Maximilian Baronig, BSc

Extending Recurrent Neural Networks with Hebbian Memory

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme:
Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Robert Legenstein

Institute of Theoretical Computer Science
Graz University of Technology, Austria

Graz, March 2021

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Abstract

Memory in the brain is crucial. Not only does it enable humans to remember specific events, it also enables complex cognitive abilities of the brain. Inspired by the utilization of memory mechanisms like synaptic plasticity in the brain, artificial neural networks are sought to enhance their cognitive capabilities by incorporating memory into their computational processes. To find out how neural networks can benefit if they are extended with some sort of working memory, challenging tasks are developed to prove the superiority of this type of neural network models. One example of such a task is an artificial question answering task, where the model needs to answer questions about previously shown facts, requiring the intermediate storage and retrieval of previously provided information. In recent years, many of those models were developed, with numerous different architectures and memory types. In this work, some of the existing neural network models augmented with some type of memory are examined and two novel models, HebbLSTM and H-HebbLSTM are proposed. These models consist of a recurrent neural network architecture similar to Long short-term memory (LSTM) networks and contain a biologically inspired memory mechanism often referred to as Hebbian learning which can be trained end-to-end using gradient descent. H-HebbLSTM is shown to achieve state-of-the-art results on artificial question answering tasks.

Zusammenfassung

Ohne jegliche Form von Gedächtnis wären komplexe kognitive Vorgänge im menschlichen Gehirn nicht möglich. Das Gedächtnis ist nicht nur notwendig, um sich an gewisse Ereignisse erinnern zu können, sondern auch, um unvollständige Information kurzzeitig zwischenzuspeichern und sie im Anschluss für weitere Berechnungsvorgänge weiterzunutzen. Inspiriert durch solche Vorgänge, welche unter anderem stark mit der Plastizität der Synapsen zusammenhängen, werden künstliche neuronale Netze mit vereinfachten Formen eines Gedächtnisses erweitert. Um zu testen, ob die Modelle aus jenen Gedächtniszellen einen Nutzen ziehen können, wurden komplexe Aufgaben, die insbesondere gedächtnisrelevante Vorgänge zur Lösung voraussetzen, entwickelt. Ein Beispiel solcher Aufgaben sind künstliche Frage-Antwort Tests. Bei solchen Tests werden dem neuronalen Netzwerk sequenziell Fakten präsentiert, die es zwischenspeichern muss, um später durch Kombination dieser eine Antwort zu ermitteln. Wie die Kombination solcher Fakten auszusehen hat, beispielsweise eine Verkettung mehrerer Informationen, hängt von der jeweiligen Instanz des Tests ab. In den vergangenen Jahren wurden zahlreiche neuronale Netzwerk-Modelle vorgestellt, die bereits das Konzept eines Gedächtnisses, im Sinne einer Erweiterung bestehender Architekturen, beinhalten. In dieser Arbeit werden einige ausgewählte dieser Architekturen im Überblick beschrieben, sowie zwei neuartige Netzwerk-Architekturen, HebbLSTM und H-HebbLSTM, vorgestellt. Diese beiden Netzwerke ähneln der ”Long short-term memory”-Architektur und beinhalten eine biologisch inspirierte Form eines Gedächtnisses, oft als ”Hebb’sches” Gedächtnis bezeichnet. Des Weiteren können diese Netzwerke end-to-end durch einen auf Gradienten basierenden Algorithmus trainiert werden. Es wird gezeigt, dass das Modell H-HebbLSTM state-of-the-art Ergebnisse auf künstlichen Frage-Antwort Tests erreichen kann.

Contents

Abstract iii

1 Introduction 1
- 1.1 Summary . . . 2
- 1.2 Related Work . . . 3

2 Recurrent Neural Networks 5
- 2.1 Architectures . . . 10
  - 2.1.1 Elman Networks . . . 10
  - 2.1.2 Long Short-Term Memory . . . 12
- 2.2 Training . . . 14
  - 2.2.1 Learning Weights with Gradient Descent . . . 15
  - 2.2.2 Training Deep Networks by Backpropagating Errors . . . 19
  - 2.2.3 Backpropagation Through Time . . . 20
  - 2.2.4 Unfolding of RNNs . . . 21
  - 2.2.5 The Vanishing or Exploding Gradient Problem . . . 21
  - 2.2.6 Regularization and Validation . . . 22
  - 2.2.7 Training Improvements . . . 24

3 Memory Augmented Neural Networks 25
- 3.1 Neural Turing Machines . . . 25
- 3.2 Dynamic Neural Turing Machine . . . 26
- 3.3 Memory Networks . . . 28
- 3.4 End-to-End Memory Networks . . . 29
- 3.5 Metalearned Neural Memory . . . 30
- 3.6 Fast Weight Memory . . . 31
- 3.7 H-Mem . . . 32

4 Hebbian Plasticity **34**
4.1 The Hebbian learning rule 34
4.2 Hebbian Associative Memory 37

5 Hebbian Long Short-Term Memory Networks **39**

6 Hebbian Long Short-Term Memory Networks with Hidden Value **44**

7 Results **48**
7.1 Dictionary Learning 48
7.2 Question Answering 50
7.3 Memory Analysis for bAbI Tasks 54

8 Discussion **58**
8.1 Conclusion 59
8.2 Future Work 59

Bibliography **61**

List of Figures

2.1 Difference between feed-forward and recurrent neural networks. 8
2.2 Different settings of RNNs for different types of sequential tasks 9
2.3 Basic architecture of an Elman network 12
2.4 Architecture of LSTM 14
2.5 Perceptron consisting of one single neuron. 17
2.6 Unfolding of a RNN 22

3.1 Architecture of the Neural Turing Machine 26
3.2 Architecture of the H-Mem model 33

4.1 The principle of locality for Hebbian weight updates. 35

5.1 Schema of the HebbLSTM architecture 40

6.1 Schema of the H-HebbLSTM architecture 46

7.1 Learning curves of task 3 and 16 of the bAbI tasks 54
7.2 Similarity matrices for key-vectors $\boldsymbol{k}(t)$ and value-vectors $\boldsymbol{v}(t)$ for a sample of task 1 of the bAbI tasks. 55
7.3 Similarity matrices for key-vectors $\boldsymbol{k}(t)$ and value-vectors $v(t)$ for a sample of task 16 of the bAbI tasks. 57

1 Introduction

To build artificial models capable of solving problems, which otherwise can only be solved by humans or other intelligent beings, a common practice is to implement mechanisms and patterns which are inspired by biological brains. This way, the whole research area of connectionism was born [1]. Recent success on complex cognitive tasks like for example the game of Go [2] justify the assumption, that artificial connectionist models are capable of solving a variety of problems requiring some kind of intelligent reasoning [3]. The development of novel demanding tasks [4] encourages the engineering of more and more sophisticated models, trying to perform advanced cognitive processes. Creating a potent model able to solve challenging tasks by using state-of-the-art biologically inspired concepts was the main driver for this work.

A recently proposed data set of toy tasks for artificial question answering [4] aims to probe an artificial model's abilities to perform cognitive processes like reasoning, induction and chaining of associated patterns. This data set introduces tasks, easily solvable for humans but very challenging for conventional network architectures [4]. Recent proposals [5], [6] show that specific biologically inspired memory mechanisms allow models to perform a process referred to as *associative inference* [7], where tasks like this can be solved by deriving information from previously stored association patterns. In this terms, a specific form of memory, the neural associative memory (NAM) [8] has gained special attention [9], [10]. The reason for that is that this kind of memory is naturally capable of storing and retrieving pattern associations [11]. These mechanisms are based on work proposed by Donald Hebb in 1949 [12], who established a theory of how synapses can change their efficacy in order to persist patterns of stimuli, referred to as Hebbian synaptic plasticity.

The model proposed in this work aims to utilize a memory based on this theory by using state-of-the-art network architectures and algorithms. Its capability of performing *associative inference* is evaluated by its performance on an artificial dictionary learning task inspired by [6] and the bAbI [4] artificial question-answering data set.

This thesis consists of three parts, the theoretical background (Chapters 2 to 4), the description of the proposed model including the experiment results (Chapter 5 and 7) and the discussion (Chapter 8).

1.1 Summary

In this work, the recurrent neural network models HebbLSTM and an improved version, H-HebbLSTM are proposed. The models combine multiple established concepts from connectionist models into a potent new model architecture, yielding state-of-the-art results on a challenging question-answering task [4].

The model is implemented as recurrent neural network combining features from long short-term memory (LSTM) [13] and a key-value store implemented as a Hebbian associative memory [14]. The LSTM architecture operates as controller and learns to extract keys and values from the data to be written to the memory, as well as to extract read-out keys from data to retrieve previously stored information. In this work, two different architectures are proposed:

- The first architecture, HebbLSTM, passes the key and value of the previous time-step to the next time-step.
- The second architecture, H-HebbLSTM, passes a computed hidden value to the next time-step instead of the key and value. The model can therefore learn to pass arbitrary relevant information to the next time-step by maintaining a hidden state. The second architecture is considered more general, since the model can decide which information is contained in the hidden value passed to the next time-step.

In the first experiment, the toy dictionary translation task, the one-shot learning and memorization capabilities are tested. In this task, each sample

consists of two parts: the facts and the query. The facts are translation rules and are shown once to the model. In order to apply those rules on the query, the model must store the rules intermediately. Then, during the processing of the query, the model must retrieve the previously stored rules accurately and apply them on-the-fly. This task is similar to the toy translation problem proposed by Munkhdalai et al. [6].

The second experiment is the artificial question-answering bAbI data set proposed by Weston et al. [4]. It consists of 20 different tasks which are inspired by abilities associated with human cognition like associative inference and induction. It is shown that the second variant of the HebbLSTM model (H-HebbLSTM) is capable of solving all of the 20 bAbI tasks on the 10k data set.

1.2 Related Work

Multiple neural network models combining a neural network as controller and some sort of additional memory mechanism have been proposed. One important work and basis for further architectures was the Neural Turing Machine (NTM) [15], a combination of neural network as controller and a Turing-machine-like memory band as memory. The key feature is that the NTM is differentiable end-to-end, enabling the model to learn algorithms involving memory simply by gradient descent.

Weston et al. [16] recently developed a framework to standardize the terminology and find common features in memory-augmented neural networks. They divide the principle architecture of such a network into four basic building blocks (further explained in Section 3.3). The authors further propose a memory-augmented network model themselves, MemNN, showing an exemplary implementation of their framework.

Metalearned Neural Memory (MNM) proposed by Munkhdalai et al. [6] uses an LSTM cell as controller and an additional memory network as working memory. The controller architecture is similar to the one in H-HebbLSTM proposed in this work, where the keys and values are calculated from the input and the hidden state of the LSTM cell. The differences are, that MNM

performs all memory operations in parallel without a read-out in advance to the write operation, as well as that MNM uses a very different memory architecture. The memory in MNM is a multi-layered fast-weight network trained by gradient descent or, alternatively, by a learned local update rule based on the perceptron learning rule [17]. The term "fast" weights [18], [19] refers to the property of some weights in the neural network model, which are updated during inference to act as memory. In contrast, the "slow" weights are the weights which are learned during training and stay fixed for inference. The fast weights are similar to synaptic plasticity in biological neural networks, where information is stored in the efficacy of synaptic connections [20].

In consideration of the biologically inspired principle of synaptic plasticity, memory based on the Hebbian learning theory [12] was developed and implemented as neural memory network [9], [21], [22]. The Hebbian learning theory is thereby used to update a single layer network acting as neural associative memory [8], [11] in order to map a key pattern to a value pattern. The value pattern can then be retrieved later on by performing a read operation on this neural memory using a key similar to the one used for storing. Via this mechanism, the network model is capable of storing associations as variable-bindings instead of activities. This storing capability is considered crucial to perform cognitive operations as for example associative inference [7].

Similar to the proposed model H-HebbLSTM, Schlag et al. [7] proposed Fast Weight Memory (FWM), a model where an LSTM network is implemented as controller to operate an associative memory subject to a Hebbian learning rule. The major difference to H-HebbLSTM is, that in FWM the memory matrix is a third-order tensor instead of a two-dimensional weight matrix. This results from the fact that FWM calculates a tensor product representation [23] of two key-vectors, based on the idea that this procedure creates a unique representation for every possible combination of key vectors. The authors claim that this way, the model is capable of storing previously unseen association patterns.

2 Recurrent Neural Networks

Artificial neural networks (ANNs) are computational models which are inspired by biological neural networks. The goal of ANNs in general is not to model a biological brain as accurate as possible, but rather to solve a variety of tasks accurately and efficiently. ANNs consist of a number of small computational units called neurons, loosely modeling biological neurons, which are connected via directed weighted connections, similar to synapses in biological networks. To conduct a computation, an input is shown to a specific subset of neurons, called the input neurons, and passed along the weighted connections from neuron to neuron. Each neuron performs a pre-determined function, called activation function, on the sum of its inputs, weighted by the scalar weight of the corresponding input connection, to calculate the further propagated output, which is passed to the adjacent neurons. The activation function of neurons in ANNs is a non-linear function and is inspired by the non-linear behavior of biological neurons, although it is a very loose abstraction [24]. In simple terms, biological neurons are activated if the sum of incoming impulses reaches a certain threshold. This behavior introduces non-linearities into the computation process [25]. Finally, the signal reaches some output neurons which have no further connections. The value of these neurons is then considered as network output [24]. Neural networks are often organized in layers, where the input neurons are the first layer and the output neurons are the last layer. In between, a number of hidden layers might be situated, the number of which is declared as *depth* of the network [26]. The reason why neural networks are often constructed as deep networks, networks with multiple hidden layers, is that each layer can represent an additional layer of abstraction. For example in the task of image processing, the first layer can learn to detect the edges, the second layer learns to combine this information to detect structures and each successive layer can then learn to combine the information extracted by the previous layer. The application of deep neural

models, referred to as deep learning [26], achieved great success on several tasks by learning such deep representations of data [27]. The reason why non-linearities are necessary in any ANN is, that a neural network where each neuron calculates a linear combination of the inputs, the network would only be able to map a linear function. However, in most machine learning tasks, the function that needs to be learned is more complex and cannot be represented sufficiently well with a linear function [26].

A feed-forward network is a restricted form of a neural network, where the computational elements, the neurons, are connected in a way that no cyclic connections occur. For one given input vector the network calculates one specific output vector, independent of the previous or successive input vectors. However, for a number of tasks the constraint of independence between data points in input data is not fulfilled [24].

An example of a family of such tasks would be the processing of sentences in natural language processing (NLP), where the input and/or output is a sentence represented as a sequence of word vectors. If the feed-forward network processes one word of a sentence after another without preserving any information about the previous words, the semantic of the sentence will get lost. Another example would be the classification of music into different genres. In this case the input, the composition, could be represented by a sequence of small data vectors representing notes and the output as single vector representing the genre. Processing one note after another without any preservation of information about the previously obtained notes is not sufficient to solve this task. For dealing with such sequential time-series data feed-forward networks are not well suited due to their static nature and the lack of preserving information about previous inputs [28].

An approach to overcome this issue is the use of recurrent neuron connections, yielding the class of recurrent neural networks (RNNs) [26]. Per definition [29], a neural network is a RNN if it contains at least one recurrent connection between its neurons. In contrast to connections in feed-forward networks, where each neuron is only connected to downstream neurons into the direction of the output, recurrent connections create cycles in the network graph. A network graph is a method of visualizing a neural network by showing neurons as points and connections between them as arrows.

Network graphs help to understand the process of computation and information flow inside a neural network. Figure 2.1 visualizes two example graphs to illustrate the difference between feed-forward and recurrent networks. The cycles in a recurrent network are evaluated iteratively, which requires the network to operate in time-steps, either continuously [30] or discrete [31]. For each time-step, the recurrent feed-back connections pass the output values of the source neurons from the current time-step to the target neurons in the next time-step. Via these connections, information of the previous inputs can be preserved and used for the computation in the next time-step [24]. The recurrent connections act as memory and enable the network to temporarily store information that shall be passed along multiple time steps [28]. Due to the dynamic nature of the recurrent connections, which are able to maintain an internal state between separate items of sequences, recurrent neural networks are a natural choice when it comes to tasks including sequences. Figure 2.2 illustrates how using RNNs for different settings of such tasks looks like.

