021, p. 2716
[120] Cf. Graves/Wayne/Danihelka 2014; Zhang, W./Yu/Zhou 2015; Chandar et al. 2016; Mark Collier/Beel 2018
[121] Cf. Keren/Sabato/Schuller 2018; Mehta et al. 2024
[122] Cf. TensorFlow 2024
[123] Cf. TensorFlow 2024
[124] Cf. TensorFlow 2024

## 2.3 Spiking Neural Networks

Spiking Neural Network (SNN) architectures more closely mirror the brain's dynamic and event-driven nature compared to classical ANNs by integrating the concept of time and utilizing discrete spikes for communication, similar to biological neural interactions.[125] Thus, offering a more biologically realistic model of neural processing.[126]

### 2.3.1 Biological Neurons

Biological neurons are the fundamental unit of the nervous system.[127] These specialized cells are tasked with the transmission of information via electrical signals, termed action potentials or, more simply, spikes.[128] Thus, neurons transmit information by firing spike sequences in various temporal patterns.[129]

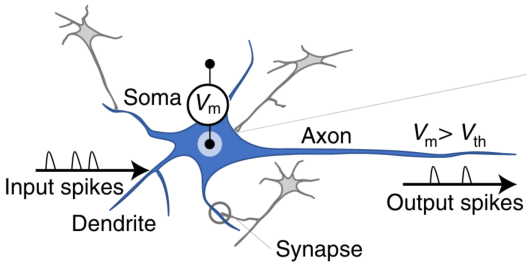



Fig. 4: Biological Neuron[130]

Figure 4 depicts a biological neuron with the dendrites, soma, axon, and synapses. The biological neuron receives input spikes, which are integrated into the membrane potential $V_m$ of the soma.[131] The membrane potential $V_m$ is the electrical potential with respect to the outside of the biological cell.[132] Once $V_m$ crosses the spiking threshold $V_{th}$ an action potential occurs,[133] which means the neuron emits an output spike with ions flowing through the axon.[134]

### 2.3.2 Mathematical Models

A multitude of mathematical SNN models have been proposed.[135] These models exhibit trade-offs between biological accuracy and computational feasibility.[136] A comprehensive review focused

[125] Cf. Zheng, H. et al. 2024, p. 13
[126] Cf. Roy/Jaiswal/Panda 2019, p. 1
[127] Cf. Yamazaki et al. 2022, p. 2
[128] Cf. Dayan/Abbott 2001, p. 1
[129] Cf. Dayan/Abbott 2001, p. 1
[130] Included in: Woźniak et al. 2020, p. 326
[131] Cf. Woźniak et al. 2020, p. 326
[132] Cf. Yamazaki et al. 2022, p. 3
[133] Cf. Woźniak et al. 2020, p. 326
[134] Cf. Yamazaki et al. 2022, p. 4
[135] Cf. Lapicque 1907; McCulloch/Pitts 1943; Hodgkin/Huxley, A. F. 1952; FitzHugh 1961
[136] Cf. Yamazaki et al. 2022, p. 6

on computational efficiency and biological plausibility was conducted by Izhikevich 2004. The comparison of the most relevant models is displayed in Appendix 3/3.

The mathematical description of first- and second-generation ANN models is outlined in Subsection 2.2.1. Similarly, SNN models, recognized as the third-generation of neural networks,[137] can be described as a hybrid system in the following manner:[138]

$$\begin{cases} \frac{d\mathbf{X}}{dt} = f(\mathbf{X}), \\ \mathbf{X} \leftarrow g_i(\mathbf{X}), \end{cases} \tag{2.12}$$

Where $\mathbf{X}$ describes the state variables of the neuron, $f(\cdot)$ are the differential equations for the evolution of the state variables.[139] In a simple Leaky Integrate-and-Fire (LIF) neuron model[140], biological spikes are emitted when the membrane potential $V_m$ crosses the spiking threshold $V_{th}$.[141] So $V_m$ would be the first component of the state variables $\mathbf{X}$. After a spike emission, the membrane potential $V_t$ needs to be reset.[142] The reset can be integrated into the hybrid system formalism by considering for example that outgoing spikes act on $\mathbf{X}$ through an additional (virtual) synapse: $\mathbf{X} \leftarrow g_i(\mathbf{X})$.[143]

To simulate an actual SNN, spiking activities are commonly modeled using the LIF neuron.[144] This is attributed to its nature as a one-dimensional spiking neural model that requires low computational resources.[145] Despite its relatively lower biological plausibility compared to other models,[146] the LIF model is widely used in the literature to simulate SNNs[147] and is particularly attractive for large-scale network simulations.[148] The LIF model represents a neuron that fires spikes when the membrane potential $V_m$ surpasses the spiking threshold. Beyond the membrane potential, this model also accounts for the membrane's leaky characteristic (ion diffusion), indicating that the membrane potential decays over time.[149] The LIF model can be described by the following differential equation:[150]

$$\tau_m \frac{dV_m(t)}{dt} = -\left(V_m(t) - E_r\right) + R_m I(t) \tag{2.13}$$

$$\text{if} \quad V_m \geq V_{th} \quad \text{then} \quad V_m \leftarrow V_{\text{peak}} \quad \text{followed by} \quad V_m \leftarrow V_{\text{reset}} \tag{2.14}$$

Where $V_m(t)$ is the membrane potential, $\tau_m$ is the membrane time constant, $E_r$ is a constant rest

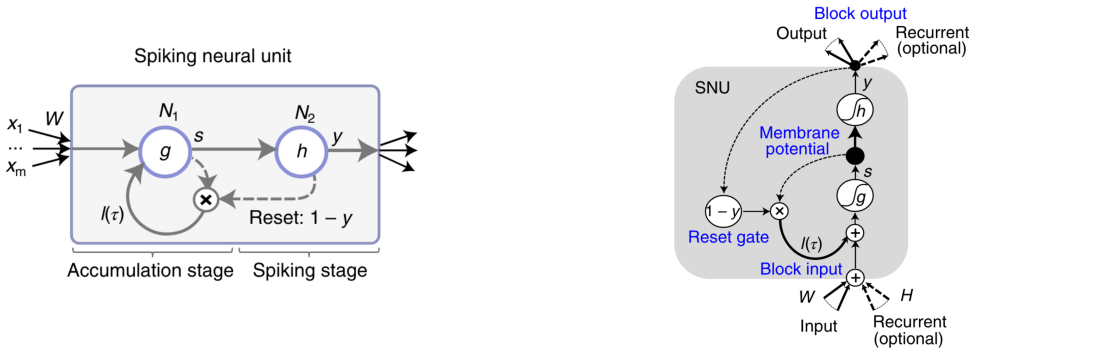
[137] Cf. Maass 1997
[138] Cf. Brette et al. 2007, p. 351
[139] Cf. Wang/Lin/Dang 2020, p. 260
[140] Cf. Woźniak et al. 2020, p. 325
[141] Cf. Wang/Lin/Dang 2020, p. 260
[142] Cf. Brette et al. 2007, p. 351
[143] Cf. Brette et al. 2007, p. 351
[144] Cf. Woźniak et al. 2020, p. 326
[145] Cf. Taherkhani et al. 2020, p. 254
[146] Cf. Izhikevich 2004, p. 1066
[147] Cf. Taherkhani et al. 2020, p. 254
[148] Cf. Brette et al. 2007, p. 350
[149] Cf. Yamazaki et al. 2022, p. 7
[150] Cf. Woźniak et al. 2020; Taherkhani et al. 2020; Yamazaki et al. 2022

potential, $R_m$ is the membrane resistance, $I(t)$ is the current potential input.[151] $V_{th}$ is the spiking threshold, $V_{\text{peak}}$ is the action potential, and $V_{\text{reset}}$ is resetting the membrane potential.[152] That means once the threshold is passed, the output spike $y(t) = 1$ is emitted, and the membrane potential $V_t$ is reset to $V_{\text{reset}}$ (often defined as 0).[153]

Other spiking neuron models rely on the same principles but differ in the complexity of the differential equations and are detailed in Izhikevich 2004, Rabinovich et al. 2006, and Yamazaki et al. 2022.

### 2.3.3 Spiking Neural Units

There are different approaches to implement the several mathematical spiking neuron models into SNN architectures,[154] some trainable via backpropagation.[155] The Spiking Neural Unit (SNU), proposed by Woźniak et al. 2020, is a high-level abstraction of the LIF dynamics into an ANN unit. The proposed SNU can be seamlessly integrated into ANN architectures like RNNs and trained via backpropagation through time.[156]




(a) Spiking Neural Dynamics[157] (b) Detailed Schematic of a Spiking Neural Unit[158]

Fig. 5: Spiking Neural Unit

As depicted in Figure 5a, the SNU comprises two ANN neurons as subunits: $N_1$, which models the membrane potential accumulation dynamics, and $N_2$, which implements the spike emission. The self-looping connection of $N_1$ realizes the integration dynamics of the membrane-potential state variable. The second neuron, $N_2$, is responsible for the spike emission and the resetting of the state variable by gating the self-looping connection at $N_1$. Figure 5b illustrates an SNU cell equivalent to the already discussed LSTM block.[159] The SNU unique features are a non-linear

[151] Cf. Taherkhani et al. 2020, p. 254
[152] Cf. Yamazaki et al. 2022, p. 7
[153] Cf. Woźniak et al. 2020, p. 326
[154] Cf. Wang/Lin/Dang 2020; Roy/Jaiswal/Panda 2019; Yamazaki et al. 2022
[155] Cf. Bohte/Kok/La Poutré 2002; Lee/Delbruck/Pfeiffer 2016
[156] Cf. Woźniak et al. 2020, p. 325 et seq.
[158] Included in: Woźniak et al. 2020, p. 326
[158] Included in: Woźniak et al. 2020, p. 327
[159] Discussed in Subsection 2.2.2 Recurrent Neural Networks

transformation $g$ within the internal state loop, a parametrized state loop connection ($l(\tau)$), a bias of the state output connection to the output activation function $h$, and a direct reset gate $(1-y)$ controlled by the output $y$.[160] Following ANN conventions, the SNU can be expressed as follows:[161]

$$\mathbf{s}_t = g(\mathbf{W}\mathbf{x}_t + l(\tau) \odot \mathbf{s}_{t-1} \odot (1 - \mathbf{y}_{t-1})) \tag{2.15}$$

$$\mathbf{y}_t = h(\mathbf{s}_t + b) \tag{2.16}$$

Where $\mathbf{s}_t$ is the vector of state variables (membrane potential) calculated by $N_1$ and $\mathbf{y}_t$ is the output vector (spike) calculated by the subunit $N_2$.[162] The activation function $g$ of $N_1$ may implement additional assumptions on the membrane potential value.[163] The input $\mathbf{x}_t$ is weighted by the synaptic weight $\mathbf{W}$ without any bias. The self-looping weight $l(\tau)$, applied to $\mathbf{s}_{t-1}$, approximates the membrane potential decay, simulating an occurring leak.[164] If $l(\tau) = 1$, the state does not decay. The last term $(1 - \mathbf{y}_{t-1})$ resets the state after a spike emission.[165] The activation function $h$ of $N_2$ determines the output of the entire SNU cell. A bias $b$ is added to the state variables $\mathbf{s}_t$ before applying the activation function.[166] With a step function, where $h(a)$ returns 1 if $a > 0$ and 0 otherwise, the SNU corresponds to the parameter LIF neuron discussed in Subsection 2.3.2. Thus, the SNU in an ANN framework corresponds to that of a LIF neuron in an SNN framework.[167] Woźniak et al. 2020 also propose a sigmoid activation function for $h$, which leads to the soft version of the SNU unit that incorporates biologically inspired neural dynamics of SNNs in ANNs. The most commonly used activation functions can be found in Appendix 3/4. Woźniak et al. 2020 provide an open-source PyTorch and Tensorflow implementation[168] with comprehensive documentation[169].

## 2.4 Neural Turing Machine

Another brain-inspired approach to tackle algorithmic tasks is the Neural Turing Machine (NTM) framework.[170] Derived from the Turing machine concept, it equips neural networks with access to external memory, expanding their problem-solving abilities.[171] Initially proposed by Graves/ Wayne/Danihelka 2014, the NTM introduces a new architecture within the domain of differentiable neural networks.

[160] Cf. Woźniak et al. 2020, p. 327
[161] Cf. Woźniak et al. 2020, p. 327
[162] Cf. Woźniak et al. 2020, p. 326 et seq.
[163] Cf. Woźniak et al. 2020, p. 327
[164] Cf. Woźniak et al. 2020, p. 327
[165] Cf. Woźniak et al. 2020, p. 327
[166] Cf. Woźniak et al. 2020, p. 327
[167] Cf. Woźniak et al. 2020, p. 327
[168] Cf. IBM Research 2024
[169] Cf. IBM Research 2021
[170] Cf. Graves/Wayne/Danihelka 2014, p. 1
[171] Cf. Asaei-Moamam et al. 2023, p. 23

### 2.4.1 High-Level Architecture

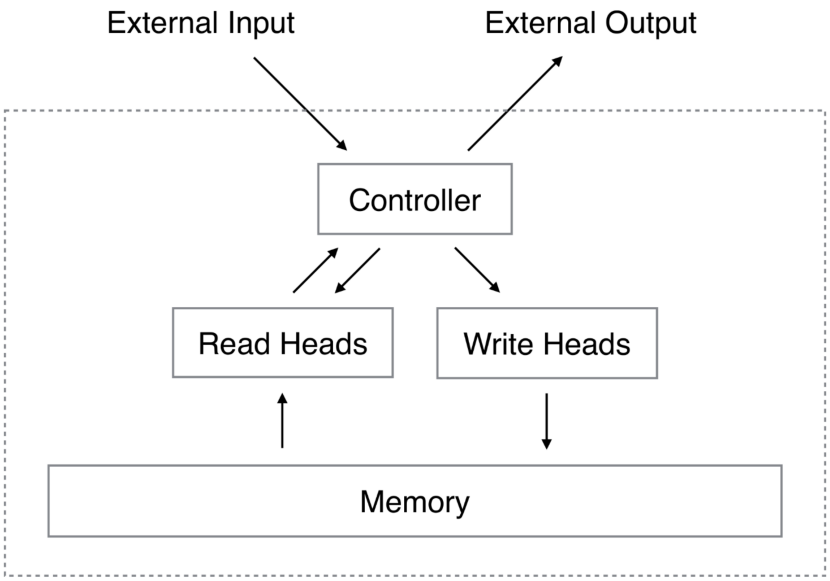



Fig. 6: Neural Turing Machine Architecture[172]

Figure 6 illustrates the typical high-level NTM architecture used, as proposed in the original paper by Graves/Wayne/Danihelka 2014.[173] The main functions of the controller network are read, write, and erase from the memory matrix through the heads.[174] During each update cycle, the controller network receives inputs from an external environment and produces corresponding outputs.[175] The controller governs the behavior of the read and write heads by providing parameters calculated from its output. These parameters are utilized by the heads to compute weights,[176] which are subsequently applied for reading and writing operations on the memory.[177] The memory matrix can be accessed through multiple heads, each capable of executing distinct operations in parallel during every update cycle.[178] Regardless of whether the controller is an RNN, the entire architecture demonstrates recurrent behavior due to the persistent nature of the contents within the memory matrix over time.[179]

All read and write operations on the memory are differentiable due to how the heads perform reading and writing, ensuring scalability regardless of the number of heads present.[180] Thus, the whole NTM architecture is differentiable end-to-end,[181] and can be trained via backpropagation algorithms using input-output examples.[182]

[172]Included in: Graves/Wayne/Danihelka 2014, p. 5
[173]Cf. Greve/Jacobsen/Risi 2016; Faradonbe/Safi-Esfahani 2020; Asaei-Moamam et al. 2023
[174]Cf. Asaei-Moamam et al. 2023, p. 23
[175]Cf. Graves/Wayne/Danihelka 2014, p. 5
[176]Discussed in Subsection 2.4.4 Addressing Mechanisms
[177]Discussed in Subsections 2.4.2 Read Head and Subsection 2.4.3 Write Head
[178]Cf. Faradonbe/Safi-Esfahani 2020, p. 135
[179]Cf. Mark Collier/Beel 2018, p. 95
[180]Cf. Asaei-Moamam et al. 2023, p. 24
[181]Cf. Graves/Wayne/Danihelka 2014, p. 22
[182]Cf. Faradonbe/Safi-Esfahani 2020, p. 135

### 2.4.2 Read Head

The memory matrix $M_t$ at time $t$ has the size of $N \times M$, where $N$ is the number of memory locations, and $M$ is the vector size of each memory location.[183] The vector $w_t$ contains $N$ weights, so one weight for each memory location.[184] The weights satisfy the following conditions:

$$\sum_{i}^{N} w_t(i) = 1, \quad 0 < w_t(i) < 1, \ \forall i \tag{2.17}$$

This formula specifies that the sum of all $N$ weights in $w_t$ equals 1, ensuring that $w_t$ is normalized[185] and all the weights are between 0 and 1.

The vector $r_t$ calculated by the read head at a certain time step $t$ is computed as:[186]

$$\mathbf{r}_t = \sum_{i}^{N} w_t(i) \cdot \mathbf{M}_t(i) \tag{2.18}$$

The resulting vector $r_t$ is the outcome of the sum of the memory matrix rows $M_t(i)$ weighted by the scalar $w_t(i)$ at time $t$. The memory matrix contains $N$ locations for $M$ size vectors,[187] so the resulting vector $r_t$ has the size of $M$, since it is the weighted sum of of the $N$ memory matrix rows.

### 2.4.3 Write Head

The write head, which is responsible for updating the memory matrix $M_t$, is inspired by the forget and update gates of the LSTM.[188] In updating the memory matrix $\mathbf{M}_t$, the write head applies modifications through two principal actions: it uses an erase vector $\mathbf{e}_t$ to remove existing data and utilizes an add vector $\mathbf{a}_t$ to insert new information.[189]

Given a weight vector $\mathbf{w}_t$ and the erase vector $\mathbf{e}_t$, the first step is described by the following formula:[190]

$$\tilde{\mathbf{M}}_t(i) = \mathbf{M}_{t-1}(i) \cdot [\mathbf{1} - w_t(i) \cdot \mathbf{e}_t], \quad 0 \leq e_t(m) \leq 1, \ \forall m \tag{2.19}$$

The old $\mathbf{M}_{t-1}$ is multiplied by the complement of the product of the weighting scalar $w_t(i)$ and the erase element $\mathbf{e}_t$ for each memory location $i$. Therefore, the memory location $\mathbf{M}_t(i)$ is not changed if either the weighting at the location or the erase element is zero.[191] If both are one,

[183] Cf. Mark Collier/Beel 2018, p. 95
[184] Cf. Asaei-Moamam et al. 2023, p. 25
[185] Cf. Graves/Wayne/Danihelka 2014, p. 6
[186] Cf. Graves/Wayne/Danihelka 2014, p. 6
[187] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 333
[188] Cf. Graves/Wayne/Danihelka 2014, p. 6
[189] Cf. Mark Collier/Beel 2018, p. 96
[190] Cf. Graves/Wayne/Danihelka 2014, p. 6
[191] Cf. Asaei-Moamam et al. 2023, p. 24

the memory location is completely erased. All $M$ elements of the erase vector $e_t$ are in range 0 to 1.[192]

After erasing outdated content, the memory matrix $M_t$ is updated with new information. Therefore, each memory location is updated by the following formula:[193]

$$\mathbf{M}_t(i) = \tilde{\mathbf{M}}_t(i) + w_t(i) \cdot \mathbf{a}_t \tag{2.20}$$

The write head adds the vector $a_t$ of length $M$ to the memory matrix $M_t$ weighted by the scalar $w_t(i)$.[194]

This methodical process of erasing outdated content and adding new data can be batched into a singular computation:[195]

$$\mathbf{M}_t(i) = \mathbf{M}_{t-1}(i) \cdot [1 - w_t(i) \cdot \mathbf{e}_t] + w_t(i) \cdot \mathbf{a}_t \tag{2.21}$$

### 2.4.4 Addressing Mechanisms

The preceding subsections, 2.4.2 and 2.4.3, discussed the reading and writing operations, both of which rely on predetermined weights. These weights are determined through specific addressing mechanisms, which are described in this subsection.

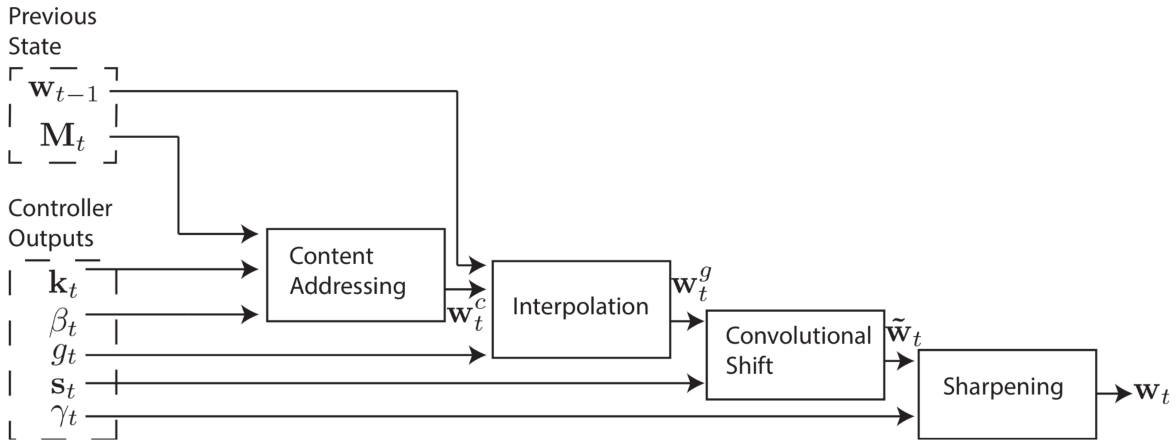



Fig. 7: Flow Diagram of the Addressing Mechanism Generating $\mathbf{w}_t$[196]

The diagram depicted in Figure 7 illustrates the process of the addressing mechanisms tasked with producing the weight vector $\mathbf{w}_t$ at time $t$. This involves a sequential procedure in four steps, where the preceding states and controller outputs are utilized to compute and enhance the weight vector $\mathbf{w}_t$.

[192] Cf. Graves/Wayne/Danihelka 2014, p. 6
[193] Cf. Graves/Wayne/Danihelka 2014, p. 6
[194] Cf. Asaei-Moamam et al. 2023, p. 24
[195] Cf. Greve/Jacobsen/Risi 2016, p. 118
[196] Included in: Graves/Wayne/Danihelka 2014, p. 7

The controller defines a set of parameters, $\mathbf{k}_t, \beta_t \geq 0$, $g_t \in [0,1]$, $\mathbf{s}_t$ s.t. $\sum_k \mathbf{s}_t(k) = 1$ *and* $\forall_k\, s_t(k) \geq 0$, and $\gamma_t \geq 1$. These parameters, obtained from the outputs of the controller network,[197] are then utilized in sequential steps to compute the weight vector $\mathbf{w}_t$.[198]

The **content-based addressing**[199] calculates the weight vector $\mathbf{w}_t^c$ based on the similarity of the key vector $\mathbf{k}_t$ to each memory location.[200] Through the **interpolation** gate, the network is able to determine the extent to which it incorporates the content-based weight vector and the previous weight vector.[201] Additionally, **shifting** allows the weights to be shifted left or right using dedicated network outputs while **sharpening** ensures their focus over time by adjusting them based on a scalar parameter.[202] A comprehensive explanation of these procedures is provided in the following subsection.

In the open-source implementation by Mark Collier/Beel 2018, the constructor of the NTM receives an argument called `addressing_mode (str)`. This argument can be configured to halt operation after content-based addressing.[203]

For the **content-based addressing** mechanism, the weight is calculated for each memory location, based on their similarity to a key vector, given by the controller network.[204] Therefore, each head has a key vector, $\mathbf{k}_t$, which is then compared to each vector $\mathbf{M}_t(i)$ in the memory matrix by a chosen similarity measure $K[\cdot,\cdot]$.[205] This measurement is then weighted by a scalar $\beta_t$ and normalized to produce the weight vector $\mathbf{w}_t^c$, which has $N$ elements.[206]

$$\mathbf{w}_t^c(i) = \frac{\exp\left(\beta_t \cdot K\left[\mathbf{k}_t, \mathbf{M}_t(i)\right]\right)}{\sum_j^N \exp\left(\beta_t \cdot K\left[\mathbf{k}_t, \mathbf{M}_t(j)\right]\right)} \tag{2.22}$$

The weight vector $w_t^c$ allows content-based addressing by storing the similarity between the key vector and each memory location, weighted by the scalar $\beta_t$.[207] The original NTM paper by Graves/Wayne/Danihelka 2014 uses the cosine similarity as a measure of similarity between the key vector $\mathbf{k}_t$ and each vector $\mathbf{M}_t(i)$ within the memory matrix. Cosine similarity serves as an indicator of vector alignment, essentially measuring the extent to which two vectors are oriented in the same direction.[208] The mathematical formula of cosine similarity is defined as:[209]

$$K[\mathbf{u}, \mathbf{v}] = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \cdot \|\mathbf{v}\|} \tag{2.23}$$

[197] Cf. Greve/Jacobsen/Risi 2016, p. 118
[198] Cf. Mark Collier/Beel 2018, p. 95
[199] Cf. Hopfield 1982
[200] Cf. Graves/Wayne/Danihelka 2014, p. 7
[201] Cf. Graves/Wayne/Danihelka 2014, p. 8
[202] Cf. Greve/Jacobsen/Risi 2016, p. 118
[203] Discussed in Subsection 2.5.2 Similar Artifacts
[204] Cf. Greve/Jacobsen/Risi 2016, p. 118
[205] Cf. Graves/Wayne/Danihelka 2014, p. 7
[206] Cf. Graves/Wayne/Danihelka 2014, p. 7
[207] Cf. Mark Collier/Beel 2018, p. 96
[208] Cf. Han/Kamber/Pei 2012, p. 77
[209] Cf. Han/Kamber/Pei 2012; Graves/Wayne/Danihelka 2014

The calculated weight vector $\mathbf{w}_t^c$ is then interpolated with the previous weight vector $\mathbf{w}_{t-1}$, based on the value ($g_t$) of an **interpolation**[210] **gate**.[211] This results in a gates weight $\mathbf{w}_t^g$ defined by the following formula:[212]

$$\mathbf{w}_t^g = g_t \mathbf{w}_t^c + (1 - g_t)\mathbf{w}_{t-1}, \quad 0 \leq g_t \leq 1 \tag{2.24}$$

The controller determines whether to use the weight $\mathbf{w}_t^c$ partially, completely, or not at all, thereby allowing flexibility in weight application.[213] The first component, $g_t \mathbf{w}_t^c$, controls the extent to which the current content-based weight vector $\mathbf{w}_t^c$ is utilized.[214] In contrast, the second component, $(1 - g_t)\mathbf{w}_{t-1}$, controls the usage level of the previous weight vector $\mathbf{w}_{t-1}$.[215] This gated interpolation mechanism grants the controller the discretion to choose between the current content-based weights and the preceding weight vector for optimal performance.[216]

After the interpolation, the **convolutional shift** is applied.[217] The process of updating all weights through shifting can be described as performing a convolution operation over all weights, updating them based on the emitted shift weighting.[218] In scenarios where the permitted shifts range from $-1$ to 1, the shift vector $s_t$ comprises three components that represent the magnitudes of shifts by $-1$, 0, and 1, respectively.[219] To accurately specify the shift weightings, one efficient approach is the adoption of a softmax layer, tailored in size to align with the controller.[220] The adjustment of $\mathbf{w}_t^g$ by $\mathbf{s}_t$ cross an array of $N$ memory locations, be mathematically modeled as a circular convolution:[221]

$$\tilde{w}_t(i) = \sum_{j}^{N-1} w_t^g(j) \cdot s_t(i - j) \tag{2.25}$$

This enables iteration through memory by applying a 1-D convolutional shift kernel to the current weights.[222] This convolution operation can cause leakage or dispersion of weightings over time if the shift weighting is not sharp.[223]

To address this issue, the resulting weights $\tilde{\mathbf{w}}_t$ get **sharpened** based on an additional scalar

[210] Interpolation is a mathematical technique used to estimate new data points within a range of a discrete set of known data points, used to predict or smooth data. Caruso/Quarta 1998, p. 109
[211] Cf. Greve/Jacobsen/Risi 2016, p. 118
[212] Cf. Graves/Wayne/Danihelka 2014, p. 8
[213] Cf. Greve/Jacobsen/Risi 2016, p. 118
[214] Cf. Graves/Wayne/Danihelka 2014, p. 8
[215] Cf. Graves/Wayne/Danihelka 2014, p. 8
[216] Cf. Mark Collier/Beel 2018, p. 96
[217] Cf. Graves/Wayne/Danihelka 2014, p. 8
[218] Cf. Greve/Jacobsen/Risi 2016, p. 118
[219] Cf. Graves/Wayne/Danihelka 2014, p. 8
[220] Cf. Graves/Wayne/Danihelka 2014, p. 8
[221] Cf. Graves/Wayne/Danihelka 2014, p. 9
[222] Cf. Mark Collier/Beel 2018, p. 96
[223] Cf. Graves/Wayne/Danihelka 2014; Greve/Jacobsen/Risi 2016

parameter $\gamma_t$ to produce the final weights $\mathbf{w}_t$.[224] Thus, $\mathbf{w}_t$ is given by the following formula:[225]

$$w_t(i) = \frac{\tilde{w}_t(i)^{\gamma_t}}{\sum_j^N \tilde{w}_t(j)^{\gamma_t}}, \qquad \gamma_t \geq 1 \tag{2.26}$$

This adjustment involves raising the preliminary weighting to the power of $\gamma_t$ and then normalizing the result across all $N$ memory locations. As $\gamma_t$ increases, the weights become sharper around the memory locations with higher initial weightings and, therefore, corrected from any blurring occurring as a result of the convolution operation.[226]

Following the completion of these four sequential steps, the final weight vector $\mathbf{w}_t$ is obtained, which is then utilized for the read and write operations.[227]

### 2.4.5 Controller Network

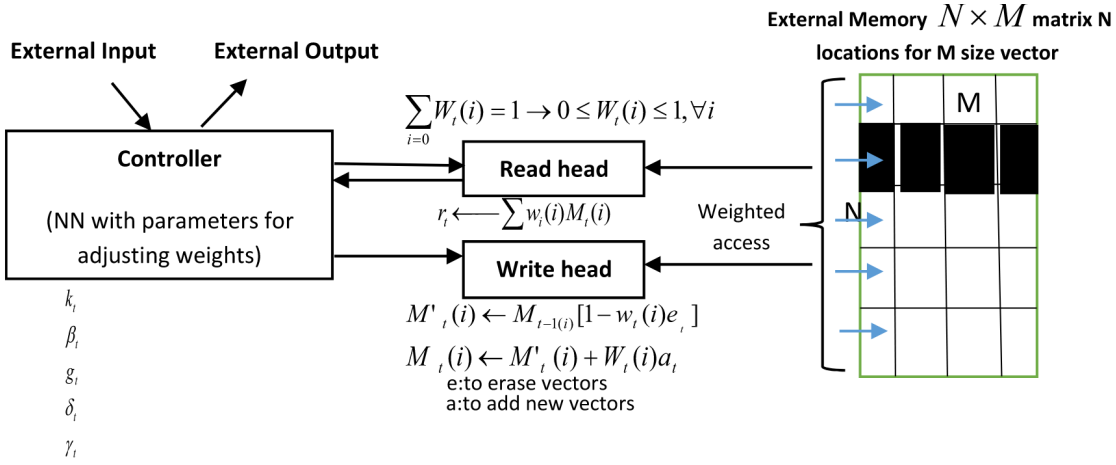



Fig. 8: Controller Access to External Memory[228]

Figure 8 illustrates the conceptual framework depicting how the controller interacts with the memory via the heads. The controller accesses the memory matrix through the read and write heads by providing parameters used for adjusting weights, thereby controlling the operations executed by the heads.[229]

The controller network has the flexibility to be either an RNN or a feed-forward network.[230] Given the persistent nature of the memory matrix, the NTM inherently exhibits recurrent behavior.[231] The accuracy of the NTM is significantly influenced by the complexity of the controller network.[232] Thus, the controller stands out as the most crucial hyperparameter in the entire NTM

[224] Cf. Greve/Jacobsen/Risi 2016, p. 118
[225] Cf. Graves/Wayne/Danihelka 2014, p. 9
[226] Cf. Mark Collier/Beel 2018, p. 96
[227] Cf. Greve/Jacobsen/Risi 2016, p. 118
[228] Included in: Asaei-Moamam et al. 2023, p. 24
[229] Cf. Asaei-Moamam et al. 2023, p. 24
[230] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 135
[231] Cf. Mark Collier/Beel 2018, p. 95
[232] Cf. Zaremba/Sutskever 2015, p. 8

framework.[233]

## 2.5 Review of Neural Turing Machine Artifacts

In this section, the NTM artifacts are reviewed and compared. Two identified types of artifacts are distinguished: those focusing on improving the NTM architecture and those applying the NTM to real-world use cases. This section primarily focuses on the NTM enhancements. Before reviewing the artifacts, some popular tasks commonly used to evaluate the capabilities of the NTM are presented.

