Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Neural Networks with Memory

Bc. Ondřej Kužela

Supervisor: doc. RNDr. Ing. Marcel Jiřina, Ph.D.

8th May 2016

Acknowledgements

I would like to express gratitude to my supervisor doc. RNDr. Ing. Marcel Jiřina, Ph.D. for all his help and time he invested into me and this thesis. Without him this thesis wouldn't be in the state it is right now if I even were able to finish it without him. I would like to also thank my advisors Martin Bálek and Peter Hroššo for all their advices and help, including introducing me to the first neural networks with memory and actually coming up with the topic of this thesis. Another person that deserves my thanks is my roommate Jan Duchač, because without him constantly motivating me into writing I would never be able to finish this thesis on time. At least but not last I would like to thank my family for all the support they gave me during my whole studies and writing of this thesis, without their help none of this would be possible.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 8th May 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kužela, Ondřej. *Neural Networks with Memory.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Neuronové sítě s pamětí jsou rodinou neuronových sítí, které kromě klasické paměti ve formě vah, sloužících pro dlouhodobé závislosti, obsahují také jinou formu paměti. Ta slouží pro uchovávání střednědobých, občas také nazývaných dlouho-krátkodobých, závislostí. Taková paměť může být buď interní nebo externí. V rámci této práce poskytuji souhrnný náhled na rodinu neuronových sítí s pamětí. Na základě analýzy existujících modelů také navrhuji nový model, který nazývám Recurrent Neural Modules with External Memory. Tento model nabízí nový a inovativní přístup k použití externí paměti v rámci neuronových sítí, jelikož nasazuje externí paměť na úrovni částí sítě a tudíž obsahuje několik externích pamětí v rámci jedné sítě. Výkonnost nově navrženého modelu byla testována na Air Travel Information System (ATIS) datasetu.

Klíčová slova neuronové sítě, rekurentní neuronové sítě, neuronové sítě s pamětí, externí paměť, moduly, střednědobé závislosti, dlouho-krátkodobé závislosti

Abstract

Neural networks with memory are the family of the neural networks that except the classic memory for the long-term dependencies, in a form of the weights, also contain another form of a memory. Such a memory serves to retain the mid-term, sometimes also called long-short-term, dependencies and can be of two different types, either internal or external. Within this thesis I offer a summarizing overview of the family of the neural networks with memory. Based on the analysis of the existing models I also propose a new model of the Recurrent Neural Modules with External Memory. This model offers a new and innovative approach to the usage of the external memory within the neural networks, since it deploys the external memory on the scope of parts of the network and thus deploys multiple external memories within one network. The performance of the newly proposed model was evaluated on the Air Travel Information System (ATIS) dataset.

Keywords neural networks, recurrent neural networks, neural networks with memory, external memory, modules, mid-term dependencies, long-short-term dependencies

Contents

List of Figures

List of Tables

Introduction

Motivation and Objectives

One of the biggest dreams of the mankind is the idea of machines being able to replace humans in their duties. But currently we are not much closer to fulfilment of this dream than we were years ago. The reason is not that there were not enough people trying, but that no one has so far found the way how to do so. Human brain is a very complicated device we don't fully understand at this moment, but this can't stop us from trying.

It is a well known fact that the human brain consists of many cells called neurons. These cells are connected together in a way so they can communicate. There are actually billions of neurons within the brain[1] and they can all work parallely. That is something we can't simply recreate using today's computers, for example common NVIDIA graphic cards with CUDA architecture can manage "only" up to tens of thousands of concurrent threads[1].

Another limitation that we are usually encountering while working with the neural networks is the fact that most of the current neural network models are struggling on problems that require keeping track of a context that is longer than the network models allows to. Through the history there have been several attempts to overcome these limitations of artificial neural networks. This thesis focusses on the ones I am calling the Neural networks with memory.

Problem Statement

The Neural networks with memory are those models of the neural networks that except the long term memory in the form of the weights, also propose some sort of a memory for managing the mid-term, sometimes also called

[1] `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

long-short-term, dependencies. The memory for managing the mid-term dependencies can be of two types, internal or external.

Probably the simplest example of a neural network, that uses an internal memory for managing the mid-term context, are the recurrent neural networks. A basic example of such a network might be the Elman network, even though the length of the dependencies it can retain is quite limited.[2] Another well known example of such a network is the current state-of-the-art the Long Short Term Memory[3], which is probably the best performing universal neural network as of now. Other examples of such networks are for example the Liquid State Machine[4] or the Clockwork RNN[5].

The neural networks with external memory are something, at least for me, more interesting. Over the last few decades there have been several attempts in supplementing the neural network with an additional external memory that would allow the network to retain longer dependencies. This trend was especially very popular among the researchers within the last few years. Some examples of such models are the Neural Turing Machine[6], the Stack RNN[7] or the Recurrent Neural Network with External Memory[8]. Even though these models might be perceived as something not natural (nature inspired), I believe in their great potential.

Goals of the thesis

The primary goals of this thesis can be summarized as follows:

- Study and summarize the family of the neural networks with memory, both internal and external.
- Analyze selected neural networks with memory in order to find the possibilities for an improvement.
- Design a modification, possible improvement, of one of the selected neural networks.
- Compare the performance of the proposed modification against the original model and other neural network models on a selected problem.

Structure of the thesis

The rest of this thesis will be organized into 6 chapters as follows. In the Chapter 1 I will give a brief introduction into the topic of neural networks in general. Since there is no paper that would summarize the whole family of the neural networks with memory, the Chapter 2 will be devoted to this summary. In the same chapter some of the selected networks will be further described in detail. The Chapter 3 will focus on the analysis of a modification of one

of the selected networks and also on a design of such a modification. In the Chapter 4 I will focus on a further description of the proposed modification and also on the implementation of its prototype and all the other network models necessary for the testing phase. The Chapter 5 will be devoted to the testing of the proposed model and the comparison of its performance to other network models. The last part of this thesis will be the summary of the results and the contribution of this thesis.

Chapter 1

Introduction to Neural Networks

Before I will be able to introduce the concept of neural networks with memory, few basic concepts needs to be defined. First of them is the concept of basic artificial neural networks (ANN). The inventor of one of the first neurocomputers, Dr. Robert Hecht-Nielsen, defines a neural network as "*...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.*"[9]. But how do the artificial neural networks actually work? To be able to answer this we have to go back to the source system we are trying to simulate here, neural network inside the human brain.

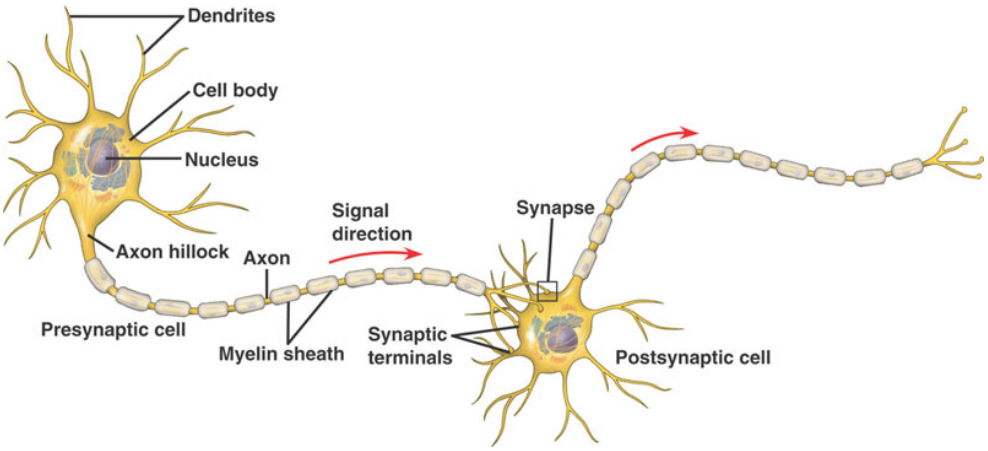

Figure 1.1: A visualization of the communication between neurons[2]

On the above picture you can see the basics of communication between neurons. Every neuron has multiple processes called dendrites, which create a

[2]Source: http://biomedicalengineering.yolasite.com/resources/neuron_structure.jpg

branch-like structure. Those serve as receivers of the information from other cells, like other neurons, sensory receptors or muscle cells. From there the received information is transferred as an electrical signal through the cell body to an axon. Axon is the part of the neuron cells which has the function to carry the information to other cells. On the end of an axon there is a branch-like structures called the axon terminals. Those are the points of transfer of the information to other cells. The transfer site is called synapse and the transfer itself is handled using chemical processes between axon terminals of the presynaptic neuron and receptors on the end of dendrites of the postsynaptic neuron. After the exchange the postsynaptic neuron handles the transfer of the information.[1]

So now we stated basics of how the communication in real neural network works, but how do we simulate this concept using artificial neural networks? Artificial neural network in general can be viewed as a graph, where nodes are the neurons and edges are the synapses (plus dendrites and axons). Every node contains three basic components, which are shown on Figure 1.2:

1. **Weight vector**, vector of synapses, assignment of weights to edges of the graph
2. **Transfer** (summation) **function**, which computes the sum of signals multiplied by concrete weights
3. **Activation function**, which maps the result of the net input to the output of the neuron

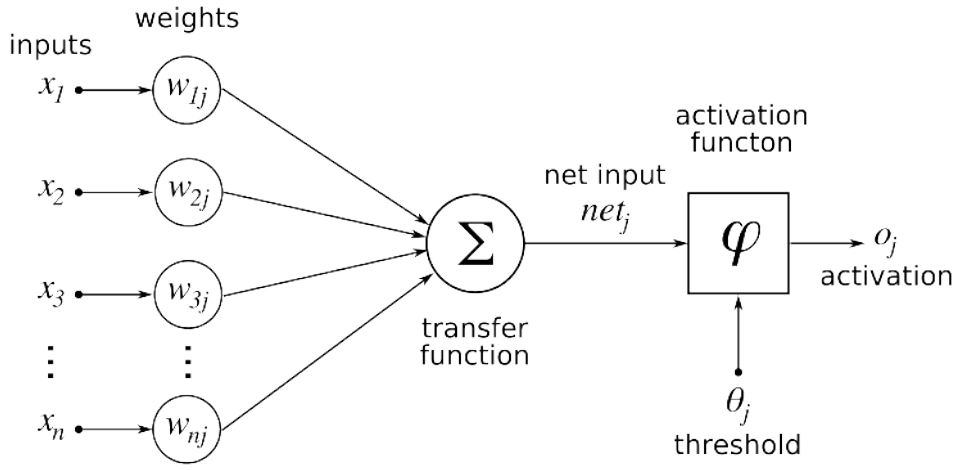

Figure 1.2: Artificial neuron model[3]

[3]Source: `http://andrewjamesturner.co.uk/images/ArtificialNeuronModel.png`

On the Figure 1.2 you can also see threshold by the activation function, which means that if the signal is too weak (weaker than the set threshold), it is not propagated further and is stopped in the neuron.

In general the network can have one or more layers. If there are more layers, the first one is usually called the input layer and the last one the output layer. Input layer is the part of the network, where the external input enters the network in means of signals. Neurons of this layer serve to process the external input and transfer it further. After the input layer there might be one or more hidden layers, whose neuron acts exactly like stated on the Figure 1.2. The neurons of the output layer acts the same as the ones from the hidden layer, but their output is propagated into the final output of the whole network. If we allow the feedback edges, edges that go within the same layer or to one of the previous layers, it means that there can appear cycles. Neural networks with cycles are called recurrent and they will be in detail described in further chapters.

The last thing that needs to be explained before we move to more advanced topics is how do the artificial neural networks learn. I will only explain this for one subtype of learning, which is the supervised learning. There are also other types of learning like unsupervised or reinforcement learning, but those are beyond the scope of our needs right now. You can imagine the newly created artificial neural network as a brain of a newly born child. The newly born child also doesn't know how to classify items based on their color and has to learn it first. It learns the way that it tries to assign an item to the category he thinks it fits the most, for example green. After that his mother tells him that he either assigned it correctly or it should have belonged to another color, for example red. On the next attempt of the same or similar color item it will be more likely that he will assign the item correctly. And this is almost the same way as how the supervised learning works.

The commonly used supervised learning algorithm is the Backpropagation algorithm. For every input in the learning set there is also given a model output. The Backpropagation algorithm has two major phases:

1. **Forward phase** – computation of outputs of all the neurons in the network, at the end error is computed based on the model and real output,

2. **Backward phase** – error is propagated back through the network, the weights are being changed in order for the error rate to be minimized.

The Backpropagation leads to minimization of the error function to minimum, which doesn't necessary have to be the global one. The speed of the training can be changed in order to prevent the overlearning with every single incorrectly recognized input.[10][11]

1.1 Recurrent Neural Networks

Even in our lives we barely decide any piece of information without a context. For an artificial neural networks to be able to recognize the context we need to allow the existence of cycles. The architecture which doesn't allow cycles inside of the networks is called Feed-forward networks (the left picture in Figure 1.3). Since memorization in general inclines to some context-based decision making I will put this architecture aside. Even though in the later chapters I will talk about a way how to use Feed-forward networks together with preprocessing using a recurrent component for a context-based decision making.