Except from the above mentioned loss of context information, feed-forward neural network architectures have additional drawbacks when dealing with sequential time-series data [32]. Firstly, sequential data, for example a sentence in natural language, can vastly vary in input length. Since a feed-forward network processes data in one forward pass (separating the data into multiple forward-passes would result in the above described loss of context information), the whole sequence needs to be passed to the network at once. This can be achieved, if the input vector is composed in a way, that the first element corresponds to the first item in the sequence, the second one to the second item, and so on. Elman et al. [32] refer to this representation as spatial encoding. If the sequences are very long, the model would require a large number of input neurons, making it unpractical. Secondly, the absolute position of the key features in the sequence is often not relevant, but rather the relative position of multiple features at different time-steps to each other. If a sequence is passed to a feed-forward network in one time-step, it might be hard to learn to extract features relative to each other and ignore their absolute position. Goodfellow et al. [26] pointed out, that a traditional feed-forward network would have to learn the extraction of the same features for every possible position. For example, a network shall extract the feature ”2019” from the two sentences ”I went to Nepal

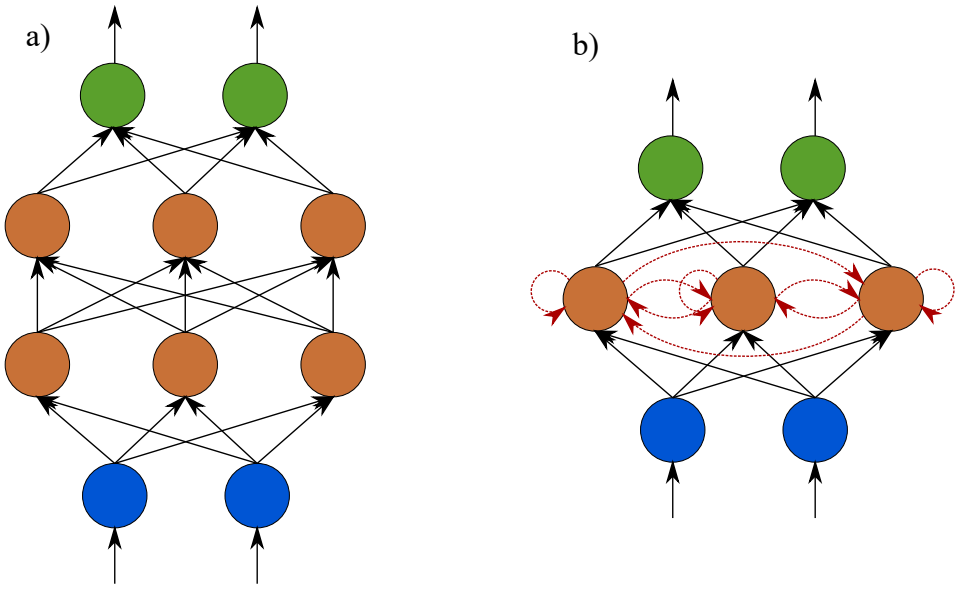


Figure 2.1: Two examples of neural network architectures to demonstrate the difference between feed-forward and recurrent neural networks. a) denotes an example of a feed-forward architecture with two hidden layers, b) denotes an example recurrent architecture with one fully-recurrent hidden layer. Each circle represents a neuron, input neurons are blue, hidden neurons are orange and output neurons are green. Arrows denote weighted connections between neurons, whereas the black arrows show feed-forward connections and the red dotted arrows show recurrent connections. The significant difference between the two architectures is, that each connection in feed-forward networks is only chosen in a way, that the target neuron is fewer connections away from the output layer than the source neuron. Therefore, no cycles appear in the network graph. In recurrent neural networks, this restriction does not exist, so that cyclic connections are allowed. Adapted from [24].

in 2009" and "In 2009, I went to Nepal". If a traditional fully connected feed-forward network is trained on a fixed-size version of such sentences, it would have to learn the extraction of the required feature for each position separately.

With the use of recurrent connections, the RNN can process one element of a sequence after another and still preserve the context and extract relevant features to pass them to subsequent time-steps. Since the same weights are used for each element in the sequence, the model can, in principle, generalize to previously unseen sequence lengths and scale to sequences of arbitrary length. These advantages make RNNs a natural choice for

sequential data like sentences in NLP as mentioned in the paragraph above. When sequential data needs to be processed, some information about the previous items of the sequence is often required in order to gain an accurate prediction. RNNs can learn to preserve a lossy summary of the task-relevant information of previously shown sequence elements to include them in the processing of future elements [26]. The recurrent connections therefore act as an activity-based memory. In Chapter 3 RNNs with other types of memory are introduced.

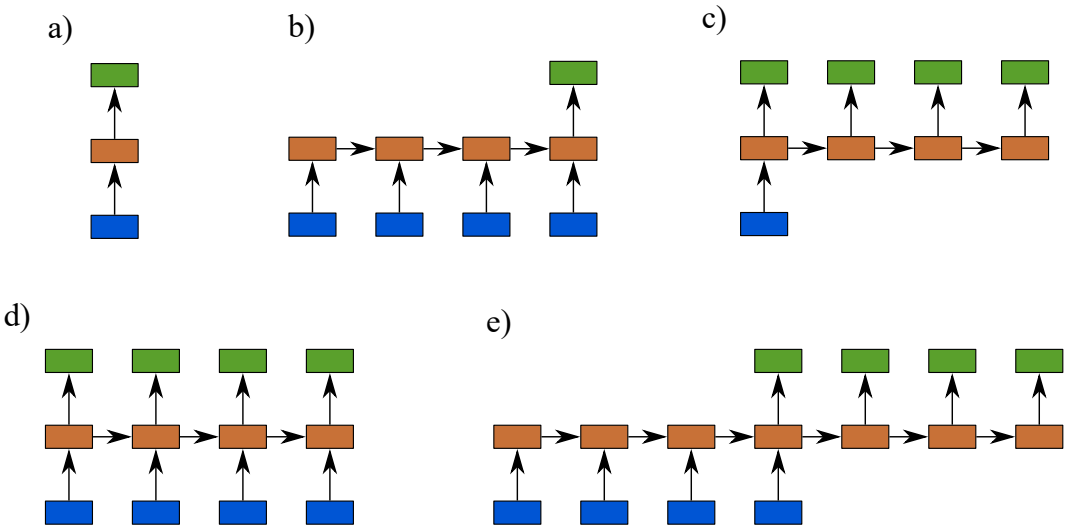


Figure 2.2: Different settings of RNNs for different types of tasks with or without sequences shown as unfolded computational graphs (see Section 2.2.4 for details about unfolding). Blue nodes are input neurons, green nodes represent hidden neurons, red nodes are output neurons. In (a), a fixed-size input vector is processed to produce a fixed-size output vector. This is the basic setting for any feed-forward neural network. No sequences are involved, each input vector is processed independently. (b) shows the setting where one sequence of input vectors produces one fixed-size output vector. The horizontal arrows between green elements show the unfolded recurrent connections, which pass information along through the computational time-steps. This setting appears in sequence labeling tasks, for example text classification, video classification and music classification [24]. In (c), a fixed-size input vector is used to generate an output sequence. Image captioning is an example of a task for this kind of architecture. (d) and (e) both show variations of a sequence-to-sequence model. Language translation would be an example of a task where one sequence is translated into another. The difference between (d) and (e) is that in (e), the model reads the input sequence to the end before the output sequence is generated, whereas in (d) the model produces one output sequence item for each input sequence item. Adapted from [33].

In the supervised learning setting, each sample consists of an input and a target [26]. Via training, the neural network weights are are optimized in a way that the network performs well on a task by successfully estimating the target values for previously unseen examples [24]. The network training is discussed in detail in Section 2.2.

2.1 Architectures

Several fundamental architectures of recurrent neural networks evolved over time. For example Jordan networks [34], which contain recurrent connections from the output to a hidden layer in the network or Time-Delay Networks [35], which accumulate neuron states over multiple time-steps. Jaeger et al. introduced Echo State Networks (ESNs), those are recurrent networks with randomly assigned and fixed connections, not organized in layers, where the recurrent dynamics are used to map the input to higher-dimensional space and the only learned weights are the ones of the connections incorporating output neurons [36]. This approach is also referred to as reservoir computing [37]. One further fundamental architecture is the one proposed by Elman et al. [32] and is discussed further in section 2.1.1. The basic idea is to extend a two-layer feed-forward network by adding fully recurrent self-connections to the hidden layer. This allows the network to maintain an internal state and pass along context information to the next time-step.

2.1.1 Elman Networks

Elman et al. [32] stated, that time plays a major role in cognitive tasks. The authors point out, that the explicit representation of time, for example by implementing a spatial time representation, introduces multiple additional challenges (some of them were covered in the beginning of this chapter). In contrast, if time is represented implicitly by the effect it has on processing, some of these challenges can be solved. Therefore they proposed an architecture for recurrent neural networks (named Elman networks) to cope with time-dependence of sequential time-series data. The authors propose a method to encode time implicitly in the form of step-wise computational

processes rather than explicitly via a spatial representation in the input [32].

The network architecture represents a very basic architecture of RNNs and is therefore also referred to as Simple Recurrent Network (SRN) [38]. The Elman network architecture is in principle similar to a feed-forward network with one hidden layer and consists of an input layer, a hidden layer and an output layer. The difference is in the context neurons, which copy the content of the hidden layer at each time-step. The content of those context neurons is then passed to the hidden layer alongside with the next element of the input sequence in the next time step. In mathematical formulation the output vector of the hidden layer at a time-step t can be stated as

$$h(t) = a_h(W_h\mathbf{x}(t) + U_h\mathbf{h}(t-1)) \tag{2.1}$$

where a_h denotes the activation function of the hidden layer, $W_h \in \mathbb{R}^{d\times m}$ the weight matrix storing the connection weights from the d input neurons x to the m hidden neurons h and $U_h \in \mathbb{R}^{m\times m}$ the weights of the recurrent self-connections in the hidden layer [39]. The output of the network is then calculated by

$$y(t) = f(W_y\mathbf{h}(t)) \tag{2.2}$$

with output function f.

Hence, the hidden layer of the Elman network is a recurrently self-connected layer with learned weights. Figure 2.3 illustrates this basic architecture graphically. The feed-forward connections therefore yield the output of each time-step and the feed-back connections allow the network to maintain a state which is passed to the computation in the next time-step, when the next item of the input sequence will be presented to the input neurons. For each time-step, the network uses the same weights. This allows the network in principle to generalize over time.

The architecture of Elman networks is often referred to as most simple, basic RNN architecture [24], [26]. It is shown, that an RNN with as few as one recurrently connected hidden layer, like in the Elman network, is sufficient to approximate any open dynamical system with arbitrary accuracy [40].

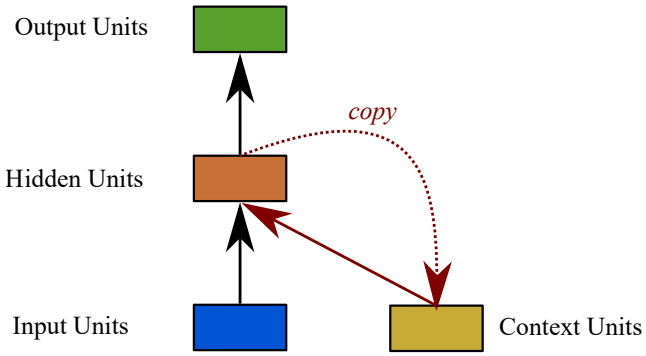


Figure 2.3: Basic architecture of an Elman network. Feed-forward connections are represented as solid black lines, red lines denote feed-back (recurrent) connections. The dashed red line denotes a copy operation which is performed at each time-step. The context neurons allow the network to preserve relevant information from sequence items which previously were processed. Adapted from [32].

2.1.2 Long Short-Term Memory

Commonly used training algorithms for RNNs often incorporate error gradients (see Section 2.2 for details) [41]. These algorithms calculate an error between the actual output of the RNN and a desired target. To find network weights which minimize this error, a gradient of the error function is calculated and propagated from the output neurons all the way back to the input neurons. However, since for each neuron this gradient is multiplied with the associated weight of the connection it traverses back, the gradient tends to either vanish or explode exponentially with increasing network depth [42] (see Section 2.2.5). The computational depth of a RNN increases with increasing sequence size, therefore the vanishing gradient problem negatively affects the training on long sequences. Long short-term memory is an approach to tackle this kind of problem and was introduced by Hochreiter et al. [13].

The approach to counter the vanishing gradient problem is to keep the gradient stable through the unfolded network. To achieve this, Hochreiter et al. [13] introduce LSTM, a gated memory cell maintaining an internal state. The main property of this internal state is that it is recurrently connected to itself via a constant-weighted connection of weight 1. In the Backpropagation through time (BPTT) algorithm [41] the gradient is back-propagated by

multiplying it with the incoming weights of a neuron. If the incoming weight is fixed to the value 1, as it is with the internal state in LSTM, the gradient gets multiplied by 1, keeping it constant through all time-steps. The internal state which is recurrently connected to itself "traps" the gradient in this constant loop. The authors call this effect the constant error carousel (CEC) and describe it as the central feature of LSTM. Despite the CEC, a full-blown LSTM cell implements several other features to enable the cell to be trainable in the context of a RNN. The internal state, also called hidden state, enables the network to store information over a longer period of time to preserve it for later use. To utilize this hidden state, read and write operations are necessary. The RNN learns how to utilize the memory by learning to perform these read and write operations via using a mechanism which the authors call gate units. A gate unit is a neuron which controls the flow of information into the memory or from the memory. The vanilla LSTM architecture contains two gates, an input gate which controls the flow into the memory (i.e. the hidden state) and an output gate to control the flow of information from the memory.

Gers et al. [43] introduced an extension to the vanilla LSTM architecture, called the forget gate. The forget gate can trigger the cell to erase the previous value, so that it can be replaced by a new one more easily. The authors showed, that this feature improved the capabilities of the LSTM cell when dealing with sequences, which are not segmented *a priori*. In a further publication, Gers et al. [44] introduced another improvement to the vanilla LSTM cell, called the peephole connections. These connections enable the gates to incorporate the state of the memory cell into the calculation of the gate values. The authors show, that these connections enhance the LSTM cell's performance on tasks, where precise timing measurement is important. In figure 2.4, the peephole connections are indicated with dashed lines.

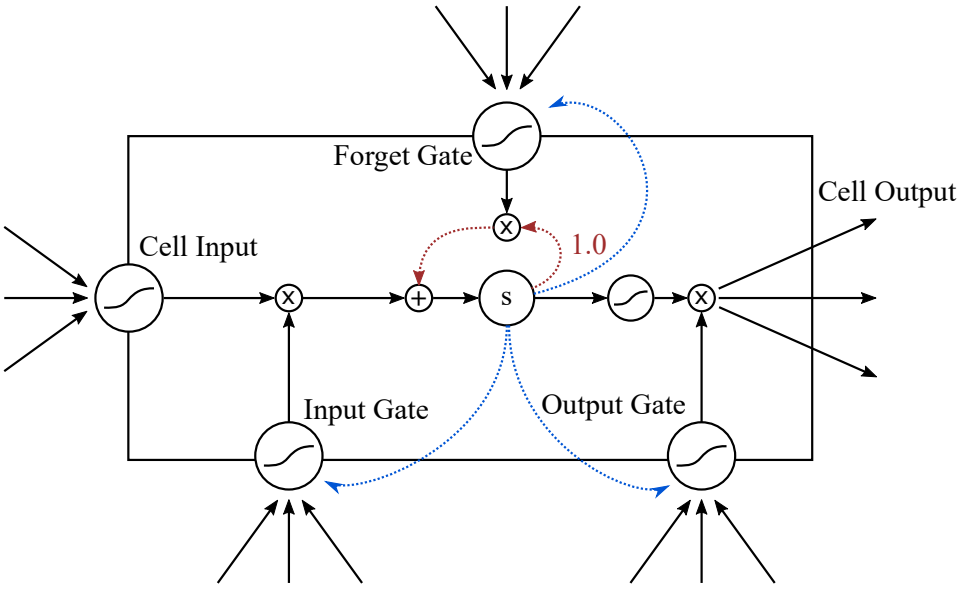


Figure 2.4: The basic architecture of a vanilla LSTM cell including the extensions from Gers et al. [43]. The arrows outside of the block denote connections from and to other neurons. The circles with sigmoidal curves denote the activation functions of the gate units and the cell unit, whereas different activation functions may be used for the different gates. The circles with an "x" denote element-wise vector multiplications, which are the points where the gate-values are applied. The circle with the "s" label in the center is the memory unit, which is connected to itself via a recurrent connection (dashed red arrow) of weight 1, and no activation function. This recurrent connection implements the constant error carousel (CEC), which is the core of the LSTM cell. The dashed blue lines are called "peephole connections" and are alongside with the forget gate an extension to LSTM, proposed by Gers et al. [43], [44]. Adapted from [24].