### 2.5.1 Evaluation Tasks

This subsection highlights the primary tasks critical for demonstrating the NTM's effectiveness. An extensive literature review[234] has identified several tasks frequently used to evaluate the NTM architecture's performance. These encompass basic memory operations such as *copy* and *repeat copy*, as well as more complex tasks like *associative recall* and *reverse*. Additionally, advanced challenges, including the *bAbI dataset* for reasoning and comprehension and the *sequential MNIST* for image processing, are also employed. The initial NTM paper by Graves/ Wayne/Danihelka 2014 identifies fundamental sequential tasks such as *copy*, *repeat copy*, and *associative recall* as key benchmarks for evaluating the model's performance.

The *copy task* serves as a fundamental benchmark to assess the NTM proficiency in storing and accurately retrieving information.[235] In this task, the NTM is presented with a sequence of random binary vectors, which it is required to duplicate exactly after a delay.[236] The operation can be mathematically represented as $x_1x_2, x_3 \dots x_n \rightarrow x_1x_2x_3 \dots x_n$, where $x_i$ denotes the individual bits within the sequence, and $n$ represents the sequence's total length.[237] This task assesses the capability of the NTM architecture to accurately store and retrieve information, thus evaluating its core memory functions: the processes of memory writing and subsequent unaltered reading.[238] Such evaluation is crucial for testing the fundamental memory functionalities inherent to the NTM design.[239]

The *repeat copy task* extends the complexity of the *copy task* by obligating the NTM to duplicate the input sequence several times.[240] This enhancement aims to assess whether the NTM can comprehend and execute a simple nested function, thereby increasing the task's complexity.[241]

[233] Cf. Graves/Wayne/Danihelka 2014, p. 10
[234] Discussed in Section 2.5.2 Similar Artifacts
[235] Cf. Faradonbe/Safi-Esfahani 2020, p. 136
[236] Cf. Zaremba/Sutskever 2015, p. 7
[237] Cf. Yang/Rush 2017, p. 7
[238] Cf. Graves/Wayne/Danihelka 2014, p. 10
[239] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 11
[240] Cf. Zaremba/Sutskever 2015, p. 7
[241] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 11

The operation can be mathematically described as: $Mx_1 \dots x_n \rightarrow x_1 \dots x_n \dots x_1 \dots x_n$ (repeated $M$ times), where $x_i$ signifies the individual bits in the sequence, $n$ is the sequence's total length, and $M$ denotes the repetition count.[242]

The *reverse task* challenges the NTM by requiring it to reverse the input sequence, building upon the basic *copy task*. This heightened complexity aims to evaluate the ability of the NTM to handle rightward orientation.[243] Mathematically, the task involves transforming an input sequence $x_1x_2 \dots x_n$ into its reverse order $x_nx_{n-1} \dots x_1$, with $x_i$ representing individual bits and $n$ indicating the sequence length.[244]

The previously mentioned tasks evaluate the storing and the retrieval of information, and the ability to handle simple nest functions and rightward orientation. The next order of complexity is to handle organizing data with "indirection", which is when one data item points to another.[245] The *associative recall task* involves random bit vectors organized into items of $N \times M$ matrices.[246] The network is trained to predict the next item in the input sequence after receiving a query item. The correct output is the item after the query item.[247] The task can be mathematically represented as: $X_1X_2X_3 \dots X_n, X_q \rightarrow X_{q+1}$, where $X_i$ denotes the individual items in the sequence, $n$ represents the total number of items, and $q$ signifies the query item.[248]

Beyond the simpler algorithmic tasks, NTM architectures are also evaluated on more complex challenges, including the *bAbI dataset* for natural language processing and the *sequential MNIST* task for image data processing.[249] The *bAbI dataset*, proposed by Facebook AI researchers Weston et al. 2015, serves as a benchmark dataset aimed at evaluating the reasoning and comprehension capabilities of machine learning models. The dataset tests on 20 tasks, each tailored to address a specific aspect of reasoning such as counting, pathfinding, and deduction.[250] Training models on these tasks can be facilitated through the generation of multiple training questions, utilizing the open-source code provided by the original authors.[251] The *sequential MNIST task* involves processing image data, requiring the NTM to predict the digit label from a sequence of pixels.[252] There are further tasks beyond the focus of this thesis. Malekmohamadi Faradonbe/ Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 11 et seqq. provide a comprehensive overview of the tasks on which various NTMs have been tested and the associated task complexities.

[242]Cf. Yang/Rush 2017, p. 7
[243]Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 12
[244]Cf. Yang/Rush 2017, p. 7
[245]Cf. Graves/Wayne/Danihelka 2014, p. 14
[246]Cf. Graves/Wayne/Danihelka 2014, p. 14
[247]Cf. Mark Collier/Beel 2018, p. 14
[248]Cf. Yang/Rush 2017, p.
[249]Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 11 et seqq.
[250]Cf. Weston et al. 2015, p. 3 et. seqq.
[251]Cf. Facebook Archive 2017
[252]Cf. Chandar et al. 2016, p. 16

### 2.5.2 Similar Artifacts

Since the initial publication by Graves/Wayne/Danihelka 2014, the NTM framework has seen significant advancements proposed by various institutions, including Google DeepMind[253], IBM[254] and Harvard University[255].

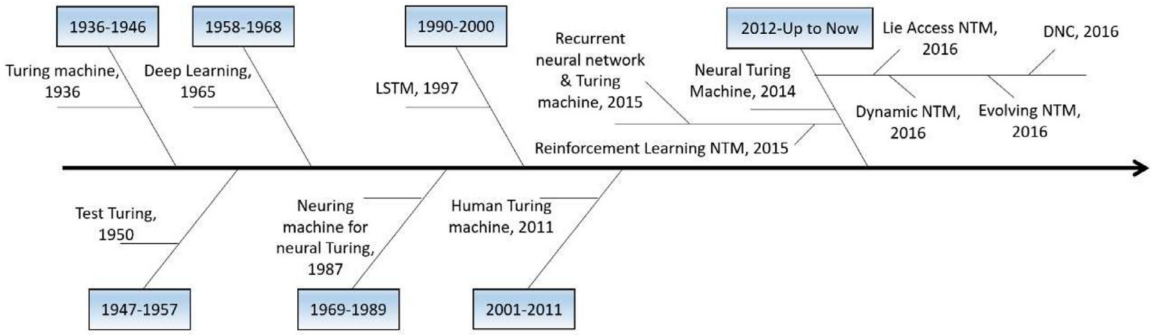



Fig. 9: NTM Timeline Activities[256] (Current State of 2019)

Figure 9 presents a comprehensive overview of the NTM framework evolution, showcasing the multiple enhancements that have arisen since its initial development. Additionally, it illustrates the evolution of foundational frameworks and concepts essential for the original NTM, including the Turing machine, Deep Learning principles, and the LSTM architecture. The literature review conducted in this section will explain these enhancements in detail and provide a more comprehensive overview of the most relevant artifacts.

Table 1 depicts a concept matrix featuring the identified artifacts in the field of NTM.[257] It categorizes the different controller networks utilized in the papers and specifies the tasks on which the NTM is tested, sorted by citations. The table is divided into articles that focus on improving the NTM and articles that apply the NTM to real-world use cases. Table 1 shows that the papers focusing on improving the NTM predominantly test their enhancements on similar tasks, with the copy task being the most commonly evaluated. Additionally, the repeat copy and associative recall tasks are also frequently tested. Furthermore, the reverse copy task is examined in some papers. These simple sequential tasks emerge as the most popular benchmarks for assessing the functionality of the NTM.

The following section discusses the relevant NTM papers in detail, focusing solely on the description and hyperparameters provided in the papers. Available open-source implementations and the corresponding hyperparameters are analyzed independently and listed separately in Appendix 4.

[253]Cf. Wayne et al. 2016
[254]Cf. Zhang, W./Yu/Zhou 2015
[255]Cf. Yang/Rush 2017
[256]Included in: Faradonbe/Safi-Esfahani 2020, p. 137
[257]Cf. Webster/Watson 2002, p. 17

| **Articles: NTM Enhancements** | **Concepts** | | | | | | | | Cits.[258] |
|---|---|---|---|---|---|---|---|---|---|
| | Controler Network | | | Tasks | | | | | |
| | **LSTM** | **RNN** | **OC** | **C** | **RepC** | **AR** | **RevC** | **OT** | |
| Graves/Wayne/ Danihelka 2014 | X | | | X | X | X | | | 2917 |
| Wayne et al. 2016 | X | | | | | | | X | 1826 |
| Zaremba/ Sutskever 2015 | X | | | X | X | | X | X | 297 |
| Danihelka et al. 2016 | X | | | X | | | | X | 190 |
| Rae et al. 2016 | X | | | X | | X | | | 175 |
| Gülçehre/Chandar/ Bengio 2017 | X | | | X | X | X | | X | 72 |
| Mark Collier/Beel 2018 | X | | | X | X | X | | | 60 |
| Chandar et al. 2016 | | X | X | X | X | X | | X | 59 |
| Greve/Jacobsen/ Risi 2016 | | | X | X | | | | X | 49 |
| Yang/Rush 2017 | X | | | X | X | | X | X | 21 |
| Boloukian/Safi-Esfahani 2020 | | | X | | | | | X | 19 |
| Zhang, W./Yu/Zhou 2015 | X | | | X | | X | | | 18 |
| Merrild/Rasmussen/ Risi 2018 | | | X | X | | | | | 14 |
| Malekmohamadi Faradonbe/Safi-Esfahani/ Karimian-kelishadrokhi 2020 | X | | | X | X | X | X | X | 10 |
| **Articles: NTM Applications** | | | | | | | | | |
| Tkačík/Kordík 2016 | X | | | | | | | X | 34 |
| Zheng, Z./Wu/Weng 2019 | | | X | | | | | X | 24 |
| Faradonbe/Safi-Esfahani 2020 | X | | | | | | | X | 11 |
| Falcon et al. 2022 | X | | | | | | | X | 5 |

Tab. 1: Concept Matrix of Relevant Artifacts Based on Webster/Watson 2002; Legend: OC (Other Controller), C (Copy), RepC (Repeat Copy), AR (Associative Recall), RevC (Reverse Copy), OT (Other Tasks), Cits. (Citations), retrieved from Google Scholar, accessed on March 18, 2024;

Graves/Wayne/Danihelka 2014 initiated the first NTM in their paper by testing a set of elementary tasks, including copy, repeat copy, and associative recall, as well as more intricate challenges such as N-Gram and Priority Sort.[259] All subsequent developments are based upon this first NTM architecture. This architecture incorporates an LSTM and a feed-forward controller network.[260] Despite lacking publication in a journal, their work stands as the most cited paper in the field of NTM, underscoring its relevance. All networks featured logistic sigmoid output layers, trained with a cross-entropy objective function, and the sequence prediction errors are reported in bits per sequence.[261] They employed the RMSProp algorithm with a momentum of 0.9. The LSTM controller network comprised 3 stacked hidden layers and 100 controller units per stacked layer for the elementary tasks.[262] For the more complex tasks, the number of units was increased to 512, and 8 heads were used.[263] Given their experimentation with 20 bits per input and 128 memory locations, the memory matrix was configured as 128x20. For the copy and repeat copy tasks, they utilized one head for each operation, while for the associative recall task, 4 heads were employed. The learning rate used for these tasks was $10^{-4}$.[264] Despite not being official, there are numerous open-source implementations of the NTM architecture available on GitHub.[265]

Wayne et al. 2016 proposed the Differentiable Neural Computer (DNC), which enhances the original NTM architecture with a more sophisticated LSTM controller architecture and a more complex memory access mechanism, which can also link memory locations together.[266] The DNC paper has 1826 citations and is the second most cited paper in the field and was published in the journal "Nature". The DNC is tested using graph experiments, the bAbI dataset, and block puzzle experiments.[267] Wayne et al. 2016 used up to 512 memory locations. An open-source implementation of the DNC is available on GitHub.[268]

Zaremba/Sutskever 2015 proposed an NTM architecture featuring an LSTM controller that is trainable through reinforcement learning. This Reinforcement Neural Turing Machine (R-NTM) underwent testing across tasks, including copy, repeat copy, and reverse.[269] The model was trained using a bit set of size 30, employing a stochastic gradient descent, learning rate of 0.05 with a fixed momentum of 0.9, and a batch size of 200.[270] Zaremba/Sutskever 2015 initialized with a Gaussian distribution with a standard deviation of 0.1. Initially, the 35 memory locations were instantiated with 0.[271] However, either the model parameters were kept under 20,000, or the update process failed after an arbitrary number of steps.[272] The open-source implementation

[259]Cf. Graves/Wayne/Danihelka 2014, p. 10 et seqq.
[260]Cf. Graves/Wayne/Danihelka 2014, p. 10
[261]Cf. Graves/Wayne/Danihelka 2014, p. 21
[262]Cf. Graves/Wayne/Danihelka 2014, p. 21 et seq.
[263]Cf. Graves/Wayne/Danihelka 2014, p. 22 et seq.
[264]Cf. Graves/Wayne/Danihelka 2014, p. 22
[265]Cf. Snipsco 2015; Chiggum 2016; Yeoedward 2016; Camigord 2017; Loudinthecloud 2018; Snowkylin 2019
[266]Cf. Faradonbe/Safi-Esfahani 2020, p. 137
[267]Cf. Wayne et al. 2016, p. 472 et seqq.
[268]Cf. Google DeepMind 2024
[269]Cf. Zaremba/Sutskever 2015, p. 7
[270]Cf. Zaremba/Sutskever 2015, p. 9
[271]Cf. Zaremba/Sutskever 2015, p. 9
[272]Cf. Zaremba/Sutskever 2015, p. 9

of the R-NTM can be accessed on GitHub.[273]

Danihelka et al. 2016 enhanced the LSTM controller to enable the storage of key-value pairs. While not directly constituting an NTM architecture, it serves as an enhancement of LSTM memory that competes with the original NTM architecture. Nevertheless, it remains relevant for the MANN field, demonstrating how memory enhancements can be tested and validated. They experimented by taking a sequence of ImageNet images[274] and feeding them into the network, essentially performing a more complex copy task. The optimization was conducted using the ADAM optimizer without gradient clipping.[275] Danihelka et al. 2016 utilized 128 to 512 LSTM units, and their test models had up to 1,256,691 trainable parameters.[276]

Rae et al. 2016 trained an Sparse Access Memory (SAM), which enhances the NTM architecture by introducing a sparse read-and-write mechanism. This leads to a more efficient memory access mechanism, which can be used with much larger memory sizes. They tested their model on the copy task with up to 20 random bits of input. Additionally, the model was also trained on the associative recall and Priority Sort tasks. They selected these tasks to demonstrate that their model can compete with the traditional NTM architecture.[277] Furthermore, Rae et al. 2016 evaluated their model on the bAbI dataset and other non-synthetic large-scale sequence datasets.[278] Rae et al. 2016 only describe the concepts of their model without specifying the hyperparameters. However, there is an open-source implementation of the SAM on GitHub by different authors, where experiment parameters can be found.[279]

Gülçehre/Chandar/Bengio 2017 propose a MANN with wormhole connections called "Temporal Automatic Relation Discovery in Sequences". They enhance the NTM and DNC by storing connection information in the memory and using it to improve the memory access mechanism.[280] Gülçehre/Chandar/Bengio 2017 train on the original NTM tasks with 16 memory locations (N), each with a size of 32 (M), and 120 hidden units per layer. They employ the Adam optimizer with a $3 \times 10^{-3}$ learning rate.[281] Additionally, to prove the stability on large sequential tasks, they test their model on natural language inference and the sequential MNIST task.[282]

The work of Mark Collier/Beel 2018 contributes to enhancing the NTM architecture through a thorough examination of constant, random, and learned instantiations of the memory. The constant initialization, with a value of $10^{-6}$, learned initialization through backpropagation, and the random initialization, modeled as a Normal distribution with a mean of 0 and standard deviation of 0.5.[283] Their research findings suggest that, particularly in sequential tasks, the con-

[273] Cf. Ilyasu123 2024
[274] Cf. Russakovsky et al. 2015
[275] Cf. Danihelka et al. 2016, p. 4
[276] Cf. Danihelka et al. 2016, p. 8
[277] Cf. Rae et al. 2016, p. 6
[278] Cf. Rae et al. 2016, p. 7
[279] Cf. Ixaxaar 2024
[280] Cf. Gülçehre/Chandar/Bengio 2017, p. 1
[281] Cf. Gülçehre/Chandar/Bengio 2017, p. 21
[282] Cf. Gülçehre/Chandar/Bengio 2017, p. 19 et seqq.
[283] Cf. Mark Collier/Beel 2018, p. 97

stant initialization scheme demonstrates the most rapid learning.[284] Furthermore, they employ backpropagation through the initialization to refine other read and write parameters, denoted as $\mathbf{r}_0$ and $\mathbf{w}_0$, thereby establishing a specific initialization scheme.[285] Additionally, to enhance the architecture, the authors address the challenge of unstable training, reported by some open-source implementations,[286] aiming to provide a new, open-source implementation of the NTM with a more stable training process. Mark Collier/Beel 2018 effectively addresses the issue of gradients becoming NaN during training. To evaluate their implementation, the authors utilize both the official DNC implementation and a traditional LSTM network as benchmarks.[287] Their experimental MANN setup includes one read and one write head, with a memory size of $128 \times 20$, and an LSTM controller comprising 100 units in one layer.[288] The benchmark LSTM network consists of 3 layers, each containing 256 units.[289] All networks undergo training utilizing the ADAM optimizer with a learning rate set at 0.001.[290] Mark Collier/Beel 2018 evaluate their enhancements across three initial NTM tasks: copy, repeat copy, and associative recall.[291] Both copy tasks entail the use of 8-bit random vectors. In the repeat copy task, these vectors are repeated from 1 to 10 times. The associative recall task employs $3 \times 6$ dimensional random vectors. They conducted the tests 10 times, with each memory instantiation schema.[292] Their network demonstrates faster convergence across all tasks compared to the classical LSTM, albeit being 1.0 - 1.2 times slower than the official DNC implementation.[293] The open-source implementation from Mark Collier/Beel 2018 can be found on GitHub.[294]

Chandar et al. 2016 propose a Dynamic Neural Turing Machine (D-NTM) with both differentiable continuous and non-differentiable discrete read/write mechanisms, thereby enhancing the D-NTM to learn a wide variety of addressing strategies. The continuous part of the D-NTM is trained using backpropagation with stochastic gradient descent, while the discrete part is trained via reinforcement learning.[295] Chandar et al. 2016 evaluate the D-NTM performance on various tasks, including the bAbI dataset, sequential MNIST, and the initial copy and associative recall tasks. For the initial NTM tasks, the setup includes the utilization of a GRU controller with 100 units.[296] Chandar et al. 2016 establish the success criterion for their model, achieved when the cost function (binary cross-entropy) falls below 0.2.[297] They employ the ADAM optimizer with a learning rate of $1 \times 10^{-3}$.[298] While the memory size for the other tasks is specified as $128 \times 20$, it is not explicitly mentioned for the initial NTM tasks.[299]

[284] Cf. Mark Collier/Beel 2018, p. 103
[285] Cf. Mark Collier/Beel 2018, p. 97
[286] Cf. Snipsco 2015; Carpedm20 2015; Camigord 2017
[287] Cf. Mark Collier/Beel 2018, p. 99
[288] Cf. Mark Collier/Beel 2018, p. 99
[289] Cf. Mark Collier/Beel 2018, p. 99
[290] Cf. Mark Collier/Beel 2018, p. 99
[291] Cf. Mark Collier/Beel 2018, p. 98
[292] Cf. Mark Collier/Beel 2018, p. 99
[293] Cf. Mark Collier/Beel 2018, p. 101 et seq.
[294] Cf. MarkPKCollier 2018
[295] Cf. Chandar et al. 2016, p. 7
[296] Cf. Chandar et al. 2016, p. 19
[297] Cf. Chandar et al. 2016, p. 19
[298] Cf. Chandar et al. 2016, p. 19
[299] Cf. Chandar et al. 2016, p. 11

Greve/Jacobsen/Risi 2016 developed the Evolving Neural Turing Machine (E-NTM), which features a unique attention mechanism that enables the network to selectively focus on specific parts of the memory. Thus, the architecture does not require accessing the entire memory content at each time step.[300] The E-NTM is originally tested on the copy task and a more complex reinforcement-like task known as the continuous double T-Maze.[301] For the copy tasks, the parameters include 25 memory locations.[302] The vector size of a memory location is determined by the sequence length of the bit vectors to be copied.[303] Greve/Jacobsen/Risi 2016 utilized 8-bit random vectors for the copy task.[304] An open-source implementation is provided on GitHub.[305]

Yang/Rush 2017 introduce a Lie Access NTM using an addressing algorithm based on Lie group theory. This algorithm has a specific focus on shifting, aiming to address the lack of relative indexing in the original NTM.[306] Their enhancements are evaluated through testing on tasks including copy, repeat copy, reverse, and several other simple sequential tasks.[307] For the copy, repeat copy, and reverse tasks, they compare their model against a benchmark LSTM with 1 to 4 layers, each containing 256 cells. In contrast, the enhanced NTM features a single-layer LSTM controller with 50 cells and a memory size of $128 \times 20$. Yang/Rush 2017 provide an official open-source implementation on GitHub.[308]

Boloukian/Safi-Esfahani 2020 tackle a classification task using an NTM, employing various autoencoder architectures as the controller network. Although the specifics of the training parameters are not relevant to this thesis, their significant contribution lies in demonstrating the feasibility of employing entirely different controller architectures constructed from deep neural networks.[309]

Zhang, W./Yu/Zhou 2015 propose several different memory structures[310] to address the challenges of overfitting and the occasional failure of the NTM model to converge to the optimal solution. They evaluate their approaches on tasks, including copy and associative recall. For the copy task, binary vectors of length 8 are utilized. The training criterion is binary cross-entropy, with RMSProp employed for optimization, using a learning rate of $1 \times 10^{-4}$, momentum of 0.9, and decay of 0.95. Zhang, W./Yu/Zhou 2015 observe that employing 2 heads per operation leads to faster convergence compared to using only 1 head per operation.[311]

Merrild/Rasmussen/Risi 2018 extend the E-NTM architecture, introduced by Greve/Jacobsen/Risi 2016, by refining the memory access algorithm, allowing for improved memory accessibility

[300] Cf. Greve/Jacobsen/Risi 2016, p. 1
[301] Cf. Greve/Jacobsen/Risi 2016, p. 120 et seqq.
[302] Cf. Greve/Jacobsen/Risi 2016, p. 119
[303] Cf. Greve/Jacobsen/Risi 2016, p. 119
[304] Cf. Greve/Jacobsen/Risi 2016, p. 119
[305] Cf. Rasmusgreve 2015
[306] Cf. Yang/Rush 2017, p. 1
[307] Cf. Yang/Rush 2017, p. 7
[308] Cf. HarvardNLP 2017
[309] Cf. Boloukian/Safi-Esfahani 2020, p. 197
[310] Cf. Zhang, W./Yu/Zhou 2015, p. 2
[311] Cf. Zhang, W./Yu/Zhou 2015, p. 4

even with larger memory sizes and sequences.[312] They evaluate their approach primarily on the initial copy task,[313] and conduct tests using 100 random bit vectors of random lengths ranging from 1 to 10.[314] Moreover, the model is assessed on sequences with lengths up to 1000, demonstrating superior performance compared to the E-NTM.[315] While additional hyperparameters are not outlined in the paper, the open-source implementation can be accessed on GitHub.[316]

Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020 present a comprehensive literature review on the NTM and its advancements, offering insights into the various tasks and applications it has been tested on, including the copy, repeat copy, associative recall, and reverse tasks.[317] Furthermore, Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020 conduct experiments on different types of NTM architectures, focusing on the initial and several simple sequential tasks. They compare the performance of a classical NTM with an LSTM and a feed-forward controller network. The memory size is set to 128×20, and the controller comprises 100 units in one layer with 4 heads per operation.[318] This applies to the copy and the repeat copy task. However, for other tested tasks, the controller size can vary.[319] The experiments employ the RMSProp optimizer with 0.9 momentum and a learning rate of $10^{-4}$.[320] For the benchmark LSTM network, they utilize a setup consisting of 3 layers with 256 units each, and the learning rate is set to $3 \times 10^{-5}$ for the copy and repeat copy tasks.[321] Additionally, the reverse task is evaluated against a benchmark LSTM network with 4 layers and 256 units per layer.[322]

The NTM architectures are also applied to real-world use cases. Tkačík/Kordík 2016 employ an NTM architecture to predict human mobility patterns. Their setup includes an LSTM controller with 75 units, a memory size of 32×16, and one head for each operation.[323] They utilize the RMSProp optimizer with 0.9 momentum and a learning rate of $10^{-4}$.[324] Other NTM applications are the prediction of the remaining useful life of machinery[325], classification tasks[326], and visual navigation[327].

In addition to the conducted literature review, various open-source implementations of the NTM architecture are analyzed. This analysis includes both official implementations provided by the authors of the respective papers and unofficial implementations. Details of the review are available in Appendix 4 and will be discussed in Subsection 4.2.3.