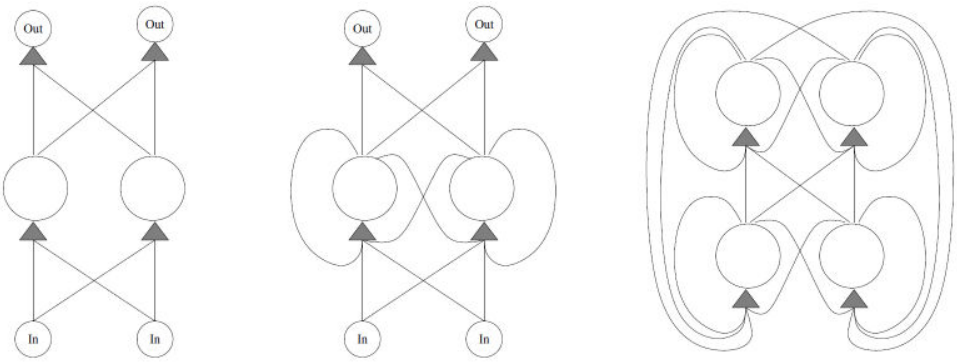

Figure 1.3: Neural network models[12], Left: Feed-forward network model, Middle: Recurrent neural network model, Right: Fully connected recurrent neural network model

Recurrent Neural Networks (RNN) are Feed-forward networks supplemented by additional feedback edges, which provide back the context. In general this context might be viewed as the state of the network in the previous time step. Using this context RNN gain the ability to learn time/context dependent problems including sequence prediction, handwriting recognition or speech recognition. We can call this ability to remember the context a memory. But how powerful this memory actually is? How long back in time is it able to remember the context? To be able to understand this we have to take a look back in the history of recurrent neural networks.

The foundations of RNN research took place in the 1980s and early 1990s. With Hopfield networks in 1982, the first introduced recurrent network, the main progress can be dated between 1986 and 1990 with Jordan and Elman networks. Both of these networks introduced the concept of so called context units. Context unit was a special unit which had the purpose to save the state of a given unit (neuron) in the hidden (for Jordan network output) layer in

time t and feed the same state into a the same (for Jordan given) unit in the hidden layer. Even though these networks had achieved success in learning short-range dependencies, they haven't been showing any worth mentioning achievement with learning mid-range dependencies. This was mainly caused by the problems of vanishing and exploding gradients.[2]

When backpropagating the error across many time steps, using standard learning algorithms like Backpropagation through time, which is the generalization of classic Backpropagation algorithm for RNN, both vanishing or exploding gradient problems can occur. As detailed in Bengio et al.[13] both caused the limited ability of RNN to learn the mid-range dependencies. The exploding gradient problem appears when the long-term components grow exponentially more than the short term ones, which leads to their explosion. The vanishing gradient problem is the opposite behaviour, which appears when long-term components grow exponentially fast to zero. That results into being impossible for the model to find any links between distant events. Various researchers have been trying ever since to overcome this limitation and find a way to allow RNN to learn the mid-term dependencies. The overview of subset of those attempts is the theme of the next chapter. Please note that from now on the phrase "neural network" will for simplification mean an artificial neural network if not stated otherwise.

Chapter 2

State-of-the-art

2.1 Adding memory to RNN

The goal of this thesis is to study the neural networks with memory, but so far I haven't stated in detail what exactly is meant by that. Now having explained all the necessary basics I can move right onto it. The neural networks that we commonly simulate in computer are by far more simple than those complex neural systems created by the nature. We are trying to simulate all we do understand, but apparently it is not enough. So to be able to overcome all the limitations for learning mid-range context like the problems of vanishing and exploding gradients, we have to apply some upgrades to the neural network. The resulting model doesn't necessary have to actually be inspired by the biological neural networks.

One of the families of upgrades that appear over the last 30 years, is the upgrade in form of an additional memory added to the neural network. This memory can be either internal for neurons, which happens for example in Long Short Term Memory[3] or Liquid State Machine[4], or external memory shared by the whole network, which is being used in some of the latest attempts including Neural Turing Machine[6], Memory Networks[14] or Stack RNN[7]. These attempts appear to have strong impact on the latest discoveries in the field of artificial intelligence and thus I would like to further orientate strictly on them, even though there have been other worth-mentioning attempts in the learning of mid-range dependencies.

I have been trying to find any paper containing a summary of such neural networks. Probably the best ones I have found have been the ones by Josefowicz et al.[15] and Lipton et al.[2]. Even though both of them summarize the family of recurrent neural networks, they do not mainly focus on the neural networks with memory and thus I would I like to write this summarization myself. In the rest of this section I will cover the development in the family of the neural networks with memory.

2.1.1 DISCERN

The beginnings of attempts in adding a memory to neural networks can be dated back into the early 1990s, when first ones have been tested. One of the first attempts have been the DISCERN system introduced by Miikkulainen in his PhD thesis in 1990. DISCERN was a large-scale natural language processing system, which was able to process stories written in natural language and later answer questions about them. The system consists out of two basic types of components: processing and memory modules. Processing modules, consisting out of 3-layer Elman networks, are serving mainly for parsing, generating and questions answering. Memory modules consist out of sets of feature maps, where words are stored in forms of vectors and which serve as auxiliary modules for processing units. With the system combinating power of neural networks together with R/W memory, DISCERN can be considered one of the first neural network with memory.[16]

2.1.2 NNPDA

In 1992 the model of the Recurrent Neural Network Pushdown Automaton (NNPDA) was introduced by Das et al.[17]. The NNPDA consists out of recurrent neural network which is connected to an external stack memory. It has been proven that the NNPDA is not only able to learn the state transitions of the underlying finite state automaton, but also how to control the connected stack. Using these features the NNPDA is able to learn simple deterministic context-free grammars. The RNN is communicating with the stack through an error function. Interesting fact is that the use of an error function according to the authors allows the network to avoid using the stack when it's not necessary for a successful learning of the language.

To be able to understand how this model works and what is its contribution to later research, we need to show the basics of its inner functionality. As you can see on the Figure 2.1 the model consists out of more different types of neurons with each serving a different purpose. The core part of the network are the fully recurrent State neurons. Those are the part that keeps the track of the network state and allows the network to learn. They take input from three different sources: their own recurrent edges, the Input neurons and the Read neurons.

The Input neurons serve for processing the external input to the system, while the Read neurons process the input from the top of the stack. The Read neurons handle the output of the stack, but do not control the operations performed on the stack. That is a job of the Action neuron. The Action neuron is a non-recurrent neuron, which based on its continued valued activation indicates the action which is to be performed on the stack (push, pop or no-op). This model served as a source of some ideas for another stack related

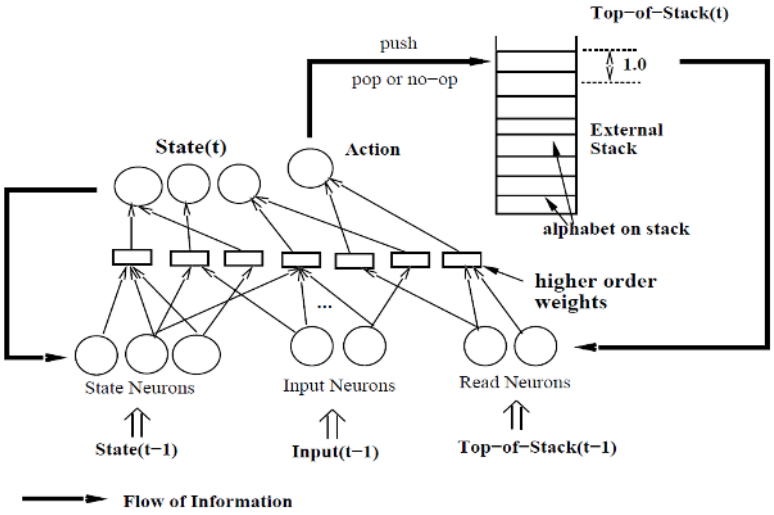

Figure 2.1: Model of the Recurrent Neural Network Pushdown Automaton (NNPDA)[17]

research from 2010s called Stack RNN, which will be mentioned later in this chapter.

2.1.3 NARX

Another research which set a ground for later researches was the Nonlinear AutoRegressive model with eXogenous (NARX) which was proposed as a model for learning mid-term dependencies in 1996 by Lin et al.[18]. Even though it doesn't contain any external memory, some of its parts serve as memory persistent units for few steps. This gives NARX the ability to remember context over more steps. I call this type of a memory an internal memory.

The NARX network model introduces units called output delays. Those output delay nodes are organized by cascading together. The number of time steps, the network can remember the old context, depends on the depth of the cascade structure. But the problem with the NARX model is that for it to remember a longer context the structure it needs to grow. This means that the NARX model may be better in retaining context information than simple RNN, but at the same time might be more complex and still vulnerable in sense of the vanishing gradient problem and thus not ideal for handling mid-term dependencies. But still this model set some interesting basics which were used in the later discoveries.

2.1.4 LSTM

Probably the most progressive and universal model in the whole recurrent networks family is the Long Short-Term Memory (LSTM) network firstly proposed by Hochreiter and Schmidhuber[3] in 1997. This network was primarily designed to fight the vanishing gradients problem, that hasn't been sufficiently solved before. The LSTM model proposes a new type of computational units for the hidden layer named memory blocks. Each memory block contains one or more memory cells. Every memory cell has at its core a unit called Constant Error Carousel (CEC). This CEC unit has a self recurrent edge and serves to enforce a constant error flow and thus prevent it from vanishing or exploding.

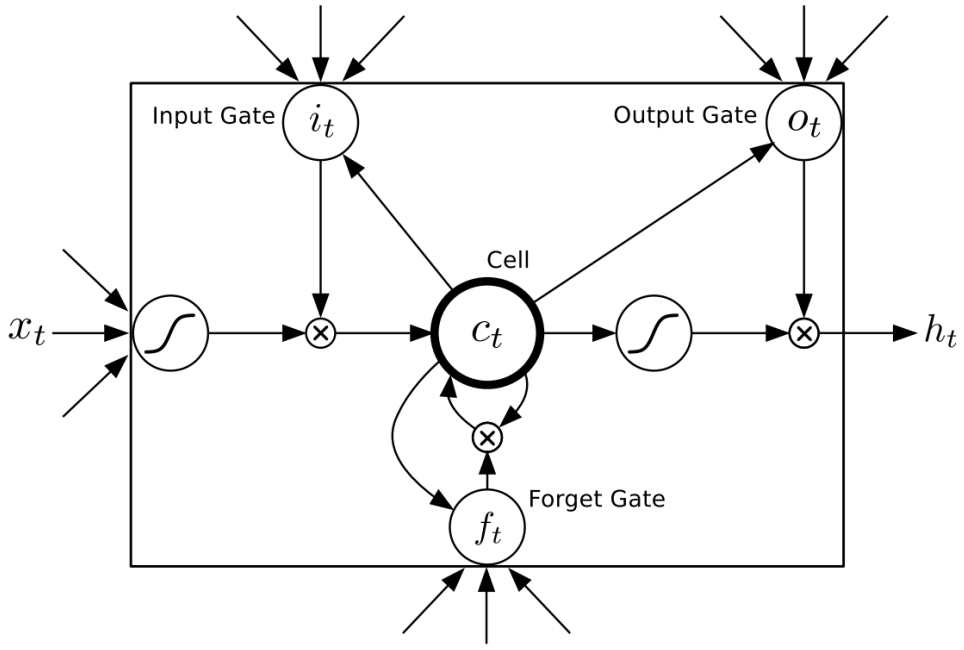

Figure 2.2: Model of a single memory block in the Long Short-Term Memory model[4]

Besides the memory cells every memory block also contains three, originally only two, adaptive gating units: input, output and forget gate. The input and output gates control the input and output into the memory cells and the forget gate learns to reset the inner state of the memory cell once its context is out of date. No one since has been able to propose any model which would be in general stronger than the LSTM and wasn't based on it. That is also the reason why newly proposed architectures are usually first compared to the

[4]Source: `http://blog.otoro.net/2015/05/14/long-short-term-memory/`

LSTM to see whether they even have at least comparable performance on the particular problem.

2.1.5 LSM

In the early 2000s two similar models were proposed independently and simultaneously: Liquid State Machine (LSM), published by Maass et al. [4], and Echo State Network (ESN), published by Jaeger et al. [19]. Both of those models share the main ideas, thus I will focus only on the Liquid State Machine. The LSM is a spiking neural network which consists out of leaky integrate-and-fire neurons. It doesn't connect to any external memory, but every neuron contains its own internal memory which can be imagined as a self recurrent edge with weight 1. This means that the neuron can save context from the last step. In general the network maintains its internal state based on all the input that came since it was in the calm state. This can be understood as that the network remembers context of any length, but it also has some limitations.

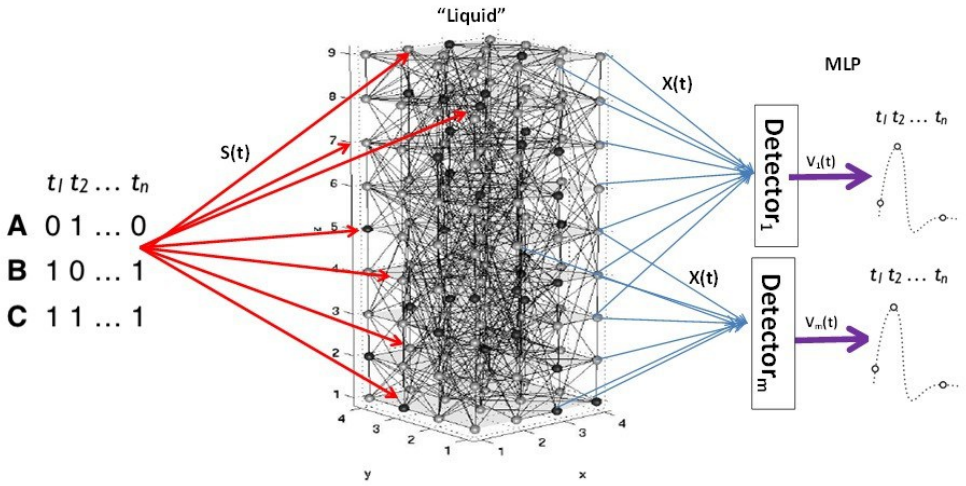

Figure 2.3: Visualization of the Liquid State Machine model[5]

Neither the LSM nor the ESN are classic neural networks that I have been talking about so far. Both of those models introduce something called reservoir computing. The network itself is called the reservoir which you can imagine as a space filled with a liquid. Any action that interferes the calm state of the liquid can be taken as an external input. After the action is performed the molecules of the liquid start to influence each other until the liquid reaches the calm state again. Through the whole process between the calm states,

[5]Source: `http://hananel.hazan.org.il/the-liquid-state-machine-lsm/`

the network reaches different internal states that can be observed. This is for example how the Liquid State Machine works and thus there is the word "liquid" in its name. The LSM is usually used as a part of some bigger model, for example to allow the classic Feed-forward network to process problems based on mid-term dependencies.