2.2 Training

Goodfellow et al. [26] describe the overall goal of any machine learning algorithm to perform well on unseen examples. This capability is called generalization, and can be measured by testing the machine learning model on a set of data examples, which were not used to train the model, namely a test set. Before the model can be tested using the test set, it needs to be trained. For example, if a neural network should learn to classify a photograph of an animal into one of the three classes "horse", "cat" and "other", it has to learn how to approximate the function $\boldsymbol{y} = f^*(\boldsymbol{x})$ which maps

the input image $\boldsymbol{x}$ to the correct class vector $\boldsymbol{y}$. It does that by learning the network parameters $\boldsymbol{\theta}$, so that the function $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$, which the network employs, approximates f^* as good as possible [26]. Neural networks with only one hidden layer can, in theory, approximate any function with arbitrary accuracy, as long as the number of hidden units is sufficiently high and the activation functions are sufficiently smooth [45]. The actual generalization performance of a neural network however, is in very dependent on its training. Training algorithms like gradient descent (see Section 2.2.1) optimize the weights of the network, which are a subset of the network parameters $\boldsymbol{\theta}$. The other parameters are for example the network architecture (in particular the number of layers and hidden units) or the activation functions. These parameters are not learned by the optimization algorithm and are called hyperparameters. The hyperparameters are the input to the training algorithm and need to be determined differently beforehand, either manually or by using separate algorithms [26].

The above described method of learning is called supervised learning, because the target value (for example the class label) is explicitly determined for each of the input examples. In addition to supervised learning, other settings are unsupervised learning, where no target value is provided and reinforcement learning, where the target is not modeled explicitly, but rather as a reward, assigning a score to the calculated result [28].

2.2.1 Learning Weights with Gradient Descent

The goal of a learning algorithm is to find network parameters to achieve the best possible approximation of the desired function. It is common in a machine learning setup, to do so by designing a function that yields a numerical error value given an input, prediction and a desired target, where the prediction is the calculated output of the machine learning model. This function is task-dependent and often referred to as cost function or error function. A learning algorithm uses this cost function to calculate how the model parameters, in the context of neural networks the weights, need to be adjusted in order to achieve accurate results [46]. The tasks can be rather complex, therefore the analytic calculation of the error surface is mostly not solvable in practice, resulting in the problem that the absolute values of

the best-performing parameters cannot be calculated, because the absolute positions of the minima of the error function are not known. A widely used algorithm for the training of feed-forward neural networks is to iteratively adjust the weights of the network, so that the cost function is reduced [47].

To get an understanding of how an algorithm can successfully adjust neural network weights, the most simple form of a neural network is considered first. This simple form is a network consisting of only one feed-forward layer and is called a perceptron, which was described by Rosenblatt et al. [17]. More complex neural networks including hidden layers or recurrent connections, as introduced in Chapter 2, are called multi-layer perceptrons.

Figure 2.5 shows the most simple form of a perceptron, consisting of only one neuron. This neuron has input weights $w_1 ... w_n$ which are multiplied element-wise with inputs $x_1 ... x_n$, and summed up to a single scalar value. Via a step function using this weighted sum as input, it calculates an output o of -1 or 1. This perceptron can then be used to perform some (very simple) tasks, for example classification into two classes, where the values -1 and 1 correspond to the different classes respectively. To solve this task, the perceptron must approximate the underlying function which determines the class for a given point in the feature space. It does this, by adjusting the weights of the inputs, using the perceptron learning rule [46].

The perceptron learning rule for the weight update of weight w_i with input x_i is denoted as

$$w_i \leftarrow w_i + \Delta w_i \tag{2.3}$$

with

$$\Delta w_i = (t_p - o_p) x_{ip} = \delta_p x_{ip} \tag{2.4}$$

where the index p indicates the training example from data set P. t_p is the desired output for this training example p, o_p the output of the perceptron for training example p. δ_p, which is equal to the term $(t_p - o_p)$ can be interpreted as error signal at the output neuron, which is the deviation of the output o_p from the desired target t_p. If this error is zero, the weights are already correct and hence Δw_i in Equation 2.4 equals to zero. This learning rule is proven to find the optimal set of weights, if, and only if,

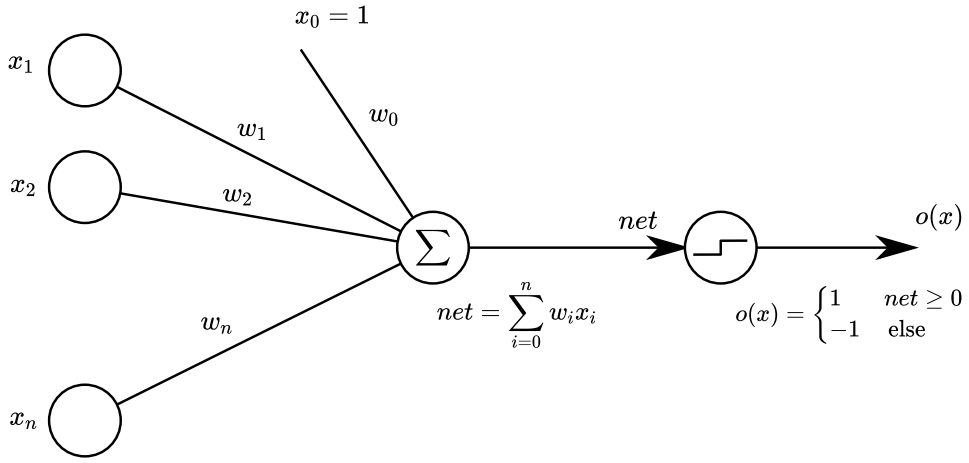


Figure 2.5: Simple perceptron consisting of one single neuron with a threshold activation function. The input values are multiplied with the corresponding weights and summed up before the threshold is applied. w_0 is called bias and is equivalent to a shift in the threshold value. Adapted from [46].

such set of weights exists. In the case of the simple single-layer perceptron, this condition requires the data set to be linearly separable. Unfortunately, this is not always the case. A trivial example of a not linearly separable function would be the XOR function [48]. To solve such a task, a more generic learning rule is needed, which is capable of training the weights of more complex networks.

A more generic form of a learning rule is the generalized delta rule. The basis of this rule is an error function, which in general should be minimized, as well as a mechanism called gradient descent, firstly appearing in [49]. The error function denotes the deviation of the network output from the desired target output and can look different for different setups. The idea of gradient descent is to calculate the direction of the steepest decrease of the error in a given point of the error surface, then follow this direction for a small step. Re-applying the procedure of following the direction of the steepest descent will converge to a local minimum, if such exists [26]. This procedure can be illustrated by a person, always walking into the direction of the steepest downhill to climb down a mountain. For the explanation of the generalized delta rule, a single layer network consisting of K output neurons is assumed. Also, this procedure is not applicable to a perceptron

with a step function, since the gradient at the step is not defined.

One example of an error function which shall be minimized using gradient descent is the sum squared error denoted by

$$E = \sum_{p \in P} E_p = \frac{1}{2} \sum_{p \in P} \sum_{k \in K} (t_{pk} - o_{pk})^2 \tag{2.5}$$

with data sample p of data set P, desired target value t_p and network output o_p.

To determine the network parameters with gradient descent, the gradient of the error function with respect to each of the network weights needs to be calculated. The general update rule for input weight w_{ki} for one iteration of gradient descent for one data example p results in

$$w_{ki} \leftarrow w_{ki} + \Delta_p w_{ki} \tag{2.6}$$

with

$$\Delta_p w_{ki} = -\eta \frac{\partial E_p}{\partial w_{ki}} \tag{2.7}$$

where $\frac{\partial E_p}{\partial w_{ki}}$ is the derivative of error E_p w.r.t. w_{ki}. The variable w_{ki} is the weight of the connection from input neuron i to output neuron k and η is a constant called learning rate which is used to moderize the weight updates [26], [46].

The term $\frac{\partial E_p}{\partial w_{ki}}$ can be interpreted as the contribution of the weight w_{ki} to the error E_p. Note, that the absolute value of the weight update is proportional to this contribution, meaning that weights which had a high contribution to the error are stronger changed than others [48].

Calculating E_p for all data examples p at each iterative weight update is often not practical. Instead, an algorithm called stochastic gradient descent (SGD) [50], [51] is used to improve the training performance. In this algorithm a random subset, also called minibatch, of training examples is chosen to calculate the error gradient and weight updates for each iteration. The batch size is a hyperparameter which can be varied to improve training performance. For the weight update, the error is averaged over the minibatch [26].

In neural networks it is common that each neuron is subject to a non-linear activation function a. This can for example be the logistic sigmoid function $a(x) = \frac{1}{1+e^{-\beta x}}$ [28]. Solving the partial derivative $\frac{\partial E_p}{\partial w_{ki}}$ for the sum squared error from Equation 2.5 for an output neuron k with arbitrary, but differentiable and monotonically increasing, activation function a_k results in

$$\Delta_p w_{ki} = \eta \delta_{pk} o_{pk} \tag{2.8}$$

with

$$\delta_{pk} = (t_{pk} - o_{pk}) a'_k(net_{pk}) \tag{2.9}$$

where net_{pk} is the weighted sum of inputs to neuron k, also written as $\sum_{i \in I} w_{ki} x_i + bias_i$ over input neurons I. The additional bias term $bias_i$ is optional and can be interpreted as similar property as the threshold in a simple perceptron [48].

Note, that the result of the generalized delta rule in 2.9 for the sum squared error is very similar to the perceptron learning rule in Equation 2.3, except with the difference that in the delta rule the derivative of the activation function is included, whereas the update in the perceptron rule is linear to x_i [46].

2.2.2 Training Deep Networks by Backpropagating Errors

In the previous paragraph, the update rule for gradient descent in the case of a single-layer perceptron is explained. However, to train deeper networks with a number of hidden layers > 0, the calculation of the error gradient $\frac{\partial E}{\partial w_{ji}}$ with respect to a weight w_{ji} from neuron i to a non-ouput neuron j is not straight-forward, since the definition of the error (for example like in Equation 2.5) is not explicitly a function of w_{ji}. To apply the update rule from Equation 2.6 to a neuron which is not in the output layer of the network, the error gradient needs to be calculated per neuron recursively by an algorithm referred to as backpropagation [47]. In this algorithm, the partial error gradient for each neuron is calculated backwards, beginning from the output layer. Hence, for each neuron, the contribution of the neuron

to the global error is calculated and used for the update of its incoming weights. This process is also called backward-pass, because in contrast to the forward-pass, where the computation is performed from the input neurons to the output neurons, the computation is performed backwards, starting at the output neurons [52].

The first step of the backpropagation is to calculate δ_k for each output neuron k as in Equation 2.9. Note that the index p is omitted because it is irrelevant if the error is calculated for one training example or multiple examples as in stochastic gradient descent.

As next steps, the error signals δ_j for each hidden neuron j are computed recursively by calculating a weighted sum over the error signals of all adjacent neurons

$$\delta_j = a'_j(net_j) \sum_{n \in N} \delta_n w_{nj} \tag{2.10}$$

with N being the set of neurons n where a weight w_{nj} from neuron j to neuron n exists. This recursive computation is repeated starting at the second-last layer and continued backwards into the direction of the input neurons until all error signals are determined.

Then, each weight can be updated according to the generalized delta rule (Equation 2.8) as if it was an output neuron.

2.2.3 Backpropagation Through Time

The backpropagation algorithm discussed in this Chapter assumes a feed-forward neural network. As the partial derivatives of the errors are computed by starting at the output layer and recursively backpropagating them layer after layer until the input layer is reached, no cycles in the computational graph are allowed. However, Rumelhart et al. [47] point out that for each recurrent neural network operating in a finite time episode, an unfolded, equivalent feed-forward network exists. Figure 2.6 shows an example of this process of unfolding. The algorithm of backpropagation can then be applied on the unfolded network as discussed in the previous paragraph, with one exception: The recurrent network re-uses the weights

for each time-step t. If the number of considered time-steps is > 2, multiple update values $\Delta w_{ji}^{(t)}$ for the same weight w_{ji} from neuron i to neuron j are computed within the backward pass. These values need to be stored intermediately, and are applied after the calculation of partial gradients is finished [41].

2.2.4 Unfolding of RNNs

RNNs operating on sequential data are processing one item after another. Recurrent connections allow to pass information to the next time-step, which can be interpreted as additional input to the current time-step, summarizing the information of all previous time-steps. To process a sequence of data one by one, the network has to perform this process as often as the number of items the sequence contains. Thus, if the size of the sequence increases, the number of computational steps will also increase. Figure 2.6 visualizes the computation process inside a RNN by re-drawing the computational graph with emphasis on the time component on a very simple example network with one input neuron and one hidden neuron. The hidden neuron maintains a state over time which is dependent on all previous inputs. After unfolding, the network graph results in a feed-forward network with separate input neurons for each sequence item [26]. Because the weights are re-used for each time-step, the absolute number of weights inside a simple RNN is fixed and does not change with varying input size. However, since the weights are applied on each item of a sequence, the unfolded connection graph grows with increasing sequence sizes. As a result the computational graph grows very large for very long sequences.

2.2.5 The Vanishing or Exploding Gradient Problem

Hochreiter et al. [42] proof, that if a conventional RNN is trained with a gradient-based algorithm like back-propagation through time (BPTT) [41], the gradient tends to either vanish or explode, leading to very small or very large weight changes respectively. The resulting effect is that the weights are either not sufficiently changed or start oscillating. The longer

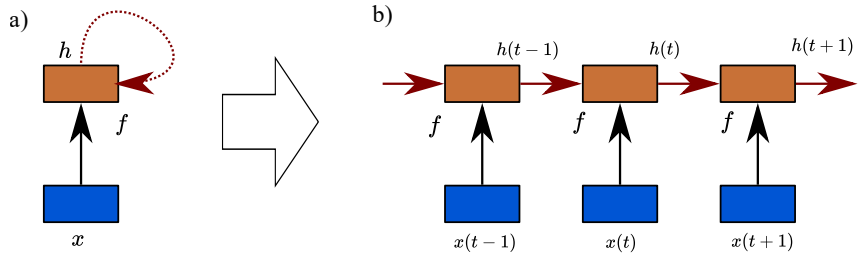


Figure 2.6: Unfolding of the computational graph of a RNN. The RNN in this figure consists of one input neuron and one hidden neuron. The hidden neuron has a recurrent self-directed connection, where the black square marks a time-delay of 1 time-step. In each time-step, the input $\boldsymbol{x}(t)$ is shown to the network. For each time-step, the hidden value $\boldsymbol{h}^{(t)}$ is calculated via $\boldsymbol{h}(t) = f(\boldsymbol{h}(t-1), \boldsymbol{x}(t); \boldsymbol{\theta})$, where the network parameters $\boldsymbol{\theta}$ parametrize the network function f. Note that in the unfolded computational graph each sequence item $\boldsymbol{x}(t)$ has its own input neuron. For each connection of these neurons to the hidden neurons, the same weights are used. Adapted from [26]

the path in the computational graph the gradient is propagated back, the stronger is the effect of exploding/vanishing gradients. When looking at the unfolded computational graph in a task where a sequence is mapped to a single output value, it appears that the first items of the input sequence are further away from the output neurons than the latest items in a sequence. Thus, the training with back-propagation, where the error calculation is performed iteratively starting at the output neurons, is much more sensitive to items which appear later in the sequence, because the gradient passes more neurons to reach the earlier sequence inputs and is therefore more prone to explode or vanish. This leads to problems in the overall training performance and can be a problem if the relevant information of a sequence is at the beginning or if large time lags of irrelevant inputs are situated between relevant sequence elements. Long short-term memory [13] is an approach to tackle this issue and is discussed in Section 2.1.2.