[312] Cf. Merrild/Rasmussen/Risi 2018, p. 761
[313] Cf. Merrild/Rasmussen/Risi 2018, p. 755 et seq.
[314] Cf. Merrild/Rasmussen/Risi 2018, p. 765 et seq.
[315] Cf. Merrild/Rasmussen/Risi 2018, p. 765 et seqq.
[316] Cf. Jakobmerrild 2017
[317] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 11
[318] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 14
[319] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 18
[320] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 14
[321] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 18
[322] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 18
[323] Cf. Tkačík/Kordík 2016, p. 2795
[324] Cf. Tkačík/Kordík 2016, p. 2795
[325] Cf. Falcon et al. 2022
[326] Cf. Faradonbe/Safi-Esfahani 2020
[327] Cf. Zheng, Z./Wu/Weng 2019

# 3 Objective Specification and Research Methodology

The literature review in Chapter 2 discusses key concepts, their theoretical foundations, and NTM artifacts. Subsequently, this chapter specifies the research objectives and goals of this thesis and outlines the research design to achieve these goals.

## 3.1 Objective Specification

As discussed in Subsection 1.1, SNNs are increasingly important in machine learning due to their efficient hardware,[328] positioning them as a competitive third-generation neural network alongside traditional ANNs.[329] Their ability to solve complex cognitive tasks highlights their significance.[330] A crucial aspect of addressing cognitive challenges is the storage and access of information,[331] which NTM architectures facilitate by enhancing neural networks with an additional memory matrix controlled by a dedicated controller network.[332] Current literature primarily employs RNN architectures like LSTM and GRU as NTM controllers.[333] However, NTM performance strongly depends on the controller type,[334] and exploration of alternative architectures like SNNs remains limited. Exploring different controllers could enhance NTM performance and open new pathways.[335] Inspired by the brain's processing methods,[336] SNN architectures offer a novel approach to potentially improve the NTM framework. The SNU, capable of simulating SNNs within a traditional ANN framework,[337] is a promising controller candidate.

This leads to the following research questions: **Is it feasible to employ an SNN as an NTM controller network?** Additionally, **how does the resulting SNTM perform compared to classical NTM and LSTM architectures?** This inquiry also involves examining the **possible limitations** and **advantages** of the **SNTM**.

The **goal** of this thesis is to **develop** an NTM with an SNN as its controller, referred to as **Spiking Neural Turing Machine (SNTM)**. The SNN controller network is simulated using SNUs.[338] The developed artifact is **benchmarked against** a classical **LSTM network** and an **NTM with an LSTM controller** on three selected tasks: **copy**, **reverse**, and **repeat copy**. The critical selection of benchmark networks, their hyperparameters, and the tasks employed are derived from the literature and discussed in Section 4.2.

[328] Cf. Zheng, H. et al. 2024, p. 10 et seq.
[329] Cf. Maass 1997
[330] Cf. Gorgan Mohammadi/Ganjtabesh 2024
[331] Cf. Greve/Jacobsen/Risi 2016, p. 117
[332] Cf. Graves/Wayne/Danihelka 2014, p. 2
[333] Discussed in Section 2.5 Review of Neural Turing Machine Artifacts
[334] Cf. Zaremba/Mikolov, et al. 2016, p. 8
[335] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 111
[336] Cf. Yamazaki et al. 2022, p. 1
[337] Cf. Woźniak et al. 2020, p. 326
[338] Discussed in Subsection 2.3.3 Spiking Neural Units

## 3.2 Research Design

To achieve the specified research goal, the research methodology employed in this thesis is based on the DSR framework, as proposed by Gregor/Hevner, A. 2013. One concrete definition of DSR is provided by Hevner, A. R./Chatterjee 2010:

> "DSR is a research paradigm in which a designer answers questions relevant to human problems via the creation of innovative artifacts, thereby contributing new knowledge to the body of scientific evidence. The designed artifacts are both useful and fundamental in understanding that problem."[339]

Examples of innovative artifacts, which are artificial objects or processes,[340] are detailed in Appendix 1/2, presenting various types that contribute to the existing knowledge base.[341] One significant contribution can be the proof of concept for a new idea or demonstrating the feasibility of a novel concept.[342] Thus, the DSR framework is well-suited for designing and implementing new machine learning prototypes[343] due to its iterative nature and the comprehensive phases that need to be conducted.[344] The design and evaluation of the artifact require support from additional research methodologies, including tests of validity using test data, scenario analyses, and controlled experiments.[345] Therefore, the DSR framework is enhanced with various research methodologies,[346] which are detailed as they are applied throughout this thesis.

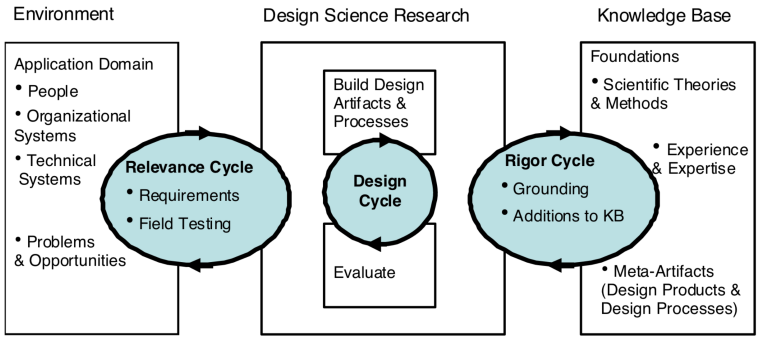



Fig. 10: Design Science Research Cycles[347]

As depicted in Figure 10, the iterative DSR framework consists of three interlinked cycles: Rigor, Design, and Relevance. The Rigor Cycle involves analyzing and contributing to the existing knowledge, the current literature.[348] This cycle provides the scientific background by exploring the state of the art in the application domain and exploring existing artifacts.[349] Additionally,

[339] Hevner, A. R./Chatterjee 2010, p. 5
[340] Cf. Gregor/Hevner, A. 2013, p. 341
[341] Cf. Gregor/Hevner, A. 2013, p. 344
[342] Cf. Gregor/Hevner, A. 2013, p. 351 et seqq.
[343] Cf. Ostheimer/Chowdhury/Iqbal 2021; Muntean/Militaru 2022; Welsch/Kowalczyk 2023; Te'eni et al. 2023
[344] Cf. Gregor/Hevner, A. 2013, p. 350
[345] Cf. Gregor/Hevner, A. 2013, p. 350
[346] Cf Nunamaker/Chen, M. 1990; Webster/Watson 2002; Thiyagalingam et al. 2022
[347] Included in: Hevner, A. 2007, p. 88
[348] Cf. Hevner, A. 2007, p. 87
[349] Cf. Hevner, A. 2007, p. 89

this cycle enhances the field's knowledge and positions the designed artifact within a scientific context.[350] The Relevance Cycle incorporates the contextual environment into the research and introduces the artifact to field experiments.[351] In this thesis, the contextual input is also provided through the literature review. Field experiments are beyond the scope of this thesis. Instead, the feasibility of an SNTM is demonstrated not in real-world field experiments but within a simulated environment. The Design Cycle, focusing on the design and evaluation of the artifact, forms the central core of the DSR methodology.[352] Requirements are informed by the Relevance Cycles, while design and evaluation theories are derived from the Rigor Cycle.[353] Through multiple iterations, the artifact is developed and refined, subsequently feeding contributions back into the Rigor and Relevance Cycles.[354]

The publication schema for DSR studies, as proposed by Gregor/Hevner, A. 2013, consists of seven components, as shown in Appendix 1/1. Chapter 1 Introduction defines the problem, outlines the research objectives, and sets the scope of this thesis, thereby specifying the goals.[355] Additionally, the relevance of the research and the structure of the thesis are explained. A literature review, following the methodology proposed by Webster/Watson 2002, is conducted in Chapter 2 Review of the Current State of Research as part of the Rigor Cycle. This chapter introduces key concepts and discusses the current state of research, including existing NTM artifacts.[356] Chapter 3 Objective Specification and Research Methodology outlines the research methodology, detailing the DSR framework and referencing existing authorities.[357] The chapter also identifies the research methodologies employed to enhance the individual cycles. Chapter 4 Design and Implementation of the Artifact describes the design iterations and the development of the artifact, using the SDR methodology.[358] The core DSR process is explained in detail, including how the Design Cycle is employed to iteratively refine the artifact.[359] Evaluation and discussion are combined in Chapter 5 Evaluation and Discussion of the Artifact, where the artifact is examined using scientific benchmarks as outlined by Thiyagalingam et al. 2022. Furthermore, the artifact is contextualized within the current state of research, representing the final iteration of the Rigor Cycle, with a discussion on the implications of the results for both research and practice.[360] The concluding Chapter 6 Conclusion restates the main contributions of the thesis, discusses their significance, and provides a future outlook.[361]

[350] Cf. Hevner, A. 2007, p. 90
[351] Cf. Hevner, A. 2007, p. 87 et. seqq.
[352] Cf. Hevner, A. 2007, p. 90
[353] Cf. Hevner, A. 2007, p. 90 et seq.
[354] Cf. Hevner, A. 2007, p. 91
[355] Cf. Gregor/Hevner, A. 2013, p. 349
[356] Cf. Gregor/Hevner, A. 2013, p. 349 et seq.
[357] Cf. Gregor/Hevner, A. 2013, p. 350
[358] Discussed in Section 4.1 Design Methodology
[359] Cf. Gregor/Hevner, A. 2013, p. 350 et seq.
[360] Cf. Gregor/Hevner, A. 2013, p. 351
[361] Cf. Gregor/Hevner, A. 2013, p. 351

# 4 Design and Implementation of the Artifact

Embedded into the DSR framework, in this chapter the SDR methodology is utilized inside the Design Cycle developing the SNTM artifact. Additionally, the chapter outlines the inputs from the Relevance and Rigor Cycles into the Design Cycle, the programming environment and the SNU simulation toolkit.

## 4.1 Design Methodology

Within the Design Cycle, the Systems Development Research (SDR) methodology is utilized to iteratively describe, implement, and refine the SNTM artifact.[362] The SDR approach leads to a successive refinement of the prototype system.[363]

According to Nunamaker/Chen, M. 1990, the SDR methodology encompasses a structured five-step research process: constructing a conceptual framework, developing a system architecture, analyzing and designing the system, building the system, and observing and evaluating the system. In this thesis, the developed system is the SNTM architecture. The components of this system are the functions within the machine learning model responsible for data handling and operations. Essential for justifying the research significance, the conceptual framework explores various innovative approaches and ideas that could enhance the new system.[364] It integrates inputs from the Relevance and Rigor Cycles into the Design Cycle,[365] including model tasks and initial hyperparameters discussed in Section 4.2. Within the next two steps, the development and design of the system's architecture detail the dynamic interactions among components and specify the system's modules and functions.[366] These steps are executed, with pseudocode presented in Section 4.5, as the first iteration of the Design Cycle. The fourth step, building the system, establishes the actual implementation, thus demonstrating the feasibility of the system.[367] In Section 4.6, during the second Design Cycle iteration, the implementation of the SNTM artifact is explained. The hyperparameter optimization, crucial for optimizing system performance, is described in the following Design Cycle iterations in Section 4.7. In the final SDR step, the system is observed and evaluated to determine if it meets the established requirements.[368] This evaluation, including benchmarking the SNTM, is conducted in Chapter 5.

The SDR methodology is integrated into the overall DSR framework, serving as a systematic approach for developing the SNTM artifact.[369]

[362] Cf. Gregor/Hevner, A. 2013, p. 350 et seq.
[363] Cf. Nunamaker/Chen, M. 1990, p. 634
[364] Cf. Nunamaker/Chen, M. 1990, p. 635
[365] Cf. Hevner, A. 2007, p. 90 et seq.
[366] Cf. Nunamaker/Chen, M. 1990, p. 635
[367] Cf. Nunamaker/Chen, M. 1990, p. 635
[368] Cf. Nunamaker/Chen, M. 1990, p. 635
[369] Cf. Nunamaker/Chen, M. 1990, p. 634

## 4.2 Conceptual Framework: Literature Review Evaluation

Within the literature review of the thesis, the current state of the art in the field of NTM architecture is presented. Subsection 2.5.2 examines similar NTM artifacts, their hyperparameters, and respective tasks. The following literature evaluation guides the selection of tasks, benchmark networks, and hyperparameters for the development of the SNTM artifact.

### 4.2.1 Benchmark Task Selection

14 NTM artifacts are identified, in the literature review, as listed in Table 1.[370] This table also provides an overview of the utilized tasks, forming the basis for the following evaluation. Boloukian/Safi-Esfahani 2020 and Wayne et al. 2016 are the only publications that do not benchmark on the copy task. Twelve of the identified NTM artifacts are tested using the simple *copy* task, originally proposed by Graves/Wayne/Danihelka 2014.[371] The other two initial tasks, *repeat copy* and *associative recall*,[372] are utilized in seven of the identified artifacts.[373] Additionally, nine publications evaluate on more complex tasks,[374] such as the *bAbI dataset*[375] and the *sequential MNIST*[376] task.

Derived from the most commonly identified tasks in the literature, the developed SNTM artifact will be evaluated on the following **tasks**: **copy**, **repeat copy**, and **reverse copy**. The first two tasks are drawn from the initial NTM paper,[377] while the reverse task, also frequently utilized,[378] is included to assess the network's capability for rightward orientation.[379] The mathematical foundation and theoretical justification for these tasks are discussed in Subsection 2.5.1, and the actual implementation details are provided in Section 4.3.

### 4.2.2 Benchmark Network Selection

While Graves/Wayne/Danihelka 2014 initially proposed using 256 units and 3 stacked hidden layers of LSTM as a benchmark for the NTM architecture, subsequent publications have expanded this approach by testing the NTM against other benchmark architectures. Studies such as those by Zaremba/Sutskever 2015, Rae et al. 2016, Greve/Jacobsen/Risi 2016, Boloukian/Safi-Esfahani 2020, Zhang, W./Yu/Zhou 2015, and Merrild/Rasmussen/Risi 2018 benchmark their enhancements only against the classical NTM architecture. Other publications, including Wayne et al. 2016, Gülçehre/Chandar/Bengio 2017, and Chandar et al. 2016, test against

[370] Discussed in Subsection 2.5.2 Similar Artifacts
[371] See Table 1 Concept Matrix of Relevant Artifacts
[372] Cf. Graves/Wayne/Danihelka 2014, p. 12 et seqq.
[373] See Table 1 Concept Matrix of Relevant Artifacts
[374] See Table 1 Concept Matrix of Relevant Artifacts
[375] Cf. Wayne et al. 2016; Chandar et al. 2016; Gülçehre/Chandar/Bengio 2017
[376] Cf. Chandar et al. 2016; Gülçehre/Chandar/Bengio 2017; Boloukian/Safi-Esfahani 2020
[377] Cf. Graves/Wayne/Danihelka 2014, p. 10 et seqq.
[378] Cf. Zaremba/Sutskever 2015; Yang/Rush 2017
[379] Cf. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020, p. 12

both the classic NTM architecture and an LSTM network, with the primary comparison being with the NTM, while the LSTM serves as an additional benchmark. Following the initial suggestion, Danihelka et al. 2016, Mark Collier/Beel 2018, and Yang/Rush 2017 benchmark against an LSTM network, with Mark Collier/Beel 2018 also testing against the official DNC implementation. Malekmohamadi Faradonbe/Safi-Esfahani/Karimian-kelishadrokhi 2020 compare the LSTM, NTM, and DNC architectures against each other. Whenever an LSTM architecture is utilized as a benchmark, the typical configuration involves 3 stacked hidden layers with 256 units.[380] Yang/Rush 2017 tests configurations ranging from 1 to 4 layers, each with 256 units.[381]

In the current literature, the primary architectures benchmarked against are the initial NTM and an LSTM architecture. Therefore, in this thesis, the **SNTM artifact** will be **benchmarked against** the LSTM controller-based **NTM architecture** and a classical **LSTM network**, which features three stacked hidden layers and 256 units. Since SNU cells and LSTM cells exhibit similarities, the SNTM artifact will be benchmarked against the LSTM-based NTM, incorporating the same hyperparameters for both architectures.

### 4.2.3 Hyperparameter Selection

As discussed in Subsection 2.5.2, many publications propose various hyperparameters for training and configuring the NTM architecture. To establish a baseline for the SNTM artifact development, a review of relevant open-source implementations is conducted, focusing only on hyperparameters for tasks that are also used as benchmarks in this thesis. The results of this review are detailed in Appendix 4/1 and Appendix 4/2. While most of the identified open-source implementations are written in Python, Merrild/Rasmussen/Risi 2018 utilized C#,[382] and the implementations by Zaremba/Sutskever 2015 and Yang/Rush 2017 are written in Lua.[383] Often, hyperparameters are not well-documented, and the actual implementation configurations differ from those described in the respective papers or individual documentations. In cases of such discrepancies, hyperparameters from the actual implementations are used as references. Some hyperparameters are not specified in the implementations and are therefore not included in the overview tables.

An overview of the NTM hyperparameters is provided in Appendix 4/1. The *sequence length* of the network input varies from 5 to 30, with 8 and 10 being the most common choices. Most open-source implementations use one *layer* with 100 *units* in the LSTM controller. Official implementations of enhanced NTM architectures often utilize less than 100 units, ranging from 25 to 64. The *memory size* commonly chosen is either $128 \times 20$ or $128 \times 8$. In most implementations, the number of *heads* is typically 1, although one instance utilizes 4 *read heads*. Additionally, the *shift range* is often unspecified. Variations in *maximum repetitions* for the repeat copy task

[380] Cf. Graves/Wayne/Danihelka 2014; Mark Collier/Beel 2018; Mark Collier/Beel 2018
[381] Cf. Yang/Rush 2017, p. 6
[382] Cf. Jakobmerrild 2017
[383] Cf. Ilyasu123 2024; HarvardNLP 2017

are discussed in Subsection 2.5.2. For the initial hyperparameter selection in this thesis, the maximum repetitions for the repeat copy task are set to 4.

Appendix 4/2 details the training hyperparameters. The *optimizer* employed in every implementation is RMSProp, typically with a *learning rate* of $1 \times 10^{-4}$. Additionally, the ADAM *optimizer* is used in two of the implementations. Every open-source implementation utilizes the cross-binary *loss function*. The *batch size* and the *epochs* show the greatest variation. The number of *epochs* is often set to 10,000, but many implementations include early stopping mechanisms. Additionally, some do not specify *epochs* but instead determine the training duration by the *number of batches* fed into the network, also incorporating early stopping. Often, the *batch size* is set to 1, although some implementations use 16 or 32.

According to Mark Collier/Beel 2018, an architecture with constant memory instantiation is the most stable version and yields the best results.[384] Consequently, the SNTM memory will be implemented with a constant memory instantiation, which differs from the initial NTM architecture proposed by Graves/Wayne/Danihelka 2014.

Based on the discussed open-source implementations and the literature review, the following hyperparameters are chosen for the initial SNTM artifact:[385]

| **NTM Parameter** | **Value** |
|---|---|
| Sequence Length | 8 |
| Layer | 1 |
| Units | 100 |
| Memory Size | $128 \times 8$ (constant instantiation) |
| Number of Heads | 1 |
| Shift Range | 0 |
| Max. Repetitions | 4 (repeat copy) |
| **Training Parameter** | |
| Optimizer | RMSProp (with default TensorFlow hyperparameters) |
| Learning Rate | $1 \times 10^{-4}$ |
| Loss Function | sigmoid cross-entropy with logits |
| Batch Size | 16 |
| Epochs | 16 (without early stopping) |

During the iterative refinements, the ADAM optimizer will also be evaluated using a learning rate of $1 \times 10^{-3}$. All other parameters for the optimizer will remain set to the default values provided by TensorFlow. These refinements will focus on adjustments to the SNTM configuration itself. This includes reducing the memory size and the number of units in the controller for more lightweight models, as well as varying the sequence lengths.

[384] Cf. Mark Collier/Beel 2018, p. 99 et seq.
[385] Discussed in Subsection 4.7 Hyperparameter Tuning

## 4.3 Task Implementation

As discussed in Subsection 4.2.1, the developed SNTM implementation is evaluated on three tasks: *copy*, *reverse*, and *repeat copy*. A detailed mathematical description and the justification for their utility in testing the NTM is provided in Subsection 2.5.1. The functions for generating the task data are located in `utils/data_mgmt.py` and are named: `copy_data()`, `reverse_data()`, and `repeat_copy_data()`.

### 4.3.1 Copy Task

**Algorithm 1** Data Generation: Copy Task

1: **Function** COPY_DATA(`sample_number`, `sequence_length`= 8)
2: `random_bits` ← np.random.randint(0, 2, `sequence_length` ×`sample_number`)
3: `y_target` ← `random_bits`.reshape(`sample_number`, `sequence_length`)
4: `x_input` ←`random_bits`.reshape(`sample_number`, 1, `sequence_length`)
5: **return** tf.cast(`x_input`, tf.float32), tf.cast(`y_target`, tf.float32)
6: **end Function**

In Algorithm 1, the process for generating the copy task data is described. The function `copy_task_data()` accepts two parameters: `sample_number` and `sequence_length`. The parameter `sample_number` specifies the number of sequences to be generated, while `sequence_length` determines the length of each sequence. Initially, an array named `random_bits` is created, which contains binary sequences derived from a uniform distribution with an equal probability of zeros and ones. The total length of the array is the product of the `sequence_length` and `sample_number`. To obtain `x_input`, the `random_bits` array is reshaped into a three-dimensional array with the shape (`sample_number`, 1, `sequence_length`). This configuration is critical for the SNU to process the data accurately, as it requires a three-dimensional input tensor. To generate `y_target`, the `random_bits` array is reshaped into a two-dimensional matrix, where each row represents a single sequence. Both the input and target arrays are then casted into a `tf.float32` tensor to ensure compatibility within the TensorFlow workflow.

### 4.3.2 Reverse Task

**Algorithm 2** Data Generation: Reverse Task

1: **Function** REVERSE_DATA(`sample_number`, `sequence_length`= 8)
2: `x_input`, `y_target` ← COPY_TASK_DATA(`sample_number`, `sequence_length`)
3: `y_target` ← np.flip(`y_target`, axis = 1)
4: **return** `x_input`, tf.cast(`y_target`, tf.float32)
5: **end Function**

The `reverse_data` function, detailed in Algorithm 2, is designated for generating the reverse task data. This process is analogous to the copy task, with the primary distinction being that the tar-

get sequences are reversed versions of the input sequences. Initially, the `copy_task_data()` function is utilized to generate the starting data, and then the `y_target` array is reversed along the second axis using the `np.flip()` function. The reversed array is subsequently cast as `tf.float32` and returned along with the `x_input` array. The shapes of the returned values maintain the same dimensions as those used in the copy task.

### 4.3.3 Repeat Copy Task

**Algorithm 3** Data Generation: Repeat Copy Task

1: **Function** REPEAT_COPY_DATA(`sample_number`, `max_reps`= 6, `sequence_length`= 8)
2: `random_bits` ← np.random.randint(0, 2, `sequence_length` × `sample_number`)
3: `x_input` ← random_bits.reshape(`sample_number`, `sequence_length`)
4: `random_reps` ← np.random.randint(2, `max_reps`+1, (`sample_number`, 1))
5: `x_input` ← np.hstack((`random_reps`, `x_input`))
6: `y_target` ← REPEAT_AND_PAD(`x_input`, `max_reps` × `sequence_length`)
7: `x_input` ← `x_input`.reshape(`sample_number`, 1, `sequence_length`+1)
8: **return** tf.cast(`x_input`, tf.float32), tf.cast(`y_target`, tf.float32)
9: **end Function**

The `repeat_copy_data()` function, outlined in Algorithm 3, accepts three parameters including `sample_number`, `max_reps`, and `sequence_length`. The `sample_number` specifies the number of sequences to generate, `max_reps` determines the maximum number of repetitions for each sequence, and `sequence_length` sets the length of the binary sequences, consistent with the other tasks. The `random_bits` array, initially a generated sequence of binary data, is reshaped into a two-dimensional array with dimensions (`sample_number`, `sequence_length`). This arrangement is essential for the subsequent function `repeat_and_pad()`. The `random_reps` array, with dimensions (`sample_number`, 1), contains the number of repetitions for each sequence, randomly chosen between 2 and `max_reps`+1 to ensure inclusion of `max_reps`. The minimum number of repetitions is set to 2 to avoid trivial cases that are already tested with the simple copy task. The `random_reps` array is then merged with the `x_input` array along the second axis using `np.hstack()`, resulting in a two-dimensional array of shape (`sample_number`, `sequence_length`+1), where the first column holds the `random_reps`. The `y_target` array is generated by the `repeat_and_pad()` function, which repeats each sequence according to the specifications in `random_reps` and pads them with zeros to achieve a total length equal to the product of `max_reps` and `sequence_length`. The padding is necessary to ensure that the SNTM can learn sequences of randomly varying lengths. A visual representation of the arrays `x_input` and `y_target` is provided in the following example, as illustrated in Equation 4.1. The details of the `repeat_and_pad()` function are presented in Appendix 5/1. The `x_input` array is subsequently reshaped into a three-dimensional array with dimensions (`sample_number`, 1, `sequence_length`+1) to fit the required input tensor format for the SNU. Finally, both the

`x_input` and `y_target` arrays are cast as `tf.float32` and returned.

$$\begin{bmatrix} 3 & 0 & 1 & 1 \\ 2 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{4.1}$$

In the provided example 4.1, the `x_input` tensor is depicted on the right and the `y_target` tensor on the left. The parameter for this example include the `sequence_length` and the `max_reps` both set to 3. The first column of the `x_input` tensor specifies the number of repetitions, while the subsequent columns contain the sequence. The `y_target` tensor includes the sequence repeated according to the specified repetitions and is padded with zeros to fill the space up to the product of the maximum number of repetitions and the sequence length. To enhance visual clarity, padding is highlighted in red and the repeated sequences in blue and black, making the repetitions more distinguishable. The `random_reps` array in this example is $\begin{bmatrix} 3 & 2 \end{bmatrix}^T$, and the `random_bits` array is $\begin{bmatrix} [0,1,1] & [1,1,1] \end{bmatrix}^T$

## 4.4 Neuro-AI-Toolkit

The `neuroaikit`, introduced in a "Nature" publication by Woźniak et al. 2020, facilitates the simulation of SNN architectures within conventional ANN frameworks. The theoretical underpinnings of the implemented SNU cells are explored in detail in Subsection 2.3.3. A comprehensive documentation[386] is provided, and the open-source version is accessible on GitHub[387]. The toolkit is released under the Apache 2.0 license, a liberal open-source license that grants users the rights to use, modify, and distribute the software with few restrictions.[388]

The open-source GitHub repository contains two main directories and five files. The files include a license, a readme, a list of requirements, a setup file, and the gitignore file. The `Website/` folder primarily stores for the materials for the documentation website.

The toolkit's core code to simulate the SNU is located within the `neuroaikit/` directory. In the public open-source release, the toolkit is developed using Python and TensorFlow. In the original publication by Woźniak et al. 2020, the SNU is evaluated across various datasets and tasks, with all pertinent code contained in the `neuroaikit/` folder. The present thesis centers on the actual implementation of the SNU and its integration into an ANN architecture, with relevant code located in the subdirectory `neuroaikit/neuroaikit/tf/`.

The file `activations.py` includes activation functions that are not present in the official TensorFlow library. For instance, `step_function()` computes the already discussed step function $h(a)$, which returns 1 if $a > 0$ and 0 otherwise. Given that the step function is non-differentiable due to the absence of a gradient, a pseudoderivative is computed during the backward pass in gradient descent.

[386] Cf. IBM Research 2021
[387] Cf. IBM Research 2024
[388] Cf. The Apache Software Foundation 2024

Within the `layers` folder, two SNU cell architectures are defined in the files `snubasiccell.py` and `snulicell.py`. The `SNUBasicCell()` class defines the fundamental SNU cell structure, where the constructor accepts parameters such as the number of `units`, cell `decay`, output `activation` function $h(a)$, and the internal state activation function `g` defaulting to `tf.identity` for no modification. Additionally, the boolean parameter `recurrent` indicates whether the SNU should include recurrent connections within the layer. The two additional functions to the `__init__()` method are `build()` for layer construction and `call()` for executing the forward pass. The `call()` function defines the actual operations which are performed on the input by the layer as described in Subsection 2.3.3. The `SNULICell()` class, which extends `SNUBasicCell()`, introduces lateral inhibition. This feature enables neurons within a network to self-inhibit,[389] as reflected in a modification to the `call()` function within the open-source code. Finally, the file `SNU.py` contains a class enabling the creation of an SNU within a `tf.keras.layers.RNN()` wrapper. This allows the SNU layer to function akin to a standard RNN layer in an ANN architecture, compatible with `tf.keras.Sequential` for layer stacking in a model.

For the implementation detailed within this thesis, only the open-source `neuroaikit` is utilized to ensure reproducibility and transparency.

## 4.5 High-Level Architecture of the Spiking Neural Turing Machine

The first iteration of the DSR Design Cycle focuses on the high-level architecture of the SNTM artifact, with the primary goal to develop an in-depth understanding of the architecture and its individual components.