2.1.6 NTM

After 2000s which were not very productive in terms of new major discoveries concerning neural networks with memory, during the 2010s a new boom arrived. Especially in the last few years a lot of new researches have been published. One of the major ones was the model of Neural Turing Machine published in 2014 by Graves et al.[6] from Google Deepmind. The Neural Turing Machine model is capable of learning simple algorithms as copying or sorting. NTM contains two basic components: controller and memory matrix. Controller is a neural network that normally communicates with the external world using its input and output, but at the same time is also able to communicate with the memory matrix using read and write operations. Interesting fact is that as the controller it is possible to use the LSTM network, which has its own internal memory that can complement the larger memory in the matrix.

2.1.7 ClockWork RNN

Clockwork RNN was also published in 2014 by Koutník et al.[5]. This model is not a typical example of the area I focus on studying, but some of its features suggest that it deserves to be mentioned in this chapter. Its concept is similar to some researches from 1990s like the previously mentioned NARX model. It presents simple, but worth-mentioning, modifications to a simple RNN which allows this model to retain longer-term dependencies than the simple RNN, or for some problems even LSTM, can.

The Clockwork RNN works on a principle of dividing the hidden layer of the RNN into multiple parts, called modules. Each of these modules runs on a different clock speed, which is the key difference from all other previously mentioned models. Each module is assigned the time period T_i and the communication between the modules is allowed only from the slower ones, with bigger T_i, to the faster ones. This feature allows the Clockwork RNN to retain the context information for a longer time which leads into a better performance on learning mid-term dependencies. Another positive outcome of this feature is that the amount of neurons, which are active at a certain time step, is smaller than for simple RNN of the same size and thus it allows the Clockwork RNN to work faster in means of the real time.

2.1.8 MemNN

Another concept that was published in the 2014 is the concept of Memory Networks, which was published by Weston et al.[14] from Facebook AI Research. Memory Network model follows some of the researches from the 1990s including the DISCERN system. Memory networks are primarily proposed for the context of question answering (QA), where the long-term memory acts as a dynamic knowledge base. The model consist of a memory, an array of objects (for example vectors or strings), and four inference components: input feature map, generalization (updating stored memories based on the new input), output feature map and response (converting output into desired response). The response component is designed as an recurrent neural network that is conditioned on the output of the output feature map, which works based on the k-most relevant stored memories. The proposed model appears to perform in the context of QA better than standard recurrent neural networks, including LSTM.

2.1.9 Stack RNN

One of the latest researches in the field of the neural networks with memory is the Stack RNN proposed by Joulin and Mikolov[7] from Facebook AI Research. From the name you can already see that this model proposes nothing completely new, but follows the researches from early 1990s by Pollack and Das et al.. Stack RNN is proposed for the problem of recognition of algorithmically generated sequences, that are beyond the scope of learnability for basic recurrent networks. The proposed model consists of recurrent neural network that has increased learning capability being allowed to control an external infinite structured memory. For the memory authors propose two basic topologies suitable for the given problem, which are a pushdown stack and a list, with the stack having better performance for the given problem.

2.1.10 RNN-EM

Another research that was published in 2015 is the RNN-EM model proposed by Peng and Yao[8]. The model proposes the use of a RNN with an additional external memory, which stores the past hidden layer activities in order to increase the potential in learning the mid-term dependencies. The memory consists of n slots with each slot being a vector and allows read and update operations. Instead of feeding the past hidden layer activity directly back using a recurrent edge, the content of the external memory is used as one of the inputs of the hidden layer. The content of the memory is retrieved using a weight vector, which is created based on the similarity of the current hidden activity to the content of the external memory. After the new hidden layer activity is calculated, the memory is updated based on the outputs of the forget and update gates. The forget and update gates together create the update vector that is afterwards applied on the memory.

2.1.11 Neural Stack, Queue and DeQue

The latest research I would like to mention here are the Neural Stack, Queue and DeQue models proposed by Grefenstette et al.[20]. Over the last 25 years many neural networks that can control a stack have been proposed, but majority of them acted with the stack as a structure of discrete operations push and pop. But in the model of Neural Stack these discrete operations are rendered as continuous and thus are a real values in the interval (0,1). This can be interpreted as the degree of certainty with which the controller (RNN) wants to push/pop a vector onto/of the stack. As the controller the authors in the paper are using the LSTM network. A visualization how the controller uses the stack is shown on the Figure 2.4.

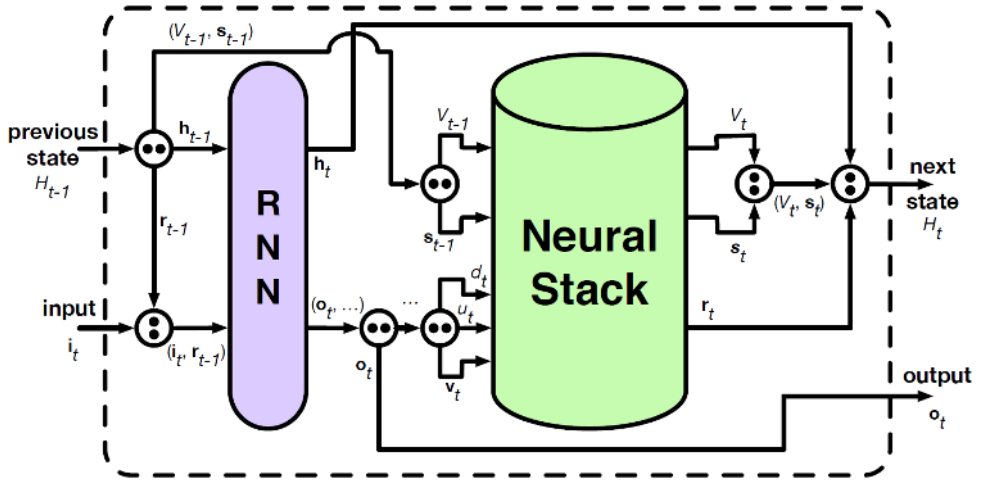

Figure 2.4: Model of the RNN controlling the Neural Stack[20]

But more importantly almost on the same base as the Neural Stack, two other structure are being proposed: Neural Queue and Neural DeQue. The Neural Queue operates almost the same way as the Stack with one exception that the pop operation reads the bottom of the structure instead of the top. The Neural DeQue means that instead of a stack a double ended queue is used. This allows to perform the push and pop operation on both ends of the memory. According to the paper both the structures show on some tasks like Bigram Flipping or Gender Conjugation better performance than both the LSTM and the Neural Stack.

2.2 Detailed analysis of selected networks

In the last section I presented many network architectures that have been proposed over the last 25 years. Some of them may be similar to another

ones which also is one of the reasons why is it not worth to study further all of them, but just some. After a discussion with my supervisor I have come to a conclusion to put the following five networks through a more detailed observation:

1. Liquid State Machine,
2. Clockwork RNN,
3. Memory Networks,
4. Stack RNN,
5. RNN-EM.

In the incoming subsections I will explain the basics of all these networks and in the end one of them will be selected for a further analysis and improvement. Another network, I won't be focusing on improving, but deserves to be described in detail, is the universal Long Short Term Memory network and thus I will also dedicate it one of the following subsections.

2.2.1 Long Short-Term Memory

The Long Short-Term Memory (LSTM) model can be as of today considered the universal state of the art as there so far hasn't been proposed any other model that would be in general more powerful than the LSTM. That is also the reason why this network is usually used as a baseline while proposing a new model. As long as the newly proposed model doesn't achieve better performance in at least one problem area than the LSTM it is probably not even worth the attention. Even though the LSTM model has already been proposed about 19 years ago, no one has been so far able to propose a better model not strongly based on the LSTM. That is one of the reasons why the LSTM deserves to be mentioned is this section even though I don't plan to focus on upgrading it.

As already previously mentioned the LSTM was originally proposed in order to deal with the vanishing and exploding gradient problem, which was achieved by introducing some innovative approaches. The key component of the of the hidden layer of the LSTM model is the memory block. The memory block is a unit that consists out of one or more memory cells and three gating units, input, output and forget gate. Each of these components has a special function that in total create the core of the innovate approach of the LSTM model. In the original model proposed by Hochreiter and Schmidhuber[3] in the 1997 there were only two gating units, the input and output ones. The forget gating unit was introduced in the early 2000s by Gers et al.[12], which is the version I will from now focus on. An example of a memory block with a single memory cell can be seen in the Figure 2.5.

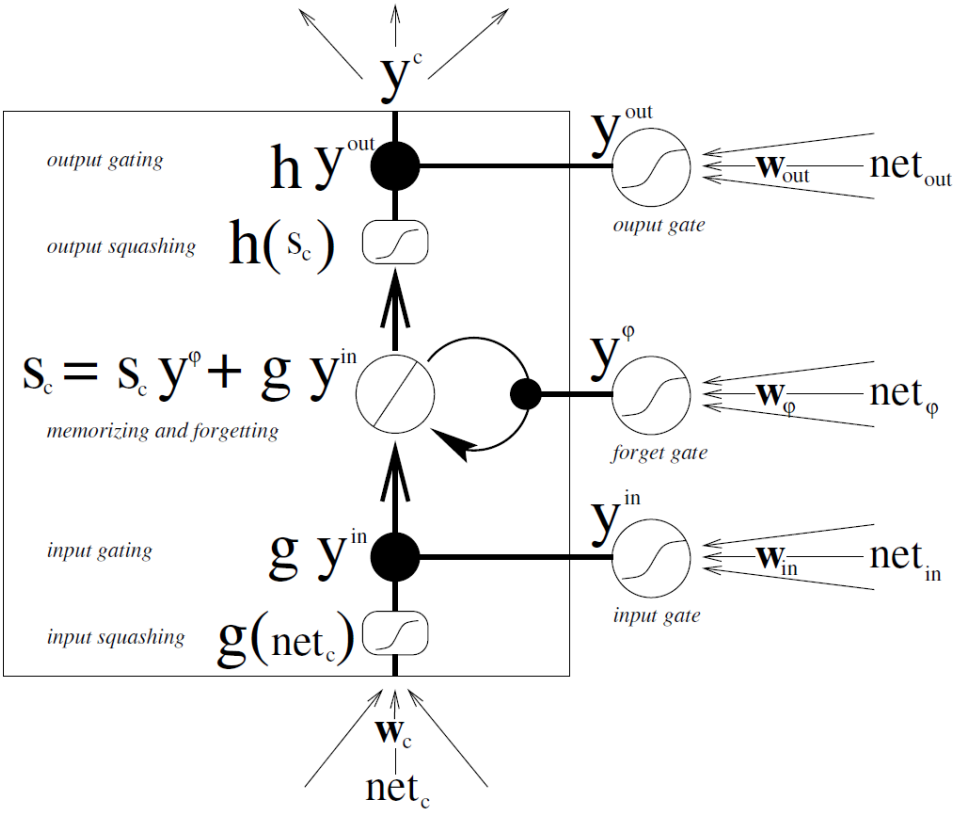

Figure 2.5: Visualization of one computational step in the memory block of LSTM[12]

As can be seen on the picture at time t the memory cell j on the beginning receives the network input net_c which is the weighted sum of the outputs of all the neighbour cells from time $t-1$. After the summation is performed the input is then being squashed by a sigmoid activation function $g(net_c)$ with range $[-2, 2]$. It doesn't necessary have be the sigmoid function, but can also be another activation function like for example the tanh function. So the whole input part is calculated and stored in a variable g as follows

$$net_{c_j^v}(t) = \sum_m w_{c_j^v m} \cdot y^m(t-1), \tag{2.1}$$

$$g(x) = \frac{4}{1+e^{-x}} - 2, \tag{2.2}$$

$$g = g(net_{c_j^v}(t)). \tag{2.3}$$

After the input is processed it is then to be multiplied by the input gate y^{in_j}. The input gate is a sigmoidal unit that controls the flow of input into memory cells of the given memory block. If its value is equal to 0 then the flow is completely cut off, otherwise if its value is equal to 1 then the whole flow is passed through. The gate computes its value as follows

$$net_{in_j}(t) = \sum_m w_{in_j m} \cdot y^m(t-1), \tag{2.4}$$

$$y^{in_j}(t) = f_{in_j}(net_{in_j}(t)), \tag{2.5}$$

which means that likely as the cell input it takes as its activation the weighted output from the state of the network in the previous time step. The f_in function is a standard logistic sigmoid activation function on the interval $[0, 1]$.