2.2.6 Regularization and Validation

Minimizing the error on the set of data vectors used for training, the training set, does not automatically conclude that the model performs equally well on unseen examples. In machine learning, the expected error on an unseen

example is called generalization error [53]. This generalization error is a performance measure of how well the model generalizes over the data set, or in other words learns the true function behind the data. Minimizing this error is the overall goal of training. However, since the algorithm can only train the model on a finite set of examples, only the training error can be directly reduced by the algorithm. Under certain circumstances, the generalization error can be very high, even though the training error is very low. This phenomenon is called *overfitting* [54] and results from the fact, that the model fits the training set very well, but does not learn the true function of the whole data set. This can be compared to "learning by heart" and is something which in general shall be avoided. If the model is too simple to represent the complex underlying function which created the data set, it can happen that a low error on the training set cannot be achieved overall. This phenomenon is called *underfitting* .

The complexity of the function a model can learn is determined by a property called *model capacity*. The capacity must be chosen in a way, where the model yields the best generalization performance. To estimate the generalization performance, a validation set can be used. This validation set is a set of examples, which are neither used for training, nor for the calculation of the test error [53]. Instead, the validation set is used to calculate a separate validation error, which can be used to estimate if the model complexity is appropriate for the task at hand. In general, any measure that is taken to reduce the generalization error but not the test error, is called *regularization* [26]. Regularization is used to find the simplest model which is capable of fitting to the true underlying function which is represented by the data. An example of regularization is weight decay, where a function of the norm of the model parameters $\boldsymbol{\theta}$ is added to the error function, to favor low weight coefficients in the model.

2.2.7 Training Improvements

Several ways to improve the training of neural networks evolved over time, some concepts are described briefly below.

Momentum One way to further improve the gradient descent algorithm is to add a momentum term to the learning rule [47]. This momentum term is simply the weight change of the last iteration, but discounted with a factor α with $0 < \alpha < 1$. The effect can be compared to the momentum in physics, where for example a particle slipping down a surface accelerates. This momentum term is meant to speed up the overall learning procedure and to filter out high-frequency variations of the error surface.

Adapting the Learning Rate The choice of the learning rate affects the learning process of the model significantly [26]. It is common to start with a high value of the learning rate and decay it successively during training. This approach is called learning rate decay and aims not only to accelerate learning by applying larger weight updates at the beginning of the training procedure, but also to escape local minima of the error function [53]. A different approach to adapting the learning rate is employed by the algorithm ADADELTA [55]. In this algorithm, the learning rate is adapted dynamically per-dimension, resulting in training which is more robust to the choice of the learning rate.

Some algorithms, for example ADAM [56], combine multiple of such improvements. The name ADAM is an abbreviation for *adaptive moment estimation* and employs both of the previously explained training improvements, *momentum* and *adaptive learning rates*.

The previous Chapters briefly introduced RNNs and how they can be trained. In the following Chapter some exemplary architectures of RNNs are shown, in particular models which are additionally augmented with different types of memory.

3 Memory Augmented Neural Networks

According to Du et al. [10], memory is a system with three functions: recording - storing information, preservation - persisting the information and recalling - retrieving the stored information. The general idea of combining memory mechanisms with neural network architectures is to enable the network to learn how to perform these three operations on memory to solve tasks where memory is required. One memory mechanism which was already discussed in Section 2.1.2 is long short-term memory (LSTM) [13]. This Section introduces some alternative concepts of how different types of memory can be utilized in neural networks.

3.1 Neural Turing Machines

Graves et al. [15] proposed a neural network model, named Neural Turing Machine (NTM), where a neural network acts as controller to perform read and write operations on a memory which the authors compare to the memory band of a classical Turing machine. In contrast to classical Turing machines, NTM is fully differentiable, which means it can be trained end-to-end by the use of a gradient descent algorithm. Figure 3.1 shows the basic architecture of the NTM, consisting of the controller, the memory and the logic used for reading and writing from and to the memory. Read and write logic are implemented in a way that the NTM can utilize two different memory addressing schemes, content-based and location-based addressing. The content-based mechanism is used to retrieve a pattern from the memory by using a partial sub-pattern, similar to the mechanism of Hopfield networks [57]. The location-based retrieval mechanism is used to

implement iteration over the memory or random-access, similar to the use of pointers in Turing machines. The calculation of the memory address is implemented in multiple steps, enabling the NTM to use either of the two memory addressing schemes. As input for the address calculation process, the network receives the address vector of the previous time-step in addition to the network input. The network can then choose to either re-use and modify the previous address vector (location-based) or to calculate a new one using an algorithm which implements the content-based addressing scheme. This choice is implemented by using a gating mechanism, similar to the gated mechanisms of LSTM [13] (see Section 2.1.2). The authors show that the NTM is capable of learning a variety of tasks, for example copying of sequences and associative recall and thereby outperforms LSTM.

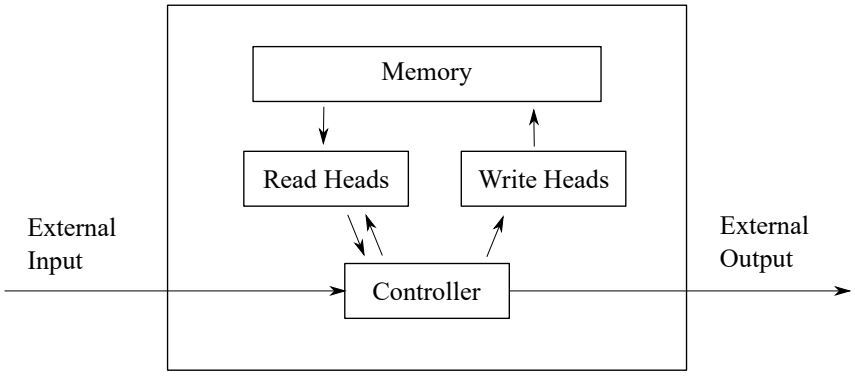


Figure 3.1: Architecture of the Neural Turing Machine. The network can learn to utilize the memory by using multiple read and write heads. The controller is either a recurrent or a feed-forward neural network. Adapted from [15].

3.2 Dynamic Neural Turing Machine

Gülçere et al. [58] extended the concept of the NTM by introducing the Dynamic Neural Turing Machine (D-NTM). As in the NTM, the main components of the model are the controller, which in most tasks is chosen to be an RNN, and the memory. In contrast to NTM, D-NTM implements a learnable addressing scheme for location-based addressing. The memory consists of a number of memory cells where each cell is divided into two

parts, the learnable address vector and the content vector [58]. The address vectors of the memory are learned during training and remain unchanged during inference. The content vectors are not fixed during inference and represent the slots where information can be stored and are initialized with zeros for each episode. The authors claim that this mechanism enables the model to utilize sophisticated location-based addressing strategies. At each time-step, the model performs a read operation on the memory by computing a read address. The information which was read out from the memory is then again used to calculate the network output for the time-step and a new content vector, which is then inserted into the memory by using a separately calculated write address. For the calculation of the addresses, a mechanism similar to the content-based addressing scheme of NTM is used. The authors also implement a memory operation referred to as Dynamic Least Recently Used Addressing (LRU), where the memory address is influenced in a way that addresses not used recently are emphasized. This influence of this mechanism to the actual processing is controlled by a separate shallow network.

The D-NTM can operate in two different modes, continuous and discrete. In continuous mode, the address vector can have multiple non-zero components which sum up to 1. In discrete mode, the address vector is converted to a one-hot vector with only one non-zero component with the value 1. Therefore, in discrete mode only one memory cell is addressed whereas in continuous mode multiple cells are. The addressing operation in discrete mode is not differentiable, thus straight-forward cost function minimization with gradient descent is not feasible. Instead, a method similar to the REINFORCE algorithm [59] is used to perform the minimization of the cost function. To improve the training performance of the discrete D-NTM, a binary coefficient is used to determine whether to use the continuous or the discrete weights for a sample. This coefficient is chosen in a way, that the network uses the continuous addresses more often in the beginning of training and the discrete weights more often at the end [58].

3.3 Memory Networks

Weston et al. [16] elaborated a general framework to describe the fundamental components of a memory-augmented machine learning model. They introduce the term memory neural network (MemNN), if the model is a neural network augmented with a separate memory mechanism. These fundamental components are abbreviated with the letters I, G, O and R.

- Input feature map I converts the network input into an internal representation.
- Generalization component G performs operations on the memory to update its state. This component is therefore responsible to persist the relevant information of the input in the memory. This component is also responsible for organization of the memory. It can potentially perform several operations.
- Output feature map O is the component to calculate an output given the memory state and the input. This component might perform multiple memory operations as well, however it is not responsible for organizing the content of the memory, but rather of using the memory to retrieve relevant information which was stored previously.
- Response component R converts the output of O to match the desired output format, in the example of question answering tasks this could be one word.

Despite the definition of the general building blocks, the authors propose a rather simple example of an implementation of such a MemNN. The proposed implementation is designed to solve similar question answering tasks as described in Section 7.2. The input of this task is a sequence of facts followed by a query, provided as sentences in natural language, whereas the output is one single word which answers the query. In the proposed implementation, the memory consists of a band of slots, where the number of slots is at least as high as the number of facts in the input. The input component I does not transform the input sentence, it simply passes it to the G component, which then stores it in its original form into the next available memory slot. The inference is performed by the O component, which performs a two-step operation on the stored items to calculate an answer by the use of scoring functions. In the first step, the scoring function

compares the network input vector with each previously stored vector and retrieves the vector with the highest score. In the second step, the input vector and the previously retrieved vector are again compared to the other stored vectors by the scoring function to again retrieve the one with the highest score. The input vector alongside the two retrieved vectors are then passed to the final component R. To calculate an answer from the retrieved information and the input sentence different approaches are proposed. The calculation can either consist of simply returning the second retrieved sentence, or, especially for textual responses, be performed by a separate RNN. In the case of a one-word response as in the conducted question answering experiment, the model uses another scoring function, applies it to each previously seen word and finally returns the word with the highest score. Multiple tweaks to further improve the performance on the conducted experiments are introduced, for example the computation of answers in G is adapted to incorporate the relative write timing of facts.

3.4 End-to-End Memory Networks

Sukhbaatar et al. [5] propose a model, which in principle implements the framework proposed in the MemNN paper by Weston et al. [16]. Their model is an improvement from the one proposed by [16], because it is trainable end-to-end by using input-output pairs and requires less supervision. It does that by implementing a differentiable continuous memory mechanism. The authors also incorporate the concept of hops, where for each output signal the internal computation process is executed multiple times, where each time is considered as a hop.

The model is applied to a task where an example consists of a set of facts, a query, and an answer. The model calculates two continuous embeddings for each of the facts and stores all of them into a memory. Then the memory is queried by using a separate embedding for the query, yielding an output vector, which is further combined with the embedded query and processed by a final weight matrix. The embedding weights and the final weight matrix are learned during training. To implement multiple hops, the authors stack multiple of those processing layers on top of each other.

In addition to the basic model the authors propose position encoding (PE), a method, where the position of each word is included in the embedding of the sentence. In a straight-forward bag-of-words (BoW) representation, this positional information gets lost, since the representation only includes the information which words the sentence contains, but not where they are located within the sentence. Furthermore, the authors propose a temporal encoding, to include the timing information of a fact into the embedding. In sequential tasks like the ones used to evaluate this model, temporal information is crucial in order to answer a query. The temporal encoding is a learned representation of the time-step, which is equivalent to the index of the element within the sequence.

3.5 Metalearned Neural Memory

Munkhdalai et al. [6] propose a model where a LSTM network serves as a controller for an associative memory.

Each time the network receives an input, it performs multiple read and write operations on the memory in parallel. A write operation is defined in the way that the memory rapidly binds a write key to a target value, a read operation is defined by retrieving previously stored patterns by querying the memory using one or more read keys. The model uses the hidden state of the LSTM controller together with the read-out vector of the previous time-step and the current input to calculate a set of write keys with their corresponding target values, a set of read keys and a scalar rate value which controls the strength of the memory update.

The memory is considered as an adaptive function with weights as parameters. These weights are adapted each time a write operation is performed and therefore represent the stored content. The authors propose two different variants of the implementation how the weights of the memory function can be trained. The first variant is a gradient descent based update, where the loss is defined as mean-squared-error between the target output vector and the output vector of the memory network given the input key. This procedure however requires expensive gradient computations and a sequential back-propagation, which results in a computational bottleneck. The

second update is a novel gradient-free mechanism where the gradient of the activation function is approximated and the weights updated according to a perceptron learning rule. The authors point out, that this update mechanism enables parallel update of weights since no backpropagation is needed which results in faster performance. The depth of the neural memory is determined by a hyperparameter and set to 3 for the tasks described in the paper. Since each computation in the model is differentiable, it can be trained end-to-end using gradient descent.

As results, the model is able to solve all 20 tasks in the word-by-word representation of the bAbI [4] synthetic question answering tasks, outperforming its predecessors.

3.6 Fast Weight Memory

Schlag et al. [7] recently proposed a network architecture where, similar to MNM (see Section 3.5), a LSTM network is used to control a hetero-associative memory. The authors refer to the memory as fast-weight memory (FWM) since, in contrast to the LSTM controller, its weights are updated using a Hebbian like learning rule (see Chapter 4) at every time-step during training and testing. The weights of the LSTM model are trained via gradient descent during training and remain unchanged during inference. The idea of the model is to improve its generalization on tasks where associative inference, the resolution of relations between different features, is needed. The authors claim that it is important for a model to generalize to all unseen compositions of different features. Due to the vast number of possible combinations between features, the model implements a mechanism to construct structures by combining multiple components called tensor product representation (TPR) [23]. The idea of TPR is to store context-specific associations via the combination of two key-vectors using the tensor product. The TPR mechanism guarantees unique representations for all possible combinations of components, under certain constraints [23].

To store the TPRs, the memory matrix is a third-order tensor updated by a rule closely related to the perceptron learning rule. For each time-step two separate key vectors k_1 and k_2 are calculated by the LSTM controller and

combined via a tensor multiplication to obtain the TPR of the key. This new key is then multiplied with a value vector, again using the tensor product, to obtain a third-order tensor for the memory update. An additional scalar strength value β, bounded between 0 and 1, is calculated for each update and acts as gate to determine the strength of the weight update for each time-step.

The authors claim that their model is very stable, because it can be trained with very long sequences. This property is shown in an experiment, where the bAbI question answering task [4] (see also Section 7.2) is modified in a way that samples are concatenated instead of processed one-by-one. This adds the additional difficulty, that the model needs to erase entries of previous samples as they are out of their relevance scope. Also, this results in very long sequences and requires a model which is not prone to suffer from exploding or vanishing gradients.

3.7 H-Mem

Limbacher et al. proposed an architecture where a controller consisting of multiple small networks operates a hetero-associative memory subject to Hebbian plasticity [9]. The controller networks are separated into two branches, one for storing data in the memory and one for retrieving data from the memory. The model uses the networks of the store branch to extract a key and a value for each element of an input sequence and stores an association between key and value in memory using a Hebbian learning rule (also see Section 4.1). If the model is presented a query, the recall branch is invoked which extracts a key vector from the query input and retrieves the associated value vector from the memory. To implement associative inference following a path of multiple relations (for example resolving $a \rightarrow c$ via $\{a \rightarrow b, b \rightarrow c\}$), the recall branch can perform multiple read operations in a row. It does that, by re-computing a new key vector after each read operation using the result of the previous read operation. The architecture of the model is shown in figure 3.2.

The model is further improved via the concept of memory-dependent memorization. This extension implements a read operation preceding the

write operation in the store branch to incorporate the current state of the memory into the calculation of the new value. For the read operation, the same key is used as for the write operation, allowing the model to calculate the new value vector with additional information about the current value retrieved by the key. This extension enables the model to join existing stored patterns with a new one, instead of just overwriting the previously stored pattern for a specific key.