Facilitating integration into a conventional TensorFlow workflow, the conceptual architecture demonstrates how to embody the theoretical architecture within a TensorFlow layer. Key functions are identified and described at a high-level, drawing parallels to the theoretical foundation discussed in Section 2.4. Pseudocode is employed to clarify concepts, detailing the functions and their interactions. This conceptual design lays the groundwork for the subsequent TensorFlow implementation of the NTM artifact. Further details and the actual implementation based on this conceptual design are elaborated in Section 4.6. The high-level architecture is based on the theoretical foundation presented by Graves/Wayne/Danihelka 2014 and the open-source implementation[390] by Mark Collier/Beel 2018. The referenced open-source implementation is developed in TensorFlow version 1.$x$, which includes elements not present in the current TensorFlow version 2.$x$. As discussed in Section 4.6.1, the conceptual design is specifically tailored to align with TensorFlow version 2.15.0. Thus, while it draws inspiration from the cited sources, it also adapts to the specifications of the current TensorFlow version. To ensure the operational workflow of the SNTM, four fundamental functions are required: `__init__()`, `call()`,

[389] Cf. Woźniak et al. 2020, p. 328
[390] Cf. MarkPKCollier 2018

`addressing()`, and `init_states()`, explained in the following subsections. These functions are encapsulated within the `NTMCell` class, a subclass of the TensorFlow `tf.keras.layers.Layer` class.

### 4.5.1 Network Initialization

The constructor of the NTM cell, which is the initial function, is described in Algorithm 4.

**Algorithm 4** `__init__` of the `SNTMCell` class

```
1: function __INIT__(configuration parameters)
2:     Inherit from tf.keras.layers.Layer                    ▷ base layer setup
3:     Initialize controller using dynamic parameters
4:     Set configuration parameters for NTM cell             ▷ memory size, heads, etc.
5:     Initialize dense layers for cell operations           ▷ parameter, memory and output layers
6: end function
```

The `NTMCell` class must inherit from `tf.keras.layers.Layer` to ensure seamless integration into the TensorFlow workflow. The controller is dynamically constructed, allowing users to specify the number and configuration of layers and units, as well as hyperparameter for individual SNU cells. This flexibility enables the implementation to accommodate various configurations of the NTM cell with different SNU cell architectures. In the constructor, NTM configuration parameters such as memory size, the number of read and write heads, and other relevant settings are established. These parameters will be discussed in detail in Section 4.6.

The memory operation parameters are computed by the controller output, as explained in Subsection 4.5.3. To ensure these parameters align with the memory size and the number of heads, a dense layer is utilized to adjust their scale appropriately. Furthermore, dense layers are required for generating the final output of the cell and for the initial instantiation of the states.

### 4.5.2 Network Operations

The `call()` function is the primary function of the `NTMCell` class, responsible for executing the NTM cell operations on the input. The theoretical foundation of this function is discussed in Subsection 2.4.1, and its conceptual execution is outlined in Algorithm 5.

Two arguments are accepted by the `call()` function: the input $x$ and the previous states. If no previous states are provided, the function initializes the states using the `init_states()` function, as described in Algorithm 7. These states include the controller state, the read vector $\mathbf{r}_{t-1}$, the addressing weights $\mathbf{w}_{t-1}$, and the memory matrix $\mathbf{M}_{t-1}$. Within the `call()` function, there are four major steps: controller operation, addressing operation, memory read and write operations, and the output generation. For the controller operation, the input $x$ is concatenated with the read vector from the previous state to form the controller input. The computed controller output is utilized to generate parameters for the read and write heads via the `addressing()`

function. This function computes the new addressing weights based on the controller output, the current memory state, and the previous addressing weights. The process is explained in Algorithm 6. With the new addressing weights, the memory is read and written accordingly. The read vector $\mathbf{r}_t$ is generated from the memory using the addressing weights $\mathbf{w}_t$,[391] and the memory is updated to $\mathbf{M}_t$ based on the write operation.[392] To generate the final SNTM output, the controller output and the read vector are merged and reshaped through a dense layer to match the output dimension. Subsequently, the output is clipped to the specified bounds before being returned, along with the updated states, which include the controller states, read vector, addressing weights, and memory matrix.

**Algorithm 5** `call()` function of the `SNTMCell` class

```
1: function CALL(x, previous states)
2:     if no previous states is provided then
3:         states ← INIT_STATES(x)                          ▷ init state using x
4:     end if
5:     Concatenate x with previous read vectors to form controller input   ▷ controller
6:     Compute controller output and new controller state                  ▷ controller
7:
8:     w_t ← ADDRESSING(controller output, M_{t-1}, w_{t-1})               ▷ addressing weights
9:
10:    Perform reading from memory using w_t → read vector                 ▷ read operation
11:    Update memory using w_t → M_t                                       ▷ write operation
12:
13:    Generate SNTM output: combining controller output and read vector   ▷ output
14:    Reshape trough dense layer to match output dimension                ▷ output
15:    Clip the output to given bounds                                     ▷ output
16:    return output, updated states
17: end function
```

### 4.5.3 Addressing Mechanism

The `addressing()` function is responsible for updating the addressing weights of the memory, thus enabling the read and write heads to modify the memory effectively. The mathematical foundation of the addressing mechanism is already discussed in detail in Subsection 2.4.4. This function receives the controller output, the current memory state ($\mathbf{M}_t$), and the previous addressing weights ($\mathbf{w}_{t-1}$) as parameters and returns the updated addressing weights. Algorithm 6 provides a high-level overview of the `addressing()` function.

The addressing mechanism requires the parameters $\mathbf{k}_t, \beta_t, g_t, \mathbf{s}_t, \gamma_t$, which are calculated from the controller's output. These parameters facilitate the addressing operations such as content addressing, interpolation, convolutional shift, and sharpening. The new addressing weights are collectively computed based on these operations and are afterwards returned. Specific details

[391] Discussed in Subsection 2.4.2 Read Head
[392] Discussed in Subsection 2.4.3 Write Head

**Algorithm 6** `addressing()` function of the `SNTMCell` class

**function** ADDRESSING(contoller output, $\mathbf{M}_{t-1}$, $\mathbf{w}_{t-1}$)
    Extract addressing parameter from controller output ▷ $\mathbf{k}_t, \beta_t, g_t, \mathbf{s}_t, \gamma_t$
    Apply content addressing ▷ $\mathbf{M}_{t-1}, \mathbf{k}_t, \beta_t \rightarrow \mathbf{w}_t^c$
    Apply interpolation ▷ $\mathbf{w}_{t-1}, \mathbf{w}_t^c, g_t \rightarrow \mathbf{w}_t^g$
    Apply convolutional shift ▷ $\mathbf{w}_t^g, \mathbf{s}_t \rightarrow \tilde{\mathbf{w}}_t$
    Apply sharpening ▷ $\tilde{\mathbf{w}}_t, \gamma_t \rightarrow \mathbf{w}_t$
    **return** new addressing weights ▷ updated $\mathbf{w}_t$
**end function**

regarding which parameters are employed for each operation are provided in the pseudocode and in the previously discussed Figure 7.[393]

### 4.5.4 State Instantiation

Cell states are employed in the SNTM to retain information across time steps and to facilitate the computation of operations. To secure proper functionality initially, these states must be initialized at the first time step ($t = 0$) of the SNTM. The pseudocode for this initialization process is outlined in Algorithm 7.

**Algorithm 7** `init_states()` function of the `SNTMCell` class

**function** INIT_STATES(x)
    Initialize controller state ▷ based on controller architecture
    Initialize read vector list ▷ based on the head number and batch size → $\mathbf{r}_{t=0}$
    Initialize addressing weights ▷ based on the head number and batch size → $\mathbf{w}_{t=0}$
    Initialize memory ($10^{-6}$ constant) ▷ based on memory size → $\mathbf{M}_{t=0}$
    **return** all initial states ▷ tuple of the states
**end function**

The function `init_states()` requires the model input $x$ as an argument because certain states are initialized based on the input and the batch size. The controller state in the `NTMCell` class is initialized according to the controller's architecture, which is specified during the class initialization. The memory matrix ($\mathbf{M}_{t=0}$) is initialized to a constant value of $10^{-6}$, following the suggestion from Mark Collier/Beel 2018.[394] The size of the memory matrix is determined by the memory size parameter set during the initialization of the `NTMCell` class. The initial read vector list ($\mathbf{r}_{t=0}$) and addressing weights ($\mathbf{w}_{t=0}$) are set based on the number of heads and the batch size. To ensure dimensional compatibility, the dense layers which are instantiated within the `__init()__` function are utilized,. At the conclusion of the `inits_states()` function, all the states are returned.

[393] Discussed in Subsection 2.4.4 Addressing Mechanisms
[394] Discussed in Section 4.2 Conceptual Framework: Literature Review Evaluation

## 4.6 Implementation of the Spiking Neural Turing Machine

The second iteration of the Design Cycle involves the actual implementation of the SNTM artifact. This subsequently described implementation is based on the high-level architecture outlined in Section 4.5.

### 4.6.1 Programming Environment

The entire development process of the SNTM artifact is organized within the `SNTM_jonasMichel/` directory. Paths to files referenced in this thesis are relative to this directory. The implementation of the SNTM artifact is saved in the `ntm_cell.py` file, which contains the `NTMCell` class.

All files are executable locally using `Python` version 3.10.13. Required libraries are listed in the `requirements.txt` file, which can be installed via `pip install -r requirements.txt` from within the same directory. Key libraries utilized include `neuroaikit`[395] and `TensorFlow` version 2.15.0. The development environment was configured using Anaconda on a 2019 MacBook Pro, equipped with a 2.4 GHz 8-Core Intel Core i9 processor and 32 GB RAM.

### 4.6.2 Network Initialization

In the constructor, the `__init__()` function of the `NTMCell` class, the network is instantiated. The instantiation begins with the creation of the controller. This is achieved by defining the `controller_cells` list with dynamic parameters, where the input parameters `cell`, `cell_args`, `controller_units`, and `controller_layers` are utilized. The chosen `cell` must be a subclass of the TensorFlow `tf.keras.layers.Layer` class. Each `cell` in the list is initialized with a specified number of `controller_units` and its unique `cell_args`. The dictionary `cell_args` contains the initial parameters for the controller `cell`, allowing for the use of different SNU cells or classical LSTM cells as controllers with their parameters dynamically assigned. The controller is then created by stacking the cells in the `controller_cells` list into a TensorFlow `StackedRNNCells` layer, which is subsequently wrapped up in a `RNN` layer. The initialized `controller` only returns the last output and the state of the controller cell, since the complete output sequence from all individual units is not required.

Subsequently, the given hyperparameters are instantiated: `memory_size`, `memory_vector_dim`, `read_head_num`, `write_head_num`, `shift_range`, `init_mode`, `addressing_mode`, `output_dim`, `clip_value`. The shape of the memory matrix is defined as `memory_size` × `memory_vector_dim`. The number of read and write heads are specified by `read_head_num` and `write_head_num`, respectively. The `shift_range`, set to `0` by default, allows shifting of the read and write heads within the addressing mechanism.[396] By default, `init_mode` is set to `"constant"` to align with

[395] Discussed in Section 4.4 Neuro-AI-Toolkit
[396] Discussed in Subsection 2.4.4 Addressing Mechanisms

the constant memory instantiation schema employed in this thesis. However, the framework is designed to be extendable by allowing implementation adjustments through the `init_mode` parameter.[397] The `addressing_mode` is set to `"content_and_location"`, the standard for the NTM architecture, with an alternative `"content"` mode.[398] The `output_dim` is `None` by default, indicating that the input and output dimensions are equal, is adjustable based on network utilization. The `clip_value` is set to `20` by default for gradient clipping to mitigate issues like the vanishing or exploding gradients.[399] All default parameters are chosen based on the open-source implementation from Mark Collier/Beel 2018.

The function `calculate_head_parameters()` computes the following values: the total number of parameters (`total_parameter_num`), the number of heads (`num_heads`), and the number of parameters per head (`num_parameters_per_head`). These values are essential for subsequent calculations within the network. The number of parameters per head and the total number of parameters is calculated using the formulas:[400]

$$\texttt{num\_p\_per\_head} = \texttt{memory\_v\_dim} + 1 + 1 + (\texttt{shift\_range} \times 2 + 1) + 1 \tag{4.2}$$

$$\texttt{total\_p\_num} = \texttt{num\_p\_per\_head} \times \texttt{num\_heads} + \texttt{memory\_v\_dim} \times 2 \times \texttt{write\_head\_num} \tag{4.3}$$

For Equation 4.2, the first $+1$ accounts for the key strength $\beta$, and the second $+1$ accounts for the interpolation gate $g$. The term $\texttt{shift\_range} \times 2 + 1$ calculates shifts to the left and right ($\times 2$) and includes the center shift ($+1$). The final $+1$ is for the sharpening parameter $\gamma$.[401] The number of heads is the sum of the read and write heads. The total number of parameters, described in Equation 4.3, is calculated by multiplying the number of heads by the number of parameters per head and adding the product of $\texttt{memory\_vector\_dim} \times 2 \times \texttt{write\_head\_num}$. This is due to each write head requiring two parameters for every element of a memory vector: one to erase using vector $\mathbf{e}$ and another to write using vector $\mathbf{a}$ to the memory.[402]

Additionally, dense layers are employed within the network to adjust the dimensions of specific vectors. The `parameter_dense` layer, instantiated with the calculated total number of parameters, transforms the `controller_output` into parameters used by the heads for the addressing mechanism. Meanwhile, the `output_layer` modifies the `controller_output` and the `read_vector` to produce the actual output of the network.[403] This layer is initially instantiated with `None` as a placeholder. If the `output_dim` is not specified, the `output_layer` must be configured based on the model's input `x`, which can only occur within the `call()` function.

The function `init_states()` utilizes dense layers to initialize certain states.[404] Therefore, the `read_init_layer` is instantiated with `memory_vector_dim` as the output dimension. This

[397] Discussed in Section 2.4.4 Addressing Mechanisms
[398] Discussed in Section 4.6.4 Addressing Mechanism
[399] Cf. Pascanu/Mikolov/Bengio 2013, p. 1317
[400] Due to space constraints, abbreviations are used; Legend: p (parameter), v (vector)
[401] Discussed in Subsection 2.4.4 Addressing Mechanisms
[402] Discussed in Subsection 2.4.3 Write Head
[403] Discussed in Subsection 4.6.3 Network Operations
[404] Discussed in Subsection 4.6.5 State Instantiation

layer initializes the read vector, which is read from the memory matrix and therefore matches the `memory_vector_dim` dimensions.[405] Finally, the `weight_init_layer` is instantiated with `memory_size` as the output dimension. This layer initializes the weights **w** for each head, which are used to address the memory locations, and thus corresponds to the `memory_size` dimensions.[406]

### 4.6.3 Network Operations

The `call()` method serves as the main entry point for the SNTM artifact and is invoked by the `tf.keras.Model` class whenever the model processes a batch of input data. This function is responsible for the forward pass of the network, performing all principal SNTM operations on the input data `x`.

Two parameters are accepted by the `call()` function: the input data `x` and the `PREV_state`, which defaults to `None`. The function first checks if a previous state exists (`PREV_state` $\neq$ `None`). If no previous state exists (`PREV_state` = `None`), one is initialized using the `init_states()` function.[407] The previous state of the SNTM is encapsulated in a `collections.namedtuple` object, which comprises the `controller_state`, the `read_vector_list`, the weights for read and write operations in `w_list`, and the memory matrix `M`.

Subsequently, the `controller_output` is computed by the `controller` network, instantiated within the `__init()__` function. The integration of the previous read vector from the memory into the controller's decision-making process is essential for the SNTM's functionality.[408] Consequently, the `PREV_read_vector_list` is retrieved from the previous state and expanded along a new axis, facilitating the concatenation with the current input data `x` using the `tf.concat()` function. This concatenation, referred to as `controller_input`, incorporates information from both the previous read vector state and the current input, thereby enriching the data fed into the controller. The `controller` is then called with the `controller_input` and previous `controller_state` to generate the `controller_output` and the updated `*controller_state`. The asterisk notation `*` implies that all returned values from the controller are unpacked and assigned to the `controller_state` variable.

Utilizing the `controller_output`, the weights `w` for the read and write operations are calculated and stored in the `w_list`. The `controller_output` is first processed through the `parameter_dense` layer to generate the `parameters` tensor, which has dimensions equal to the `total_parameter_num`. This tensor is then sliced into the `head_parameter_list` for the read and write heads, and the `erase_add_list` for the erase and add operations from the write head. Parameters for the $i^{th}$ head are accessed via `head_parameter_list[i]`, and similarly

[405] Discussed in Subsection 2.4.2 Read Head
[406] Discussed in Subsection 2.4.2 Read Head and Subsection 2.4.3 Write Head
[407] Discussed in Subsection 4.6.5 State Instantiation
[408] Discussed in Subsection 2.4.1 High-Level Architecture

through `erase_add_list[i]`, but only within the range of write heads. To compute the addressing parameters $\mathbf{k}, \beta, g, \mathbf{s}$, and $\gamma$, both the `prev_w_list` and `prev_M` are retrieved from the previous state. Additionally, `w_list` is initialized as an empty list. A `for loop` iterates over the `head_parameter_list` to determine the weights `w` for each head. Initially, the addressing parameters are derived using the `get_addressing_parameters()` function with the $i^{th}$ `head_parameter`.[409] Next, the `addressing()` function computes the weights for the $i^{th}$ head using these parameters.[410] The calculated weights `w` are then appended to the `w_list`.

Using the calculated weights `w` in the `w_list`, both read and write operations are executed.[411] For the read operation, the `w_list` is sliced into the `read_w_list`, containing individual weights for the read heads. An empty `read_vector_list` is initialized to store vectors retrieved from the memory. The loop `for i in range(read_head_num)` executes the read operation for each defined read head. A `read_vector` is calculated for each head based on the previously discussed Equation 2.18.[412] Each `read_vector` is appended to the `read_vector_list`, ensuring the list contains one read vector for each defined read head. Subsequently, to obtain the individual write head weights, the `w_list` is sliced to create the `write_w_list`. Also the memory matrix `M` is retrieved from the previous state. A `for loop` is employed to iterate over the number of write heads for executing the write operations. Initially, the weights `w` are expanded along axis 1 to align with the dimensions of the memory matrix `M`. Similarly, the `erase_vector` and the `add_vector` are derived from the `erase_add_list` for the $i^{th}$ head and expanded to match these dimensions. The `erase_vector` parameters are processed through a sigmoid activation function to ensure the values range between 0 and 1, as illustrated in Equation 2.19. Conversely, the `add_vector` is processed through a hyperbolic tangent activation function, maintaining values between $-1$ and 1, which is essential for accurately incrementing or decrementing the memory content at specific locations. The memory matrix `M` is updated utilizing the formula described in Equation 2.21,[413] with each write head updating the matrix once.

As the final step, the output of the SNTM cell is calculated. If not specified differently as hyperparameter provided to the `__init__()` function, the `output_dim` is set to the input dimension of `x`.[414] Subsequently, the dense `output_layer` is defined with `output_dim` as the output dimension. The `NTM_output`, is a concatenation of the `controller_output` and the `read_vector_list`, processed through the `output_layer`, to ensure the correct output dimension The `final_NTM_output` is then produced by clipping it with the `clip_value` to maintain output values within a specific range. This enhances stability during training and prevents issues with vanishing and exploding gradients.[415] The `call()` function returns the `final_NTM_output`, serving as the output of the SNTM cell, along with the `NTMControllerState`, which contains the SNTM updated states: `controller_state`, `read_vector_list`, `w_list`, and `M`.

[409] Discussed in Subsection 4.6.4 Addressing Mechanism
[410] Discussed in Subsection 4.6.4 Addressing Mechanism
[411] Discussed in Subsection 2.4.2 Read Head and Subsection 2.4.3 Write Head
[412] Discussed in Subsection 2.4.2 Read Head
[413] Discussed in Subsection 2.4.3 Write Head
[414] Discussed in Subsection 4.6.2 Network Initialization
[415] Cf. Pascanu/Mikolov/Bengio 2013, p. 1317

### 4.6.4 Addressing Mechanism

Addressing mechanisms are utilized to calculate the weights `w` for both read and write operations.[416] The process involves two functions: `get_addressing_parameters()` and `addressing()`. Both functions are invoked within the `call()` function to compute the `w_list` from the controller's output, the `head_parameter_list`.[417]

The first function, `get_addressing_parameters()`, is used to calculate the parameters, utilized in the addressing mechanism, with given constraints: $\mathbf{k}_t, \beta_t \geq 0$, $g_t \in [0, 1]$, $\mathbf{s}_t$ s.t. $\sum_k \mathbf{s}_t(k) = 1$ *and* $\forall_k\ s_t(k) \geq 0$, and $\gamma_t \geq 1$.[418] Derived from an LSTM architecture,[419] the contents of the memory matrix are constrained within the range of $[-1, 1]$ by applying a hyperbolic tangent activation function to $\mathbf{k}$,[420] similar to the previously discussed `add_vector`.[421] To satisfy the given constraints, $\beta$ is passed through a $\text{softplus}(x) = \log(\exp(x) + 1)$ function, ensuring that $\beta \geq 0$.[422] The logistic sigmoid function is applied to $g$ to ensure that $g \in [0, 1]$. To fulfill the constraints of the shift vector $\mathbf{s}$, the simplest way is to utilize a softmax layer.[423] To guarantee $\gamma \geq 1$, the softplus transformation is applied to $\gamma$ as follows: $\gamma = \text{softplus}(\gamma)+1$.[424] The activation functions selected ensure the defined constraints are met and parameters stay within required ranges. Finally, the `get_addressing_parameters()` function returns the calculated parameters. Within the `call()` function, the parameters are passed to the `addressing()` function.

The second function, `addressing()`, calculates the weights `w` based on addressing parameters, the memory matrix `M`, and the previous weights `w_prev`, as depicted in Figure 7.[425] The process starts with calculating content addressing by determining the cosine similarity between $\mathbf{k}$ and each vector in $\mathbf{M}_t(i)$, as defined in Equation 2.22, yielding the content weights `w_c`. If the `addressing_mode` is set to `"content"`, the `addressing()` function returns `w_c` as the final weights `w`. Otherwise, the interpolation occurs where `w_c` is combined with `w_prev` using the interpolation gate $g$, as described in Equation 2.24, resulting in interpolated weights `w_g`. The interpolated weights undergo a convolution shift based on the shift vector $\mathbf{s}$. To facilitate this shift, $\mathbf{s}$ is adjusted to align with the correct memory dimensions. This adjustment involves zero-padding, achieved using `tf.concat`. A new matrix is created to represent the shift operations: the `s_matrix`. Each row of the `s_matrix` corresponds to a different shift direction (left or right) applied to $\mathbf{w}_g$, enabling the convolutional shift mechanism. In the last step, `w_` is refined through the sharpening process, detailed in Equation 2.26, producing the final weights `w`. Finally the `addressing()` function returns the calculated weights `w`.

[416] Discussed in Subsection 2.4.4 Addressing Mechanisms
[417] Discussed in Subsection 4.6.3 Network Operations
[418] Discussed in Subsection 2.4.4 Addressing Mechanisms
[419] Discussed in Subsection 2.2.2 Recurrent Neural Networks
[420] Cf. Mark Collier/Beel 2018, p. 98
[421] Discussed in Subsection 4.6.3 Network Operations
[422] Cf. Mark Collier/Beel 2018, p. 98
[423] Cf. Graves/Wayne/Danihelka 2014, p. 8
[424] Cf. Mark Collier/Beel 2018, p. 98
[425] Discussed in Subsection 2.4.4 Addressing Mechanisms

### 4.6.5 State Instantiation

The final function within the `NTMCell` class is the `init_states` function, which is responsible for initializing the states when the `NTMCell` class is first invoked. The states initialized are: `controller_state`, `read_vector_list`, `w_list`, and the memory matrix `M`. The utilization of these states in the `call` function has been previously discussed.[426] The `init_states` function receives the input `x` as a parameter, and initially defines the `batch_size` as the first dimension of `x`.

To determine the initial state of the controller, the function `controller.get_initial_state()` is invoked with the input `x`. This function, integral to the `tf.keras.layers.RNN` class, retrieves the initial state required for the network's operation. With as many elements as there are read heads, the `read_vector_list` is created. Each read vector is normalized using a tangent activation function to ensure the vector remains within the range of $[-1, 1]$. The initialization is achieved by invoking the dense `read_init_layer` with a tensor of ones. This not only ensures the read vector matches the correct dimensions but also enables a learned initialization.[427] The resulting read vectors are subsequently expanded to match the batch size, ensuring alignment with the dimensions of the input `x`. Similarly to the `read_vector_list`, the `w_list` is initialized using a dense `weight_init_layer`. The weights within the `w_list` are normalized using a softmax activation function to ensure they sum to one. The result is then expanded to match the batch size. The `w_list` contains as many weight elements as there are read and write heads in the SNTM. Finally, the memory matrix `M` is initialized with the shape [`memory_size,memory_vector_dim`], which is then expanded to match the batch size. The initialization uses the `constant` `init_mode` to set the memory values to $10^{-6}$, following Mark Collier/Beel 2018 who reported the best results with these setting. To enhance the developed framework, additional memory initialization methods can be integrated using the `init_mode` parameter and conditional `if` statements.

The `init_states` function concludes by returning a tuple containing the initialized states.

## 4.7 Hyperparameter Tuning

Subsequently to the first two Design Cycle iterations, the concept and implementation of the SNTM architecture, the hyperparameter tuning is conducted within the next iterations.

The first hyperparameters are chosen in Subsection 4.2.3 as a staring point for the artifact development. During the hyperparameter tuning iterations, the goal is to improve the performance of the SNTM architecture on the selected tasks and find out possible limitations of the architecture. Each set of hyperparameters is saved in a dictionary and stored in the `config_sntm.py` file. A comprehensive overview over all used hyperparameters during the iterations is given in Table 2. To get a statistical significance of the loss and accuracy, the SNTM model is trained for 500 runs

[426] Discussed in Subsection 4.6.3 Network Operations
[427] Cf. Mark Collier/Beel 2018, p. 97

with the same hyperparameters. The test runs are conducted within the `ntm_run_test.ipynb` notebook. The notebook architecture is detailed in Appendix 5/2.

For each iteration a table is shown in this thesis to summarize the results. This table includes the mean and standard deviation of the accuracy and loss, the fail ratio, the hyperparameters and the version of the model. The accuracy of the model is calculated using the `calculate_accuracy()` function located in `utils/data_mgmt.py`. This function accepts the target values and predicted outputs as parameters, computes the total number of errors by comparing these parameters, and calculates the accuracy. The accuracy is determined by subtracting the number of errors from the total number of elements and then dividing by the total number of elements to produce a percentage. The function returns both the calculated accuracy and the total number of errors. The fail ratio is calculated by dividing the number of failed runs by the total number of runs. A run is considered failed if the loss is `NaN` or the accuracy is `NaN`, thus the model is not correctly trained. The version of the model `vX`, is given to easily match each iteration and their results with the hyperparameters in the `config_sntm.py` file. All referred results are taken from the notebook `evaluation.ipynb`, and stored in the `model_comparison.csv`.

| V | SL | C[428] | M[429] | H | SR | MR | LR | O | LF |
|---|---|---|---|---|---|---|---|---|---|
| **v1** | 8 | $1 \times 100$ | $128 \times 8/20$ | 1 | 0 | 4 | $10^{-4}$ | RMSProp | cross-entropy |
| **v2** | 8 | $1 \times 100$ | $128 \times 8/20$ | 1 | 0 | 4 | $10^{-3}$ | ADAM | cross-entropy |
| **v3** | 8 | $1 \times 100$ | $128 \times 8/20$ | 1 | 0 | 4 | $10^{-3}$ | RMSProp | cross-entropy |
| **v4** | 8 | $1 \times 100$ | $128 \times 8/20$ | 1 | 0 | 4 | $3 \times 10^{-3}$ | ADAM | cross-entropy |
| **v5** | 8 | $1 \times 50$ | $16 \times 8/20$ | 1 | 0 | 4 | $3 \times 10^{-3}$ | ADAM | cross-entropy |
| **v6** | 20 | $1 \times 100$ | $128 \times 20/80$ | 1 | 0 | 4 | $3 \times 10^{-3}$ | ADAM | cross-entropy |
| **v7** | 20 | $3 \times 256$ | $128 \times 20/80$ | 1 | 0 | 4 | $3 \times 10^{-3}$ | ADAM | cross-entropy |

Tab. 2: Comparative Overview of Hyperparameter Configurations for SNTM Artifact Versions; Grey highlighted cells indicate changes in the hyperparameter configuration compared to the previous version. Legend: V (SNTM Version), SL (Sequence Length), C (Controller), M (Memory), H (Number of Heads), SR (Shift Range), MR (Max Repetitions for Repeat Copy), LR (Learning Rate), O (Optimizer), LF (Loss Function), cross-entropy (sigmoid cross-entropy with logits);

### SNTM Version 1 (Design Cycle Iteration 3)

As defined in Subsection 4.2.3, the initial set of hyperparameters, referred to as **v1**, is outlined in Table 2. This configuration was intended to serve as a baseline for identifying potential improvements based on the literature review.[430] However, all 500 test runs failed, resulting in loss and accuracy being recorded as `NaN`. Therefore for this version of the SNTM, no summary

[428] Notated as: `layer` × `units`
[429] Notated as: `memory size` × `memory locations (copy, reverse)`/ `memory locations (repeat copy)`
[430] Discussed in Subsection 4.2.3 Hyperparameter Selection

table is presented. Theoretically, the model has 16,020 trainable parameters for the copy and reverse tasks, and 31,772 for the repeat copy task.[431]

### SNTM Version 2 (Design Cycle Iteration 4)

To address the training issues observed in SNTM Version 1, several adjustments are made. The learning rate is increased from $10^{-4}$ to $10^{-3}$, and the optimizer is switched from RMSProp to ADAM, as indicated in Table 2. These changes maintain the other hyperparameters as in the initial version. The main objective of these adjustments is to initiate training activity by employing ADAM, an optimizer noted in the literature for its robustness compared to RMSProp.[432]

As presented in Table 3, the model now successfully trains, achieving a fail ratio of 50.60% for the copy task, 48.20% for the repeat copy task, and 52.60% for the reverse task across 500 test runs. The mean accuracies are notably consistent, with the copy task at 0.988108 and the reverse task closely following at 0.987820, while the repeat copy task records a slightly lower mean accuracy of 0.960120. The minimal standard deviation observed across all tasks indicates stable performance, provided the model trains successfully. The loss value of the model is also very consistent on every task, and the values differ in the same pattern as the accuracy. The total number of trainable parameters is 16,020 for the copy and reverse tasks, and 31,772 for the repeat copy task.

| **Task** | **Accuracy** | | **Loss** | | **Fail Ratio** | **Parameter** | **Version** |
|---|---|---|---|---|---|---|---|
| | mean | std | mean | std | | | |
| **C** | 0.988108 | 0.009293 | 0.513295 | 0.007195 | 50.60% | 16,020 | v2 |
| **RC** | 0.960120 | 0.007670 | 0.569178 | 0.003878 | 48.20% | 31,772 | v2 |
| **R** | 0.987820 | 0.010419 | 0.514107 | 0.005906 | 52.60% | 16,020 | v2 |

Tab. 3: SNTM Results Iteration 4 (v2); Legend: C (Copy), RC (Repeat Copy), R (Reverse)

### SNTM Version 3 (Design Cycle Iteration 5)

During the next iteration, the focus will be on understanding why the initial model version failed to train. To isolate the issue, the RMSProp optimizer is reintroduced with the same learning rate of $10^{-3}$ used in SNTM Version 2. This approach will help determine whether the training issues are due to the optimizer or the learning rate.