The core of each memory cell is a node s_c which is the internal state of the cell. The internal state has a self connected recurrent edge called the constant error carousel (CEC). This edge serves in order so the error can flow across the time without vanishing or exploding. But the addition proposed by Gers et al.[12] is that this flow is controlled by a unit called the forget gate and thus the inner state is calculated as follows

$$s_{c_j^v}(t) = y^{\varphi_j}(t) \cdot s_{c_j^v}(t-1) + y^{in_j}(t) \cdot g, \tag{2.6}$$

where y^φ is the forget gate. The forget gate is a unit that provides the network the ability to learn to flush the content stored in the internal state. This is especially useful in continuously running networks, where the inputs doesn't have marked starts and ends. The activation of the forget gate is calculated, almost the same as for the other gates, according to the equations stated below

$$net_{\varphi_j}(t) = \sum_m w_{\varphi_j m} \cdot y^m(t-1), \tag{2.7}$$

$$y^{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t)). \tag{2.8}$$

The last part that needs to be described here is how the output of the network y_c is calculated from the inner state s_c. The whole process is similar to the input one. First the output is being squashed by the activation function $h(x)$ and afterwards multiplied by the value of the output gate y^{out}. The output gate is a unit that controls whether the output is let out of the cell or not. Its activation is computed as follows

$$net_{out_j}(t) = \sum_m w_{out_j m} \cdot y^m(t-1), \tag{2.9}$$

$$y^{out_j}(t) = f_{out_j}(net_{out_j}(t)). \tag{2.10}$$

The final output of the cell can be then calculated as follows

$$y^{c_j^v}(t) = y^{out_j}(t) \cdot h(s_{c_j^v}(t)), \tag{2.11}$$

$$h(x) = \frac{2}{1 + e^{-x}} - 1, \tag{2.12}$$

where the function $h(x)$ is the central sigmoid activation function with range $[-1, 1]$.

In the previous few paragraphs I tried to give a short summary how the forward pass of the LSTM network works. The forward pass but serves only for the calculation of the value, not the training itself. For the training of the network Gers et al.[12] proposed a backward pass training algorithm for the proposed LSTM model. This training algorithm is a combination of the back propagation, which is used for training the output units, the truncated backpropagation through time (BPTT, Williams & Peng 1990), which is used for the training of the output gates, and the truncated version of the real time recurrent learning (RTRL, Robinson & Fallside 1987), which is used for training the weights to cells, input gates and forget gates. Even though the backward pass is a really important part of the LSTM model, its complexity puts it out of the scope of this thesis, especially because the LSTM and its learning is not the main focus. If you are interested in the process behind the learning of the LSTM I would like to refer you to the work published by Gers et al.[12].

Even though many papers have been published on the LSTM model since it was first proposed in 1997, not much have been published about the true source of its performance or its limitations. Probably the first deeper study of how LSTM model works internally and why, was published in 2015 by Karpathy et al.[21]. The authors of the study focus on studying how different parts of the network act while being exposed to real world data. Based on the cell activation statistics they were able to reveal that the different neurons of the LSTM serve different function after being read. For example if they are presented a text input they might act as some of the following:

- cell sensitive to position in line,
- cell that turns on inside quotes,
- cell that activates inside if statement or
- cell that acts like a line length counter.

Even though this sounds really interesting there is still a big amount of cells that are not easily interpretable and we can't easily guess their relevance. But deep understanding of the LSTM network might lead to a great progress in the field of neural networks.

2.2.2 Liquid State Machine

The Liquid State Machine (LSM) was the first neural network with memory I got into a contact with. I previously already performed a research[22], available in English on GitHub, of this model. Since I came across some interesting thoughts published by several authors about this network, I think it is worth to be mentioned in this section. The LSM is not a typical model of a recurrent neural network as for example the LSTM and contains some characteristics that no other network does. But before I will be able to explain how the LSM actually works, its key component needs to be described first: the leaky integrate-and-fire neurons.

The leaky integrate-and-fire neurons are the basic unit of the LSM and define many of its characteristics. The integrate-and-fire comes from the fact that the neurons keep integrating the incoming spikes until the firing limit is reached. After that they fire a spike to all connected neurons. This is also why the LSM is called a spiking network, because its neurons communicate using spikes. The word leaky comes from a fact that through the time, if no input comes, the inner state (potential) of the neurons is slowly decreased until it reaches the state of calm. The inner state of neuron n through the time is computed as follows

$$IS_a(t) = IS_a(t-1) + EI_a(t) + \sum_{i \in neighbours\ of\ n} IO_i(t-1), \tag{2.13}$$

where $IS_n(t)$ is the inner state of the neuron n in time t, $EI_n(t)$ is the external (outside of the LSM) input for the neuron n in time t and $IO_i(t)$ is the internal output of neuron i in time t, which equals to the internal input of all its neighbours in time $t+1$.

Before I will proceed further in the explanation of the LSM, let's see the natural process which is the source of the main idea of how the LSM works. Have you ever thrown a stone into the water and watch how it changes its surface? All those small waves floating on the water surface? This is probably the closest real life example of how the Liquid State Machine works. With any external stimuli the water changes its inner (liquid) state until it reaches the calm state again. And this is the process after which the Liquid State Machine was named. The LSM network works actually similar to how does the water do.

On the Figure 2.6 you can see the visualization of how the LSM works. At every time step t the network receives the input $u(.)$. The neurons of the network based on their inner state, external input and internal input compute a new inner state and fire a spike if necessary. Based on the inner state of the neurons, a new inner state of the network at time t $x^M(t)$ is computed as a set

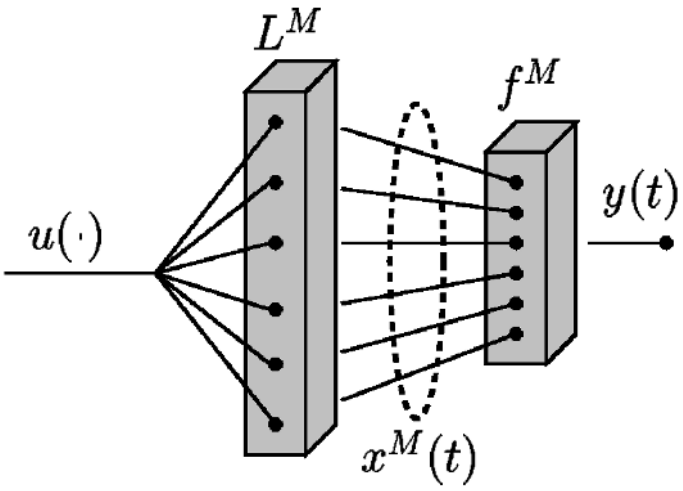

Figure 2.6: The Liquid State Machine[4]

of all non-external input neurons. After that the inner state is processed by a detector f^M and an output $y(t)$ is given. The reason why we use a detector is that the LSM itself doesn't usually learn during the process, it only encodes the incoming input based on its current inner state, and thus the detector of the network can be any classifier system that is able to be trained to recognize a pattern. Those could for example be:

- 3 layer Feed-Forward network,
- Support Vector Machine (SVM),
- Perceptron or Multi Layer Perceptron (MLP),
- Adaline (Widrow & Hoff, 1960),
- Tempotron (Gutig & Sompolinsky, 2006),
- etc.

The LSM is sometimes called a random network. What does this means is that the network consist out of just one layer of neurons that are not fully-interconnected and the input neurons are selected randomly among them. Based on the given connectivity (normally about 5 – 20%) each neuron is selected pseudorandomly, usually with some additional restriction, a given amount of neighbours. The restriction on the selection of neighbours defines the topology of the network. The originally proposed topology by prof. Maass was based on the close neighbours, having neurons places into 3D rectangular structure, this topology prefers connections between neurons that are closer to each other.

It has been proven by Hazan and Manevitz[23] in 2011 that the topology proposed by prof. Maass is not robust enough in the means of vulnerability to failures in the parts (neurons) of the network. This was proven using the dead (never firing) and noisy (firing as often as the refractory period allows it) neurons inside of the network. Even though the topology based on the close neurons haven't proven to be robust enough, some other topologies proposed by the authors have shown a much bigger robustness in the means of the damaged neurons. Those have been for example the small world topologies based on the power law.

So it has been proven that the LSM can be robust in the means of damaged neurons or noise in the input data, which has been proven by prof. Maass himself. Together with the Echo State Network, which is a similar network proposed by Jaeger et al.[19], they create the family of networks called the reservoir computing. Currently their field of expertise is the temporal patterns recognition. Temporal patterns are the patterns that come over time and need to be recognized as one pattern. The speech recognition is for example one of the fields where LSM is being successfully deployed. According to my opinion, with the LSM being a strongly biologically inspired network, with some breakthrough it can become a really strong model for even wider (maybe universal) field of use.

2.2.3 Clockwork RNN

Even though the concept of the Clockwork RNN[5] doesn't add any additional memory to the neural network, it organizes the hidden layer in a way that some of its parts serve as a memory for another ones. This feature makes this model perspective in a way that it may be combined with another model to create a new powerful architecture without putting any serious limitations on it. This is probably the main reason why I selected this model for a further study. In the upcoming paragraphs I will try to summarize how the Clockwork RNN network actually works.

I have already previously mentioned that the Clockwork RNN is based on the Simple Recurrent Neural Network (SRN). Just as the classic RNN it consists out of three basic layers: input, hidden and output layer. The input and output layer are the same as for the SRN, all the changes appear in the hidden layer. Unlike in the SRN in the Clockwork RNN the hidden layer is divided into g modules, each of them consisting out of k neurons. Each module is assigned a clock period $T_i \in T_1, \ldots, T_g$. This clock period says how often the module processes a new input and thus different modules work at different time steps. This is the key features which makes the Clockwork RNN stronger than the simple RNN.

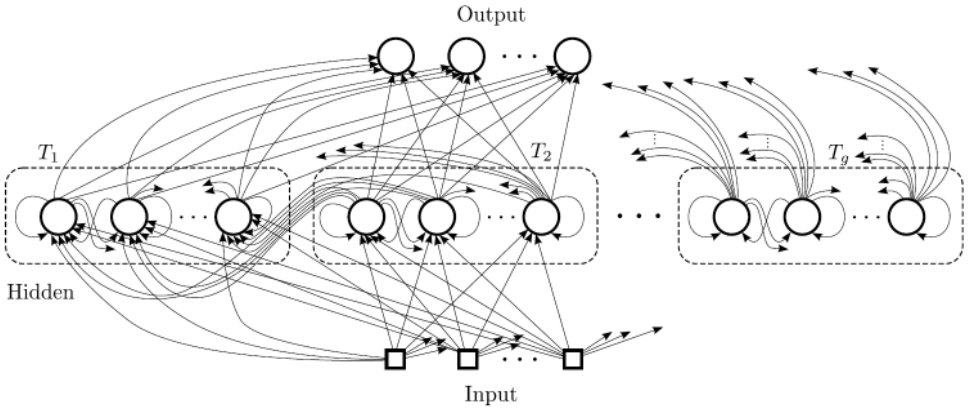

Figure 2.7: The Clockwork RNN model[5]

All neurons within one module are fully-interconnected. As of connections between neurons it depends on their clock periods. The recurrent connections from module i to module j exist if and only if the clock period T_i is bigger than T_j, i.e. if module i is slower than module j. If we would sort the modules by increasing period than we would get a similar as the one shown on the Figure 2.7 where all the connections that propagate the inner state lead from right to left. The reason why we allow only connections from slower to faster modules and not the other way around is obvious. If we think about the fact that those edges propagate the state of a hidden layer from the past and the other way around they would propagate a state from the future.

The SRN uses the following equations, without the omitted neuron biases, to calculate output at a time step t:

$$y_h(t) = f_H(W_H \cdot y(t-1) + W_i \cdot x(t)), \tag{2.14}$$

$$y_O(t) = f_O(W_O \cdot y_H(t)). \tag{2.15}$$

The main difference between the equations of the SRN and the Clockwork RNN is that for the Clockwork RNN only a subset of modules is active at a time step t. Specifically those are the modules that satisfy the equation $(t \ mod \ T_i) == 0$. This is satisfied by editing the weights W_H and W_I matrices into partition of g block as you can see in the following equations:

$$W_H = \begin{pmatrix} W_{H_1} \\ \vdots \\ W_{H_g} \end{pmatrix} \qquad W_I = \begin{pmatrix} W_{I_1} \\ \vdots \\ W_{I_g} \end{pmatrix} \tag{2.16}$$

where W_H is an upper-triangular matrix where all the rows are organized as $\{0_1, \ldots, 0_{i-1}, W_{Hi,i}, \ldots, W_{Hi,g}\}$, where i is the number of the row and W_{H_i} is a vector assigned to module i. W_I has rows organized as vectors W_{I_i} of length equal to the length of the input vector. Both W_H and W_I change through the time according to which modules are active at the given time step as stated in the following equations:

$$W_{H_i} = \begin{cases} W_{H_i} & \text{if } (t \ mod \ T_i) = 0 \\ 0 & \text{otherwise,} \end{cases} \tag{2.17}$$

this again shows that the only active modules are the ones that are supposed to run at the given time step. An example of this calculation can be seen on the Figure 2.8. As for the backward pass it is again similar with the SRN. The only difference is that the error propagates only from the modules that were active at the given time step. For the others the error gets copied back in time.