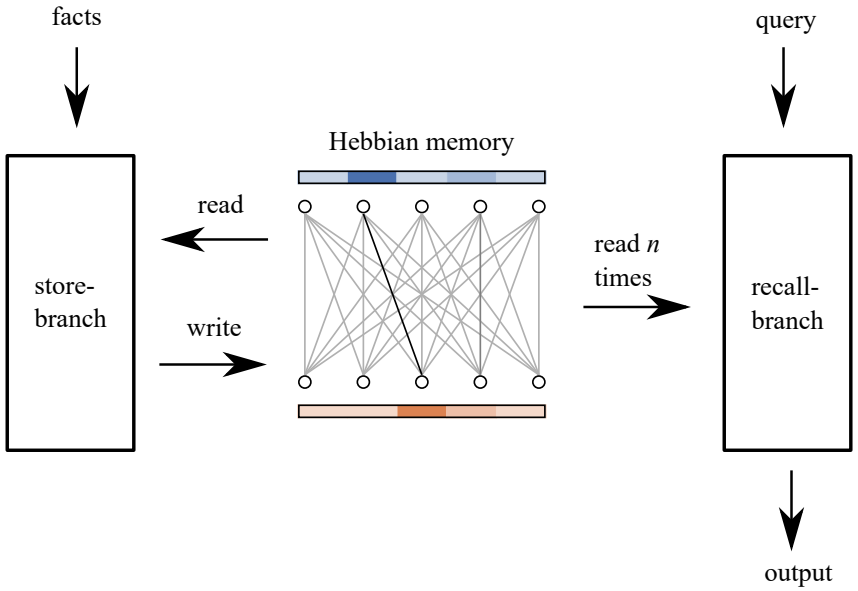


Figure 3.2: The architecture of the H-Mem model with memory-dependent memorization. The input is processed either via the store branch or the recall branch respectively, depending if it is a fact or a query respectively. Before each write operation a read operation is performed using the same key. Adapted from [9].

Despite the simple architecture, the model is capable of solving all of the 20 bAbI [4] tasks in the 10k data-set, delivering state-of-the-art results.

4 Hebbian Plasticity

Even though the learning and memory mechanisms in the brain are not fully understood yet, synaptic plasticity, the change of the weights between neurons, is believed to be the basis for the memory and learning mechanisms in the brain [20]. Donald Hebb [12] proposed a theory about these synaptic weight changes:

> *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells, such that A's efficientcy, as one of the cells firing B, is increased."*

A commonly used simplified version of this postulate is "cells that fire together, wire together". Hebb proposed, that the efficacy of a synaptic connection from one neuron to the other is increased if the activation of the source neuron contributed in the activation of the target neuron. Neuronal plasticity following this principle is referred to as Hebbian plasticity and is assumed to play a major role on how memory in the brain works [60].

4.1 The Hebbian learning rule

The increase in synaptic efficacy subject to Hebbian plasticity can be applied to neural networks by changing the weights between neurons according to rules following the Hebbian learning principle. Gerstner et al. [61] derived a generic mathematical formulation of the Hebbian learning rule as function of the pre- and postsynaptic activity by considering the requirements and constraints it is bound to. The authors describe six aspects which are important for the formulation of a Hebbian learning rule:

- *Locality.* The weight update of a weight w_{ji} from neuron j to neuron i must depend solely on the state of these two neurons as well as the weight between them, but not on the state of other neurons k, this is illustrated in figure 4.1.
- *Cooperativity.* The function must implement Hebb's theory [12] whereas the simultaneous activity of neurons i and j is required for a weight increase of the connection between them.
- *Synaptic Depression.* Constantly increasing the weights without any subtraction term would result in infinite growth of the weights. Therefore, a depression term needs to be included in the learning rule to avoid this issue.
- *Boundedness.* In biological neural networks, the weights cannot be increased indefinitely, hence they should be bounded between 0 and w_{max}, which is the upper bound for weight values.
- *Competition.* The idea of competition is to increase weights at the cost of decreasing others. One example of such a competition term would be Oja's rule [62], which is explained below.
- *Long-term stability.* To avoid simple overwriting of previously stored information, converging the weights to binary values of 0 and 1 can be useful. However, the authors claim that mostly this aspect is disregarded.

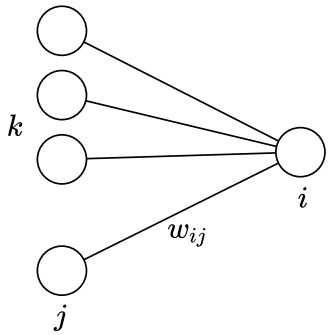


Figure 4.1: The principle of locality for Hebbian weight updates. An update of weight w_{ji} from neuron j to neuron i must solely depend on the state of neurons j and i as well as the current state of w_{ji}, but not on states of neurons k other than i and j. Modified from [61].

To find a general formulation of the Hebbian learning rule under the *locality*

constraint, where the weight update for Δw_{ji} [1] can only be a function $F(w_{ji}, a_i, a_j)$ of locally available information, the authors performed a series expansion at $a_i = a_j = 0$, yielding

$$\begin{aligned}\Delta w_{ji} \approx\ & c_2^{\text{corr}}(w_{ji})a_i a_j + c_2^{\text{post}}(w_{ji})a_i^2 + c_2^{\text{pre}}(w_{ji})a_j^2 \\ & + c_1^{pre}(w_{ji})a_j + c_1^{\text{post}}(w_{ji})a_i + c_0(w_{ji}) \\ & + \mathcal{O}(a).\end{aligned} \tag{4.1}$$

In this expression, a_i and a_j are the activity values of neurons i and j respectively. The coefficients $c_k^{\text{pre}}(w_{ji})$ and $c_k^{\text{post}}(w_{ji})$ with $x \in \{0, 1\}$ are functions of the current weight value w_{ji} and differ between different implementations of the learning rule. The term $c_2^{\text{corr}}(w_{ji})a_i a_j$ implements the basis of the Hebbian learning as product between the activations of the neurons connected by w_{ji} [63]. By choosing all other coefficients to be zero and choosing $c_2^{\text{corr}}(w_{ji}) > 0$ results in

$$\Delta w_{ji} = c_2^{\text{corr}}(w_{ji})a_i a_j, \tag{4.2}$$

which is the most simple implementation of a Hebbian learning rule, implementing the principles of *locality* and *cooperativity*. However, this simple rule is neither *bounded* nor implements *synaptic depression*, potentially resulting in unlimited growth of the weights over time. An approach to introduce *synaptic depression* via the principle of *competition* was proposed by Oja [62] and is referred to as Oja's rule and is denoted by

$$\Delta w_{ji} = \eta_0 a_i a_j - \eta_0 w_{ji} a_i^2 \tag{4.3}$$

with a constant $\eta_0 > 0$. This rule can be derived from 4.1 by setting $c_2^{\text{post}} = -\eta_0 w_{ji}$ and $c_2^{corr} = \eta_0$. The quadratic subtraction term not only assures that the weight decreases if the post-synaptic neuron i is activated without activation of the pre-synaptic neuron j, it also regularizes the magnitude of the weight update due to the term quadratic in a_i. These two properties result in a more stable weight update and a soft bound [63]. Oja's rule has several other properties, not further discussed in this work.

[1] The authors formulate the update rule in terms of the firing rate of spiking networks. Since in this work solely discrete-time operations are considered, the weight update is denoted as Δw_{ji} instead of $\frac{d}{dw} w_{ji}$.

The learning rule used by [9] as well as by HebbLSTM proposed in this work is denoted by

$$\Delta w_{ji} = \eta_0 (w^{\max} - w_{ji}) a_j a_i - \eta_0 w_{ji} a_j^2. \tag{4.4}$$

This rule is similar to Oja's rule [62], except for the additional soft bound term $(w^{\max} - w_{ji})$ to ensure $w_i j \leq 1$ as well as the slightly adapted subtraction term $\eta_0 w_{ji} a_j^2$, where the post-synaptic activity is replaced by the post-synaptic activity. For details on the Hebbian weight update in this work see Chapter 5.

4.2 Hebbian Associative Memory

The Hebbian learning rule can in general be applied to train neural network weights. For conventional neural networks, where the weights are trained once and stay fixed for inference, gradient descent is usually the state-of-the-art choice for a training algorithm [21]. A different application of Hebbian learning is a neural associative memory (NAM). Formally, a NAM is defined as a fully connected single layer network, which maps a set of input patterns $X = \{x_1, ..., x_M\}$ to a set of corresponding output patterns $Y = \{y_1, ..., y_M\}$ in a way that input pattern x_μ is associated with pattern y_μ for $\mu \in \{1, ..., M\}$ [8]. The associations are learned via weight changes using a specific learning rule, for example the Hebbian rule. Thus, association learning is realized using synaptic plasticity [64].

One feature to classify associative memory is the kind of mapping it represents. If the memory maps input pattern x to output pattern y where pattern x is different to pattern y, the memory is referred to as *hetero-associative*. If x and y are equivalent, the memory associates a pattern to itself which is called *auto-associative*. An auto-associative memory can restore the full pattern x by performing a query using a pattern $\tilde{x}$, which must be sufficiently close to x according to a specific metric [8].

Hebbian learning has successfully been applied in machine learning models by combining it with conventional approaches, some examples are already described in Sections 3.6 and 3.7. One further model using Hebbian memory

is proposed by Schmidhuber et al. [18]. The authors trained a feed-forward network to store information for short-term use by changing weights in a separate fast-weight network. This fast-weight memory represents the hetero-associative Hebbian memory and is similar to the Hebbian memory used in the model proposed in this thesis. The authors point out, that the difference between this fast-weight network and the activity-based memory in conventional RNNs is, that the fast-weight network is naturally able to store connections between variables, which is an important feature of associative memories. Therefore, associative memories are helpful for tasks where a temporary variable binding is involved.

5 Hebbian Long Short-Term Memory Networks

Long short-term memory (LSTM) networks [13] (see Section 2.1.2) are shown to solve a variety of tasks that require the extraction and temporal storage of information in sequential data. By using a gated memory unit and feedback connections it can solve tasks involving sequences and time-dependent information. Furthermore it provides a viable solution to the problem of vanishing gradients, therefore it can handle long input sequences and deal with long time lags [13]. However, a study by [9] indicates that the more long-term nature of Hebbian synaptic plasticity [12] is superior to the activity-based memory of LSTM networks in many tasks. The idea behind Hebbian long short-term memory (HebbLSTM) networks is to combine the capabilities of LSTM networks and Hebbian plasticity.

HebbLSTM is a LSTM-like network that is augmented with a form of Hebbian memory (see Figure 5.1). This memory is implemented as a single-layer hetero-associative neural network subject to Hebbian plasticity. With such a form of memory, the HebbLSTM network can learn to extract and store associative information from its input and use this input to solve tasks that require the resolution of such associations, while maintaining the benefits of LSTM networks.

HebbLSTM is a recurrent neural network and hence can, in principle, be applied to input sequences of arbitrary length. It can furthermore be integrated into larger RNN architectures. Since the model uses a gated architecture like LSTM does, it is assumed that it does not suffer from the vanishing gradient problem.

The hetero-associative memory module implemented in HebbLSTM is capable of storing associations between input- and output pattern pairs. These

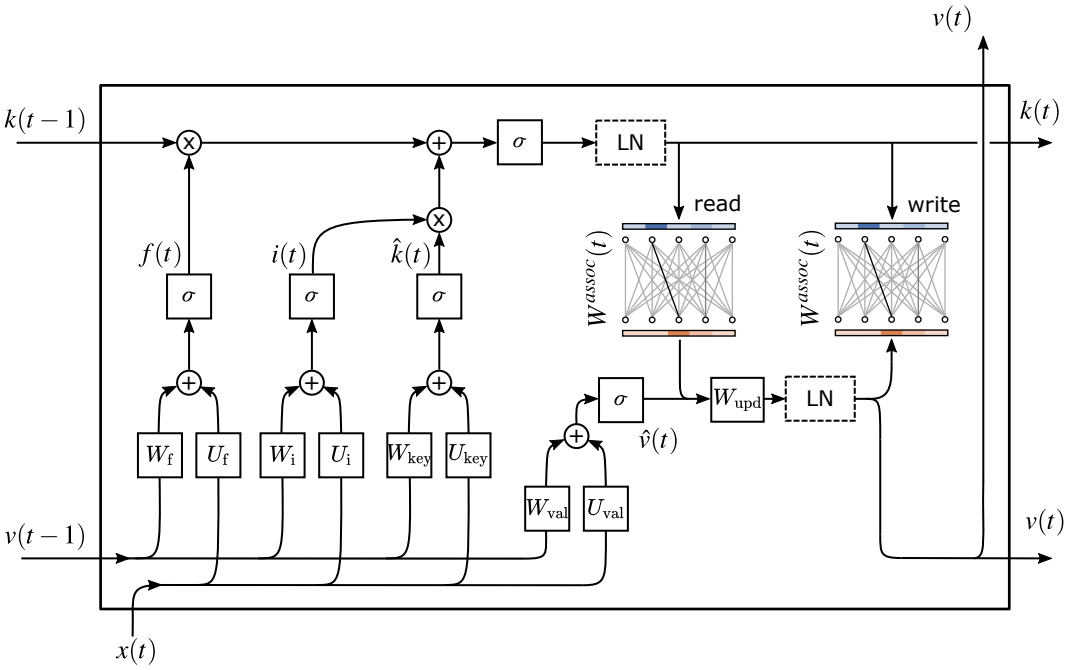


Figure 5.1: Schema of the HebbLSTM architecture. At each time-step t, the cell is presented with input-vector $\boldsymbol{x}(t)$, key-vector $\boldsymbol{k}(t-1)$ and value-vector $\boldsymbol{v}(t-1)$ of the previous time-step t. Dense layers (marked with U and W with corresponding indices) followed by an activation function σ are used to compute a forget gate vector $\boldsymbol{f}(t)$, an input gate vector $\boldsymbol{i}(t)$ and a preliminary new key-vector $\hat{\boldsymbol{k}}(t)$. This vector $\hat{\boldsymbol{k}}(t)$ is then summed with the previous key-vector $\boldsymbol{k}(t-1)$ (which is beforehand multiplied by the input gate vector $\boldsymbol{i}(t)$) to calculate the final key-vector $\boldsymbol{k}(t)$. This key-vector is then used to perform a read operation on the Hebbian memory matrix $W^{\text{assoc}}(t)$. The result of this read operation is then concatenated with a preliminary value-vector $\hat{\boldsymbol{v}}(t)$ to calculate the final value-vector $\boldsymbol{v}(t)$ by using a weight W_{upd}. Finally, in the store step, the Hebbian matrix is updated according to the Hebbian plasticity rule using $\boldsymbol{k}(t)$ and $\boldsymbol{v}(t)$.

patterns are represented as vectors. In order to store a task-relevant association from one vector to another, the model needs to compute these two vectors from a given input. The first vector is the pre-synaptic vector and acts as key and the second one is the post-synaptic vector which is referred to as value. The Hebbian memory maps the pre-synaptic key-vector to the post-synaptic value-vector, such that (under certain conditions) the post-synaptic value-vector can be retrieved by querying the memory with the pre-synaptic key-vector. To compute the key-vector and value-vector

from the input, the model uses a combination of fully-connected weight layers and gates. This section explains the computation steps in detail.

At time-step t, an input vector $\boldsymbol{x}(t) = [x_1(t), x_2(t), \ldots, x_n(t)]$ of dimension n is shown to the network. The network computes two vectors, a key-vector $\boldsymbol{k}(t) \in \mathbb{R}^m$ and a value-vector $\boldsymbol{v}(t) \in \mathbb{R}^m$ from input $\boldsymbol{x}(t)$ at time-step t. The network learns that it can establish an association between key-vector $\boldsymbol{k}(t)$ and value-vector $\boldsymbol{v}(t)$ in memory, by using a Hebbian plasticity rule, and to retrieve the associated value-vector later by querying the memory with a vector similar to $\boldsymbol{k}(t)$.

The following paragraphs explain the computational steps that are performed to achieve this previously described extraction of two vectors from the input.

An input gate vector $\boldsymbol{i}(t) \in \mathbb{R}^m$ and a forget gate vector $\boldsymbol{f}(t) \in \mathbb{R}^m$, similar to a LSTM-architecture, are computed using

$$\boldsymbol{f}(t) = \sigma_g(W^f \boldsymbol{x}(t) + U^f \boldsymbol{v}(t-1)) \tag{5.1}$$

$$\boldsymbol{i}(t) = \sigma_g(W^i \boldsymbol{x}(t) + U^i \boldsymbol{v}(t-1)). \tag{5.2}$$

where σ_g is a non-linear activation function, $W^f \in \mathbb{R}^{m \times n}$, $U^f \in \mathbb{R}^{m \times n}$, $W^i \in \mathbb{R}^{m \times n}$ and $U^i \in \mathbb{R}^{m \times n}$ and $\mathbf{v}(t-1)$ is the value-vector of the previous time-step (the initial value-vector $\mathbf{v}(0)$ is set to the zero-vector).