This version fails to train for both the copy and reverse tasks, as indicated by a 100% failure ratio, detailed in Table 4. However, the repeat copy task shows some success, training with a

[431] These figures are not explicitly listed in `evaluation.ipynb` for this version but are consistent with those of SNTM Version 4, where only the optimizer and learning rate differ.
[432] Discussed in Subsection 2.2.3 Model Training and Subsection 4.2.3 Hyperparameter Selection

fail ratio of 88.60%. In the 11.40% of cases where the model does train successfully, it achieves a mean accuracy of 0.944217 with a standard deviation of 0.009524. The mean loss is recorded at 0.574430, with a standard deviation of 0.003882. These figures indicate a slightly lower accuracy and a slightly higher loss compared to SNTM Version 2. The fact that this version also fails to train so often indicates that the RMSProp optimizer, commonly used in the literature, is not suitable for the developed SNTM architecture. As a result, the ADAM optimizer will be employed in subsequent refinement iterations.

| **Task** | **Accuracy** | | **Loss** | | **Fail Ratio** | **Parameter** | **Version** |
|---|---|---|---|---|---|---|---|
| | mean | std | mean | std | | | |
| **C** | - | - | - | - | 100% | - | v3 |
| **RC** | 0.944217 | 0.009524 | 0.574430 | 0.003882 | 88.60% | 31,772 | v3 |
| **R** | - | - | - | - | 100% | - | v3 |

Tab. 4: SNTM Results Iteration 5 (v3); Legend: C (Copy), RC (Repeat Copy), R (Reverse)

### SNTM Version 4 (Design Cycle Iteration 6)

The fourth iteration of the SNTM architecture explores the influence of the learning rate on training stability. Previous findings suggested that the RMSProp optimizer is inadequate, frequently leading to training failures. To investigate the influence of the learning rate, it is increased from $10^{-3}$ to $3 \times 10^{-3}$, while switching the optimizer back to ADAM. All other hyperparameters remain unchanged from the previous version, as detailed in Table 2.

The increased learning rate results in a significantly reduced fail ratio of 20% to 30%. As indicated in Table 5, the mean accuracy for both the copy and reverse tasks has improved, reaching 0.999259 and 0.999191 respectively. Similarly, the repeat copy task demonstrates an enhanced mean accuracy of 0.966982. When compared to the outcomes from version 2, detailed in Table 3, the metrics observed across all tasks in version 4 demonstrate a higher performance.

| **Task** | **Accuracy** | | **Loss** | | **Fail Ratio** | **Parameter** | **Version** |
|---|---|---|---|---|---|---|---|
| | mean | std | mean | std | | | |
| **C** | 0.999259 | 0.001631 | 0.504848 | 0.001984 | 23.40% | 16,020 | v4 |
| **RC** | 0.966982 | 0.009342 | 0.566736 | 0.003658 | 29.60% | 31,772 | v4 |
| **R** | 0.999191 | 0.001600 | 0.506665 | 0.004060 | 20.40% | 16,020 | v4 |

Tab. 5: SNTM Results Iteration 6 (v4); Legend: C (Copy), RC (Repeat Copy), R (Reverse)

Apparently, a higher learning rate correlates with fewer training failures. However, an excessively high learning rate might cause the loss function to fluctuate around a minimum, hindering

convergence.[433] Therefore, it's not a solution to simply increase the learning rate infinitely to reduce the fail ratio.

### SNTM Version 5 (Design Cycle Iteration 7)

Since increasing the learning rate indefinitely is not a viable solution to reduce the fail ratio, the next iteration focuses on reducing the model's complexity. Zaremba/Sutskever 2015 report that their model's training failed randomly once the total number of parameters surpassed 20,000.[434] To reduce the trainable parameters, the number of units in the controller is decreased from 100 to 50, and the number of memory locations is reduced from 128 to 16. All other hyperparameters remain unchanged from the previous version. The results of this iteration are summarized in Table 6.

Reducing the controller's complexity results in $5,558$ trainable parameters for the copy and reverse tasks and $14,110$ for the repeat copy task, down from $16,020$ and $31,772$ in the earlier versions. The fail ratio for the copy task decreased from 23.40% to 19%, and for the repeat copy task, it dropped from 29.60% to 25.40%. The fail ratio for the reverse task remains around 20%. These observations suggest that simplifying the model's complexity does not necessarily reduce the fail ratio. Also noteworthy is the model's loss and accuracy, which remain relatively constant despite the reduction in trainable parameters. The mean accuracy is slightly lower than in previous versions, also with minimal variability as indicated by the standard deviations. This trend is also observed in the loss values. This shows that substantial reducing the trainable parameters[435] does not significantly impact the model's performance, as the accuracy and loss values are only slightly lower[436] than in the previous version. The changes of optimizer and learning rate in the previous versions have a more significant impact on the performance..

| **Task** | **Accuracy** | | **Loss** | | **Fail Ratio** | **Parameter** | **Version** |
|---|---|---|---|---|---|---|---|
| | mean | std | mean | std | | | |
| **C** | 0.998204 | 0.002580 | 0.509639 | 0.003219 | 19.00% | $5,558$ | v5 |
| **RC** | 0.958673 | 0.004763 | 0.568467 | 0.001866 | 25.40% | $14,110$ | v5 |
| **R** | 0.997740 | 0.003053 | 0.501858 | 0.002075 | 21.20% | $5,558$ | v5 |

Tab. 6: SNTM Results Iteration 7 (v5); Legend: C (Copy), RC (Repeat Copy), R (Reverse)

### SNTM Version 6 (Design Cycle Iteration 8)

As discussed in Subsection 4.2.3, the SNTM artifact also needs to be evaluated on longer sequences. To address this goal, the sequence length is increased from 8 to 20 in this iteration.

[433] Cf. Sun et al. 2020, p. 3672
[434] Cf. Zaremba/Sutskever 2015, p. 9
[435] $14,110 \rightarrow 5,558$ and $31,772 \rightarrow 16,020$
[436] $0.999259 \rightarrow 0.998204$, $0.966982 \rightarrow 0.958673$, and $0.999191 \rightarrow 0.997740$

The controller complexity is restored to $1 \times 100$, and the number of memory locations is set to 128 again. All other hyperparameters remain unchanged from the previous version.

The results for longer sequences, detailed in Table 7, show that the copy and reverse tasks have very similar accuracies again, with 0.979265 and 0.979622 respectively. The performance on the repeat copy task is slightly lower, at 0.929274. The loss values follow a similar pattern. Notably, the fail ratio for the repeat copy task at 13.60% is the lowest observed in all iterations.

| Task | Accuracy | | Loss | | Fail Ratio | Parameter | Version |
|---|---|---|---|---|---|---|---|
| | mean | std | mean | std | | | |
| **C** | 0.979265 | 0.004740 | 0.521056 | 0.003700 | 28.60% | 23,976 | v6 |
| **RC** | 0.929274 | 0.011908 | 0.585228 | 0.004506 | 13.60% | 67,676 | v6 |
| **R** | 0.979622 | 0.004539 | 0.520516 | 0.003215 | 24.40% | 23,976 | v6 |

Tab. 7: SNTM Results Iteration 8 (v6); Legend: C (Copy), RC (Repeat Copy), R (Reverse)

### SNTM Version 7 (Design Cycle Iteration 9)

SNTM Version 6 demonstrates a strong performance on longer sequences, yet there is room for improvement on the repeat copy task. Given the strong correlation between controller complexity and performance,[437] the network's complexity is increased to $3 \times 256$ in this iteration.

Table 8 shows that the more complex controller leads to worse accuracy across all tasks compared to version six. The mean accuracies for the copy and reverse tasks are 0.923078 and 0.918945, respectively, while the repeat copy task records a lower mean of 0.805355. The increased standard deviation suggests less stability in model performance. Moreover, the overall fail ratio is significantly higher, but compared to the other tasks for the repeat copy task its lower at 17.40%. This result indicates that longer sequences may help mitigate training failures for the repeat copy task. The total number of trainable parameters for the copy and reverse tasks is $362,292$, and for the repeat copy task, it is $426,152$. These figures represent a significant increase compared to all previous versions. However, the results suggest that a higher number of trainable parameters does not necessarily enhance the model's performance.

| Task | Accuracy | | Loss | | Fail Ratio | Parameter | Version |
|---|---|---|---|---|---|---|---|
| | mean | std | mean | std | | | |
| **C** | 0.923078 | 0.011812 | 0.564215 | 0.006854 | 39.20% | 362,292 | v7 |
| **RC** | 0.805355 | 0.037327 | 0.639893 | 0.012541 | 17.40% | 426,152 | v7 |
| **R** | 0.918945 | 0.011966 | 0.564302 | 0.006958 | 41.80% | 362,292 | v7 |

Tab. 8: SNTM Results Iteration 9 (v7); Legend: C (Copy), RC (Repeat Copy), R (Reverse)

[437] Cf. Zaremba/Sutskever 2015, p.8

# 5 Evaluation and Discussion of the Artifact

In the subsequent chapter, the designed SNTM artifact is evaluated by benchmarking it against an LSTM and a classical NTM model. The results are then discussed, outlining their theoretical and practical significance.

## 5.1 Evaluation Methodology and Specification

Scientific machine learning benchmarking is employed to evaluate the designed SNTM artifact.[438] Benchmarks form the quantitative foundation of research in computer science[439] and play a critical role in exploring the potential benefits and limitations of emerging software.[440] Benchmarking involves obtaining quantitative measures that facilitate meaningful comparisons across various systems or components.[441]

A *scientific machine learning benchmark* is underpinned by a specific scientific objective and incorporates two main components: a dataset and a reference implementation.[442] The dataset, which could be either synthetic or real-world, is selected based on the research objective.[443] The reference implementation, which can be written in any programming language, exemplifies a standard approach to tackling the benchmark problem.[444] Tuning hyperparameters is not part of the benchmarking process, and they are best handled through explicit specification.[445] In the context of machine learning, benchmarking not only supports the testing of algorithmic performance on fixed datasets but also serves as a framework for algorithmic improvements.[446] The collected metrics should be rich enough to enable diverse methods of analysis and exploration, with example metrics including training accuracy and time to solution.[447] Moreover, the energy consumption of the employed algorithms can serve as an additional metric, reflecting the growing importance of efficiency in computational practices.[448] It is essential to employ state-of-the-art algorithms and techniques to ensure the validity and relevance of the benchmark results.[449]

How scientific machine learning benchmarking is applied to the SNTM artifacts, and the specifically defined benchmark criteria, is outlined in the subsequent Section 5.2.

[438] Cf. Thiyagalingam et al. 2022
[439] Cf. Bienia et al. 2008, p. 72
[440] Cf. Ihde et al. 2022, p. 98
[441] Cf. Ihde et al. 2022, p. 99
[442] Cf. Thiyagalingam et al. 2022, p. 414
[443] Cf. Thiyagalingam et al. 2022, p. 415
[444] Cf. Thiyagalingam et al. 2022, p. 414
[445] Cf. Thiyagalingam et al. 2022, p. 414
[446] Cf. Thiyagalingam et al. 2022, p. 414
[447] Cf. Thiyagalingam et al. 2022, p. 414
[448] Cf. Thiyagalingam et al. 2022, p. 415
[449] Cf. Bienia et al. 2008, p. 72

## 5.2 Benchmark Against a Classical NTM and LSTM

The previously developed artifacts SNTM Version 4 and 6 have archived the best results on the 8-bit and 20-bit sequences, in comparisons to the other SNTM versions. Therefore these two artifacts will be evaluated using the scientific machine learning benchmarking. The selection on the two main components, the dataset and the reference implementation, is already discussed based on the current literature in Section 4.2. This ensures the scientific validity and relevance of benchmark results.[450] The datasets are the *copy*, *repeat copy*, and *reverse copy* tasks, and the reference implementation is the LSTM controller-based NTM architecture and a classical LSTM network, which features three stacked hidden layers and 256 units. To ensure the validity of the benchmark results, the training hyperparameters will be the same as for all tested implementations. The classical NTM hyperparameters are chosen to match the SNTM artifact hyperparameters, as discussed in Section 4.7.

| Mo | V | SL | C[451] | Me[452] | H | SR | MR | LR | O | LF |
|---|---|---|---|---|---|---|---|---|---|---|
| SNTM | **v4** | 8 | $1 \times 100$ | $128 \times 8/20$ | 1 | 0 | 4 | $3 \times 10^{-3}$ | ADAM | ce |
| NTM | **v1** | 8 | $1 \times 100$ | $128 \times 8/20$ | 1 | 0 | 4 | $3 \times 10^{-3}$ | ADAM | ce |
| LSTM | **v1** | 8 | $3 \times 256$ | - | - | - | - | $3 \times 10^{-3}$ | ADAM | ce |
| SNTM | **v6** | 20 | $1 \times 100$ | $128 \times 20/80$ | 1 | 0 | 4 | $3 \times 10^{-3}$ | ADAM | ce |
| NTM | **v2** | 20 | $1 \times 100$ | $128 \times 20/80$ | 1 | 0 | 4 | $3 \times 10^{-3}$ | ADAM | ce |
| LSTM | **v2** | 20 | $3 \times 256$ | - | - | - | - | $3 \times 10^{-3}$ | ADAM | ce |

Tab. 9: Hyperparameter Configurations for SNTM Evaluation; Legend: Mo (Model), V (SNTM Version), SL (Sequence Length), C (Controller/Network), Me (Memory), H (Number of Heads), SR (Shift Range), MR (Max Repetitions for Repeat Copy), LR (Learning Rate), O (Optimizer), LF (Loss Function), ce (sigmoid cross-entropy with logits);

Table 9 shows the hyperparameter configurations for the evaluation of the SNTM artifact. The SNTM Version 4 is tested against the NTM Version 1 and the LSTM Version 1 on the 8-bit sequencesa. The SNTM Version 6 is tested against the NTM Version 2 and the LSTM Version 2 on the 20 bits sequence length. The evaluation is conducted with 500 runs to ensure statistical significance, and the results are averaged over the runs. The test architecture is detailed in Appendix 5/2. The subsequently discussed numerical results and plots are taken from the `evaluation.ipynb` notebook.

The defined metrics that will be used to evaluate the artifacts are the *accuracy* and the cross-entropy *loss*, on an unseen test dataset. The *accuracy* is the ratio of correctly predicted bits to the total number of predicted bits. Additionally, the *training time*, in seconds, will be measured and compared.[453] The *parameters* of the model and the *fail ratio*, the ratio of failed runs to the total number of runs, are also considered.

[450] Cf. Bienia et al. 2008, p. 72
[451] Notated as: `layer` × `units`
[452] Notated as: `memory size` × `memory locations (copy, reverse)`/ `memory locations (repeat copy)`
[453] To assure a fair measurement, the individual test runs are conducted without background applications running.

| Task | Accuracy | | Loss | | T-Time | | FR | P | Model |
|---|---|---|---|---|---|---|---|---|---|
| | mean | std | mean | std | mean | std | | | |
| **C** | 0.9993 | 0.0016 | 0.5048 | 0.0020 | 6.40 | 0.80 | 23.40% | 16,020 | SNTM v4 |
| **C** | 0.9979 | 0.0039 | 0.5048 | 0.0050 | 7.29 | 1.02 | 0% | 48,720 | NTM v1 |
| **C** | 0.9704 | 0.0162 | 0.5137 | 0.0082 | 15.13 | 1.18 | 0% | 1,324,112 | LSTM v1 |
| **RC** | 0.9670 | 0.0093 | 0.5667 | 0.0037 | 6.89 | 1.24 | 29.60% | 31,772 | SNTM v4 |
| **RC** | 0.9999 | 0.0001 | 0.5534 | 0.0025 | 8.15 | 0.75 | 0% | 71,672 | NTM v1 |
| **RC** | 0.8907 | 0.0826 | 0.5962 | 0.0322 | 15.82 | 1.19 | 0% | 1,355,104 | LSTM v1 |
| **R** | 0.9992 | 0.0016 | 0.5067 | 0.0041 | 6.44 | 0.66 | 20.40% | 16,020 | SNTM v4 |
| **R** | 0.9977 | 0.0045 | 0.5036 | 0.0040 | 7.30 | 0.74 | 0% | 48,720 | NTM v1 |
| **R** | 0.9690 | 0.0176 | 0.5130 | 0.0086 | 15.19 | 1.12 | 0% | 1,324,112 | LSTM v1 |

Tab. 10: Evaluating Benchmark 1 Performance on 8-Bit Sequences[454]; Color highlighting indicates the best (green), moderate (yellow), and lowest (red) model in comparison, per metric and task. Legend: T-Time (Time of Training in sections), FR (Fail Ratio), P (Parameter), C (Copy), RC (Repeat Copy), R (Reverse)

The first benchmark employs SNTM_v4, LSTM_v1, and NTM_v1, assessing their performance on the copy, reverse, and repeat copy tasks. These tasks utilize 8-bit sequences with a maximum of four repetitions for the repeat copy task. The results of this benchmark are presented in Table 10.

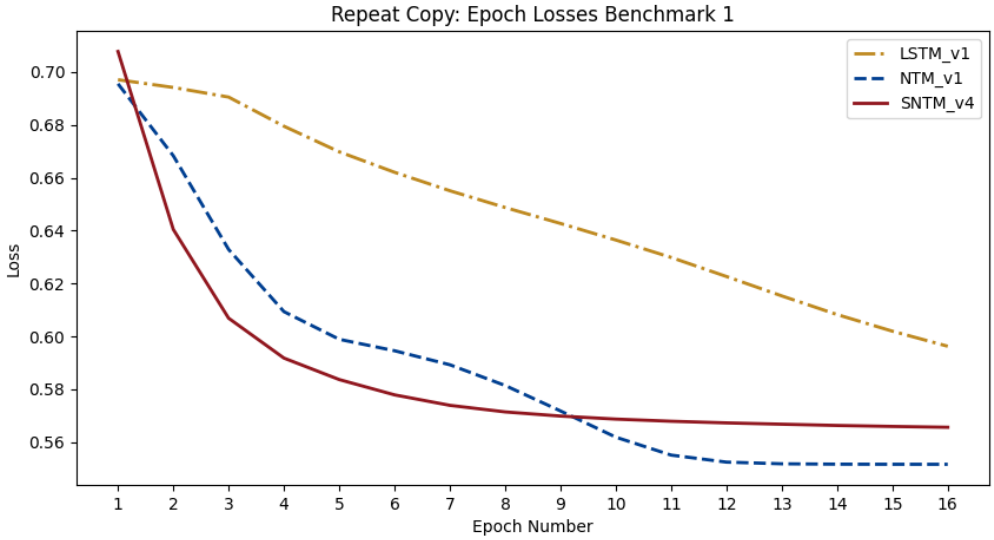



Fig. 11: Benchmark 1 (Repeat Copy): Epoch Losses on Train Dataset[455]

Figure 11 depicts the loss values per epoch on the train split for the repeat copy task. For the loss values, the average of 500 runs is also considered. Despite starting with the highest loss, the SNTM model demonstrates a quicker learning rate over all epochs compared to the LSTM. Until the ninth epoch, the SNTM even maintains a lower loss than the NTM. However, by the end of the epochs, the NTM records the lowest loss. For other tasks, depicted in Appendix 6/1,

[454] All numerical results are taken from the `evaluation.ipynb` notebook.
[455] Plot created within the `evaluation.ipynb` notebook.

the SNTM model also converges faster than the LSTM. The NTM and SNTM models display similar loss values for the copy and reverse tasks, with the NTM slightly outperforming the SNTM at the end of the epochs for the copy task. Regarding training duration, the SNTM is the fastest, averaging around 6.5 seconds, which is around one second less than the NTM. The LSTM model requires the longest training time, approximately 15 seconds for all tasks. However, the SNTM exhibits a higher failure rate, with fail ratios of 23.40% for the copy task, 29.60% for the repeat copy task, and 20.40% for the reverse task. In contrast, the other two models do not fail to train once for all tasks. The SNTM model achieves the highest accuracy scores for two tasks, recording 0.9993 for the copy task and 0.9992 for the reverse task, outperforming the other models. The NTM closely follows with accuracies of 0.9979 for the copy and 0.9977 for the reverse task, also with a slightly higher standard deviation than the SNTM. For the repeat copy task, the SNTM scored 0.9670, while the NTM led with 0.9999, demonstrating the lowest variability with a standard deviation of 0.0001. In contrast, the LSTM model has the lowest accuracy for all tasks, with 0.9704 for the copy task, 0.8907 for the repeat copy task, and 0.9690 for the reverse task. The boxplots in Appendix 6/2 illustrate that the SNTM has fewer outliers in accuracy for the copy and reverse tasks compared to other models. The loss values mirror these accuracy patterns, except for the reverse task, where the NTM exhibits a slightly lower loss than the SNTM. The LSTM model has the highest trainable parameter count, utilizing over 1.3 million, which is more than 80 times the number of parameters in the SNTM. The SNTM has the fewest parameters, with counts of 16,020 and 31,772 for different tasks. The NTM model holds the intermediate position, with 48,720 and 71,672 parameters.

| **Task** | **Accuracy** | | **Loss** | | **T-Time** | | **FR** | **P** | **Model** |
|---|---|---|---|---|---|---|---|---|---|
| | mean | std | mean | std | mean | std | | | |
| **C** | 0.9793 | 0.0047 | 0.5211 | 0.0037 | 8.06 | 1.24 | 28.60% | 23,976 | SNTM v6 |
| **C** | 0.9975 | 0.0020 | 0.5070 | 0.0017 | 9.58 | 2.11 | 0% | 60,276 | NTM v2 |
| **C** | 0.9551 | 0.0126 | 0.5360 | 0.0084 | 15.32 | 2.39 | 0% | 1,339,832 | LSTM v2 |
| **RC** | 0.9293 | 0.0119 | 0.5852 | 0.0045 | 8.55 | 1.43 | 13.60% | 67,676 | SNTM v6 |
| **RC** | 0.9858 | 0.0070 | 0.5590 | 0.0028 | 11.14 | 1.66 | 0% | 121,976 | NTM v2 |
| **RC** | 0.7062 | 0.0160 | 0.67144 | 0.0053 | 14.77 | 3.68 | 0% | 1,418,032 | LSTM v2 |
| **R** | 0.9796 | 0.0045 | 0.5205 | 0.0033 | 8.20 | 1.35 | 24.40% | 23,976 | SNTM v6 |
| **R** | 0.9976 | 0.0020 | 0.5057 | 0.0026 | 9.13 | 2.57 | 0% | 60,276 | NTM v2 |
| **R** | 0.9548 | 0.0121 | 0.5378 | 0.0081 | 15.80 | 2.39 | 0% | 1,339,832 | LSTM v2 |

Tab. 11: Evaluating Benchmark 2 Performance on 20-Bit Sequences[456]; Color highlighting indicates the best (green), moderate (yellow), and lowest (red) model in comparison, per metric and task. Legend: T-Time (Time of Training), FR (Fail Ratio), P (Parameter), C (Copy), RC (Repeat Copy), R (Reverse)

The second benchmark evaluates models on 20-bit sequences, with up to four repetitions for the repeat copy task. This benchmark includes the developed SNTM_v6 model and compares it to NTM_v2 and LSTM_v2 models. The Results are detailed in Table 11. Again, the SNTM

[456] All numerical results are taken from the `evaluation.ipynb` notebook.

emerges as the fastest model to train, recording around 8.2 seconds. This is about a second faster than the NTM and nearly half the time required by the LSTM. The second benchmark also shows a higher standard deviation for training time compared to the first benchmark. Appendix 6/3 illustrates the loss values over the epochs for all three tasks. For the copy and reverse tasks, the SNTM and NTM perform similarly, as in the first benchmark, but the SNTM outperforms the LSTM by converging in fewer epochs across all tasks. The training fail ratio is consistent with the first test, with 28.60% for the copy task, 13.60% for the reverse copy, and 24.40% for the reverse task. Neither the NTM nor the LSTM failed to train in any instance. The NTM achieves slightly higher accuracy across all tasks compared to the SNTM, with scores of 0.9975 for the copy task, 0.9858 for the repeat copy task, and 0.9976 for the reverse task. In contrast, the SNTM scores 0.9793 for the copy task, 0.9293 for the repeat copy task, and 0.9796 for the reverse task. Yet, the SNTM still outperforms the LSTM on all metrics. The loss patterns mirror those of the accuracy results. Additionally, the standard deviation indicates stability in the measured accuracy and loss values, with a few outliers in the accuracy, as depicted in Appendix 6/4, confirming this observation.

Based on the first benchmark results, the SNTM_v4 model outperforms the benchmark networks for the copy and reverse tasks. The classical NTM_v1 model performs the best on the repeat copy task, but SNTM_v4 also surpasses LSTM_v1 on this task. Overall, SNTM_v4 demonstrates superior performance compared to the LSTM_v1 across all tasks and holds a slight edge over the NTM_v1 on certain tasks. The second benchmark further demonstrates that SNTM_v6 surpasses LSTM_v2 across all tasks but slightly lags behind NTM_v2 in accuracy and loss. The classical NTM appears more proficient at handling longer sequences, consistently outperforming SNTM on 20-bit sequences and on the more complex 8-bit repeat copy task, managing a 32-bit[457] sequence output. Despite higher fail ratios, SNTM models in both benchmarks train fastest and maintain the lowest parameter count.

## 5.3 Objective Reflection

The objective of this thesis was to construct an SNTM model and evaluate its effectiveness compared to conventional NTM and LSTM architectures across three specific tasks. The evaluation addressed the central research question: **Is it feasible to employ an SNN as an NTM controller network?** This question can be answered with yes. The findings of the thesis answer the question by demonstrating the feasibility of using an SNN as a controller for an NTM architecture, as showcased by the developed SNTM model. This model, integrated within the TensorFlow `tf.keras.layers.Layer` class, seamlessly fits into a standard TensorFlow workflow, proving its practical applicability.

Subsequently, the second research question focused on examining **potential limitations** and **advantages** of the developed model: **How does the resulting SNTM perform compared**

[457] $8 \times 4$, where 8 is the bit sequences and 4 the maximum repetitions

**to classical NTM and LSTM architectures?** The benchmark architectures were derived from the literature, ensuring a scientifically relevant framework for evaluating the SNTM's performance. Compared to the chosen LSTM architectures derived from the literature, the developed SNTM model performs better on all tested tasks. Although it does not outperform the NTM model with an LSTM controller on all tasks, it demonstrates superior performance on some. Notably, the SNTM model learns faster than the benchmark models on the tested sequence tasks and does so with the fewest parameters. However, limitations were observed in the SNTM's training stability, where it occasionally failed to converge to a stable solution. This problem was not encountered with the conventional NTM and LSTM models, which trained more consistently. Although the fail ratio was reduced, it was not eliminated, suggesting room for further refinement. Additionally, the SNTM model's testing was confined to simpler sequence tasks, and its effectiveness on more complex tasks like image recognition or natural language processing remains unexplored in this thesis.

## 5.4 Theoretical and Practical Significance

According to Gregor/Hevner, A. 2013, the SNTM artifact introduced in this thesis represents a novel solution to well-known problems, thus marking an *improvement*.[458] In this context, the known problems are the benchmarking tasks, whereas the SNN-enhanced NTM model introduces the novel solution. An *improvement* presents both a theoretical and practical advancement, contributing new knowledge to the applied field.[459]

The practical significance of the proposed artifact lies in its implementation within the TensorFlow framework, which integrates the model into a classical TensorFlow workflow. This integration provides a template for further research in the field. Furthermore, the implementation facilitates testing the model on practical tasks and deploying it in real-world applications. It has been observed that the SNTM model trains faster than traditional ANN models. This is also noted by Woźniak et al. 2020 for single SNU cells. Since SNN architectures can be executed on energy-efficient neuromorphic hardware,[460] this model represents a step toward more energy-efficient deep learning models in the domain of NTMs and memory-enhanced networks. To achieve this, the implementation must be adapted to run on specific hardware. The SNTM provides a starting point for further research in this area.