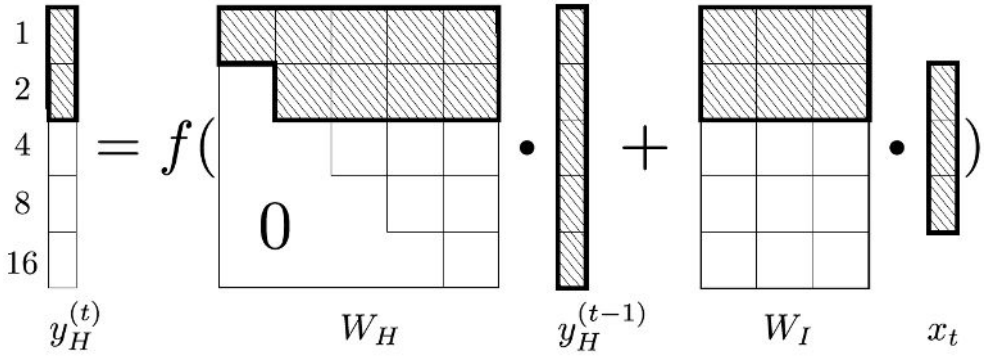

Figure 2.8: Calculation of the hidden layer activation at the time step t=6. The numbers on the left side are the clock periods and thus only the first two modules are active.[5]

According to the authors of the model and the results published in the paper, the Clockwork RNN shows better performance than the SRN on some tasks requiring mid-term context like sequence generation or spoken word classification. For spoken word classification it even shows a better performance than the LSTM network, which shows a potential hidden behind this model. Another big potential of this model lies behind the fact that the Clockwork RNN performs for the same amount of neurons fewer actions per time step than the SRN. This gives us the possibility to use a bigger network without needing additional runtime. Since the human brain consist out of billions of neurons, this might actually be one of the concepts that will allows us to get closer to its simulation.

2.2.4 Memory Networks

The Memory Networks were actually the reason why I started working on the topic of neural networks with memory. After reading its research paper I thought that it has to be something unique and revolutionary. But after reading few other research papers from the area I found out there is more to that and that it actually might be worth to study the whole family of models and not just only one. And that was probably the moment when the main idea of this thesis topic was born. After a deeper study I realized that the basic Memory Networks published by Weston et al.[14] are first of all not as revolutionary as I thought, but also has some downs that are seriously limiting it.

As about the fact that the network is nothing completely new, we can just simply look back on the DISCERN model published by Miikkulainen in 1990. The DISCERN was, same as the Memory Networks, a multi component system that was proposed for processing of natural language in means of QA systems. But what troubles me more than the revolutionary character of the model, is the fact how it is trained. The basic Memory Networks are not able to be trained end-to-end and thus require supervision at each layer of the network. This means that the model can't easily be trained using backpropagation and thus there are limitations about its general applicability. But only few months after the publication of the basic Memory Networks, their end-to-end variant was proposed by Sukhbaatar et al.[24] and thus I would like to focus on this variant instead of the basic one.

The End-To-End Memory Networks (MemN2N) are a form of Memory Networks, which is able to be trained end-to-end and thus requires less supervision during the training phase. Before producing any output the network first performs multiple reads from the long-term memory, which appears to be important for a good performance. In short the network works in the way that the MemN2N model takes a set of inputs $x1, \ldots, xn$ and stores them in the memory. Once it processes a query q, it creates a continuous representation of both the input x and the query q. Based on the continuous representation it can afterwards create the output a. All the symbols contained in the input x, query q and answer a must be stored in a predefined dictionary of words.

To be easier to understand how the networks work I will first describe just as a single layer as viewed on the Figure 2.9a. Given the input set $\{xi\}$ we create memory vectors $\{mi\}$ and an output vector $\{ci\}$, both by embedding the x into a continuous space using matrices A and C. We also embed the query q into an internal state u and create the match p_i between u and each memory m_i as follows

$$p_i = Softmax(u^T \cdot m_i), \text{ where } Softmax(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}. \tag{2.18}$$

The vector p created using these operations is a probability vector over the inputs. Using this probability vector p we can create the response vector o as follows

$$o = \sum_i p_i \cdot c_i \tag{2.19}$$

The whole transformation function from the input to the output is smooth and thus we are able to easily compute gradients and backpropagate through it. Based on the response vector o and the internal state u we can finally create the predicted label $\hat{a}$ as follows

$$\hat{a} = Softmax(W \cdot (o + u)), \tag{2.20}$$

where W is the final weight matrix. During the training of the network all matrices A, B, C and W are learned by minimizing the standard cross-entropy loss between the predicted label $\hat{a}$ and the true label a.

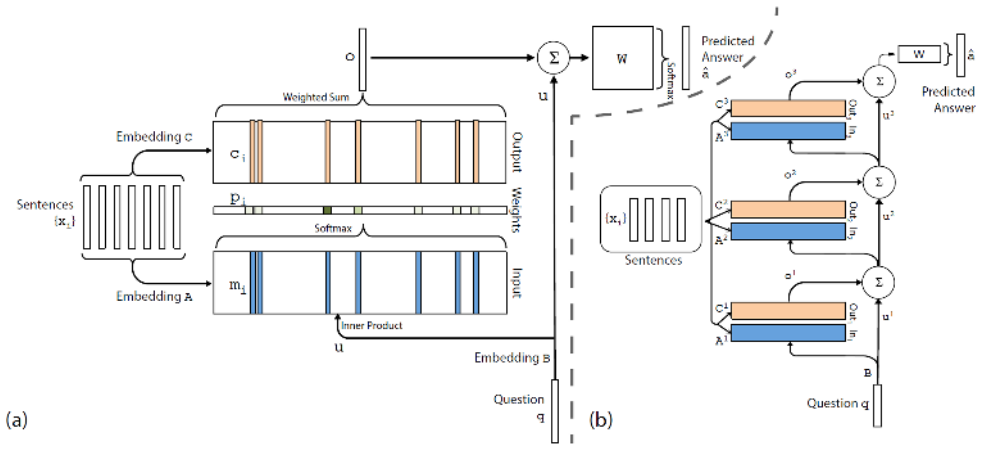

Figure 2.9: Visualization of the MemN2N model with A: one/B: multiple layers[24]

One layer means one search based on the current internal state. We can extend the model by connecting multiple layers. Let's assume we have K layers connected together, then the input for the next layer is calculated as a sum of the output o^k of the current layer and the inner state u^k of the current layer as follows

$$u^{k+1} = o^k + u^k, \tag{2.21}$$

where the k means the current layer. Every layer has its own embedding matrices A^k and C^k and they are modified during the training process. The predicted label $\hat{a}$ is then computed as follows

$$\hat{a} = Softmax(W \cdot u^{K+1}) = Softmax(W \cdot (o^K + u^K)). \tag{2.22}$$

In the paper the authors are also exploring two types of weight tying between matrices A^k and C^k in order to ease the training and reduce the number of parameters. Those two types of weight tying are:

1. **Adjacent**: The input embedding matrix of each layer is the same as the output embedding matrix of the layer before and thus $A^k = C^{k-1}$. In the same way the query embedding matrix B and final weight matrix W are also limited and thus $B = C^0 = A^1$ and $W = A^{K+1} = C^K$.

2. **Layer-wise**: The input and output embedding matrices are the same across all the layers and thus $A^1 = \ldots = A^K$ and $C^1 = \ldots = C^K$. The authors also propose the use of a linear mapping H alongside the layer-wise tying, which is used for updating the inner state u between the layers hops as $u^{k+1} = H \cdot u^k + o^k$ and is among the parameters that are learned during the training phase.

According to the result published in the paper the proposed MemN2N model not only significantly makes the network easier to use, but at the same time also shows better performance for some of the Question and Answering problems. Even though there are some interesting thoughts that were proposed in both the Memory Networks related papers[14][24], the network model on its own has a really restricted field of use, being built specifically as a QA system. I am currently not sure how the proposed model could be applied to a more universal use and thus I will not continue to work with this model in the rest of my thesis.

2.2.5 Stack RNN

There have been attempts to let the neural networks work with a stack structure since the early 1990s. Mostly they have been focusing on letting the neural networks to learn how to act as some sort of universal automatons which would be able to accept all sorts of grammars based on the data they were presented during the training stage. One of the latest researches in this field is the Stack RNN model presented by Joulin and Mikolov[7]. As you can see on the Figure 2.10, the authors proposed a recurrent neural networks working with a stack or a double linked list in order to be able to learn the regularities in sequences of symbols generated by simple algorithms like $a^n b^n$ and so on. In this section I would like to mainly focus on the RNN with stack as it shows a longer history of research and according to the authors also a better performance.

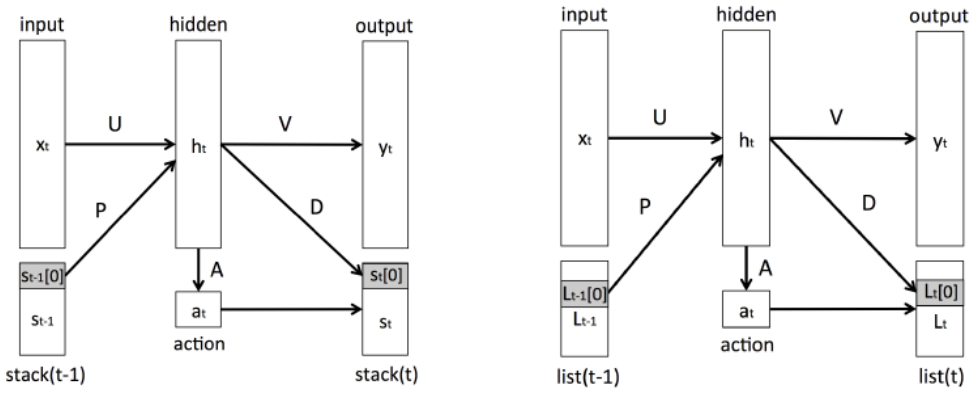

Figure 2.10: Left: Visualization of the Stack RNN model, Right: Visualization of the List RNN model, where a double linked list is used instead of the stack

But not to be mistaken I don't want to focus in the section on the performance of the Stack RNN in the proposed field of expertise, but mainly on its contribution in the field of neural networks with memory. Because the proposed network is not only learning to recognize the presented patterns, but also learns its own way to control the stack. This gives the network more freedom and at the same time makes the network more powerful. The used external memory in an infinite stack structure with three typical control operations. The POP operation which removes the top element of the stack, the PUSH operation which adds a new element to the top of the stack and the NO-OP operation which does nothing and thus allows the stack to simply keep the same state into the next iteration.

The decision which operation is going to be performed is made based on the 3-dimensional variable a_t which is computed from the current hidden layer activity h_t as follows

$$a_t = f(A \cdot h_t), \tag{2.23}$$

where A is a $3 \times n$ transformation matrix (n is the number of neurons of the hidden layer) and f is a softmax function. We set $a_t[\text{POP}]$ as the probability for the POP action, $a_t[\text{PUSH}]$ as the probability for the PUSH action and $a_t[\text{NO-OP}]$ as the probability for no action. The stack is at time t stored in a vector s_t, where the size of the vector can be increased if needed. At every time step we need to perform the operation on the stack using the following equations

$$s_t[0] = a_t[\text{PUSH}] \cdot \sigma(D \cdot h_t) + a_t[\text{POP}] \cdot s_{t-1}[1] + a_t[\text{NO-OP}] \cdot s_{t-1}[0], \tag{2.24}$$

$$s_t[i] = a_t[\text{PUSH}] \cdot s_{t-1}[i-1] + a_t[\text{POP}] \cdot s_{t-1}[i+1] \text{ for } i > 0, \tag{2.25}$$

where D is a $1 \times m$ transformation matrix and $\sigma(x) = \frac{1}{1+e^{-x}}$ is a sigmoid activation function. These equations mean that if a_t[PUSH] equals 1 then the whole stack is shifted one down and a new value is added to the top of the stack. On the other way around if a_t[POP] equals 1 then the whole stack is shifted by one up and thus the top element of the stack is removed. If the a_t[NO-OP] equals 1 then the top element stays the same and no shifting is performed. When the stack is empty then s_t is set to -1.

Using the information stored in the stack the hidden layer activity h_t is updated as follows

$$h_t = \sigma(U \cdot x_t + R \cdot h_{t-1} + P \cdot s_{t-1}^k), \tag{2.26}$$

where U is token embedding matrix, x_t is the input at the time t, R is a matrix of recurrent weights,P is a $m \times k$ recurrent matrix and s_{t-1}^k are the top k elements of the stack at time $t-1$. The choice of k is one of the parameters of the network. Using one stack has some limitation especially if we take into consideration that only 1 action can be performed at a time. The authors thus propose a model that can use multiple stacks in parallel. That allows the model to be able to learn how to process more complex patterns.

One of the limitations of this model is that the stack can be only accessed through its top most element and thus the network can't work with an earlier stored context without processing the newer ones first. This doesn't limit the network while processing the sequences of symbols generated by simple algorithms like $a^n b^n$, but might be a problem if we were trying to deploy this structure to solve some other problems. Another limitation of this model is that it deploys the external memory on the scope of the whole network. This may increase the ability to learn context of the network as the whole, but doesn't help to increase the ability of individual neurons or groups of neurons to learn mid-term context. This might be useful, because different parts of the neural network might be thanks to that learning contexts of different term lengths. These limitations might not be a problem for some types of tasks, but if we would try to focus on an universal use of the network then they might limit this model too much to be able to use it.

2.2.6 RNN-EM

The idea to allow the hidden layer of the neural network to retain the information about its past states longer is not nothing new. Similar concepts have already been proposed in the early 1990s, for example as in NARX model by Lin et al., where hidden layer contained few delay units that have been

feeding the past states of the hidden layer. But models like this have been limited in the matter of number of the past inner states they could retain. The RNN-EM model offers an upgrade in this matter since it doesn't limit the number of past inner states that can be retained. In the following paragraphs I will try to give a detailed view into how the RNN-EM actually works. But before we will be able to move onto that, the background needs to be explained.