The computation of key-vector $\boldsymbol{k}(t)$ and value-vector $\boldsymbol{v}(t)$ incorporates two steps. The first step is to calculate preliminary key- and value-vectors $\hat{\boldsymbol{k}}(t)$ and $\hat{\boldsymbol{v}}(t)$ using two weight matrices $W^{\text{key}} \in \mathbb{R}^{m \times n}$ and $U^{\text{key}} \in \mathbb{R}^{m \times n}$ respective $W^{\text{val}} \in \mathbb{R}^{m \times n}$ and $U^{\text{val}} \in \mathbb{R}^{m \times n}$ for each of the vectors. More specifically, vectors $\hat{\boldsymbol{k}}(t)$ and $\hat{\boldsymbol{v}}(t)$ are computed according to

$$\hat{\boldsymbol{k}}(t) = \sigma_g(W^{\text{key}} \boldsymbol{x}(t) + U^{\text{key}} \boldsymbol{v}(t-1)) \tag{5.3}$$

$$\hat{\boldsymbol{v}}(t) = \sigma_g(W^{\text{val}} \boldsymbol{x}(t) + U^{\text{val}} \boldsymbol{v}(t-1)). \tag{5.4}$$

For the computation of the key-vector $\boldsymbol{k}(t)$, input gate vector $\boldsymbol{i}(t)$ and forget gate vector $\boldsymbol{f}(t)$ are used to calculate which elements of the previous key-vector, that is $\mathbf{k}(t-1)$, and which elements of the current preliminary

key-vector $\hat{\boldsymbol{k}}(t)$ are summed:

$$\boldsymbol{k}(t) = \sigma_{\mathrm{u}}(\boldsymbol{f}(t) \circ \boldsymbol{k}(t-1) + \boldsymbol{i}(t) \circ \hat{\boldsymbol{k}}(t)), \tag{5.5}$$

where σ_{u} is a non-linear activation function and where $\circ$ denotes the Hadamard product. This gated update is similar to the one used in LSTM, which helps to avoid the vanishing gradient problem.

For the computation of the value-vector $\boldsymbol{v}(t)$, the model firstly performs a read operation on the memory module, which is a matrix W^{assoc} of size $m \times m$, by using the current key-vector $\boldsymbol{k}(t)$. It is assumed that this read step enhances the capabilities of dealing with redundant or correlating information in the memory matrix. More specifically, the value-vector $\boldsymbol{v}(t)$ is computed according to:

$$\boldsymbol{v}(t) = \sigma_{\mathrm{u}}(W^{\mathrm{upd}}(\hat{\boldsymbol{v}}(t)^{\intercal}, (W^{\mathrm{assoc}}(t)\boldsymbol{k}(t))^{\intercal})^{\intercal}), \tag{5.6}$$

where σ_{u} is a non-linear activation function and $W^{\mathrm{upd}} \in \mathbb{R}^{m \times n}$ is a weight matrix. The matrix $W^{\mathrm{assoc}}(0)$ is initialized with zeros.

Matrices W^{f}, U^{f}, W^{i}, U^{i}, W^{key}, U^{key}, W^{val}, U^{val} and W^{upd} are optimized during training. Note, that W^{assoc} is not optimized since it is dynamic during inference. This matrix is instead subject to Hebbian plasticity. Weight updates of this matrix are given by

$$\Delta W_{ij}^{\mathrm{assoc}}(t) = \lambda(w^{\mathrm{max}} - W_{ij}^{\mathrm{assoc}}(t))\boldsymbol{k}_j(t)\boldsymbol{v}_i(t) - \lambda W_{ij}^{\mathrm{assoc}}(t)\boldsymbol{k}_j(t)^2, \tag{5.7}$$

where $W_{ij}^{\mathrm{assoc}}(t)$, with $j = 0, ..., m$, $i = 0, ..., m$, represents the weight from the j-th input neuron to the i-th output neuron in the Hebbian memory network at time-step t. The constant λ is an update parameter and w^{max} is a constant representing the maximum weight. $v_i(t)$ and $k_j(t)$ are the i-th and j-th element of value-vector $\boldsymbol{v}(t)$ and key-vector $\boldsymbol{k}(t)$ respectively.

The term $w^{\mathrm{max}} - W^{\mathrm{assoc}_{ij}(t)}(t)$ works as a soft-bound to avoid growth of memory entries above a certain threshold w^{max}.

The strengthening of the weights is performed by the term

$$k_j(t)v_i(t) \tag{5.8}$$

which is, according to [61], the basis of a Hebbian learning rule. This term implements the hypothesis of Hebb et al. [12] in which weights between neurons that fire together are increased. Equation 5.8 only takes a high value if both of the elements $k_j(t)$ and $v_i(t)$ and therefore neurons j and i have a high value, which corresponds to firing in this implementation of neural networks.

The term $-\lambda W_{ij}^{\text{assoc}}(t)\boldsymbol{k}_j(t)^2$ is used to weaken the pre-synaptic activity of a given key $\boldsymbol{k}(t)$. Hence, this subtraction term weakens the weights that are activated by this specific key. More specific, if the element $k_j(t)$ representing pre-synaptic neuron j is of high value, all outgoing weights from this element are weakened. This term is needed because the Hebbian learning term in Equation 5.8 only increases weights but never weakens them (under the assumption that a neurons activation function cannot take negative values). This means, that without a reduction term, all of the weights would converge to $w^{\max}$. The idea is to forget values which were previously assigned to a key similar to $\boldsymbol{k}(t)$.

The association weight matrix is then updated according to

$$W^{\text{assoc}}(t+1) = W^{\text{assoc}}(t) + \Delta W^{\text{assoc}}(t). \tag{5.9}$$

6 Hebbian Long Short-Term Memory Networks with Hidden Value

In the HebbLSTM model three potential drawbacks were identified:

1. The first drawback is that the model is constrained to pass only the key and the value of the last write operation to the next time-step. However, since the key-value pair of the last time-step was already stored in the memory, they might not provide useful information for the next time-step at all.
2. In a vanilla LSTM cell [13] the hidden value can be passed through multiple time-steps without performing any operation on it. In contrast, if the HebbLSTM model would pass the same key through multiple time-steps (for example to maintain a state of the cell outside of the Hebbian memory), a write operation with this key would be performed at each time-step. Therefore, the model is not capable of maintaining an internal state outside the Hebbian memory without directly influencing the write operation of the Hebbian memory.
3. LSTM is proven to solve the vanishing gradient problem [13]. Even though there is no proof performed proving that HebbLSTM is not prone to the vanishing gradient problem, the fact that it cannot pass a constant value along multiple time-steps without direct influence on the computation might be an indicator that the constant error carousel of LSTM (for details see Section 2.1.2) is not fulfilled anymore.

To address these three issues, a second architecture called Hebbian Long Short-Term Memory with Hidden Value (H-HebbLSTM) is proposed. The main difference is that H-HebbLSTM maintains a hidden state in addition

to the Hebbian memory to enable the model to perform context-dependent read and write operations.

Figure 6.1, illustrates the architecture of H-HebbLSTM. The model contains several other differences to the HebbLSTM model.

1. Instead of using the same key-vector for the read and write steps, a separate key-vector is calculated for any interaction with the memory.
2. The model interacts three times with the Hebbian memory. The first interaction is a read operation to read out previously stored information using a preliminary key-vector. The second interaction is a write operation, where a new key-value-pair is written to the memory. The third interaction is only relevant for the downstream connections and is a final read operation, before the output value is calculated. Via this third interaction, the model can learn different weights for reading out solution-relevant information, whereas the first read operation does not necessarily yield information which is relevant for the final answer, but rather for the writing step.
3. The model can choose to not change the hidden value at all, by setting the input gate to 0 and the forget gate to 1. It is assumed, that this feature is needed to maintain an internal state over multiple time-steps, for example if the model needs to "remember" the current operation mode of the task (query or fact).

The calculation of the key-vectors, value-vectors, the hidden vector and the output-vector is explained in the next few paragraphs.

The input to the H-HebbLSTM memory cell is the input-vector $\boldsymbol{x}(t)$ of the current time-step t and the hidden value-vector $\boldsymbol{h}(t-1)$ from the previous time-step $t-1$. The first hidden vector $\boldsymbol{h}(0)$ at time-step $t=0$ is set to the zero-vector. For each time-step t, the first conducted computation is a key-vector $\hat{\boldsymbol{k}}(t)$ for the initial read-operation on the memory. $\hat{\boldsymbol{k}}(t)$ is calculated via

$$\hat{\boldsymbol{k}} = \sigma_g(W^{\hat{\boldsymbol{k}}}[\boldsymbol{h}(t-1); \boldsymbol{x}(t)]) \tag{6.1}$$

where $W^{\hat{\boldsymbol{k}}} \in \mathbb{R}^{m \times (h+d)}$ denotes a trained weight matrix and σ_g an activation function. $\boldsymbol{x}(t)$ is the input-vector of the cell at time-step t and $\boldsymbol{h}(t-1)$ the hidden vector of time-step $t-1$. The dimension m denotes the number of

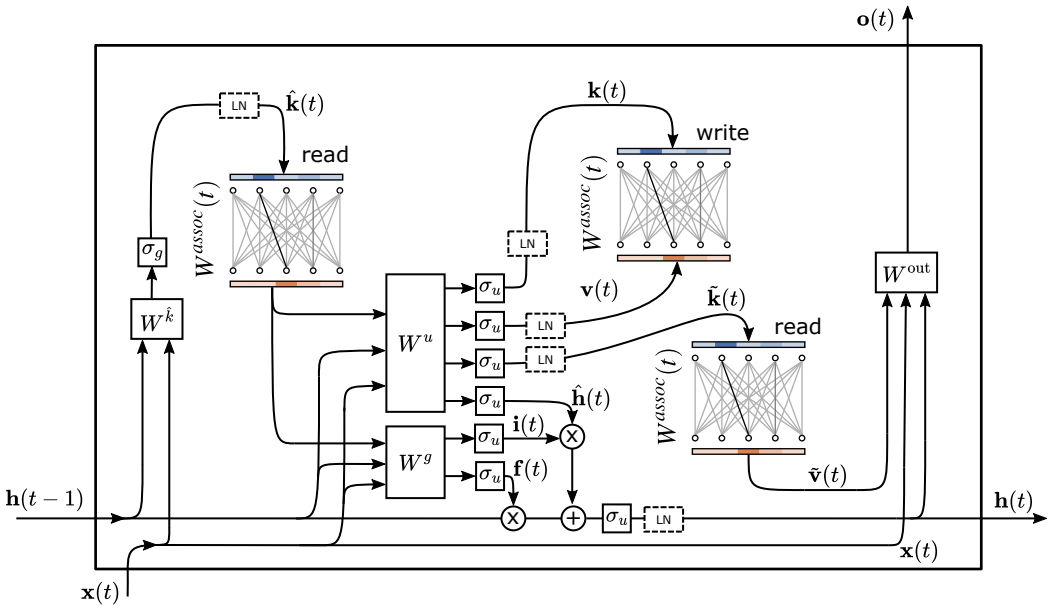


Figure 6.1: Schema of the H-HebbLSTM architecture. Each time-step t, three memory operations are performed. A preceding read-out using preliminary key-vector $\hat{k}(t)$ followed by a write-in using key-vector $\boldsymbol{k}(t)$ and value-vector $v(t)$ and a final read-out with key-vector $\tilde{\boldsymbol{k}}(t)$ used for the cell output $\boldsymbol{o}(t)$. The squares with the label "LN" denote a layer normalization. Detailed equations can be found in the text.

memory units in the Hebbian memory, h the dimension of the hidden vector and d the input dimension. The notation $[\boldsymbol{a};\boldsymbol{b}] \in \mathbb{R}^{(d_a+d_b)}$ corresponds to the concatenation of two vectors $\boldsymbol{a} \in \mathbb{R}^{d_a}$ and $\boldsymbol{b} \in \mathbb{R}^{d_b}$ and is equivalent to $(a^\intercal, b^\intercal)^\intercal$.

The preliminary key-vector $\hat{\boldsymbol{k}}(t)$ is used to perform a read operation on the Hebbian memory matrix $W^{\text{assoc}}(t) \in \mathbb{R}^{m\times m}$ to calculate a preliminary value-vector $\hat{\boldsymbol{v}}(t)$ via $\hat{\boldsymbol{v}}(t) = W^{\text{assoc}}(t)\boldsymbol{k}(t)$.

After this memory-read operation, the calculation

$$[\boldsymbol{k}(t);\boldsymbol{v}(t);\tilde{\boldsymbol{k}}(t);\hat{\boldsymbol{h}}(t)] = \sigma_u(W^u[\hat{\boldsymbol{v}}(t);\boldsymbol{h}(t-1);\boldsymbol{x}(t)]) \tag{6.2}$$

of key-vector $\boldsymbol{k}(t) \in \mathbb{R}^m$, value-vector $\boldsymbol{v}(t) \in \mathbb{R}^m$, read-out key-vector $\tilde{\boldsymbol{k}}(t) \in \mathbb{R}^m$ and preliminary hidden vector $\hat{\boldsymbol{h}}(t) \in \mathbb{R}^h$ is performed using a

weight matrix $W^u \in \mathbb{R}^{4m \times (m+h+d)}$.

The vectors $\boldsymbol{k}(t)$ and $\boldsymbol{v}(t)$ are then used to update the Hebbian memory matrix $W^{\text{assoc}}(t)$ according to Equation 5.7.

Using the read-out key-vector $\tilde{\boldsymbol{k}}(t)$, a second read operation is performed on the Hebbian memory matrix $W^{\text{assoc}}(t)$ via $\tilde{\boldsymbol{v}}(t) = W^{\text{assoc}}(t)\tilde{\boldsymbol{k}}(t)$ before the cell output $\boldsymbol{o}(t)$ is calculated. The equation for this output $\boldsymbol{o}(t)$ is then performed by

$$\boldsymbol{o}(t) = W^{\text{out}}[\boldsymbol{h}(t); \boldsymbol{x}(t); \tilde{\boldsymbol{v}}(t)]. \tag{6.3}$$

Via the equation

$$[\boldsymbol{i}(t); \boldsymbol{f}(t)] = \sigma_g(W^g[\hat{\boldsymbol{v}}(t); \boldsymbol{h}(t-1); \boldsymbol{x}(t)]), \tag{6.4}$$

the forget-gate vector $\boldsymbol{f}(t) \in \mathbb{R}^h$ and input-gate vector $\boldsymbol{i}(t) \in \mathbb{R}^h$ are calculated. $W^g \in \mathbb{R}^{2h \times (m+h+d)}$ and $W^{out} \in \mathbb{R}^{d \times (m+h+d)}$ denote trained weight matrices.

These two vectors are used in

$$\boldsymbol{h}(t+1) = \sigma_u(\boldsymbol{h}(t) \circ \boldsymbol{f}(t) + \hat{\boldsymbol{h}}(t) \circ \boldsymbol{i}(t)) \tag{6.5}$$

for the calculation of the new hidden vector $\boldsymbol{h}(t)$. The symbol $\circ$ denotes the Hadamard product.

7 Results

This section explains the conducted experiments and shows the performance of HebbLSTM and H-HebbLSTM. The experiments were chosen in order to test the associative and memory-dependent capabilities of the network.

7.1 Dictionary Learning

This task is inspired by the synthetic dictionary inference task from Munkhdalai et al. [6]. Each sample instance of this task is divided into two parts, the facts and the query. The facts are groups of letter-pairs, where each pair represents a directed mapping from one letter to the other. The separator for the source- and target-letters is "¿", the separator between facts is the semicolon. For example, the fact "*tlk ¿ axs;*" contains three mappings, t to a, l to x and k to s. After the facts, the separator "#" denotes the begin of the query, which is a string of letters to be translated by the algorithm, by applying all of the previously learned translation rules. If a letter does not appear on the left side of the separator in any fact, it must stay as it is in the query. If for example the fact from before is the only fact of a task instance, the model is expected to translate a query *rfknt* into *rfsna*.