This work merges the brain-inspired concepts of SNN and NTM architectures, establishing a foundation for enhancing SNN models with external memory, which is essential for solving complex cognitive tasks.[461] The architecture developed in this thesis performs well against selected benchmark models, marking it as a step toward integrating SNN into further classical deep learning frameworks. This represents a theoretical contribution to the fields of deep learning

[458] Cf. Gregor/Hevner, A. 2013, p. 345
[459] Cf. Gregor/Hevner, A. 2013, p. 346
[460] Cf. Carrillo et al. 2012; Merolla et al. 2014; Zheng, H. et al. 2024
[461] Cf. Greve/Jacobsen/Risi 2016, p. 117

and neuromorphic computing. To fully unleash the potential of this architecture, the stability of the training process needs improvement. The current limitations in stability stem from poor gradient stability, a common issue in RNN architectures.[462] This issue could be addressed by refining the gradient clipping mechanism, a well-known technique in deep learning used to stabilize training.[463] Additionally, testing other optimizers might prevent the model from failing to converge, thereby achieving a more stable solution.

Although the SNTM demonstrates superior performance on selected tasks compared to benchmark LSTM models, this does not necessarily imply that the SNTM artifact is generally superior to the LSTM architecture. The benchmark models were not fine-tuned for the tested tasks but were derived from the current state of the art, ensuring scientific relevance in the evaluation. However, more extensive benchmarking across a broader range of tasks and models would be required to make a more generalized statement about the SNTM model's performance. This thesis serves as a starting point, as suggested in current literature,[464] for further task implementations. Common tasks like the bAbI and the sequential MNIST datasets could be utilized for further evaluation.[465] Further research should also explore different enhanced NTM architectures, such as those proposed by Zaremba/Sutskever 2015, Greve/Jacobsen/Risi 2016, and Yang/Rush 2017, and apply an SNN as the controller. Implementing the model in PyTorch would be the next step to integrate the model within another Python deep learning framework, thereby reaching a broader research audience. Since the controller is responsible for memory operations, experimenting with different spiking neural models, such as the Hodgkin-Huxley[466] or the Fitz-Hugh Nagumo[467] model, would offer additional avenues to enhance the SNTM model.

Since the NTM is inspired by the Turing Machine,[468] more algorithmic tasks, like sorting or searching, could be tested. Comparing the NTM architecture to the theoretical definition of a Turing Machine would be an essential step to further evaluate the model's computing capabilities.

[462] Cf. Mikhaeil/Monfared/Durstewitz 2022, p. 1 et seq.
[463] Cf. Mikhaeil/Monfared/Durstewitz 2022, p. 2
[464] Cf. Graves/Wayne/Danihelka 2014; Danihelka et al. 2016; Rae et al. 2016; Mark Collier/Beel 2018
[465] Cf. Wayne et al. 2016; Chandar et al. 2016; Gülçehre/Chandar/Bengio 2017; Boloukian/Safi-Esfahani 2020
[466] Cf. Hodgkin/Huxley, A. F. 1952
[467] Cf. FitzHugh 1961
[468] Cf. Graves/Wayne/Danihelka 2014, p. 1 et seq.

# 6 Conclusion

The objective of this thesis was to design and implement an NTM with an SNN as its controller network to demonstrate the feasibility of such an SNTM. SNU cells were utilized to simulate the SNN controller, enabling the integration of the designed network architecture into a classical TensorFlow workflow. This provides a starting point for further research in the area of SNTM architecture.

Additionally, this thesis explores the advantages and limitations of the SNN controller network compared to a classical NTM with an LSTM as the controller network and a simple LSTM architecture without additional memory enhancement. The benchmark networks, tasks, and hyperparameters are derived from the literature to ensure an expressive comparison.

## 6.1 Critical Reflection of the Results and Methodology

The feasibility of the SNTM has been demonstrated through a detailed implementation within the TensorFlow framework. The developed SNTM outperforms the benchmark LSTM in all three benchmark tests but is only occasionally superior to the classical NTM. For the tested 8-bit sequence, the SNTM demonstrates the highest accuracy within the benchmark for the copy and reverse tasks with accuracies of 0.9993 and 0.9992, respectively. The classical NTM performs similarly, albeit slightly worse, with accuracies of 0.9979 and 0.9977. The LSTM records accuracies of 0.9704 and 0.9690. When handling longer sequences, such as the reverse copy task and the 20-bit sequences, the SNTM is outperformed by the classical NTM. Nevertheless, the SNTM still outperforms the benchmark LSTM in all scenarios involving longer sequences.[469] The training time for the SNTM is significantly lower, requiring less than half the time compared to the classical LSTM, which is around 15 seconds. Depending on the task and sequence length, the SNTM needs around 6.5 to 8 seconds to train 16 epochs. In contrast, the classical NTM needs 7.5 to 10 seconds. Despite being fast, one area for improvement is the stability of the training process. While the NTM and LSTM train consistently in all test cases, the SNTM sometimes fails to converge after an arbitrary number of epochs, resulting in a `NaN` loss. Through hyperparameter tuning during the Design Cycle, it was possible to reduce the failure ratio to around 25% of the test cases. Another limitation is the utilization of the simple sequential task for the benchmark evaluation, which is neither representative of the full capabilities of the SNTM nor the benchmark networks. Nevertheless, compared to the benchmark models, the SNTM is a good starting point and a promising approach to enhancing the NTM architecture in further research.

The DSR methodology provides a comprehensive framework that accounts for literature and environmental influences by integrating the Rigor and Relevance Cycles into the actual design process of the SNTM. The iterative approach of DSR is beneficial, ensuring that the design

[469] Discussed in Section 5.2

of the SNTM is not only theoretically underpinned but also practically applicable. This is achieved by continuously refining the artifact. To ensure a structured development approach, the SDR methodology is utilized within the design process. Initially, a conceptual and high-level architecture is defined, followed by the implementation of the actual artifact. Although development processes like this can be inflexible and time-consuming, the iterative nature of DSR allows for continuous refinement and the incorporation of changes from the conceptual design to the actual implementation. During this thesis, additional functions were introduced in the implementation phase, which were not part of the initial design but were necessary to achieve a robust and scalable implementation of the SNTM artifact. The benchmark methodology for the final evaluation provides a standardized way to measure and compare the performance of the SNTM against established models. However, the conducted benchmark is highly specific, comparing only two other architectures with a limited set of tasks. While it is state of the art to compare the performance of newly enhanced NTM architectures against classical NTMs and a classical LSTM, it must be noted that the benchmarking results may not necessarily generalize to other tasks or architectures. Nevertheless, the benchmark methodology is well-suited to prove the feasibility of the SNTM, explore some potential advantages and limitations, and identify further research opportunities.

## 6.2 Future Outlook

The practical implications of our research, including the unique embedding of the developed artifact into the current literature, are extensively discussed in Subsection 5.4. The key part of the theoretical and practical implications is the novel implementation of the SNTM within TensorFlow. This implementation not only provides a practical, usable framework but also serves as a foundation upon which future theoretical research can be built. These potential research areas span a wide range, from applying the SNTM to different tasks to exploring other NTM architectures with an SNN controller.

The benchmarks conducted in this thesis can be extended to more complex tasks like the bAbI dataset or the sequential MNIST dataset. It is also possible to apply the SNTM to more algorithmic tasks like sorting or searching to evaluate its computing capabilities and compare the developed artifact with the properties of an actual Turing Machine. Further research should also explore how the developed SNTM can be applied to real-world applications, such as natural language processing or image recognition.

To be applicable for further research, the SNTM controller network may be replaced with different, more complex mathematical spiking neuron models like the Hodgkin-Huxley or the Fitz-Hugh Nagumo model. These models could improve the SNTM's performance. An interesting approach would be to develop the SNTM on actual neuromorphic hardware, which could lead to a more energy-efficient deep learning model. Further elaboration is also needed in terms of the training's stability.

Since the classical NTM architecture enhanced with an SNN controller network performs well, other further enhanced NTM architectures could be tested with an SNN as the controller network. This would provide a broader understanding of the SNTM model's capabilities and limitations and further elaborate on the potential of an SNN controller network in enhancing the NTM architecture.

# Appendix

## List of Appendices

# Appendix 1: Design Science Research

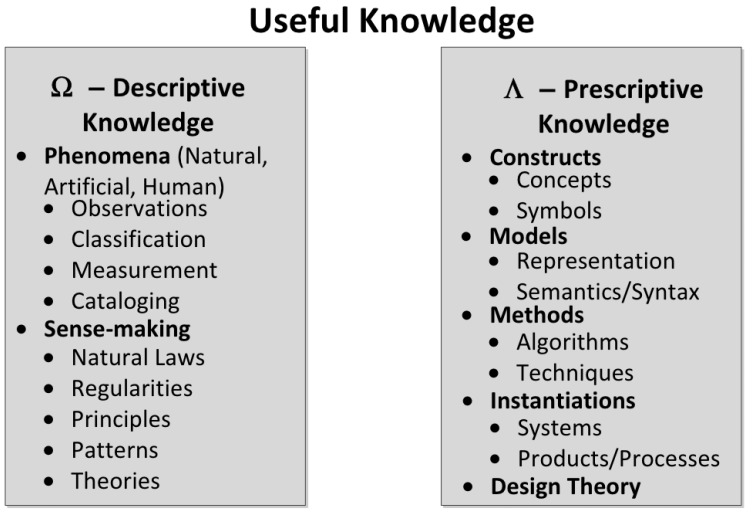
## Appendix 1/1: Publication Schema for a Design Science Research Study

| Table 3. Publication Schema for a Design Science Research Study | |
|---|---|
| **Section** | **Contents** |
| 1. Introduction | *Problem definition, problem significance/motivation, introduction to key concepts, research questions/objectives, scope of study, overview of methods and findings, theoretical and practical significance, structure of remainder of paper.*<br>For DSR, the contents are similar, but the problem definition and research objectives should specify the ***goals*** that are required of the artifact to be developed. |
| 2. Literature Review | *Prior work that is relevant to the study, including theories, empirical research studies and findings/reports from practice.*<br>For DSR work, the prior literature surveyed should include any prior design theory/knowledge relating to the class of problems to be addressed, including artifacts that have already been developed to solve similar problems. |
| 3. Method | *The research approach that was employed.*<br>For DSR work, the specific DSR approach adopted should be explained with reference to existing authorities. |
| 4. Artifact Description | A concise description of the artifact at the appropriate level of abstraction to make a new contribution to the knowledge base.<br>This section (or sections) should occupy the major part of the paper. The format is likely to be variable but should include at least the description of the designed artifact and, perhaps, the design search process. |
| 5. Evaluation | Evidence that the artifact is useful.<br>The artifact is evaluated to demonstrate its worth with evidence addressing criteria such as validity, utility, quality, and efficacy. |
| 6. Discussion | *Interpretation of the results: what the results mean and how they relate back to the objectives stated in the Introduction section. Can include: summary of what was learned, comparison with prior work, limitations, theoretical significance, practical significance,* and *areas requiring further work.*<br>Research contributions are highlighted and the broad implications of the paper's results to research and practice are discussed. |
| 7. Conclusions | *Concluding paragraphs that restate the important findings of the work.*<br>Restates the main ideas in the contribution and why they are important. |

Publication Schema for a Design Science Research Study[470]

[470]Included in: Gregor/Hevner, A. 2013, p. 350

### Appendix 1/2: The Design Science Research Knowledge Base




The Design Science Research Knowledge Base[471]

## Appendix 2: Literature Review: Search Terms

| Search Terms | Keywords/Synonyms | Concept | | |
|---|---|---|---|---|
| | | ANN | SNN | NTM |
| Artificial Neural Network | | x | | |
| ANN Definitions | | x | | |
| Review on Backpropagation Algorithms | optimizers | x | | |
| Recurrent Neural Networks | RNN, LSTM, GRU | x | | |
| Long Short-Term Memory | LSTM | x | | |
| Spiking Neural Networks Review | SNN | | x | |
| Train SNN with Backpropagation | | x | x | |
| Spiking Neural Unit | SNU | x | x | |
| Mathematical SNN Models | | | x | |
| Neural Turing Machine | NTM | x | | x |
| Neural Turing Machines Review | | x | | x |
| Neural Turing Machine Enhancements | | x | | x |

Search Term List for the Literature Review

[471] Included in: Gregor/Hevner, A. 2013, p. 344

# Appendix 3: Neural Networks

## Appendix 3/1: Comparison: Artificial Neural Networks to Biological Networks

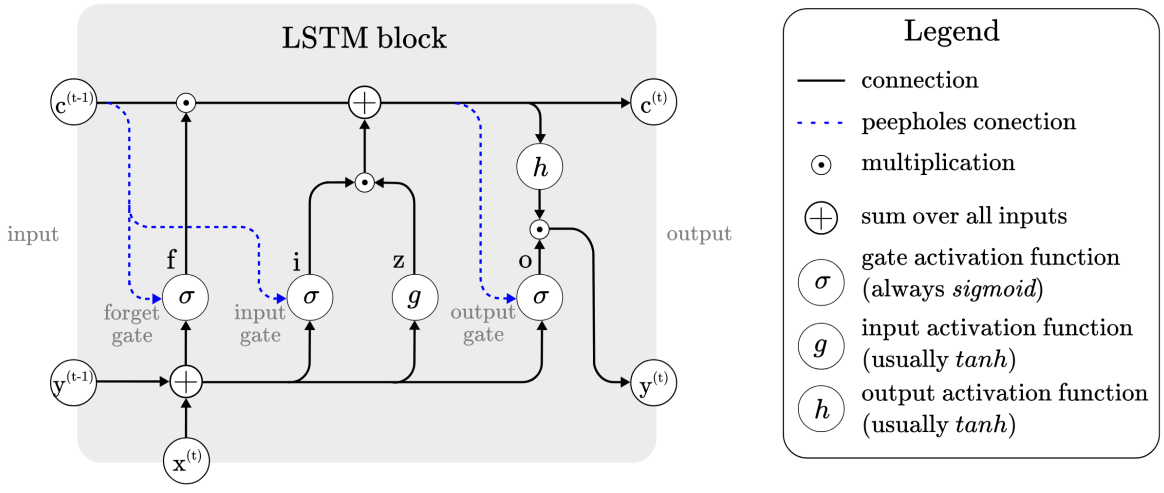
| Biological Neural Networks | Artificial Neural Networks |
|---|---|
| Stimulus | Input |
| Receptors | Input Layer |
| Neural Net | Processing Layer(s) |
| Neuron | Processing Element (Artificial Neuron) |
| Effectors | Output Layer |
| Response | Output and an entry |

Similarities Between Biological Neural Networks and Artificial Neural Networks[472]

| Neurons | Processing Elements |
|---|---|
| Synapses | Weights |
| Dendrites | Summing Function |
| Cell Body | Activation Function |
| Axon | Output |
| Threshold value | Bias |

Similarities of Neurons and Processing Elements (Artificial Neuron)[473]

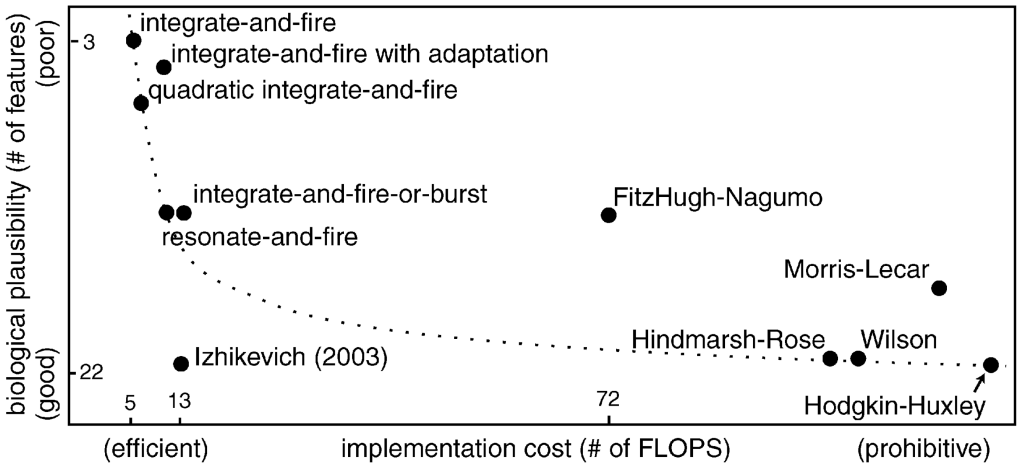
## Appendix 3/2: Long Short-Term Memory Block




LSTM Block Architecture[474]

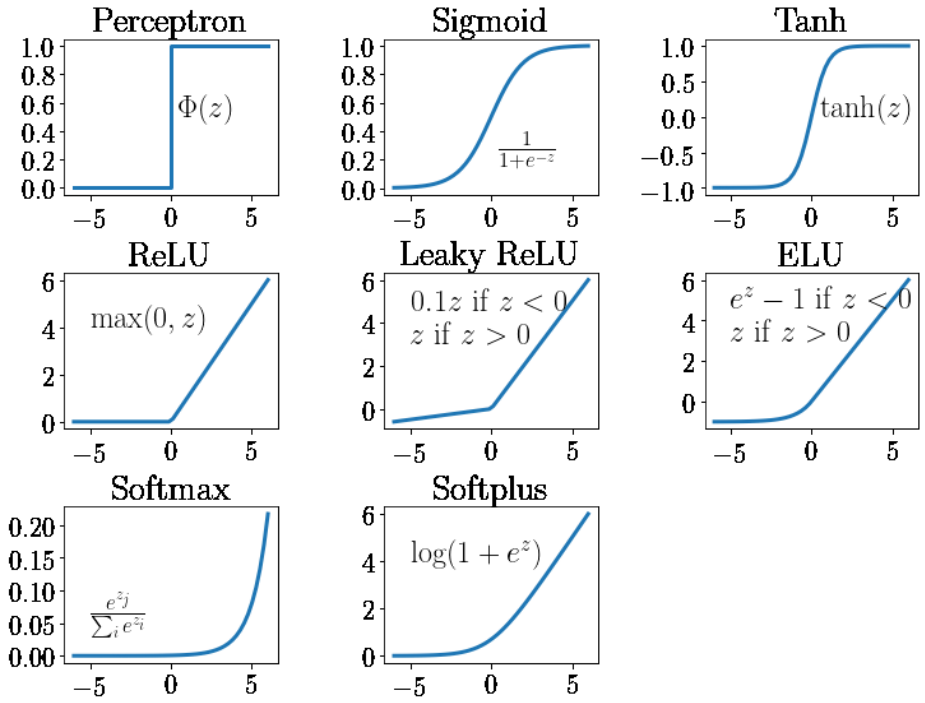
[472]Included in: Guresen/Kayakutlu 2011, p. 428
[473]Included in: Guresen/Kayakutlu 2011, p. 428
[474]Included in: Van Houdt/Mosquera/Nápoles 2020, p. 5932

## Appendix 3/3: Comparison: Spiking Neuron Models




Comparison: Spiking Neuron Models (Implementation Cost and Biological Plausibility)[475]

## Appendix 3/4: Common Activation Functions




Neural Network Activation Functions: Overview [476]

[475] Included in: Izhikevich 2004, p. 1066
[476] Included in: Johnson et al. 2020, p. 31

# Appendix 4: Open-Source Implementation Review

## Appendix 4/1: Neural Turing Machine Hyperparameter

| GitHub Repository | Type | NTM Hyperparameter | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **SL** | **CN** | **La** | **Un** | **ML** | **MLS** | **WH** | **RH** | **SR** |
| Snowkylin 2019 | NTM | 10 | LSTM | 3 | 128 | 20 | 8 | 1 | 1 | 1 |
| Loudinthecloud 2018 | NTM | 20 | LSTM | 1 | 100 | 128 | 20 | 1 | 1 | - |
| Chiggum 2016 | NTM | 1-20 | LSTM | - | - | - | - | 1 | 1 | - |
| Camigord 2017 | NTM | 10 | LSTM | 1 | 100 | 128 | 10 | 1 | 1 | 1 |
| Yeoedward 2016 | NTM | 5 | LSTM | 1 | 100 | 50 | 5 | 1 | 1 | - |
| Snipsco 2015 | NTM | 8 | LSTM | 1 | 100 | 128 | 8 | 1 | 1 | - |
| Carpedm20 2015 | NTM | 8 | LSTM | 1 | 100 | 128 | 8 | 1 | 1 | 1 |
| MarkPKCollier 2018 | NTM | 8 | LSTM | 1 | 100 | 128 | 20 | 1 | 1 | - |
| Google DeepMind 2024[477] | DNC | 16 | LSTM | 1 | 64 | 16 | 16 | 1 | 4 | - |
| HarvardNLP 2017[478] | NTM | - | LSTM | 1 | 50 | 1 | 20 | 1 | 1 | - |
| Ilyasu123 2024[479] | NTM | 30 | LSTM | - | 25 | - | - | - | - | - |
| Jakobmerrild 2017[480] | NTM | 10 | LSTM | - | - | - | - | 1 | 1 | 3 |

Open-Source Implementations: NTM Parameter; Legend: SL (Sequence Length), CN (Controller Network), La (Layers), Un (Units), ML (Memory Locations), MLS (Memory Location Size), WH (Write Heads), RH (Read Heads), SR (Shift Range); All information retrieved from GitHub, accessed on March 20, 2024.

[477] Official Implementation for Wayne et al. 2016
[478] Official Implementation for Yang/Rush 2017
[479] Official Implementation for Zaremba/Sutskever 2015
[480] Official Implementation for Merrild/Rasmussen/Risi 2018

## Appendix 4/2: Trainings Hyperparameter

| GitHub Repository | Training Hyperparameter | | | | |
|---|---|---|---|---|---|
| | **Op** | **BS** | **E / NB** | **LR** | **LF** |
| Snowkylin 2019 | RMSProp, ADAM | 10 | 1000000 | $1 \times 10^{-4}$ | binary cross-entropy |
| Loudinthecloud 2018 | RMSProp | 1 | 50000 | $1 \times 10^{-4}$ | cross-entropy |
| Chiggum 2016 | RMSProp | - | - | - | - |
| Camigord 2017 | RMSProp | 1 | 300000 | $1 \times 10^{-4}$ | binary cross-entropy |
| Yeoedward 2016 | RMSProp | 1 | - | $1 \times 10^{-4}$ | sigmoid cross-entropy |
| Snipsco 2015 | RMSProp | 1 | 1000000 | $1 \times 10^{-4}$ | binary cross-entropy |
| Carpedm20 2015 | RMSProp | 1 | 100000 | $1 \times 10^{-4}$ | softmax cross-entropy |
| Mark Collier/Beel 2018 | ADAM | 32 | 31250 | $1 \times 10^{-3}$ | sigmoid cross-entropy |
| Google DeepMind 2024[481] | RMSProp | 16 | 100000 | $1 \times 10^{-4}$ | sigmoid cross-entropy |
| HarvardNLP 2017[482] | RMSProp, ADAM | 32 | 1000 | $2 \times 10^{-3}$ | - |
| Ilyasu123 2024[483] | RMSProp | 1 | - | $5 \times 10^{-2}$ | cross-entropy |
| Jakobmerrild 2017[484] | - | - | 50 | - | - |

Open-Source Implementations: Training Hyperparameters; Legend: Op (Optimizer), BS (Batch Size), E (Epochs), NB (Number of Batches) LR (Learning Rate), LF (Loss Function); All information retrieved from GitHub, accessed on March 20, 2024.

[481] Official Implementation for Wayne et al. 2016
[482] Official Implementation for Yang/Rush 2017
[483] Official Implementation for Zaremba/Sutskever 2015
[484] Official Implementation for Merrild/Rasmussen/Risi 2018

# Appendix 5: Spiking Neural Turing Machine Implementation

## Appendix 5/1: Data Generation

**Algorithm 8** Repeat and Pad an Array

```
function REPEAT_AND_PAD(arr, max_length)
    result ← [ ]
    for each row ∈ arr do
        repeated_sequence ← np.tile(row, max_length)          ▷ repeat the sequence
        if repeated_sequence.size < max_length then           ▷ pad with zeros if necessary
            repeated_sequence ← np.pad(repeated_sequence, 0)  ▷ shortend here
        end if
        result.append(repeated_sequence)
    end for
    return np.array(result)                                   ▷ return the result
end function
```

The `repeat_and_pad()` function is designed to manipulate sequences within an array by repeating them and ensuring they all have a uniform length. This function accepts an array, `arr`, and a target `max_length`, repeating each sequence according to a specified number of repetitions. If a sequence is shorter than `max_length` after being repeated, it is padded with zeros to meet the required length. The processed sequences, referred to as `repeated_sequence`, are stored in a list named `result`, which is then casted to an `np.array` before being returned. This function plays a crucial role in generating data for the repeat copy task.

## Appendix 5/2: Test Architecture

The data used for analysis and evaluation is stored in the `data/` directory. The `metrics.csv` file contains the metrics of the trained models, while the `model_comparison.csv` file contains the summary of all the conducted test runs for each trained model. The `utils/` directory contains the Python files used for data management, analysis, and testing. The `config_benchmark_lstm.py`, `config_benchmark_ntm.py`, and `config_sntm.py` files contain the configurations for the benchmark models and the SNTM models, respectively. The `evaluation_benchmark_lstm.ipynb`, `evaluation_benchmark_ntm.ipynb`, and `evaluation_sntm.ipynb` notebooks create a summary of all the conducted test runs (saved in `metrics.csv`) for each version and update the `model_comparison.csv` with the `update_comparison()` function from `utils/analysis.py`.

The `ntm_cell.py` file contains the implementation of the NTM cell, while the `ntm_run_test.ipynb` notebook conducts the tests. The `plots/` directory contains the visualizations created in the `evaluations.ipynb` notebook. The `requirements.txt` file lists all the necessary dependencies to run the code developed within this thesis.

**Thesis Folder Structure:**

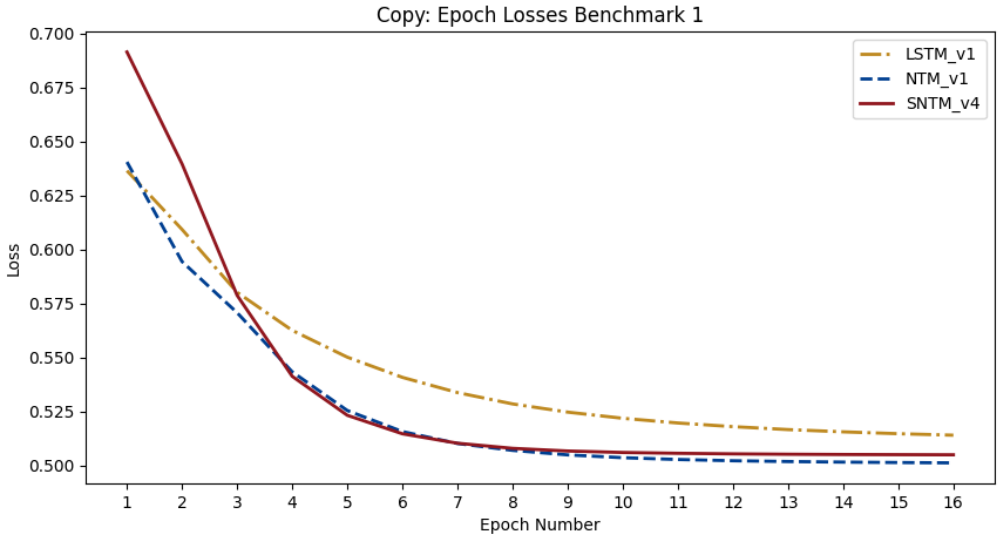
```
SNTM_jonasMichel
├── data
│   ├── metrics.csv
│   └── model_comparison.csv
├── plots
│   └── all plots
├── utils
│   ├── analysis.py
│   ├── data_mgmt.py
│   └── testing.py
├── config_benchmark_lstm.py
├── config_benchmark_ntm.py
├── config_sntm.py
├── evaluation_benchmark_lstm.ipynb
├── evaluation_benchmark_ntm.ipynb
├── evaluation_sntm.ipynb
├── evaluation.ipynb
├── ntm_cell.py
├── ntm_run_test.ipynb
└── requirements.txt
```

Within the `ntm_run_test.ipynb`, for each model version, the respective configuration file is first loaded. The function `create_datasets()` from `utils/data_mgmt.py` is used to randomly create datasets. Then, for each test run iteration, a new model is instantiated with the given hyper-parameters and trained using the `init_model()` function from `utils/testing.py`. Depending on the configuration file, the model is either an LSTM, a classical NTM, or an SNTM. The entire model is structured as a `tf.keras.models.Sequential()` with additional layers tailored to suit the input and output sizes. Because the sequential model does not support two outputs in one layer (output and hidden states), an extra `NTMLayer` is created within the `ntm_cell.py` file. This layer is essentially containing the configured NTM cell and returns only the outputs of the cell. The whole sequential model is then trained and evaluated, and all metrics are saved to the `metrics.csv` file.