In 1997 the Long short-term memory (LSTM) have introduced the gating functions inside of its memory blocks in order to prevent the back-propagated errors to vanish or explode. The memory block of the LSTM contains 3 different gating functions, input, output and forget, which have already been in deep introduced in the section about LSTM (2.2.1). But already in the past, some of the researchers have been asking questions whether it is really necessary to have all three gates. This question has been answered in 2014 by Cho et al.[25] proposing a new hidden unit, usually by others cited as the gated recurrent unit (GRU), that contains only two gates.

Each gated recurrent unit has its own separate two gates, which are named reset and update gate. The reset gate decides whether the previous hidden state is ignored or used. The update gate is used to control whether the hidden state will be updated with a new hidden state. Together these two gates act similar to the ones used in memory cells of the LSTM and help the network to remember the mid-term dependencies. But thanks to the fact that every unit has its own gates, every unit can learn to capture different dependencies. Units capturing short-term dependencies will tend to have reset gates being active more frequently, while the ones capturing mid-term dependencies will have their update gates mostly active.

With having the main part of the background covered we can start the exploring of the Recurrent Neural Network with External Memory (RNN-EM). Like LSTM and GRU models, the RNN-EM also proposes a model with gating units, but in this case they serve for controlling the external memory that is connected to the hidden layer. This allows the network to keep a longer context tracking, with the context being fed from the external memory content and not directly from the past hidden layer activities as for the classic RNN. The weight of the fed external memory content is calculated based on how relevant it is to the current hidden layer activity.

On Figure 2.11 you can see a model of the RNN-EM which is actually classic RNN model supplemented by the external memory M_t, forget gate f_t, update gate u_t and their supporting vectors and operations. On the top line you can see the standard components as input x_t, hidden layer activity h_t and output y_t. If we would put the context c_t equal to h_{t-1} and would omit the rest of the components, than we would get a model of simple recurrent neural network (SRN). But the more interesting parts are the ones where there are performed

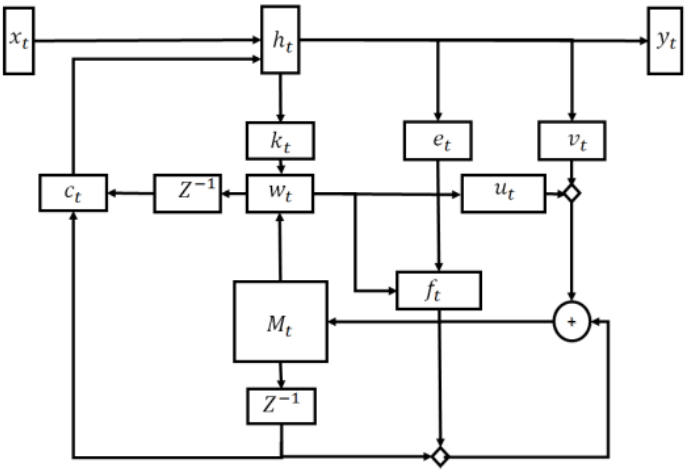

Figure 2.11: The RNN-EM model. The Z^{-1} denotes a time delay operator.[8]

any operations on the content of the external memory. We could divide those into two basic parts: read and update.

The external memory M_t of the RNN-EM is an memory matrix of n slots where each slot contains a vector of m elements. If we want to select the most relevant context from the external memory we do so based on the current hidden layer activity and the content of the external memory. First of all we need to create relevant weight vector w_t. This is done using the key vector $k_t = W_k \cdot h_t$, which is nothing else than the current hidden layer activity h_t transferred using a linear transformation matrix W_k into a vector of desired length. Once we have the key vector k_t we can compare it to the context vectors stored in the memory using the cosine distance $K(u, v) = \frac{u \cdot v}{\|u\| \cdot \|v\|}$ and get the weight for each context vector $\hat{w}_t(c)$ as follows

$$\hat{w}_t(c) = \frac{exp(\beta_t \cdot K(k_t, M_t(:, c)))}{\sum_q exp(\beta_t \cdot K(k_t, M_t(:, q)))}, \tag{2.27}$$

where β_t is a scalar calculated as follows

$$\beta_t = log(1 + exp(W_\beta \cdot h_t)) \tag{2.28}$$

and either sharpens the weight vector when it's bigger than 1.0 or smooths/dampens the vector when it's between 0.0 and 1.0. The temporal weight vector $\hat{w}_t(c)$ can be then used to compute the final weight vector w_t as follows

$$w_t = (1 - g_t) \cdot w_{t-1} + g_t \cdot \hat{w}_t(c), \tag{2.29}$$

where g_t is a scalar coefficient used to interpolate the temporal weight vector with the past weight vector. The relevant context c_t can be retrieved from the

external memory using the weight vector as follows

$$c_t = M_{t-1} \cdot w_{t-1}. \tag{2.30}$$

As for the update of the external memory the most two important components are the forget and update gate. The forget gate f_t acts as follows

$$f_t = 1 - w_t \odot e_t, \tag{2.31}$$

where e_t is an erase vector generated by linear transformation from the hidden layer activity h_t. From the equation can be observed that the c-th element of the forget gate is 0 if and only when both c-th elements of the read weight w_t and the erase e_t vectors are set to one. This means that a memory can only be forgotten if it is to be read. The update gate just uses the read weight vector w_t as follows

$$u_t = w_t \tag{2.32}$$

and thus a memory can be updated only if it is to be read. With the given forget and update gates the external memory is updated as follows

$$M_t = M_{t-1} \cdot diag(f_t) + v_t^T \cdot u_t, \tag{2.33}$$

where $diag(\cdot)$ is a transformation, which takes a vector and creates a diagonal matrix with the diagonal elements taken from the vector.

According to the original paper the RNN-EM is proposed mainly for the problem of language understanding where its performance according to the test results presented by the authors is better than for some of the previously presented models including the SRN, the LSTM or the network of gated recurrent units (GRNN). But to my opinion its biggest idea lies in the usage of external memory together with the gating units. This feature allows the network to retain a context that would normally have to be retained by a bigger network. This similarly like the Clockwork RNN allows us to simulate a bigger network using a smaller one and might be the right path in progressing in the matter of simulation huge neural networks on today's personal computers.

Chapter 3

Research of improvement

In the last chapter I have introduced many models of neural networks with memory that have been proposed over the past 25 years. Some of them had bigger impact than others, for some of them their time can even still come. But before I will move forward I would like to talk a little bit about why do I think that adding external memory to the networks or making the networks act that they provide context to themselves might be the way to push the research in the field of the neural network forward.

Most of the above presented models are not build in the way how it works in the biological world. But to be honest we still don't know how it actually works in the real biological world and we probably do not have the computing resources to build something so complex and big as the nature does. Some can say that it is not natural to add additional external memory to the neural networks. I don't want to argue about this here, but I am going to say that adding memory has its place until we prove there is another way to get around. Because in my opinion adding external memory to the neural networks is currently the best way how to simulate those complex systems we do not understand.

The external memory provides a lot to the neural networks especially in the means that it allows us to simulate bigger networks using a smaller ones. And how does this work? In general the external memory allows the neural network to maintain a longer context which normally have to be retained using a bigger network. Thus using the external memory allows us, if used wisely, to reduce the size of the neural network in some way. The main goal in the neural networks field is to be able to simulate the human brain using an artificial neural network. This is currently impossible because of the limitations in means of computing resource. But what if we were able to reduce the size of the network or at least its computing load?

While reading the particular research papers I came across an interesting fact that most of the neural networks, deploying an external memory, deploy it on the scope of the whole network. This fact lead for me to a lot of questions whether there could be a model that would be deploying the external memory on the scope of individual neurons or groups of neurons. An advantage of such a model could be that different groups of neurons might be used to track contexts of different lengths. Similar ideas to this one were introduced in models like the Long Short-Term Memory, where the neurons were organized into the memory blocks, or the Clockwork RNN, where the neurons were organized into the modules. Within those groups the neurons share more context than outside of them and thus this context could be replaced by some external memory. But probably not all the models with the external memory could be modified to work this way.

A model that caught my attention was the RNN-EM model by Peng and Yao[8]. This model is being strongly inspired by the gating units proposed in the Long Short-Term Memory or the Gated Recurrent Unit models. Both of those models deploy the gating units on the scope of the individual neurons or groups of them. This inspired me to consider this model for the above mentioned modification. Big advantage of the RNN-EM model is that the external memory has predefined size and thus can't grow during the computation like for example in many of the stack based models. The limited memory size might allow us to deploy the memory into the network not only once but multiple times. Thus in the rest of this thesis I would like to focus mainly on just one model and that would be the newly proposed model of the **Recurrent Neural Modules with External Memory**.

3.1 Recurrent Neural Modules with External Memory

The model of Recurrent Neural Modules with External Memory (RNM-EM) is strongly influenced by the model of Recurrent Neural Network by Peng and Yao[8] and thus also the almost same name. The whole network consists out of multiple modules, where each module a slightly modified RNN-EM network. But deploying only multiple modules without any recurrent edges wouldn't probably lead to any major improvement or even a big contribution to the family of neural networks with memory, even though it would allow the network to work significantly faster than the RNN-EM with the same total number of hidden layer neurons if the modules would be implemented in the way to work in parallel. Thus I also propose to implement a recurrent propagation of the combined context $R(t)$ as viewed on the Figure 3.1.

As already previously mentioned the RNM-EM model consists out of k modules, where each modules is a modified RNN-EM network. At each time step

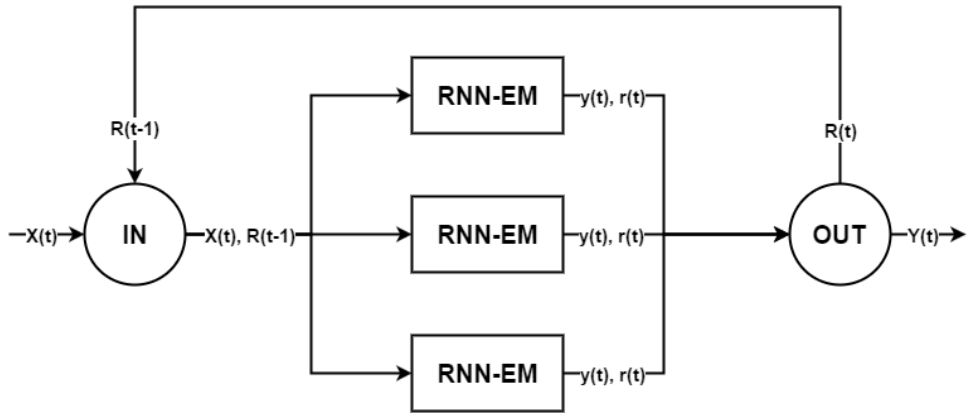

Figure 3.1: Model of the RNM-EM network with 3 modules.

t all the modules receive the input $X(t) \in R^{d\times 1}$ and the combined context $R(t-1)$ which was generated by the modules in the previous time step. After the modules process the input and the context, each of them produces its output $y_i(t)$ and context $r_i(t)$. These partial outputs and contexts are afterwards combined into the output of the network $Y(t)$ and the combined context $R(t)$ as follows

$$Y(t) = g(\sum_{i<k} y_i(t) + b_y), \tag{3.1}$$

$$R(t) = \sum_{i<k} (W_i \cdot r_i(t)) + b_r, \tag{3.2}$$

where g is a softmax function, W_i is a matrix assigned to the particular module and b_y and b_r are biases for the equations. The dimensions of the combined context $R(t)$ and contexts $r_i(t)$ will be further described in the chapter Implementation (4). The dimensions of the outputs $Y(t)$ and $y_i(t)$ depend on the size of the desired output. The rest of the computation takes place inside of the modules.

On the Figure 3.2 you can see how each module internally works. The model of the module is really similar to the one published in the RNN-EM paper. The two biggest differences in the visualization are the context r_t and the circles with the question marks next to the external memory M_t and the memory weight w_t. These circles represent a place that might contain a delay node Z^{-1}. Exactly one of these places needs to be assigned a delay node in order to set the dependency between the memory M_t and the weight w_t, which is not explained in the RNN-EM research paper. This matter will be later discussed in the chapter Implementation (4). Note that the visualization

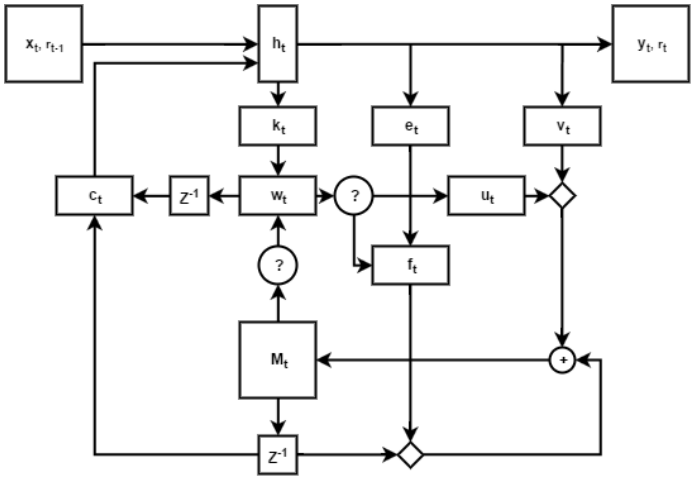

Figure 3.2: The RNM-EM module model. The Z^{-1} denotes a time delay operator and the '?' denotes a place, where an additional time delay operator might be placed.

doesn't show how the outputted context r_t is produced. This will be later described in this and the next chapters.

Even though most of the equations are similar to the ones previously stated in the section about the RNN-EM (2.2.6), I will summarize the computation process once again. The reason for this is the fact that I will describe them in ordered and more detailed way than they were previously mentioned. I would also like to put all the changed equations into a proper context rather than just pointing out the differences, especially because these equations should be the main source of information if someone would be trying to reproduce the proposed model.