The facts are chosen in a way, that each letter pair appears only once throughout all facts, to ensure that the model learns the mappings in one shot. Each task instance consists of k facts, where each fact contains l letter-pairs followed by a query of length q. The target value for training is the correctly translated sequence, also of length q. Note, that the larger l is chosen, the further away is the source-letter from its target-letter, because each fact is constructed in a way that at first, all of the source letters are presented, then all the target letters, with the separator in between. The

Table 7.1: Data sample of the dictionary learning task with $k = 6$, $l = 2$ and $q = 10$.

Facts	sb>hf;qr>nx;hy>wc;ei>ax;ld>sx;oz>dg;#
Query	ywfbtcxwzd
Answer	cwfftcxwgx

Table 7.2: Results for different combinations of fact count k, fact length l and sequence length q for H-HebbLSTM compared to a LSTM baseline model in [% test accuracy]. The table shows the results with the lowest validation error out of three separate runs using different random seeds.

k	l	q	LSTM	H-HebbLSTM
26	1	10	15.8	100
2	1	100	92.4	100
2	13	100	7.5	23,1
6	2	10	61.0	100
6	4	50	17.9	100

model must store all of the intermediate source-letters in order to assign them to their corresponding target-letters as soon as the separator symbol is presented.

For this task, several different combinations of the parameters k, l and q were chosen according to Table 7.2. The data was encoded using a dictionary encoding, where firstly all available letters are collected in a vocabulary list and secondly all letters in the task instances are replaced by the corresponding integer index within the vocabulary. Then, each letter was embedded into continuous multidimensional fixed-size vector of dimension $d = 30$. Each embedded vector represents the input of one time-step, hence the network receives one symbol per time-step. During the presentation of the facts, the output of the networks was ignored, whereas during the presentation of the query, the translated letter for the presented letter of the current time-step was expected and set as target.

H-HebbLSTM is capable of solving most of the settings, a higher fact length

however increases the difficulty, because firstly the corresponding inputs are further away from each other and secondly, the model must store more information intermediately. The network outperformed LSTM in each tested setting.

Training and Hyperparameters The optimizer to minimize the sparse categorical cross-entropy was **ADAM** [56] and the networks were trained for up to 300 epochs, but the training was stopped if a validation accuracy of 100% was achieved. The initial value of the **learning rate** was 0.003 and it was decayed by the factor 0.85 every 50 training epochs. For all weights within the models, a L2 regularizer with parameter l of 10^{-3} was used. For weight initialization of all weights within the H-HebbLSTM and LSTM cells the **Glorot normal initializer** [65] was applied. For the embedding layer, the **He uniform initializer** [66] was used. The final output of the models was mapped onto the target feature space by a linear fully connected layer followed by a softmax. The number of **units** for the **Hebbian memory** was 90 (therefore the dimension m of the memory matrix was 90×90). For the **embedding** of the input a dimension d of 30 was used. The initial key vector $\boldsymbol{k}(0)$ and value vector $\boldsymbol{v}(0)$ were zero-vectors and the memory matrix W^{assoc} was initilized with zeros for each sample. The model was trained on a training set of $8,100$ instances and validated on a validation set of 900 instances. The test set consisted of 1000 instances. For the LSTM cell, the tanh activation function was used for the input and output and the logistic sigmoid function for the gates. The forget gate bias was initialized with ones. For H-HebbLSTM, the logistic sigmoid function was used as activation function σ_g for the calculation of the gate vectors i and f, the rectified linear unit activation function was used for all other calculations as function σ_u.

7.2 Question Answering

Understanding stories and answering questions about them requires rapid and flexible memory on long time scales. To see if HebbLSTM can learn to answer questions about stories, the bAbI data set has been chosen. This data set was proposed by Weston et al. [4] and consists of a collection of 20

Table 7.3: Three example instances from different tasks of the bAbI data set. All of the three examples require different capabilities of the model. For example, task 2 requires resolving of chained statements. The bAbI data set consists of 20 different tasks.

Task 2: Two Supporting Facts	Task 8: Lists/Sets	Task 19: Path Finding
John is in the playground.	Daniel picks up the football.	The kitchen is north of the hallway.
John picked up the football.	Daniel drops the newspaper.	The bathroom is west of the bedroom.
Bob went to the kitchen.	Daniel picks up the milk.	The den is east of the hallway.
Q: Where is the football?	John took the apple.	The office is south of the bedroom.
A: playground	Q: What is Daniel holding?	Q: How do you go from the den to the kitchen?
	A: milk, football	A: west, north

question-answering tasks. The tasks are designed to test different skill sets which the authors describe as neccessary for conversing with humans. This skill set includes for example the chaining of facts, deduction, induction and some more (see Table 7.3 for examples of this data set). The model has to extract relevant information out of each sentence, then store and later retrieve this information to answer a question. At test time, stories are shown only once to the network, hence it must extract and store all relevant information at once.

The data set consists of two sub sets, one with 1k training examples and one with 10k. For this experiment, the 10k data set was chosen. Each task instance consists of a sequence of sentences $\langle x_1, ..., x_T \rangle$ with $T < 321$ where the last sentence is a question. The answer to the question is typically a single word (in some tasks the answer consists of multiple words, which were, however, interpreted as one word of the vocabulary).

To represent the tasks in a machine-readable manner, the sentences x_t were encoded into a one-hot encoding of dimension V where V is the vocabulary size of the task. The one-hot encoded sentences were passed to an encoding layer implementing a learned encoding given by $\boldsymbol{e}_t = \sum_j \boldsymbol{f}_j \circ A\boldsymbol{w}_{t,j}$, which maps the sequence of words $\langle \boldsymbol{w}_{1,t}, \boldsymbol{w}_{2,t}, ..., \boldsymbol{w}_{J,t} \rangle$ of a given sentence x_t with J words to a continuous vector $\boldsymbol{e}_t$ of embedding dimension d. The matrix $A \in \mathbb{R}^{d \times V}$ was optimized during training. Encoding vectors $\boldsymbol{f}_j$ were also optimized during training and shared between all sentences. The model can therefore learn a representation where all the necessary information, for example the positioning of each word within a sentence, is preserved. In order to enable the model to capture the temporal context of a task, a temporal encoding for sentences as introduced in [5] was

used. This encoding introduces a special matrix T_A that encodes temporal information. The modified sentence representation is then given by $\boldsymbol{e}_t = \sum_j \boldsymbol{f}_j \circ A\boldsymbol{w}_{t,j} + \text{row}_t(T_A)$

Table 7.4 provides an overview of the test results of HebbLSTM and H-HebbLSTM compared with LSTM [13] and H-Mem [9] on the 10k data set of the 20 bAbI tasks. All of the models were tested on a test set of 1,000 task instances. HebbLSTM solved 14 out of 20 tasks, H-HebbLSTM solved all tasks. A task instance is considered as solved, if the output of the model matches the expected answer. A task is considered as solved, if more than 95% of the instances of this task are solved. The test error with the lowest validation error out of 3 runs is shown in the table.

The proposed model HebbLSTM outperforms LSTM on the 10k bAbI task data-set. By failing 16 tasks, LSTM does not provide the necessary capabilities for solving this kind of associative tasks. In comparison, H-Mem and H-HebbLSTM can solve all of the 20 bAbI tasks by using the Hebbian memory module. The results of these models on the 20 bAbI task show, that associative memory can enhance the performance of RNNs on a set of association-based tasks compared to common LSTM networks.

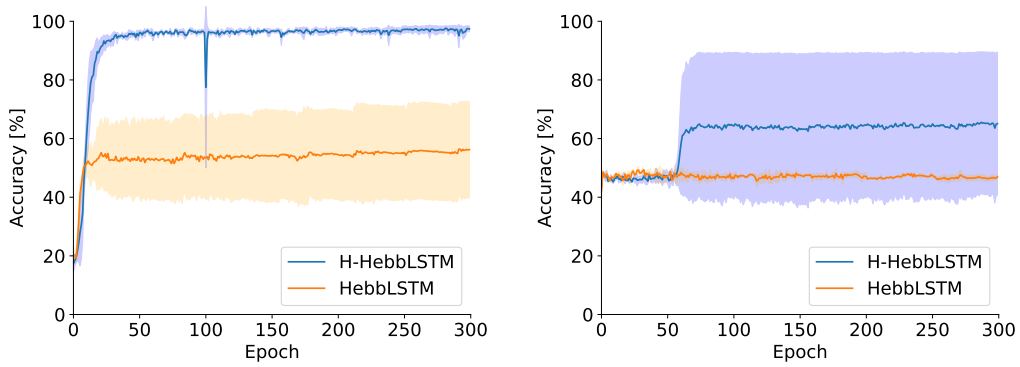
Figure 7.1 shows the learning curve during training of task 3 (three supporting facts) and task 16 (basic induction).

Training and Hyperparameters The optimizer to minimize the sparse categorical cross-entropy was **ADAM** [56] and the networks were trained for up to 300 epochs, but the training was stopped if a validation accuracy of $> 98\%$ was achieved for 10 consecutive training epochs. The initial value of the **learning rate** was 0.003 and it was decayed by the factor 0.85 every 50 training epochs. For all weights within the models, a L2 regularizer with parameter l of 10^{-3} was used. For weight initialization of all weights within the HebbLSTM and H-HebbLSTM cells the **Glorot normal initializer** [65] was applied. For the embedding layer, the **He uniform initializer** [66] was used. The final output of the models was mapped onto the target feature space by a linear fully connected layer followed by a softmax. The number of **units** for the **Hebbian memory** was 90 (therefore the dimension m of the memory matrix was 90×90). For the **embedding** of the input a dimension

Table 7.4: Comparison of the error on the test set of different models on the 10k bAbI tasks data set. The first model is an LSTM model of [4]. The second model is H-Mem, proposed in [9]. The third and fourth models are HebbLSTM and H-HebbLSTM, proposed in this thesis. For H-Mem, HebbLSTM and H-HebbLSTM, the test error of the run with the lowest validation error out of 3 runs is printed.

Task	LSTM	H-Mem	HebbLSTM	H-HebbLSTM
1: Single Supporting Fact	0.0	0.0	0.0	0.0
2: Two Supporting Facts	81.9	0.0	9.9	0.6
3: Three Supporting Facts	83.1	3.7	18.4	1.8
4: Two Arg. Relations	0.2	0.0	0.0	0.0
5: Three Arg. Relations	1.2	0.3	0.6	0.6
6: Yes/No Questions	51.8	1.0	0.4	1.7
7: Counting	24.9	0.2	0.5	1.1
8: Lists/Sets	34.1	0.0	0.0	0.1
9: Simple Negation	20.2	0.1	0.2	0.0
10: Indenite Knowledge	30.1	1.5	0.4	0.0
11: Basic Coreference	10.3	0.0	0.0	0.0
12: Conjunction	23.4	0.0	0.0	0.0
13: Compound Coref.	6.1	0.0	0.0	0.0
14: Time Reasoning	81.0	0.4	8.1	1.1
15: Basic Deduction	78.7	0.0	0.0	0.0
16: Basic Induction	51.9	0.3	50.4	0.6
17: Positional Reasoning	50.1	0.0	0.0	4.7
18: Size Reasoning	6.8	0.1	0.3	1.9
19: Path Finding	90.3	4.7	39.9	1.9
20: Agent's Motivations	2.1	0.0	0.0	0.0
Mean Error	36.4	0.6	6.5	0.8
Failed Tasks (err. $>$ 5%)	16	0	5	0

d of 80 was used. The initial key vector $\boldsymbol{k}(0)$ and value vector $\boldsymbol{v}(0)$ were zero-vectors and the memory matrix W^{assoc} was initilized with zeros for each sample. For each of the 20 tasks, the model was trained on a training set of $9,000$ task instances and validated on a validation set of $1,000$ instances. The dimension of the hidden value of the H-HebbLSTM model was 90, for all activation functions the rectified linear unit (ReLU) was used.



(a) Learning curve of task 3 (three supporting facts)

(b) Learning curve of task 16 (basic induction)

Figure 7.1: Learning curves of the two variants HebbLSTM and H-HebbLSTM on two out of the 20 bAbI question answering tasks using the 10k data-set. The graph shows the average validation accuracy over 3 independent runs with different random seeds, the shaded area shows the standard deviation of the validation accuracy between the runs. HebbLSTM could not solve any of the two tasks, whereas H-HebbLSTM solves task 3 consistently. In task 16, different random seeds lead to very different results.

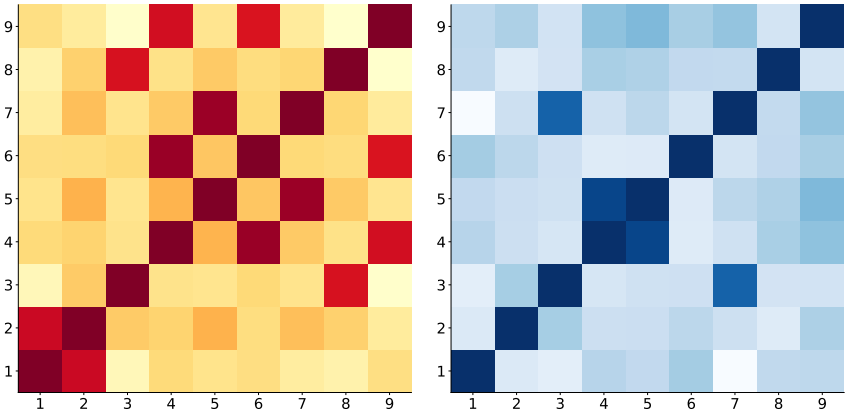
7.3 Memory Analysis for bAbI Tasks

This section provides some insights of how the model H-HebbLSTM uses the Hebbian memory to solve the bAbI tasks [4]. Table 7.5 shows one example of task 1 (single supporting fact) of the bAbI tasks [4]. In Figure 7.2, the key $\boldsymbol{k}(t)$ (in (a)) and value $\boldsymbol{v}(t)$ (in (b)) of the memory update step for each time-step t (axis labels) are compared with each other using the cosine similarity.

This task does not require the chaining of multiple facts, since the answer can be derived from one of the 8 input facts (in this particular example statement 6). The model needs to store each fact in a key-value like fashion. By comparing which keys and values are similar to each other, it can be assumed what the model encodes with each key and value. In the example of Table 7.5 and Figure 7.2, one can see from (a) that the keys of fact 4 and 6 show a high similarity. In fact, both of the statements (4 and 6) contain information about the location of "daniel". The same result can be seen in (b), where the value vectors $\boldsymbol{v}(t)$ for each time-step t are compared. Value

Table 7.5: Example instance of task 1 (One Supporting Fact).

```
1.  sandra went to the office.
2.  sandra travelled to the bathroom.
3.  mary went back to the bedroom.
4.  daniel moved to the garden.
5.  john journeyed to the garden.
6.  daniel went back to the hallway.
7.  john journeyed to the bedroom.
8.  mary travelled to the kitchen.
9.  (Query) where is daniel?
    Correct answer: hallway
```



(a) Similarity matrix of key vectors $\boldsymbol{k}(t)$. (b) Similarity matrix of value vectors $\boldsymbol{v}(t)$

Figure 7.2: Similarity matrices for key-vectors $\boldsymbol{k}(t)$ and value-vectors $\boldsymbol{v}(t)$. The task instance is described in Table 7.5. The matrices show the cosine similarity between vectors of different time-steps t. Darker areas denote higher similarity.

vectors of time-steps 3 and 7 show a high similarity, which means that the model probably encodes the location ”bedroom” in these values.