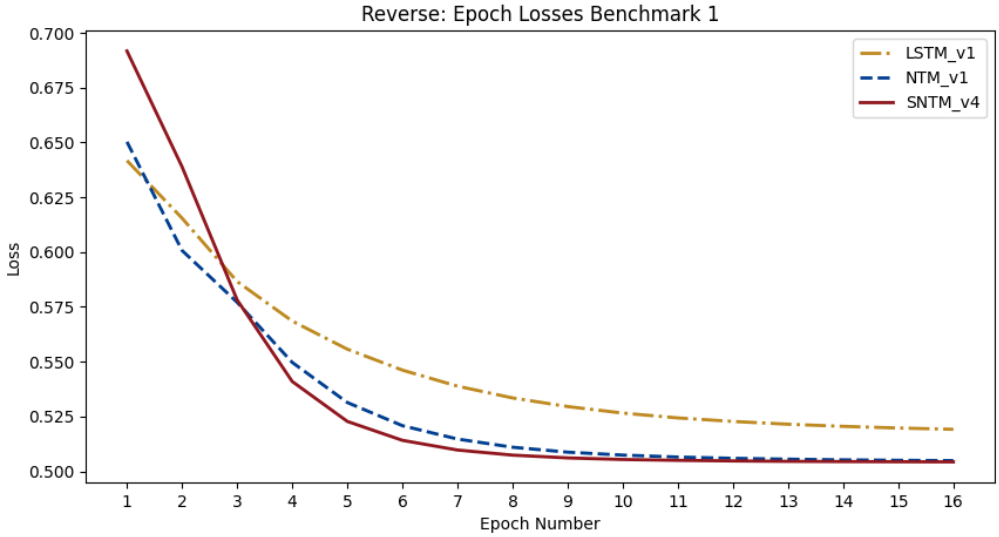
# Appendix 6: Artifact Evaluation

## Appendix 6/1: Benchmark 1: Epoch Losses

The following line charts illustrate the epoch losses of the SNTM model compared to the LSTM and NTM models. The line charts are generated in the `evaluation.ipynb` notebook, using the Python library `matplotlib`.



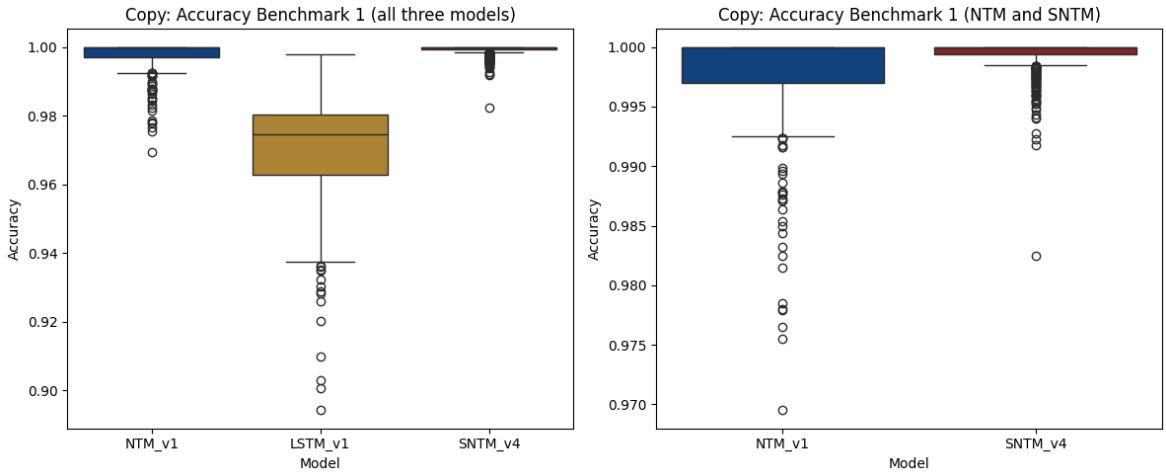

Benchmark 1 (Copy): Epoch Losses Line Chart



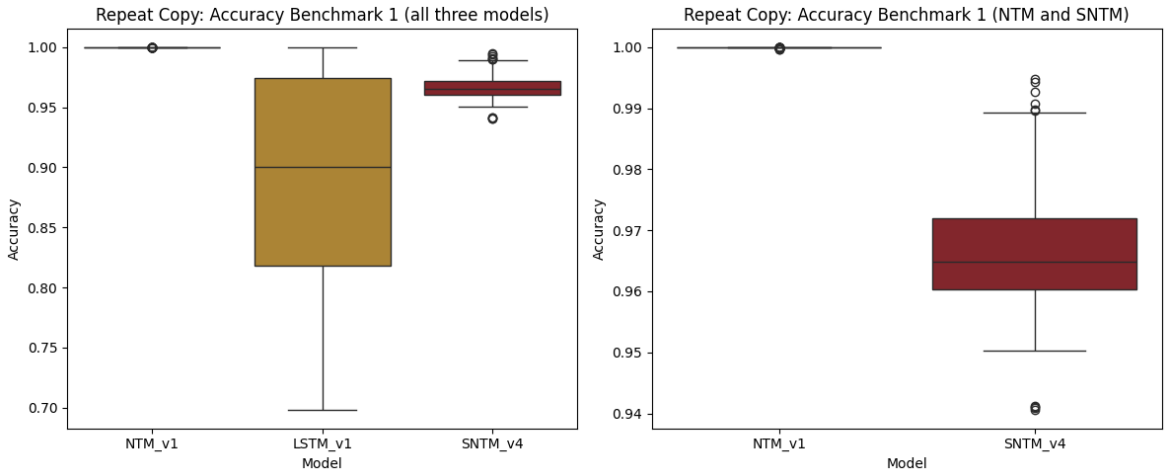

Benchmark 1 (Reverse): Epoch Losses Line Chart

## Appendix 6/2: Benchmark 1: Accuracy Boxplots

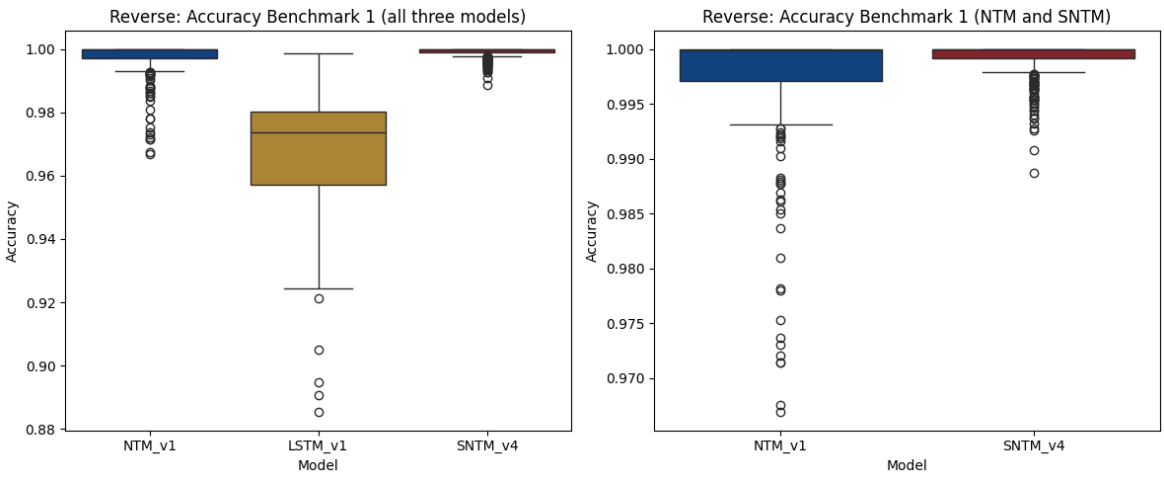
The following boxplots illustrate the accuracy of the SNTM model compared to the LSTM and NTM models. The boxplots are generated in the `evaluation.ipynb` notebook, using the Python library `seaborn`.




Benchmark 1 (Copy): Accuracy Boxplot



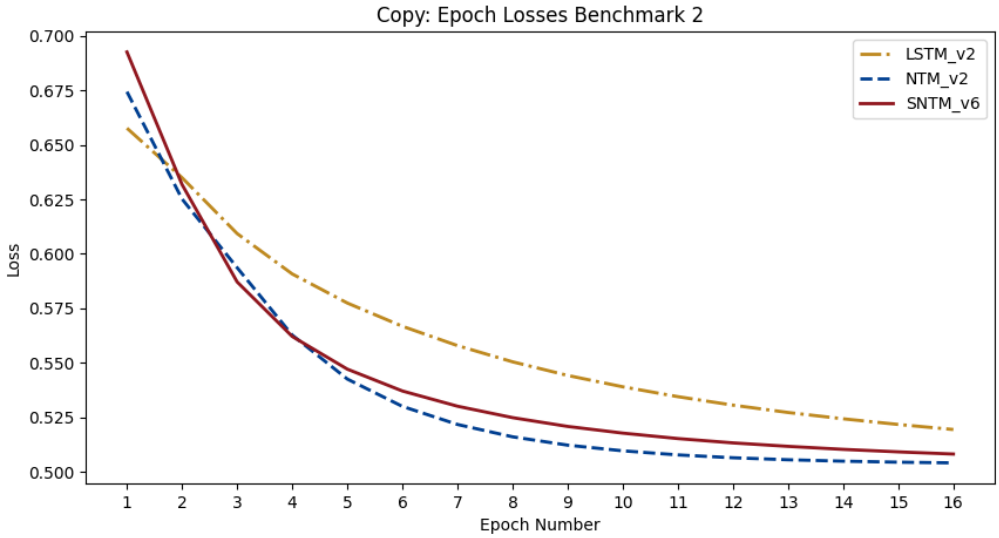

Benchmark 1 (Repeat Copy): Accuracy Boxplot



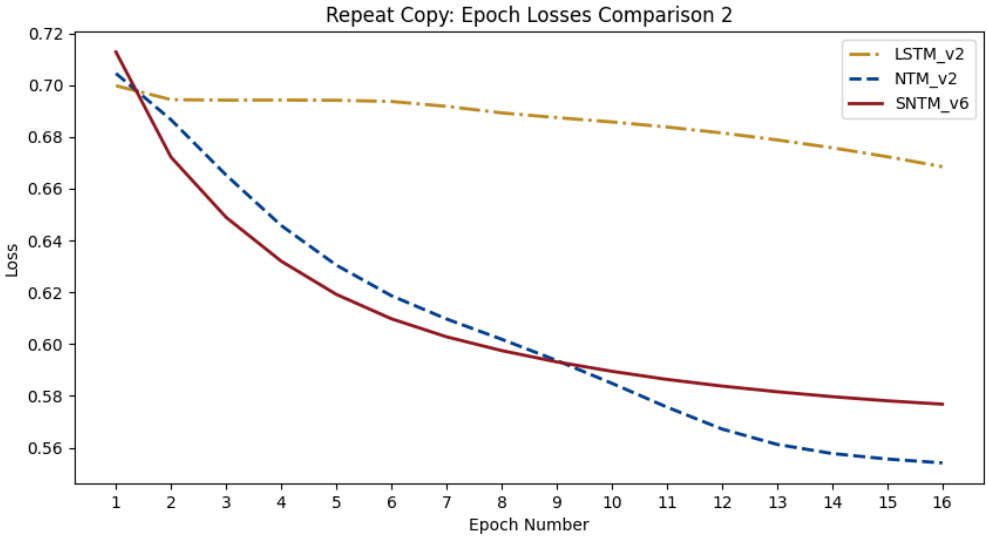

Benchmark 1 (Reverse): Accuracy Boxplot

## Appendix 6/3: Benchmark 2: Epoch Losses

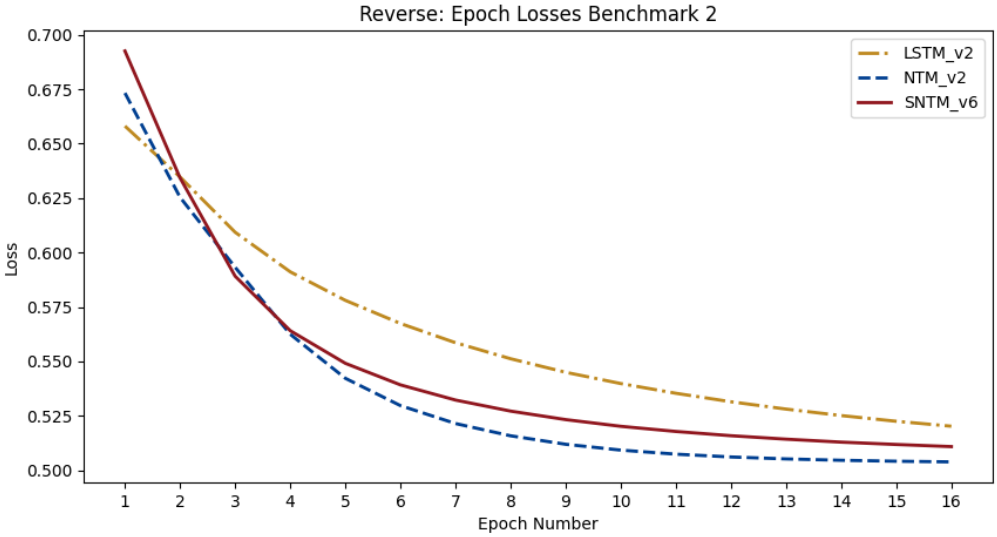
The following line charts illustrate the epoch losses of the SNTM model compared to the LSTM and NTM models. The line charts are generated in the `evaluation.ipynb` notebook, using the Python library `matplotlib`.




Benchmark 2 (Copy): Epoch Losses Line Chart



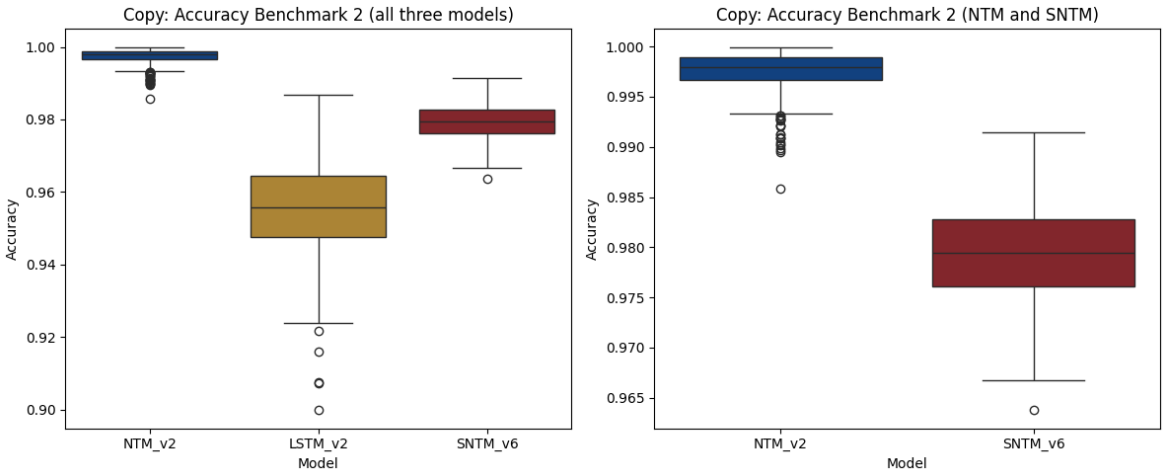

Benchmark 2 (Reverse): Epoch Losses Line Chart



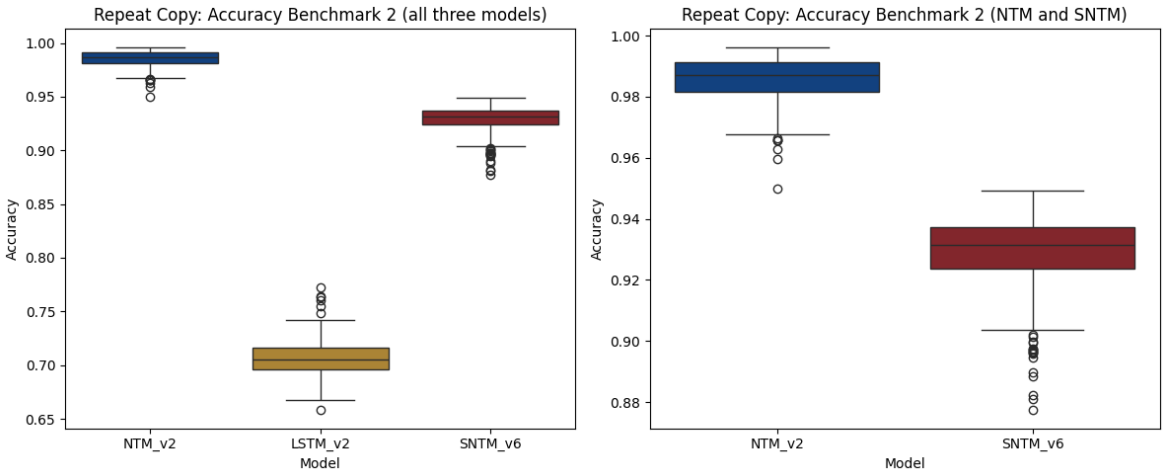

Benchmark 2 (Reverse): Epoch Losses Line Chart

## Appendix 6/4: Benchmark 2: Accuracy Boxplots

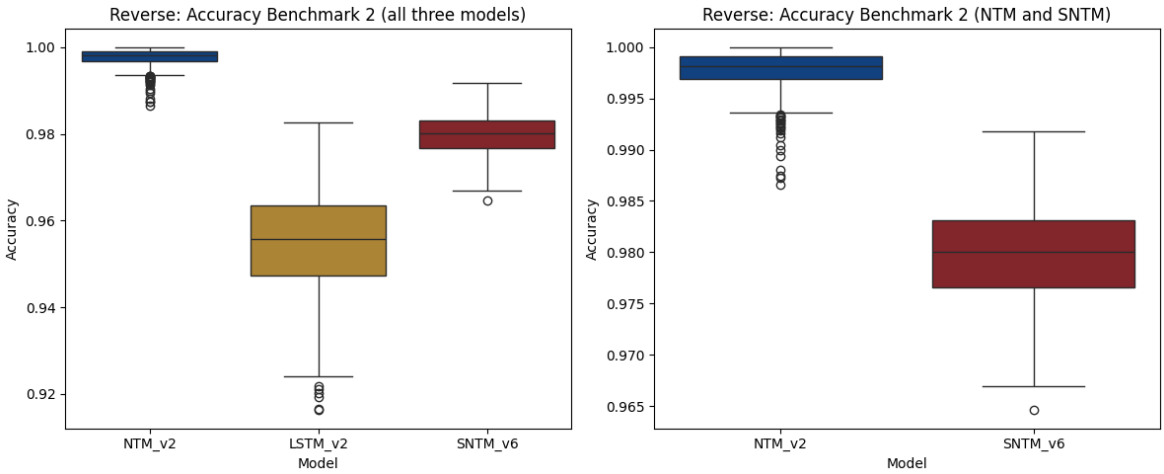
The following boxplots illustrate the accuracy of the SNTM model compared to the LSTM and NTM models. The boxplots are generated in the `evaluation.ipynb` notebook, using the Python library `seaborn`.




Benchmark 2 (Copy): Accuracy Boxplot




Benchmark 2 (Repeat Copy): Accuracy Boxplot




Benchmark 2 (Reverse): Accuracy Boxplot

## References

**Asaei-Moamam, Z.-S./Safi-Esfahani, F./Mirjalili, S./Mohammadpour, R./Nadimi-Shahraki, M.-H. (2023)**: Air Quality Particulate-Pollution Prediction Applying GAN Network and the Neural Turing Machine. In: *Applied Soft Computing* 147, p. 110723. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2023.110723. URL: https://www.sciencedirect.com/science/article/pii/S156849462300741X.

**Bienia, C./Kumar, S./Singh, J. P./Li, K. (2008)**: The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. New York, NY, USA: Association for Computing Machinery, pp. 72–81. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454128. URL: https://doi.org/10.1145/1454115.1454128.

**Bohte, S. M./Kok, J. N./La Poutré, H. (2002)**: Error-Backpropagation in Temporally Encoded Networks of Spiking Neurons. In: *Neurocomputing* 48.1, pp. 17–37. ISSN: 0925-2312. DOI: 10.1016/S0925-2312(01)00658-0. URL: https://www.sciencedirect.com/science/article/pii/S0925231201006580.

**Boloukian, B./Safi-Esfahani, F. (2020)**: Recognition of Words from Brain-Generated Signals of Speech-Impaired People: Application of Autoencoders as a Neural Turing Machine Controller in Deep Neural Networks. In: *Neural Networks* 121, pp. 186–207. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2019.07.012. URL: https://www.sciencedirect.com/science/article/pii/S089360801930200X.

**Botev, A./Lever, G./Barber, D. (2017)**: Nesterov's Accelerated Gradient and Momentum as Approximations to Regularised Update Descent. In: *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 1899–1903. DOI: 10.1109/IJCNN.2017.7966082.

**Bräunl, T. (2003)**: 'Neural Networks'. In: *Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 273–285. ISBN: 978-3-662-05099-6. DOI: 10.1007/978-3-662-05099-6_19. URL: https://doi.org/10.1007/978-3-662-05099-6_19.

**Brette, R./Rudolph, M./Carnevale, T./Hines, M./Beeman, D./Bower, J. M./Diesmann, M./Morrison, A./Goodman, P. H./Harris, F. C./Zirpe, M./Natschläger, T./Pecevski, D./Ermentrout, B./Djurfeldt, M./Lansner, A./Rochel, O./Vieville, T./Muller, E./Davison, A. P./El Boustani, S./Destexhe, A. (2007)**: Simulation of Networks of Spiking Neurons: A Review of Tools and Strategies. In: *Journal of Computational Neuroscience* 23.3, pp. 349–398. ISSN: 1573-6873. DOI: 10.1007/s10827-007-0038-6. URL: https://doi.org/10.1007/s10827-007-0038-6.

**Camigord (2017)**: Implementation GitHub: Neural-Turing-Machine. URL: https://github.com/camigord/Neural-Turing-Machine (retrieval: 03/20/2024).

**Carpedm20 (2015)**: Implementation GitHub: NTM-TensorFlow. URL: https://github.com/carpedm20/NTM-tensorflow (retrieval: 03/20/2024).

**Carrillo, S./Harkin, J./McDaid, L./Pande, S./Cawley, S./McGinley, B./Morgan, F. (2012)**: Advancing Interconnect Density for Spiking Neural Network Hardware Implementations Using Traffic-Aware Adaptive Network-on-Chip Routers. In: *Neural Networks* 33,

pp. 42–57. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2012.04.004. URL: https://www.sciencedirect.com/science/article/pii/S0893608012001104.

**Caruso, C./Quarta, F. (1998)**: Interpolation Methods Comparison. In: *Computers & Mathematics with Applications* 35.12, pp. 109–126. ISSN: 0898-1221. DOI: 10.1016/S0898-1221(98)00101-1. URL: https://www.sciencedirect.com/science/article/pii/S0898122198001011.

**Chakrabarti, K./Chopra, N. (2021)**: Generalized AdaGrad (G-AdaGrad) and Adam: A State-Space Perspective. In: *2021 60th IEEE Conference on Decision and Control (CDC)*, pp. 1496–1501. DOI: 10.1109/CDC45484.2021.9682994.

**Chandar, S./Gülçehre, Ç./Cho, K./Bengio, Y. (2016)**: Dynamic Neural Turing Machine with Soft and Hard Addressing Schemes. arXiv: 1607.00036 [cs.LG].

**Chiggum (2016)**: Implementation GitHub: Neural-Turing-Machines. URL: https://github.com/chiggum/Neural-Turing-Machines (retrieval: 03/20/2024).

**Cox, D. D./Dean, T. (2014)**: Neural Networks and Neuroscience-Inspired Computer Vision. In: *Current Biology* 24.18, R921–R929. ISSN: 0960-9822. DOI: 10.1016/j.cub.2014.08.026. URL: https://www.sciencedirect.com/science/article/pii/S0960982214010392.

**Danihelka, I./Wayne, G./Uria, B./Kalchbrenner, N./Graves, A. (2016)**: Associative Long Short-Term Memory. In: *CoRR* abs/1602.03032. arXiv: 1602.03032. URL: http://arxiv.org/abs/1602.03032.

**Dayan, P./Abbott, L. (2001)**: Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems. Cambridge: MIT Press.

**Facebook Archive (2017)**: Implementation GitHub: bAbI-tasks. URL: https://github.com/facebookarchive/bAbI-tasks.

**Falcon, A./D'Agostino, G./Lanz, O./Brajnik, G./Tasso, C./Serra, G. (2022)**: Neural Turing Machines for the Remaining Useful Life Estimation Problem. In: *Computers in Industry* 143, p. 103762. ISSN: 0166-3615. DOI: 10.1016/j.compind.2022.103762. URL: https://www.sciencedirect.com/science/article/pii/S0166361522001592.

**Faradonbe, S. M./Safi-Esfahani, F. (2020)**: A Classifier Task Based on Neural Turing Machine and Particle Swarm Algorithm. In: *Neurocomputing* 396, pp. 133–152. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2018.07.097. URL: https://www.sciencedirect.com/science/article/pii/S0925231219304278.

**Fei, Z./Wu, Z./Xiao, Y./Ma, J./He, W. (2020)**: A New Short-Arc Fitting Method with High Precision Using Adam Optimization Algorithm. In: *Optik* 212, p. 164788. ISSN: 0030-4026. DOI: 10.1016/j.ijleo.2020.164788. URL: https://www.sciencedirect.com/science/article/pii/S0030402620306240.

**Fiesler, E. (1994)**: Neural Network Classification and Formalization. In: *Computer Standards & Interfaces* 16.3, pp. 231–239. ISSN: 0920-5489. DOI: 10.1016/0920-5489(94)90014-0. URL: https://www.sciencedirect.com/science/article/pii/0920548994900140.

**FitzHugh, R. (1961)**: Impulses and Physiological States in Theoretical Models of Nerve Membrane. In: *Biophysical Journal* 1.6, pp. 445–466. ISSN: 0006-3495. DOI: 10.1016/S0006-3495(61)86902-6. URL: https://www.sciencedirect.com/science/article/pii/S0006349561869026.

**García Cabello, J. (2022)**: Mathematical Neural Networks. In: *Axioms* 11.80. ISSN: 2075-1680. DOI: 10.3390/axioms11020080. URL: https://www.mdpi.com/2075-1680/11/2/80.

**Gewaltig, M.-O./Diesmann, M. (2007)**: NEST (NEural Simulation Tool). In: *Scholarpedia* 2.4, p. 1430.

**Google DeepMind (2024)**: Implementation GitHub: DNC. URL: https://github.com/google-deepmind/dnc (retrieval: 03/14/2024).

**Gorgan Mohammadi, A./Ganjtabesh, M. (2024)**: On Computational Models of Theory of Mind and the Imitative Reinforcement Learning in Spiking Neural Networks. In: *Scientific Reports* 14.1, p. 1945. ISSN: 2045-2322. DOI: 10.1038/s41598-024-52299-7. URL: https://doi.org/10.1038/s41598-024-52299-7.

**Graves, A./Wayne, G./Danihelka, I. (2014)**: Neural Turing Machines. In: *CoRR* abs/1410.5401. arXiv: 1410.5401. URL: http://arxiv.org/abs/1410.5401.

**Greff, K./Srivastava, R. K./Koutník, J./Steunebrink, B. R./Schmidhuber, J. (2017)**: LSTM: A Search Space Odyssey. In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10, pp. 2222–2232. DOI: 10.1109/TNNLS.2016.2582924.

**Gregor, S./Hevner, A. (2013)**: Positioning and Presenting Design Science Research for Maximum Impact. In: *MIS Quarterly* 37, pp. 337–356. DOI: 10.25300/MISQ/2013/37.2.01.

**Greve, R. B./Jacobsen, E. J./Risi, S. (2016)**: Evolving Neural Turing Machines for Reward-Based Learning. In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. GECCO '16. New York, NY, USA: Association for Computing Machinery, pp. 117–124. ISBN: 978-1-4503-4206-3. DOI: 10.1145/2908812.2908930. URL: https://doi.org/10.1145/2908812.2908930.

**Gülçehre, Ç./Chandar, S./Bengio, Y. (2017)**: Memory Augmented Neural Networks with Wormhole Connections. In: *CoRR* abs/1701.08718. arXiv: 1701.08718. URL: http://arxiv.org/abs/1701.08718.

**Guresen, E./Kayakutlu, G. (2011)**: Definition of Artificial Neural Networks with Comparison to Other Networks. In: *Procedia Computer Science* 3, pp. 426–433. ISSN: 1877-0509. DOI: 10.1016/j.procs.2010.12.071. URL: https://www.sciencedirect.com/science/article/pii/S1877050910004461.

**Haji, S. H./Abdulazeez, A. M. (2021)**: COMPARISON OF OPTIMIZATION TECHNIQUES BASED ON GRADIENT DESCENT ALGORITHM: A REVIEW. In: *PalArch's Journal of Archaeology of Egypt / Egyptology* 18.4, pp. 2715–2743. URL: https://archives.palarch.nl/index.php/jae/article/view/6705.

**Han, J./Kamber, M./Pei, J. (2012)**: 'Getting to Know Your Data'. In: *Data Mining (Third Edition)*. Ed. by Jiawei Han/Micheline Kamber/Jian Pei. Third Edition. The Morgan Kaufmann Series in Data Management Systems. Boston: Morgan Kaufmann, pp. 39–82. ISBN: 978-0-12-381479-1. DOI: 10.1016/B978-0-12-381479-1.00002-2. URL: https://www.sciencedirect.com/science/article/pii/B9780123814791000022.