Before I will move further I first need to explain few constants that will appear in the following paragraphs. The RNM-EM model consist out of k identical modules. Each of these modules contains p neurons in the hidden layer and an external memory M_t that consists out of n memory slots with each being a vector of m elements. The last constant, that will appear in the following equations and will not be explained further, is the input vector's length d.

The computation itself could be divided into 3 major parts: the hidden layer and output computation, memory read and the memory update. I will start with the computation of the context $c_t \in R^{m\times 1}$ and the hidden layer activity $h_t \in R^{p\times 1}$ that are computed as follows

$$c_t = M_{t-1} \cdot w_{t-1}, \tag{3.3}$$

$$h_t = \sigma(W_x \cdot x_t + W_c \cdot c_t + W_r \cdot r_{t-1} + b_n), \tag{3.4}$$

where σ is an activation function (tanh or sigmoid), $W_x \in R^{p \times d}$ is the input weight matrix, $W_c \in R^{p \times m}$ is the context weight matrix, W_r is the combined context weight and b_n is a bias. From the hidden layer activity the output of the module y_t can be simply computed as follows

$$y_t = W \cdot h_t, \tag{3.5}$$

where W is the weight to the hidden layer activity.

In the memory read part the weight to the memory content w_t needs to be calculated. The module has an external memory $M_t \in R^{m \times n}$. To read the memory the weight $w_t \in R^{n \times 1}$ is used. This weight w_t is updated at every time step using the memory M_a and h_t, where a is either t or $t-1$ based on where we place the delay node. The value of a will be based on the tests described and performed in the further chapters. First of all the key vector $k_t \in R^{m \times 1}$, which is used to search the content of the external memory, is calculated as follows

$$k_t = W_k \cdot h_t, \tag{3.6}$$

where $W_k \in R^{m \times p}$ is the weight matrix to the hidden layer activity. We compare this key vector using the cosine distance $K(u, v) = \frac{u \cdot v}{\|u\| \cdot \|v\|}$. Based on the cosine distance we create the weight estimate $\hat{w}_t$ as follows

$$\hat{w}_t(c) = \frac{exp(\beta_t \cdot K(k_t, M_a(:, c)))}{\sum_q exp(\beta_t \cdot K(k_t, M_a(:, q)))}, \tag{3.7}$$

where the β_t is a scalar which is used to sharpen or smooth the weight vector, depends whether it is bigger or smaller than 1.0, and is computed as follows

$$\beta_t = log(1 + exp(W_\beta \cdot h_t)), \tag{3.8}$$

where $W_\beta \in R^{1 \times p}$ is a weight matrix that maps the hidden layer activity to the scalar. The final weight vector w_t is then computed as follows

$$w_t = (1 - g_t) \cdot w_{t-1} + g_t \cdot \hat{w}_t, \tag{3.9}$$

where g_t is a scalar that is used to the interpolate the weight estimate $\hat{w}_t$ with the past weight and is computed as follows

$$g_t = \sigma(W_g \cdot x_t + b_g), \tag{3.10}$$

where σ is a sigmoid function, W_g is weight to the input vector and b_g is a bias. This equation might be modified and will be a subject to testing in the further chapters.

The last part of the calculation is the external memory update. The update of the memory is performed based on the two gating units: the forget and update gates. The forget gate f_t is computed as follows

$$f_t = 1 - w_b \odot e_t, \tag{3.11}$$

where b is either t or $t-1$, based on where we place the delay node. The value of b will be based on the tests described and performed in the further chapters. The $e_t \in R^{n\times 1}$ is an erase vector that is calculated as follows

$$e_t = W_e \cdot h_t + b_e, \tag{3.12}$$

where $W_e \in R^{n\times p}$ is the weight to the hidden layer activity and b_e is a bias. The update gate $u_t \in R^{n\times 1}$ simply uses the weight vector w_b as follows

$$u_t = w_b. \tag{3.13}$$

The update of the external memory M_t itself can be then performed as follows

$$M_t = M_{t-1} \cdot diag(f_t) + v_t^T \cdot u_t, \tag{3.14}$$

where $diag(\cdot)$ transforms a vector into a diagonal matrix with diagonal elements from the vector, v^T means transposition of the vector v and v_t is a new content vector that is calculated as follows

$$v_t = W_v \cdot h_t + b_v, \tag{3.15}$$

where $W_v \in R^{m\times p}$ is a weight to the hidden layer activity and b_v is a bias.

The last thing that needs to be described in this chapter is the context r_t the module returns, since it was in the previous paragraphs omitted on purpose. Since this is something new and thus has no background in the RNN-EM model I have decided to put this under testing before deciding what is the best value to use as the returning context. The most obvious would be to use the retrieved context c, but the question is from which time. Should it be the context c_t used to calculate the current step or should it be the context c_{t+1} that will be used for the next step? This matter will be deeper described in the chapter Implementation (4).

Even though the proposed model might not seem to be something super revolutionary, it sets an important background that is necessary for a further research in the field of neural networks with memory. Thus even if the model wouldn't show a significant increase in performance in compare to the RNN-EM model with similar number of parameters, it would still have a significant contribution for the research of neural networks with memory, since it is unique in a way that it deploys multiple external memories within a single neural network.

The last thing I would like to mention before I will move on to the implementation is the fact that the model described in this chapter is not final, but will be evolving based on testing. In the above described equations there is actually a lot of room for modification and also network setting. More on this will be described in the upcoming chapters.

Chapter 4

Implementation

In the previous chapter I have proposed the model of the Recurrent Neural Modules with External Memory. In the model there was a lot of place for modifications, which is something that I will focus on in this and the following chapter. But before I will move onto that, I first need to describe the platform I will be implementing the prototype of the model on.

4.1 Selection of the platform

From the very first beginning I knew it will be better to implement the network on some platform which is widely used for the neural networks. Currently there are a lot of available platforms thanks to the big boom the artificial intelligence is experiencing in the last few year. After a brief research I came up with 4 platforms I was deciding between: Brain Simulator, Theano, TensorFlow and Torch. In the following paragraphs I will try to point out the pluses and minuses of each platform.

4.1.1 Brain Simulator

Brain Simulator[26] is an artificial intelligence platform being developed by GoodAI, company based in Prague, Czech Republic. Being build on C# platform and NVIDIA Cuda technology, it provides a powerful and fast platform for artificial intelligence tasks with a great graphical user interface. A big advantage of the Brain Simulator is the fact that I am already familiar with this platform thanks to the summer internship I had at GoodAI last summer, where I performed the previously mentioned research[22] on the Liquid State Machine. But the problem with the Brain Simulator platform would be that it is a platform that only few people outside of the company uses and thus is not well spread among the other researches.

4.1.2 Theano

Theano[27] is a Python based library primarily developed by the people of the Montreal Institute for Learning Algorithms at the Université de Montréal. Theano offers an easy to use library for machine learning for artificial intelligence with the support for the NVIDIA Cuda technology. But the biggest power lies in its community, since Theano is probably the most spread machine learning platform among the researches. This not only offers many third party additional libraries and functions, but also a lot of working examples of networks, including most of the networks covered in the section Detailed Analysis (2.2). For example the implementation of the RNN-EM model, that was used for the testing in the main paper, was build on the Theano library.

4.1.3 TensorFlow

The newest from the platforms is the TensorFlow[28] platform developed by the Google Brain team. The TensorFlow proposes an interesting concept of data flow diagrams, where the whole network is viewed as a graph. The nodes of the graph are the mathematical operations and the edges are tensors, multidimensional data arrays. Even though the platform proposes APIs for both Python and C++ and is supported by a big research team of the Google Deepmind, it faces a similar problem as the Brain Simulator platform. Since the platform was publicly released under an open source licence in November 2015, it still is not as widespread as for example the Theano platform.

4.1.4 Torch

Another platform that is widely used among the researchers is the Torch[29] platform primarily developed by Collobert et al., researches from several companies including Facebook AI or Google Deepmind. Even though this platform is widely supported by researchers, even within companies like Facebook AI or Google Deepmind, and using a third party software allows the usage of NVIDIA Cuda technology, it shows two major disadvantages for me. First of them is the fact that this framework is based on the Lua programming language which I have never came across before. The second one is the fact that this platform does not work on the Windows platform. This might not be such a big obstacle, but it is something nice to have, even though I will probably be developing mainly on the Linux platform anyway.

4.2 Theano

Even though I have a significant experience with the Brain Simulator platform I have decided to use the Theano[30][31] platform instead. The main reasons behind this decision is that the Theano platform is really widespread among the AI research community and there are also many third party libraries and networks build on it. This is especially seminal in the current phase

of the research where I need to build the prototype of the RNM-EM model and compare it to other networks. In the rest of this section I will mainly talk about the third party libraries and network implementations I am going to be reusing in my work.

4.2.1 Used third party libraries

Lasagne[32] is a neural network library build on Theano. It introduces the functionality to build and train several neural networks. In the RNM-EM prototype I am using this library for training purposes. Namely for the updating phase of all the trainable variables (weights, biases, etc.) using the Adadelta algorithm, which will be further described in the section about RNM-EM implementation (4.4).

Another Python library I am using is the **Pydot**[33] library. Pydot is a library for generating of graphs from the inner graph flow of the computation. It is printing the graphs using the GraphViz tool. I am using the generated graphs mainly for the debugging and testing purposes, but they are also great for the visualization of the computational data flow. Unfortunately these graphs tend to be large, especially when using multiple modules, and thus I am not showing any of them in the text here.

4.2.2 Used implementations

For the testing purposes I am reusing few networks implementations. The probably most important one is the Nissan Pow's implementation[34] of the RNN-EM network. Even though the original testing implementation in the RNN-EM paper was implemented in Theano, the code hasn't been publicly posted. Thus I am using the Nissan Pow's implementation as the starting point for my prototypes of the RNN-EM and RNM-EM networks. I have compared this implementation to the one implemented[35] on the CNN neural network library by Kaisheng Yao, the author of the RNN-EM paper, and found just a few minor differences I am going to describe in the later chapters.

For the Comparison testing I am using two other implementations except the RNN-EM and RNM-EM ones and those are the Elman network and LSTM. Even though both of the implementation I have build by modifying the Pow's RNN-EM code, I have been using another implementations as sources. For the Elman network I have been using the Grégoire Mesnil's code[36] and for the LSTM the Charles Ollion's code[37] and the LSTM Networks for Sentiment Analysis online tutorial[38].

4.3 RNN-EM

As already previously mentioned, I am using the Nissan Pow's RNN-EM[34] implementation as the starting point for the prototyping in the current re-

search phase. At this point it is for me more important to test the performance of the proposed model rather than implement my own version of the code, that would anyway be almost the same. Comparing the implementation to the original research paper[8] and the code[35] published by the authors of the original paper, I have discovered two minor changes I considered worth testing.

The first one was the fact that for some reason instead of the cosine distance in the equation 3.7 as mentioned in the paper, the implementation uses "1 − cosine distance". The second and more important change is the activation function in the equation 3.4 for computing the hidden layer activity. In the Testing chapter (5) I will put these changes under a testing to see whether they improve the performance of the network or not.

I am additionally also proposing few modifications that could have an impact on the performance of the model. The first modification is in the computing of the scalar g_t from the equation 3.10. The equation 3.9, where the scalar g_t is used, is similar to the equation 7 from the GRU paper[25] that looks as follows

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \hat{h}_t, \tag{4.1}$$

where z_t is the update gate, a vector that is used to update the hidden layer activity, and is computed as follows

$$z_t = \sigma(W_{xz} \cdot x_t + W_{hz} \cdot h_{t-1}). \tag{4.2}$$

The equation for computing the g_t hasn't been proposed in the original research paper, but has been written in both the Pow's and Yao's implementation as follows

$$g_t = \sigma(W_g \cdot x_t + b_g). \tag{4.3}$$

Being influenced by the equation for the update gate z_t from the GRU model, I would like to test how would the performance of the network change if I would also base the scalar g_t on the past memory weight w_{t-1}. Thus the equation, I will put under the testing, will look as follows

$$g_t = \sigma(W_g \cdot x_t + W_i \cdot w_{t-1} + b_g). \tag{4.4}$$

Probably the most important model modification, I would like to test, is the relationship between the hidden layer activity h_t, the memory weight w_t and the memory content M_t, mainly on the time scale. All of those three are dependant on the rest of them, but the original paper doesn't fully cover how in the means of the time scale. I have previously already mentioned this when I was talking about the need to put a delay node between the memory weight

w_t and the memory M_t. The rest of this section I will talk about this problem and all the possible solutions I am proposing.

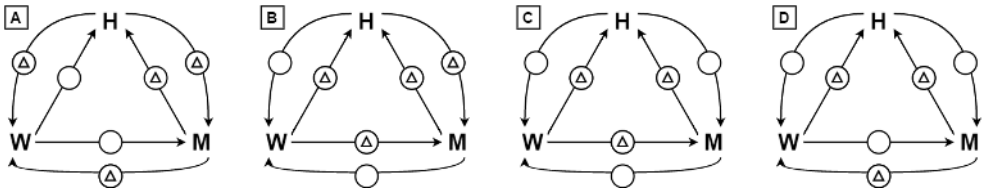

Figure 4.1: Visualization of the four possible models of the computation order inside of the RNM-EM module. The Δ denotes a time delay operator.