Since no statements are chained in task 1 it is considered very easy compared to other tasks. In contrast, task 16 (basic induction) turned out to be very challenging for the model, because it takes more iterations to solve than the average of the other tasks, and does not converge for 2 out of 3 runs within the predefined number of epochs. One example of an instance of task 16 is

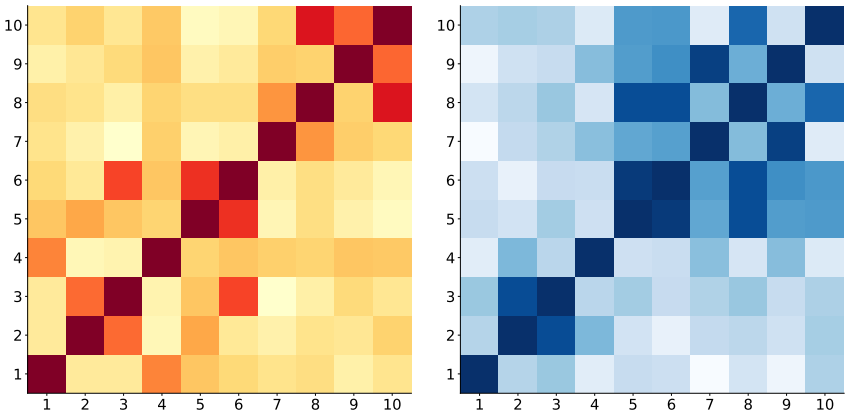
shown in Table 7.6 and Figure 7.3.

Table 7.6: Example instance of task 16 (Basic Induction).

```
1.  lily is a rhino.
2.  brian is a swan.
3.  bernhard is a swan.
4.  lily is gray.
5.  brian is white.
6.  bernhard is white.
7.  julius is a frog.
8.  julius is white.
9.  greg is a frog.
10. (Query) what color is greg?
    Correct answer: white
```

When having a look at Table 7.6, one can see that firstly, three statements need to be chained in order to solve the task and secondly, that it is not clear anymore what key-value association needs to be stored for each fact. To solve the question, the relations ”greg” $\rightarrow$ ”frog” and ”frog” $\rightarrow$ ”white” need to be resolved in order to be able to derive the answer ”white”. Therefore it is not sufficient that ”frog” is either a key or a value, it needs to be both, resulting in a mix-up of keys and values in this task. The mixture of keys and values can also be seen when having a look at the similarity matrices in Figure 7.3. In (a), the similarity between keys of facts 3 and 6 indicates that ”bernhard” is encoded in both of the keys. However, since the keys of facts 5 and 6 also show relatively high similarity, it is assumed that these key-vectors also encode ”white”, since it is the only word which these two facts have in common.

In conclusion, it is obvious that the model solves the samples of task 16 by some sort of mix-up between different keys and values, but the exact processes of how the inference is performed in this task is very hard to understand. Nevertheless, the model is capable of solving this task (see Table 7.4).



(a) Similarity matrix of key vectors $\boldsymbol{k}(t)$. (b) Similarity matrix of value vectors $\boldsymbol{v}(t)$.

Figure 7.3: Similarity matrices for key-vectors $\boldsymbol{k}(t)$ and value-vectors $\boldsymbol{v}(t)$. The task instance is described in Table 7.6. The matrices show the cosine similarity between vectors of different time-steps t. Darker areas denote higher similarity.

8 Discussion

The improved model H-HebbLSTM vastly outperforms HebbLSTM in every task (see Section 7 for details). It is assumed, that the improvements are able to eliminate all the identified drawbacks of the HebbLSTM model discussed in Section 6. In the bAbI question answering tasks [4], H-HebbLSTM performs at a state-of-the level with a mean test error of 0.8%, yielding comparable results to the closely related H-Mem [9] model. A very interesting result is that H-HebbLSTM is capable of answering queries in task 3 of the 20 bAbI tasks [4], "three supporting facts", in only one time-step. In this task three of the input facts need to be chained together in order to resolve the query. In only one time-step, H-HebbLSTM is capable of resolving all of the three relations to correctly answer the piece of information requested by the query. It is not exactly clear how the model accomplishes this process, however, it is assumed that on the one hand the hidden state of LSTM helps the model preparing these one-time-step resolutions and on the other hand, the fact that two read operations are performed in one time-step, the model can actually query the memory twice per time-step.

8.1 Conclusion

This work shows that a RNN with an architecture similar to LSTM [13] can be augmented with a Hebbian-style hetero-associative neural memory to successfully solve tasks requiring the extraction, storage and retrieval of variable bindings. This hypothesis was justified using two different tasks, an artificial toy dictionary learning task and an artificial question-answering task [4]. Despite its simplicity, Hebbian learning is a powerful tool to implement a working memory instance which can enrich the computational capabilities of a RNN on tasks where association and memorization of patterns plays a central role. Both of the proposed models are fully differentiable, allowing end-to-end training for the optimization of the parameters. Due to the nature of the RNN architecture, the model is flexible and can be embedded in several other architectures such as convolutional networks or encoder-decoder architectures, but that was not scope of this work.

8.2 Future Work

On examples with long sequences (with sequence length > 300), the model shows some instabilities during gradient-based training. The architecture can be further optimized by additional gradient stabilization mechanisms like [67].

Further use-cases of the model can be examined, for example reinforcement learning tasks. The combination of different types of memory could possibly help the model to short-term memorize discoveries of an agent [6].

H-HebbLSTM uses an input and forget gate to calculate the hidden value. According to literature [68], [69], the gates shall be bounded between 0 and 1, which is accomplished by using the logistic sigmoid function as activation for the gate values. In H-HebbLSTM however, the best performance was achieved by using the ReLU activation function for the gates, which is unbounded due to the linear positive part. During development, no model could be found to solve all of the 20 bAbI tasks using logistic sigmoid gate activations. No explanation for this behaviour could be found so far. For

future work, one could further examine the behaviour of the gates and try to find a model which achieves equal performance by using logistic sigmoid activation functions for the gating mechanisms, which might further enhance the stability when training on long sequences.

Furthermore, H-HebbLSTM performs a layer normalization and on the hidden value at each time-step. This is needed, since the input activation function for the hidden value is the ReLU function, which is unbounded on the positive side. To avoid that the hidden value grows rapidly, this layer normalization is required, however, it is assumed that this step weakens the back-propagation for long sequences, even though the model is capable of solving task 3 of the bAbI tasks. This task is the longest of the task, including sequences with up to 320 sentences.

It is worth noting, that in addition to the layer normalization, an activation function (which was ReLU in all of the tasks) was applied on the hidden value. Even though this seems counter-intuitive, it resulted in increased stability during training, because it counteracts the unboundedness of adding a value determined by the ReLU activation function at each time-step.

Another approach which might be worth further examination is to replace the single-layered networks within the cell with more complex multi-layered network. Even though this increases the complexity of the model, it presumably enhances its computational capabilities. This might especially be required in more complicated tasks, for example the translation of sentences from one language to another.

Bibliography

[1] W. Bechtel and A. Abrahamson, *Connectionism and the mind; an introduction to parallel processing in networks*, 1st. USA: Blackwell Publishers, Inc., 1990.

[2] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 2016.

[3] S. J. Hanson and D. J. Burr, "What connectionist models learn: Learning and representation in connectionist networks," *Behavioral and brain sciences*, vol. 13, no. 3, pp. 471–489, 1990.

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

[5] S. Sukhbaatar, A. Szlam, J. Weston and R. Fergus, "End-to-end memory networks," in *Advances in neural information processing systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama and R. Garnett, Eds., vol. 28, Curran Associates, Inc., 2015, pp. 2440–2448.

[6] T. Munkhdalai, A. Sordoni, T. Wang and A. Trischler, "Metalearned neural memory," in *Advances in neural information processing systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019, pp. 13 331–13 342.

[7] I. Schlag, T. Munkhdalai and J. Schmidhuber, *Learning associative inference using fast weight memory*, 2021. arXiv: 2011.07831 [cs.LG].

[8] G. Palm, F. Schwenker, F. T. Sommer and A. Strey, "Neural associative memories," *Biological cybernetics*, vol. 36, pp. 36–19, 1993.

[9] T. Limbacher and R. Legenstein, "H-mem: Harnessing synaptic plasticity with hebbian memory networks," English, in *Advances in neural information processing systems*, vol. 33, 2020.

[10] K.-L. Du and M. Swamy, "Associative memory networks," in 2014, pp. 187–214.

[11] T. Kohonen, *Self-organization and associative memory: 3rd edition*. Berlin, Heidelberg: Springer-Verlag, 1989.

[12] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. New York: Wiley, 1949.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 1997.

[14] M. Tsodyks, "Associative memory in neural networks with the hebbian learning rule," *Modern physics letters b*, vol. 03, no. 07, pp. 555–560, 1989.

[15] A. Graves, G. Wayne and I. Danihelka, *Neural turing machines*, 2014. arXiv: 1410.5401 [cs.NE].

[16] J. Weston, S. Chopra and A. Bordes, *Memory networks*, 2015. arXiv: 1410.3916 [cs.AI].

[17] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological review*, pp. 65–386, 1958.

[18] J. Schmidhuber, "Learning to control fast-weight memories: An alternative to dynamic recurrent networks," *Neural computation*, vol. 4, no. 1, pp. 131–139, 1992.

[19] G. E. Hinton and D. C. Plaut, "Using fast weights to deblur old memories," in *In proceedings of the 9th annual conference of the cognitive science society*, Erlbaum, 1987, pp. 177–186.

[20] L. Squire and E. Kandel, *Memory: from mind to molecules*, ser. Owl book. Henry Holt and Company, 2003.

[21] J. Melchior and L. Wiskott, *Hebbian-descent*, 2019. arXiv: 1905.10585 [cs.LG].

[22] T. Munkhdalai and A. Trischler, *Metalearning with hebbian fast weights*, 2018. arXiv: 1807.05076 [cs.NE].

[23] P. Smolensky, "Tensor product variable binding and the representation of symbolic structures in connectionist systems," *Artif. intell.*, vol. 46, no. 1–2, pp. 159–216, 1990.

[24] A. Graves, *Supervised Sequence Labelling with Recurrent Neural Networks*, ser. Studies in Computational Intelligence. Berlin: Springer, 2012.

[25] O. E. Arslan, "Chapter 3 - computational basis of neural elements," in *Artificial neural network for drug design, delivery and disposition*, M. Puri, Y. Pathak, V. K. Sutariya, S. Tipparaju and W. Moreno, Eds., Boston: Academic Press, 2016, pp. 29–82.

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

[27] Y. LeCun, Y. Bengio and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[28] K.-L. Du and M. N. Swamy, *Neural networks and statistical learning*. Springer Publishing Company, Incorporated, 2013.

[29] D. E. Rumelhart, G. E. Hinton and J. L. McClelland, "A general framework for parallel distributed processing," in *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*. Cambridge, MA, USA: MIT Press, 1986, pp. 45–76.

[30] K.-i. Funahashi and Y. Nakamura, "Approximation of dynamical systems by continuous time recurrent neural networks," *Neural networks*, vol. 6, no. 6, pp. 801–806, 1993.

[31] A. C. Tsoi and A. Back, "Discrete time recurrent neural network architectures: A unifying review," *Neurocomputing*, vol. 15, no. 3, pp. 183–223, 1997.

[32] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

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

[34] M. I. Jordan, "Attractor dynamics and parallelism in a connectionist sequential machine," in *Proceedings of the eighth annual conference of the cognitive science society*, Hillsdale, NJ: Erlbaum, 1986, pp. 531–546.

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

[36] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks," *GMD-report 148, german national research institute for computer science*, 2001.

[37] G. Tanaka, T. Yamane, J. B. Héroux, R. Nakane, N. Kanazawa, S. Takeda, H. Numata, D. Nakano and A. Hirose, "Recent advances in physical reservoir computing: A review," *Neural networks*, vol. 115, pp. 100–123, 2019.

[38] D. Servan-Schreiber, A. Cleeremans and J. L. Mcclelland, "Graded state machines: The representation of temporal contingencies in simple recurrent networks," *Machine learning*, vol. 7, pp. 161–193, 1991.

[39] T. Mikolov, A. Joulin, S. Chopra, M. Mathieu and M. Ranzato, *Learning longer memory in recurrent neural networks*, 2015. arXiv: 1412.7753 [cs.NE].

[40] A. M. Schäfer and H. G. Zimmermann, "Recurrent neural networks are universal approximators," in *Artificial neural networks – ICANN 2006*, S. D. Kollias, A. Stafylopatis, W. Duch and E. Oja, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 632–640.

[41] R. J. Williams and D. Zipser, "Gradient-based learning algorithms for recurrent networks and their computational complexity," in *Backpropagation: theory, architectures, and applications*. USA: L. Erlbaum Associates Inc., 1995, pp. 433–486.

[42] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *Int. j. uncertain. fuzziness knowl.-based syst.*, vol. 6, no. 2, pp. 107–116, 1998.

[43] F. Gers, J. Schmidhuber and F. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural computation*, vol. 12, pp. 2451–71, 2000.

[44] F. A. Gers, N. Schraudolph and J. Schmidhuber, "Learning precise timing with LSTM recurrent networks," *Journal of machine learning research*, vol. 3, pp. 115–143, 2002.

[45] Kurt and Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[46] T. M. Mitchell, *Machine learning*. New York: McGraw-Hill, 1997.

[47] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning internal representations by error propagation," in *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*. Cambridge, MA, USA: MIT Press, 1986, pp. 318–362.

[48] D. E. Rumelhart, G. E. Hinton and J. L. McClelland, "Training hidden units: The generalized delta rule," in *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*. Cambridge, MA, USA: MIT Press, 1986, pp. 121–159.

[49] A. Cauchy, "Methode generale pour la resolution des systemes d'equations simultanees," *C.r. acad. sci. paris*, vol. 25, pp. 536–538, 1847.

[50] J. Kiefer and J. Wolfowitz, "Stochastic estimation of the maximum of a regression function," *Ann. math. statist.*, vol. 23, no. 3, pp. 462–466, 1952.

[51] H. Robbins and S. Monro, "A stochastic approximation method," *Ann. math. statist.*, vol. 22, no. 3, pp. 400–407, 1951.

[52] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors," in *Neurocomputing: foundations of research*. Cambridge, MA, USA: MIT Press, 1988, pp. 696–699.

[53] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*, 3rd ed. Pearson, 2009.

[54] G. I. Webb, "Overfitting," in *Encyclopedia of machine learning*, C. Sammut and G. I. Webb, Eds. Boston, MA: Springer US, 2010, pp. 744–744.

[55] M. D. Zeiler, *ADADELTA: An adaptive learning rate method*, 2012. arXiv: 1212.5701 [cs.LG].

[56] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of 3rd international conference on learning representations, ICLR 2015*, Y. Bengio and Y. LeCun, Eds., 2015.

[57] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982. eprint: https://www.pnas.org/content/79/8/2554.full.pdf.

[58] Ç. Gülçehre, S. Chandar, K. Cho and Y. Bengio, "Dynamic neural turing machine with continuous and discrete addressing schemes," *Neural computation*, pp. 857–884, 2018.

[59] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. learn.*, vol. 8, no. 3–4, pp. 229–256, 1992.

[60] K. Fox and M. Stryker, "Integrating hebbian and homeostatic plasticity: Introduction," *Philosophical transactions of the royal society b: biological sciences*, vol. 372, no. 1715, p. 20 160 413, 2017.

[61] W. Gerstner and W. Kistler, "Mathematical formulations of hebbian learning," *Biological cybernetics*, vol. 87, pp. 404–15, 2003.

[62] E. Oja, "Simplified neuron model as a principal component analyzer," *Journal of mathematical biology*, vol. 15, no. 3, pp. 267–273, 1982.

[63] Y. Munakata and J. Pfaffly, "Hebbian learning and development," *Developmental science*, vol. 7, no. 2, pp. 141–148, 2004.

[64] G. Palm, "Neural associative memories and sparse coding," *Neural networks: the official journal of the international neural network society*, vol. 37, 2012.

[65] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In proceedings of the international conference on artificial intelligence and statistics (AISTATS'10). society for artificial intelligence and statistics*, 2010.

[66] K. He, X. Zhang, S. Ren and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision (ICCV)*, 2015.

[67] J. Zhang, Q. Lei and I. Dhillon, "Stabilizing gradients for deep neural networks via efficient SVD parameterization," in *Proceedings of the 35th international conference on machine learning*, J. Dy and A. Krause, Eds., vol. 80, Stockholmsmässan, Stockholm Sweden: PMLR, 2018, pp. 5806–5814.

[68] R. Jozefowicz, W. Zaremba and I. Sutskever, "An empirical exploration of recurrent network architectures," in *Proceedings of the 32nd international conference on international conference on machine learning - volume 37*, Lille, France: JMLR.org, 2015, pp. 2342–2350.

[69] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2017.