**HarvardNLP (2017)**: Implementation GitHub: Lie-Access-Memory. URL: https://github.com/harvardnlp/lie-access-memory.

**Héricé, C./Khalil, R./Moftah, M./Boraud, T./Guthrie, M./Garenne, A. (2016)**: Decision Making under Uncertainty in a Spiking Neural Network Model of the Basal Ganglia. In: *Journal of Integrative Neuroscience* 15.04, pp. 515–538. DOI: 10.1142/S021963521650028X. eprint: https://doi.org/10.1142/S021963521650028X. URL: https://doi.org/10.1142/S021963521650028X.

**Hevner, A. (2007)**: A Three Cycle View of Design Science Research. In: *Scandinavian Journal of Information Systems* 19.

**Hevner, A. R./Chatterjee, S. (2010)**: Design Research in Information Systems: Theory and Practice. 1st ed. NY: Springer New York.

**Hindmarsh, J. L./Rose, R. M./Huxley, Andrew Fielding (1984)**: A Model of Neuronal Bursting Using Three Coupled First Order Differential Equations. In: *Proceedings of the Royal Society of London. Series B. Biological Sciences* 221.1222, pp. 87–102. DOI: 10.1098/rspb.1984.0024. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rspb.1984.0024. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspb.1984.0024.

**Hinton, G./Nitish, S./Kevin, S. (2012)**: Neural Networks for Machine Learning. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (retrieval: 04/04/2024).

**Hochreiter, S./Schmidhuber, J. (1997)**: Long Short-Term Memory. In: *Neural Computation* 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

**Hodgkin, A. L./Huxley, A. F. (1952)**: A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve. In: *The Journal of Physiology* 117.4, pp. 500–544. DOI: 10.1113/jphysiol.1952.sp004764. eprint: https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.1952.sp004764. URL: https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1952.sp004764.

**Hopfield, J. J. (1982)**: Neural Networks and Physical Systems with Emergent Collective Computational Abilities. In: *Proceedings of the National Academy of Sciences* 79.8, pp. 2554–2558. DOI: 10.1073/pnas.79.8.2554. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.79.8.2554. URL: https://www.pnas.org/doi/abs/10.1073/pnas.79.8.2554.

**IBM Research (2021)**: Welcome to Neuro-Inspired AI! URL: https://ibm.github.io/neuroaikit/ (retrieval: 12/04/2024).

– **(2024)**: Implementation GitHub: Neuroaikit. URL: https://github.com/IBM/neuroaikit/ (retrieval: 12/04/2024).

**Ihde, N./Marten, P./Eleliemy, A./Poerwawinata, G./Silva, P./Tolovski, I./Ciorba, F. M./Rabl, T. (2022)**: A Survey of Big Data, High Performance Computing, and Machine Learning Benchmarks. In: *Performance Evaluation and Benchmarking*. Ed. by Raghunath Nambiar/Meikel Poess. Cham: Springer International Publishing, pp. 98–118. ISBN: 978-3-030-94437-7.

**Ilyasu123 (2024)**: Implementation GitHub: Reinforcement-NTM. URL: https://github.com/ilyasu123/rlntm (retrieval: 03/18/2024).

**Ixaxaar (2024)**: Implementation GitHub: PyTorch-DNC, SAM. URL: https://github.com/ixaxaar/pytorch-dnc (retrieval: 03/18/2024).

**Izhikevich, E. (2004)**: Which Model to Use for Cortical Spiking Neurons? In: *IEEE Transactions on Neural Networks* 15.5, pp. 1063–1070. DOI: 10.1109/TNN.2004.832719.

**Jakobmerrild (2017)**: Implementation GitHub: ENTM-CSharpPort. URL: https://github.com/jakobmerrild/ENTM_CSharpPort.

**Janiesch, C./Zschech, P./Heinrich, K. (2021)**: Machine Learning and Deep Learning. In: *Electronic Markets* 31.3, pp. 685–695. ISSN: 1422-8890. DOI: 10.1007/s12525-021-00475-2. URL: https://doi.org/10.1007/s12525-021-00475-2.

**Jo, T. (2023)**: Deep Learning Foundations. 1st ed. Cham: Springer International Publishing.

**Johnson, N. S./Vulimiri, P. S./To, A. C./Zhang, X./Brice, C. A./Kappes, B. B./Stebner, A. P. (2020)**: Machine Learning for Materials Developments in Metals Additive Manufacturing. arXiv: 2005.05235 [physics.app-ph].

**Keras (2024)**: Keras API Documentation: RMSprop. URL: https://keras.io/api/optimizers/rmsprop/ (retrieval: 04/04/2024).

**Keren, G./Sabato, S./Schuller, B. (2018)**: Fast Single-Class Classification and the Principle of Logit Separation. In: *2018 IEEE International Conference on Data Mining (ICDM)*, pp. 227–236. DOI: 10.1109/ICDM.2018.00038.

**Kester, L./Kirschner, P. A. (2012)**: 'Cognitive Tasks and Learning'. In: *Encyclopedia of the Sciences of Learning*. Ed. by Norbert M. Seel. Boston, MA: Springer US, pp. 619–622. ISBN: 978-1-4419-1428-6. DOI: 10.1007/978-1-4419-1428-6_225. URL: https://doi.org/10.1007/978-1-4419-1428-6_225.

**Kheradpisheh, S. R./Ganjtabesh, M./Thorpe, S. J./Masquelier, T. (2018)**: STDP-based Spiking Deep Convolutional Neural Networks for Object Recognition. In: *Neural Networks* 99, pp. 56–67. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2017.12.005. URL: https://www.sciencedirect.com/science/article/pii/S0893608017302903.

**Kingma, D. P./Ba, J. (2015)**: Adam: A Method for Stochastic Optimization. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio/Yann LeCun. URL: http://arxiv.org/abs/1412.6980.

**Kirkland, P./Di Caterina, G./Soraghan, J./Matich, G. (2020)**: Perception Understanding Action: Adding Understanding to the Perception Action Cycle with Spiking Segmentation. In: *Frontiers in Neurorobotics* 14. DOI: 10.3389/fnbot.2020.568319.

**Kussul, E./Baidyk, T./Wunsch, D. C. (2010)**: 'Classical Neural Networks'. In: *Neural Networks and Micromechanics*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 7–25. ISBN: 978-3-642-02535-8. DOI: 10.1007/978-3-642-02535-8_2. URL: https://doi.org/10.1007/978-3-642-02535-8_2.

**Lapicque, L. M. (1907)**: Recherches Quantitatives Sur l'excitation Electrique Des Nerfs. In: *Journal of physiology, Paris* 9, pp. 620–635.

**Lee, J. H./Delbruck, T./Pfeiffer, M. (2016)**: Training Deep Spiking Neural Networks Using Backpropagation. In: *Frontiers in Neuroscience* 10. DOI: 10.3389/fnins.2016.00508.

**Liu, L./Lu, S./Zhong, R./Wu, B./Yao, Y./Zhang, Q./Shi, W. (2021)**: Computing Systems for Autonomous Driving: State of the Art and Challenges. In: *IEEE Internet of Things Journal* 8.8, pp. 6469–6486. DOI: 10.1109/JIOT.2020.3043716.

**Loudinthecloud (2018)**: Implementation GitHub: Pytorch-NTM. URL: https://github.com/loudinthecloud/pytorch-ntm (retrieval: 02/21/2024).

**Maass, W. (1997)**: Networks of Spiking Neurons: The Third Generation of Neural Network Models. In: *Neural Networks* 10.9, pp. 1659–1671. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(97)00011-7. URL: https://www.sciencedirect.com/science/article/pii/S0893608097000117.

**Maass, W./Markram, H. (2004)**: On the Computational Power of Circuits of Spiking Neurons. In: *Journal of Computer and System Sciences* 69.4, pp. 593–616. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2004.04.001. URL: https://www.sciencedirect.com/science/article/pii/S0022000004000406.

**Malekmohamadi Faradonbe, S./Safi-Esfahani, F./Karimian-kelishadrokhi, M. (2020)**: A Review on Neural Turing Machine (NTM). In: *SN Computer Science* 1.6, p. 333. ISSN: 2661-8907. DOI: 10.1007/s42979-020-00341-6. URL: https://doi.org/10.1007/s42979-020-00341-6.

**Mark Collier/Beel, J. (2018)**: Implementing Neural Turing Machines. In: *Artificial Neural Networks and Machine Learning – ICANN 2018*. Cham: Springer International Publishing, pp. 94–104. ISBN: 978-3-030-01424-7.

**MarkPKCollier (2018)**: Implementation GitHub: NeuralTuringMachine. URL: https://github.com/MarkPKCollier/NeuralTuringMachine (retrieval: 03/20/2024).

**McCulloch, W. S./Pitts, W. (1943)**: A Logical Calculus of the Ideas Immanent in Nervous Activity. In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.

**Mehta, V./Kumar, A./Nahar, K./Poonam/Sharma, D./Tiwari, R. (2024)**: Binary Image Classification Using Machine Learning and Deep Quantum Neural Networks. In: *2024 14th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pp. 811–816. DOI: 10.1109/Confluence60223.2024.10463226.

**Merolla, P. A./Arthur, J. V./Alvarez-Icaza, R./Cassidy, A. S./Sawada, J./Akopyan, F./Jackson, B. L./Imam, N./Guo, C./Nakamura, Y./Brezzo, B./Vo, I./Esser, S. K./Appuswamy, R./Taba, B./Amir, A./Flickner, M. D./Risk, W. P./Manohar, R./Dharmendra S. Modha (2014)**: A Million Spiking-Neuron Integrated Circuit with a Scalable Communication Network and Interface. In: *Science (New York, N.Y.)* 345.6197, pp. 668–673. DOI: 10.1126/science.1254642. eprint: https://www.science.org/doi/pdf/10.1126/science.1254642. URL: https://www.science.org/doi/abs/10.1126/science.1254642.

**Merrild, J./Rasmussen, M. A./Risi, S. (2018)**: HyperNTM: Evolving Scalable Neural Turing Machines through HyperNEAT. In: *Applications of Evolutionary Computation*. Ed. by Kevin Sim/Paul Kaufmann. Cham: Springer International Publishing, pp. 750–766. ISBN: 978-3-319-77538-8.

**Mikhaeil, J./Monfared, Z./Durstewitz, D. (2022)**: On the Difficulty of Learning Chaotic Dynamics with RNNs. In: *Advances in Neural Information Processing Systems*. Vol. 35. Curran Associates, Inc., pp. 11297–11312. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/495e55f361708bedbab5d81f92048dcd-Paper-Conference.pdf.

**Mukkamala, M. C./Hein, M. (2017)**: Variants of RMSProp and Adagrad with Logarithmic Regret Bounds. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup/Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 2545–2553. URL: https://proceedings.mlr.press/v70/mukkamala17a.html.

**Muntean, M./Militaru, F. D. (2022)**: Design Science Research Framework for Performance Analysis Using Machine Learning Techniques. In: *Electronicsweek* 11.2504. ISSN: 2079-9292. DOI: 10.3390/electronics11162504. URL: https://www.mdpi.com/2079-9292/11/16/2504.

**Neil, D./Liu, S.-C. (2016)**: Effective Sensor Fusion with Event-Based Sensors and Deep Network Architectures. In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 2282–2285. DOI: 10.1109/ISCAS.2016.7539039.

**Nielsen, M. A. (2015)**: Neural Networks and Deep Learning. CA, USA: Determination Press.

**Nunamaker, J./Chen, M. (1990)**: Systems Development in Information Systems Research. In: *Twenty-Third Annual Hawaii International Conference on System Sciences*. Vol. 3, 631–640 vol.3. DOI: 10.1109/HICSS.1990.205401.

**O'Connor, P./Neil, D./Liu, S.-C./Delbruck, T./Pfeiffer, M. (2013a)**: Real-Time Classification and Sensor Fusion with a Spiking Deep Belief Network. In: *Frontiers in Neuroscience* 7. DOI: 10.3389/fnins.2013.00178.

– **(2013b)**: Real-Time Classification and Sensor Fusion with a Spiking Deep Belief Network. In: *Frontiers in Neuroscience* 7. DOI: 10.3389/fnins.2013.00178.

**Onan, A. (2022)**: Bidirectional Convolutional Recurrent Neural Network Architecture with Group-Wise Enhancement Mechanism for Text Sentiment Classification. In: *Journal of King Saud University - Computer and Information Sciences* 34.5, pp. 2098–2117. ISSN: 1319-1578. DOI: 10.1016/j.jksuci.2022.02.025. URL: https://www.sciencedirect.com/science/article/pii/S1319157822000696.

**Ostheimer, J./Chowdhury, S./Iqbal, S. (2021)**: An Alliance of Humans and Machines for Machine Learning: Hybrid Intelligent Systems and Their Design Principles. In: *Technology in Society* 66, p. 101647. ISSN: 0160-791X. DOI: 10.1016/j.techsoc.2021.101647. URL: https://www.sciencedirect.com/science/article/pii/S0160791X21001226.

**Pascanu, R./Mikolov, T./Bengio, Y. (2013)**: On the Difficulty of Training Recurrent Neural Networks. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta/David McAllester. Vol. 28. Proceedings of Machine Learning Research. Atlanta, Georgia, USA: PMLR, pp. 1310–1318. URL: https://proceedings.mlr.press/v28/pascanu13.html.

**Rabinovich, M. I./Varona, P./Selverston, A. I./Abarbanel, H. D. I. (2006)**: Dynamical Principles in Neuroscience. In: *Reviews of Modern Physics* 78.4, pp. 1213–1265. DOI: 10.1103/RevModPhys.78.1213. URL: https://link.aps.org/doi/10.1103/RevModPhys.78.1213.

**Rae, J./Hunt, J. J./Danihelka, I./Harley, T./Senior, A. W./Wayne, G./Graves, A./Lillicrap, T. (2016)**: Scaling Memory-Augmented Neural Networks with Sparse Reads and Writes. In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee/M. Sugiyama/U. Luxburg/I. Guyon/R. Garnett. Vol. 29. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper_files/paper/2016/file/3fab5890d8113d0b5a4178201dc842ad-Paper.pdf.

**Rasmusgreve (2015)**: Implementation GitHub: neuralturingmachines. URL: https://github.com/rasmusgreve/neuralturingmachines (retrieval: 03/20/2024).

**Rosenblatt, F. (1962)**: Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. Cornell Aeronautical Laboratory. Report No. VG-1196-G-8. Spartan Books. URL: https://books.google.ca/books?id=7FhRAAAAMAAJ.

**Roy, K./Jaiswal, A./Panda, P. (2019)**: Towards Spike-Based Machine Intelligence with Neuromorphic Computing. In: *Nature* 575.7784, pp. 607–617. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1677-2. URL: https://doi.org/10.1038/s41586-019-1677-2.

**Russakovsky, O./Deng, J./Su, H./Krause, J./Satheesh, S./Ma, S./Huang, Z./Karpathy, A./Khosla, A./Bernstein, M./Berg, A. C./Fei-Fei, L. (2015)**: ImageNet Large Scale Visual Recognition Challenge. In: *International Journal of Computer Vision* 115.3, pp. 211–252. ISSN: 1573-1405. DOI: 10.1007/s11263-015-0816-y. URL: https://doi.org/10.1007/s11263-015-0816-y.

**Salehinejad, H./Baarbe, J./Sankar, S./Barfett, J./Colak, E./Valaee, S. (2018)**: Recent Advances in Recurrent Neural Networks. In: *CoRR* abs/1801.01078. arXiv: 1801.01078. URL: http://arxiv.org/abs/1801.01078.

**Schaul, T./Antonoglou, I./Silver, D. (2014)**: Unit Tests for Stochastic Optimization. In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. by Yoshua Bengio/Yann LeCun. URL: http://arxiv.org/abs/1312.6055.

**Schmidhuber, J. (2015)**: Deep Learning in Neural Networks: An Overview. In: *Neural Networks* 61, pp. 85–117. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2014.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0893608014002135.

**Snipsco (2015)**: Implementation GitHub: NTM-Lasagne. URL: https://github.com/snipsco/ntm-lasagne (retrieval: 03/20/2024).

**Snowkylin (2019)**: Implementation GitHub: NTM. URL: https://github.com/snowkylin/ntm (retrieval: 02/21/2024).

**Stimberg, M./Brette, R./Goodman, D. F. (2019)**: Brian 2, an Intuitive and Efficient Neural Simulator. In: *eLife* 8. Ed. by Frances K Skinner/Ronald L Calabrese/Frances K Skinner/Fleur Zeldenrust/Richard C Gerkin, e47314. ISSN: 2050-084X. DOI: 10.7554/eLife.47314. URL: https://doi.org/10.7554/eLife.47314.

**Sukhbaatar, S./szlam, a./Weston, J./Fergus, R. (2015)**: End-to-End Memory Networks. In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes/N. Lawrence/D. Lee/M. Sugiyama/R. Garnett. Vol. 28. Curran Associates, Inc. URL: https://proceedings.

neurips.cc/paper_files/paper/2015/file/8fb21ee7a2207526da55a679f0332de2-Paper.pdf.

**Sun, S./Cao, Z./Zhu, H./Zhao, J. (2020)**: A Survey of Optimization Methods from a Machine Learning Perspective. In: *IEEE Transactions on Cybernetics* 50.8, pp. 3668–3681. DOI: 10.1109/TCYB.2019.2950779.

**Sutskever, I./Vinyals, O./Le, Q. V. (2014)**: Sequence to Sequence Learning with Neural Networks. In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Cambridge, MA, USA: MIT Press, pp. 3104–3112.

**Taherkhani, A./Belatreche, A./Li, Y./Cosma, G./Maguire, L. P./McGinnity, T. (2020)**: A Review of Learning in Biologically Plausible Spiking Neural Networks. In: *Neural Networks* 122, pp. 253–272. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2019.09.036. URL: https://www.sciencedirect.com/science/article/pii/S0893608019303181.

**Tavanaei, A./Ghodrati, M./Kheradpisheh, S. R./Masquelier, T./Maida, A. (2019)**: Deep Learning in Spiking Neural Networks. In: *Neural Networks* 111, pp. 47–63. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2018.12.002. URL: https://www.sciencedirect.com/science/article/pii/S0893608018303332.

**Te'eni, D./Yahav, I./Zagalsky, A./Schwartz, D./Silverman, G./Cohen, D./Mann, Y./Lewinsky, D. (2023)**: Reciprocal Human-Machine Learning: A Theory and an Instantiation for the Case of Message Classification. In: *Management Science* 0.0, pp. 1–26. URL: https://doi.org/10.1287/mnsc.2022.03518.

**TensorFlow (2024)**: Sigmoid Cross Entropy with Logits. URL: https://www.tensorflow.org/api_docs/python/tf/nn/sigmoid_cross_entropy_with_logits (retrieval: 04/04/2024).

**The Apache Software Foundation (2024)**: Apache License 2.0. URL: https://www.apache.org/licenses/LICENSE-2.0.txt (retrieval: 04/15/2024).

**Thiyagalingam, J./Shankar, M./Fox, G./Hey, T. (2022)**: Scientific Machine Learning Benchmarks. In: *Nature Reviews Physics* 4.6, pp. 413–420. ISSN: 2522-5820. DOI: 10.1038/s42254-022-00441-7. URL: https://doi.org/10.1038/s42254-022-00441-7.

**Tkačík, J./Kordík, P. (2016)**: Neural Turing Machine for Sequential Learning of Human Mobility Patterns. In: *2016 International Joint Conference on Neural Networks (IJCNN)*, pp. 2790–2797. DOI: 10.1109/IJCNN.2016.7727551.

**Van Houdt, G./Mosquera, C./Nápoles, G. (2020)**: A Review on the Long Short-Term Memory Model. In: *Artificial Intelligence Review* 53.8, pp. 5929–5955. ISSN: 1573-7462. DOI: 10.1007/s10462-020-09838-1. URL: https://doi.org/10.1007/s10462-020-09838-1.

**Vaswani, A./Shazeer, N./Parmar, N./Uszkoreit, J./Jones, L./Gomez, A. N./Kaiser, Ł./Polosukhin, I. (2017)**: Attention Is All You Need. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., pp. 6000–6010. ISBN: 978-1-5108-6096-4.

**Venugopalan, S./Rohrbach, M./Donahue, J./Mooney, R./Darrell, T./Saenko, K. (2015)**: Sequence to Sequence – Video to Text. In: *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 4534–4542. DOI: 10.1109/ICCV.2015.515.

**Wang, X./Lin, X./Dang, X. (2020)**: Supervised Learning in Spiking Neural Networks: A Review of Algorithms and Evaluations. In: *Neural Networks* 125, pp. 258–280. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2020.02.011. URL: https://www.sciencedirect.com/science/article/pii/S0893608020300563.

**Wayne, G./Graves, A./Reynolds, M./Harley, T./Danihelka, I./Grabska-Barwińska, A./Colmenarejo, S. G./Grefenstette, E./Ramalho, T./Agapiou, J./Badia, A. P./Hermann, K. M./Zwols, Y./Ostrovski, G./Cain, A./King, H./Summerfield, C./Blunsom, P./Kavukcuoglu, K./Hassabis, D. (2016)**: Hybrid Computing Using a Neural Network with Dynamic External Memory. In: *Nature* 538.7626, pp. 471–476. ISSN: 1476-4687. DOI: 10.1038/nature20101. URL: https://doi.org/10.1038/nature20101.

**Webster, J./Watson, R. T. (2002)**: Analyzing the Past to Prepare for the Future: Writing a Literature Review. In: *MIS Quarterly* 26.2, pp. xiii–xxiii. ISSN: 02767783. JSTOR: 4132319. URL: http://www.jstor.org/stable/4132319 (retrieval: 03/04/2024).

**Welsch, G./Kowalczyk, P. (2023)**: Designing Explainable Predictive Machine Learning Artifacts: Methodology and Practical Demonstration. arXiv: 2306.11771 [cs.SE].

**Werbos, P. (1990)**: Backpropagation through Time: What It Does and How to Do It. In: *Proceedings of the IEEE* 78.10, pp. 1550–1560. DOI: 10.1109/5.58337.

**Weston, J./Bordes, A./Chopra, S./Rush, A. M./van Merriënboer, B./Joulin, A./Mikolov, T. (2015)**: Towards Ai-Complete Question Answering: A Set of Prerequisite Toy Tasks. In: *arXiv preprint arXiv:1502.05698*. arXiv: 1502.05698.

**Woźniak, S./Pantazi, A./Bohnstingl, T./Eleftheriou, E. (2020)**: Deep Learning Incorporating Biologically Inspired Neural Dynamics and In-Memory Computing. In: *Nature Machine Intelligence* 2.6, pp. 325–336. ISSN: 2522-5839. DOI: 10.1038/s42256-020-0187-0. URL: https://doi.org/10.1038/s42256-020-0187-0.

**Xue, F./Chen, X./Li, X. (2017)**: Real-Time Classification through a Spiking Deep Belief Network with Intrinsic Plasticity. In: *Advances in Neural Networks - ISNN 2017*. Ed. by Fengyu Cong/Andrew Leung/Qinglai Wei. Cham: Springer International Publishing, pp. 188–196. ISBN: 978-3-319-59072-1.

**Yamazaki, K./Vo-Ho, V.-K./Bulsara, D./Le, N. (2022)**: Spiking Neural Networks and Their Applications: A Review. In: *Brain Sciences* 12.863. ISSN: 2076-3425. DOI: 10.3390/brainsci12070863. URL: https://www.mdpi.com/2076-3425/12/7/863.

**Yang, G./Rush, A. (2017)**: Lie-Access Neural Turing Machines. In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=Byiy-Pqlx.

**Yeoedward (2016)**: Implementation GitHub: Neural-Turing-Machine. URL: https://github.com/yeoedward/Neural-Turing-Machine (retrieval: 03/20/2024).

**Yi, D./Ahn, J./Ji, S. (2020)**: An Effective Optimization Method for Machine Learning Based on ADAM. In: *Applied Sciences* 10.1073. ISSN: 2076-3417. DOI: 10.3390/app10031073. URL: https://www.mdpi.com/2076-3417/10/3/1073.

**Zaremba, W./Mikolov, T./Joulin, A./Fergus, R. (2016)**: Learning Simple Algorithms from Examples. In: *Proceedings of the 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan/Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning

Research. New York, New York, USA: PMLR, pp. 421–429. URL: https://proceedings.mlr.press/v48/zaremba16.html.

**Zaremba, W./Sutskever, I. (2015)**: Reinforcement Learning Neural Turing Machines. In: *CoRR* abs/1505.00521. arXiv: 1505.00521. URL: http://arxiv.org/abs/1505.00521.

**Zhang, W./Yu, Y./Zhou, B. (2015)**: Structured Memory for Neural Turing Machines. In: *CoRR* abs/1510.03931. arXiv: 1510.03931. URL: http://arxiv.org/abs/1510.03931.

**Zhang, Y./Qu, P./Zheng, W. (2021)**: Towards "General Purpose" Brain-Inspired Computing System. In: *Tsinghua Science and Technology* 26.5, pp. 664–673. DOI: 10.26599/TST.2021.9010010. URL: https://www.sciopen.com/article/10.26599/TST.2021.9010010.

**Zheng, H./Zheng, Z./Hu, R./Xiao, B./Wu, Y./Yu, F./Liu, X./Li, G./Deng, L. (2024)**: Temporal Dendritic Heterogeneity Incorporated with Spiking Neural Networks for Learning Multi-Timescale Dynamics. In: *Nature Communications* 15.1, p. 277. ISSN: 2041-1723. DOI: 10.1038/s41467-023-44614-z. URL: https://doi.org/10.1038/s41467-023-44614-z.

**Zheng, Z./Wu, X./Weng, J. (2019)**: Emergent Neural Turing Machine and Its Visual Navigation. In: *Neural Networks* 110, pp. 116–130. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2018.11.004. URL: https://www.sciencedirect.com/science/article/pii/S0893608018303198.

**Zou, F./Shen, L./Jie, Z./Zhang, W./Liu, W. (2019)**: A Sufficient Condition for Convergences of Adam and RMSProp. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11119–11127. DOI: 10.1109/CVPR.2019.01138.

**Anlage „Erklärung zur Verwendung generativer KI-Systeme“**

Bei der Erstellung der eingereichten Arbeit habe ich die nachfolgend aufgeführten auf künstlicher Intelligenz (KI) basierten Systeme benutzt:

1. ChatGPT
2. Consensus (https://consensus.app/search/)
3. Connected Papers (https://www.connectedpapers.com/)

Ich erkläre, dass ich

- mich aktiv über die Leistungsfähigkeit und Beschränkungen der oben genannten KI-Systeme informiert habe,[1]
- die aus den oben angegebenen KI-Systemen direkt oder sinngemäß übernommenen Passagen gekennzeichnet habe,[2]
- überprüft habe, dass die mithilfe der oben genannten KI-Systeme generierten und von mir übernommenen Inhalte faktisch richtig sind,
- mir bewusst bin, dass ich als Autorin bzw. Autor dieser Arbeit die Verantwortung für die in ihr gemachten Angaben und Aussagen trage.

Die oben genannten KI-Systeme habe ich wie im Folgenden dargestellt eingesetzt:

| **Arbeitsschritt in der wissenschaftlichen Arbeit[3]** | **Eingesetzte(s) KI-System(e)** | **Beschreibung der Verwendungsweise** |
|---|---|---|
| Korrektur der Arbeit | ChatGPT | Einzelne Kapitel (sofern nicht kritisch aufgrund IBM Inhalte) ChatGPT gegeben zum Korrigieren. Erfolg: eher mäßig, nach der Korrektur wurden dennoch einige Fehler gefunden. |
| Paper finden, für sehr spezifische Fragen | Consensus | Frage gestellt und Paper begutachtet |
| Paper finden, auf Basis bereits vorhandener | Connected Papers | Vorwärts und rückwärts Recherche mit dem Tool |

(Die Tabelle ist im Bedarfsfall zu erweitern und auf den Folgeseiten fortzusetzen)

Stuttgart, 06.05.24, JMüll

Ort, Datum, Unterschrift

[1] U.a. gilt es hierbei zu beachten, dass an KI weitergegebene Inhalte ggf. als Trainingsdaten genutzt und wiederverwendet werden. Dies ist insb. für betriebliche Aspekte als kritisch einzustufen.

[2] In der Fußnote Ihrer Arbeit geben Sie die KI als Quelle an, z.B.: Erzeugt durch Microsoft Copilot am dd.mm.yyyy. Oder: Entnommen aus einem Dialog mit Perplexity vom dd.mm.yyyy. Oder: Absatz 2.3 wurde durch ChatGPT sprachlich geglättet.

[3] Beispiele hierfür sind u.a. die folgenden Arbeitsschritte: Generierung von Ideen, Konzeption der Arbeit, Literatursuche, Literaturanalyse, Literaturverwaltung, Auswahl von Methoden, Datensammlung, Datenanalyse, Generierung von Programmcodes

# Declaration

I hereby insure that I have personally authored my Bachelor thesis with the topic: *Design and Implementation of a Neural Turing Machine with a Spiking Neural Network Controller* and have used no sources and aids other than those indicated. I also insure that the submitted electronic version corresponds to the printed version.

(place, date)

(signature)