On the Figure 4.1 you can see a visualization of the possible examples of relationships in the computation order between the hidden layer activity H, the memory weight W and the memory M. Even though there are 6 possible places to put the delay node in the model and thus 2^6 total options, with omitting equivalents and nonsenses I propose here these 4 models:

1. Models **A** and **B** – The first two models are actually equivalents, the only difference is at which time step the memory weight W is computed and thus the edges of node W from model A are the opposite as on the model B. The model A is the one proposed in both the Pow's and Yao's implementations. If I would summarize this model in the means of functions and the order of computation it would look like this: $W_t(H_{t-1}, M_{t-1})$, $M_t(H_{t-1}, W_t)$ and $H_t(W_t, M_{t-1})$. While for the model B the memory weight is computed as the operation of the previous time step and thus the order of computation would look like this: $H_t(W_{t-1}, M_{t-1})$, $M_t(H_{t-1}, W_{t-1})$ and $W_t(H_t, M_t)$. Even though the model B might look unnecessary at this moment it will be further used in the RNM-EM model.

2. Model **C** – Model C is a simple modification of the model B with making the memory M_t dependant on the hidden layer activity H_t from the same time step instead on the past one. This modification also allows me to drop the hidden layer activity H from the recurrence input and output since it is not needed anymore.

3. Model **D** – Even though it looks more natural to depend the memory weight W_t on the same memory M_t as it will be later applied on to get the context c_{t+1}, I have decided also to test the option where the memory weight W_t depends on the memory $M_t - 1$ from the last time step and thus the memory M_t depends on the weight W_t from the same time step.

Performances of these models will be further tested in the Testing chapter (5) and the models will be also extensively used in the RNM-EM model.

4.4 RNM-EM

Before I will move any further I would like to mention that this implementation is just a prototype created in order to test the proposed model and evaluate whether it is worth a more detailed research. The probably two biggest limitations of this prototype are the non-parallelism on the level of modules and the partially static construction of the network. But none of these limitations influences the testing of the performance of the model.

One of the big advantages of this model is the fact that if fully implemented on a parallel scheme, it will for the same total amount of hidden layer neurons have a smaller real time complexity than the RNN-EM model. This happens thanks to the fact that the equations inside the modules are computationally independent and thus can be computed at the same time, having a sufficient number of computing resources. Since this is something that doesn't need to be tested I am not computing the modules in this prototype in parallel, but one by one.

The second limitation is the static construction of the network in means of the number of modules. This limitation is put on the network since I haven't been easily able to overcome the limitations inside of the Theano platform that are set for assembling tensors together (the theano.tensor.stack function), returning lists of tensors as the parameter of scan function or the learning of the trainable variables. The static construction is made in a way that it allows me to easily add more modules or delete some and thus doesn't limit me for the testing phase.

4.4.1 Description of the prototype

The prototype of the RNM-EM model is based on the implementation of the RNN-EM model. The model implementation itself consists out of two basic parts, the network and module models. The module model is a slightly modified RNN-EM model and thus I will not give it a lot of attention in the following text. The network model is something something completely new and thus I would like to describe it a little bit further.

But before I will move onto the network model itself, I need to talk a little bit about the input parameters of the network. This is an important part because without it, it would be hard to understand all the variables in the following network model pseudocode and the trainable variables in the next chapter. In the Table 4.1 you can see all the input parameters of the network. Note that some of them depend on the structure of the desired input and output data and thus can't be changed as freely as the other ones.

Table 4.1: Table of the input parameters of the RNM-EM network.

name	description
mc	number of RNN-EM modules
ms	number of neurons in each module
nc	output size, number of classes of the dataset
ne	number of embeddings in the dataset vocabulary, used for the embedding matrix
de	dimension of the word embeddings, first part of the input size (de*cs)
cs	size of the word context window, second part of the input size (de*cs)
m_size	size of the memory vectors
m_slots	number of the slots for memory vectors

As can be seen in the pseudocode in Algorithm 1, the one computational step of the network consist out of few simple substeps. In the first part the subresults from the modules needs to be generated, which can be seen on the lines 3 – 4. In this version the subresults from the modules are calculated in a sequential order, but in the future I would like to replace this with a parallel processing. After the subresults are generated, they can be combined into the recurrent context (line 5) and the output of the network (line 6). The rest of the computation is performed on the scope of the modules.

Algorithm 1 Single computational step of the RNM-EM model

1: **procedure** STEP($input, recurrence, M[], w[]$)
2: $rec[], y[]$
3: **for** int $i = 0$; $i < mc$; $i++$ **do**
4: $y[i], rec[i], M[i], w[i]$ = module.step($input, recurrence, M[i], w[i]$)
5: $recurrence = \sum_i (W_i * rec[i]) + br$
6: $y = softmax(\sum_i (y[i]) + b)$
7: **return** $y, recurrence, M[], w[]$

4.4.2 Description of the trainable variables of the model

In the proposed model there are two basic types of the trainable variables based on where they do appear, either on the network or the module level. Note that the trainable variables, that appear on the module level, actually appear on the network multiple times, based on the number of modules. In the following table you can see the trainable variables on the module, Table 4.2, and the network, Table 4.3, levels.

Table 4.2: Table of the trainable variables on the module level. The phase column explains the phase in which the learnable variable is used: memory read (MR), memory update (MU), hidden layer activity computation (HL) or output generation (OG).

name	dim	phase	description
W_k	$m_{size} \times ms$	MR	weight of the hidden layer activity for the key vector
b_k	m_{size}	MR	bias applied to the key vector
W_b	$1 \times ms$	MR	weight of the hidden layer activity for the scalar beta
b_b	1	MR	bias applied to the scalar beta
W_g	$m_{slots} \times de \cdot cs$	MR	weight of the input for the scalar g
W_i	$m_{slots} \times m_{slots}$	MR	weight of the memory weight for the scalar g
b_g	m_{slots}	MR	bias applied to the scalar g
W_v	$m_{size} \times ms$	MU	weight of the hidden layer activity for the new content vector
b_v	m_{size}	MU	bias applied to the new content vector
W_e	$m_{slots} \times ms$	MU	weight of the hidden layer activity for the erase vector
b_e	m_{slots}	MU	bias applied to the erase vector
W_x	$ms \times de \cdot cs$	HL	weight of the input for the hidden layer activity
W_h	$ms \times m_{size}$	HL	weight of the context for the hidden layer activity
W_r	$ms \times ms$	HL	weight of the recurrent context for the hidden layer activity
b_h	ms	HL	bias applied to the hidden layer activity
W	$nc \times ms$	OG	weight of the hidden layer activity for the output

4.4.3 Training of the model

Same as for the original RNN-EM, I used for the training of the RNM-EM model the AdaDelta[39] method. AdaDelta is a per-dimension learning rate method for gradient descent introduced by Matthew Zeiler in 2012. Based only on the first order information the AdaDelta method dynamically adapts over time. Big advantage of this method is that it doesn't require a manual tuning, especially of the learning rate. In the results[40] posted by Andrej Karpathy can be seen that the AdaDelta performs well comparing to the other methods, especially the AdaGrad method it is similar to. Even though Karpathy states

Table 4.3: Table of the trainable variables on the network level.

name	dim	description
emb	$(ne+1) \times de$	the matrix used for embedding the input into the recurrence
$h_1, \ldots, h_n$	ms	hidden layer activities of modules (learnable only in model A from Figure 4.1)
$W1, \ldots, Wn$	$ms \times TBD$	weight applied to the recurrent outputs of modules (the second dimension will be determined based on the tests performed later)
b_r	ms	bias applied to the recurrent output
b	nc	bias applied to the totaloutput of the network

that the SGD+Momentum method performs better with well-tuned settings, finding such a setting is not so easy and thus I will use the AdaDelta method.

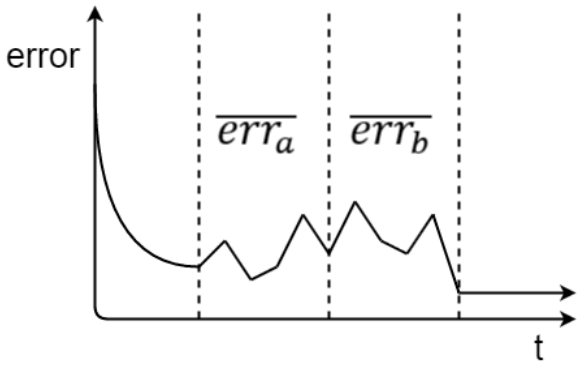

Figure 4.2: Visualization of the average error over time. After the given amount of training epochs the average error $\overline{err}$ is calculated. If the $\Delta err = \overline{err_b} - \overline{err_a}$, where the $\overline{err}$ are the average errors over the current b or last a given amount of epochs, is smaller than 0, then the training is stopped, otherwise it continues.

Another question that needs to be answered here is when does the training phase stop? The current solution I implemented in the prototype depends of the average error over a defined time period. As can be seen of the Figure 4.2 after every given time period, the default value is 5 training epochs, I calculate the average error. If the average error is smaller than the one from the previous time period, the training continues. Otherwise it stops and the best trained model, evaluation is performed after every training epoch, with the smallest error on the validation set is returned as the best one. This method also solves the problem if the network tends to make small "waves",

continuous increasing and decreasing with small difference of the error over time, on the error graph after a while, which was the behaviour I observed at the RNN-EM model.

4.4.4 Different alternatives of the model

There is one issue I have been on purpose omitting so far but is really important to be clarified before moving any further. That would be the recurrent context r_t. So far I have stated how it is used for the calculation of the hidden layer activities and how it is combined together on the network scale from the partial recurrent contexts returned by the modules. But what is actually the partial recurrent context?

Currently I am considering 3 different vectors that could serve as the recurrent context:

1. the hidden layer activity from the current time step $\mathbf{h_t}$,
2. the memory context from the current time step $\mathbf{c_t}$, which is based on the memory weight w_{t-1} and the memory content M_{t-1} from the previous time step or
3. the memory context for the next time step $\mathbf{c_{t+1}}$, which is based on the memory weight w_t and the memory content M_t from the current time step.

Every one of these approaches has its own advantages and could thus influence the network performance in a different way. But at the same time there are 3 different models as proposed on the Figure 4.1. This makes the total number of 9 different model variations. I would like to test all these variations and compare their performance in the upcoming chapter.

Another possibility of changing the performance of the model are the input parameters. There are 4 of them that are independent on the selected dataset: number of modules **mc**, number of hidden neurons within modules **ms**, size of the memory vectors **m_size** and number of memory slots (vectors) **m_slots**. Since these input parameters strongly influence the number of the trainable variables, I will also further study the influence of their settings to the network performance.

Chapter 5

Testing and Evaluation

In this chapter I will first focus on the test-driven upgrades of the model and afterward I will compare the performance of the proposed model against other selected ones. But before that I first need to describe in short where and how the testing will be performed.

The first thing that needs to be stated are the machine and the platform on which the test will be performed. Even though I won't be measuring the time performance of the model, it still is important to state this. All the test will be performed on an Ubuntu 14.04 laptop with Intel Core i5 2410M and 4GB DDR3 memory. It is also important to mention that the installed versions of the most important libraries were Python 2.7.6, Numpy 1.8.2, Theano 0.8.1 and Lasagne 0.1. Note that with different versions of the libraries you may achieve different results.

To be able to make any conclusions based on the test results I need to be able to compare them. I have decided to compare the test results based on the number of the trainable variables, basically all the weights and bias vectors, of the tested model. I believe this is a reliable measure since the networks with more trainable variables have the potential to learn more. Because all the changes of the network's structure and the input parameters influence the number of the trainable variables, I have created a simple program in Java that allows me to simply calculate the total number of the trainable variables of the RNM-EM model based on the network's setting. A screenshot from the program can be found on the Figure 4.2. The program itself can be found as an appendix of this thesis on the CD.

5.1 ATIS dataset

Since the main goal of this part of the thesis is to prove whether the newly proposed model RNM-EM is able to achieve at least similar score as the RNN-EM model, I have decided to perform the tests on the same dataset on which

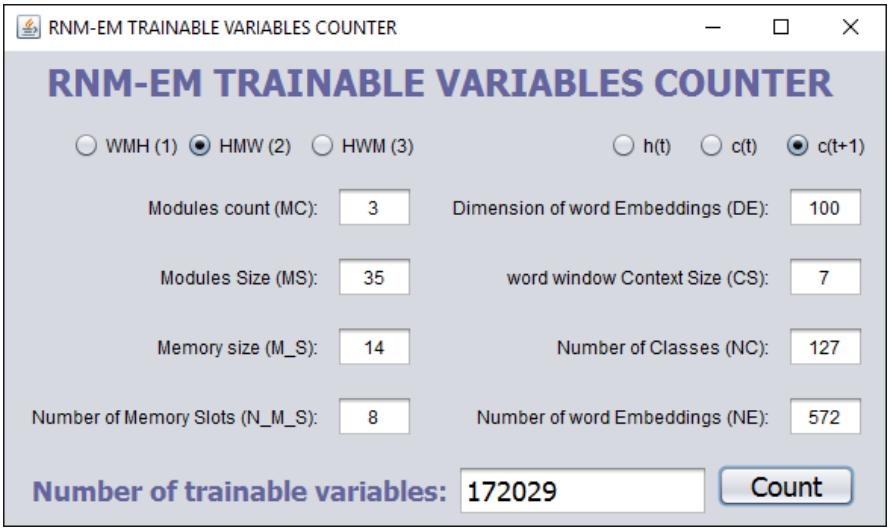

Figure 5.1: Screenshot from the Java program for computing of the total number of trainable variables in the RNM-EM model with given parameters.

the RNN-EM model was tested in the original paper. All the modifications and setting tests will be performed in order to help the RNM-EM model to achieve this goal and thus will be performed on the same dataset. The dataset I am talking about is the Airline Travel Information System (ATIS) corpus.

The ATIS dataset, collected by the DARPA, is a commonly used language understanding dataset. The goal of the language understanding is to verify whether the network is able to learn how to correctly assign tags to the words based on their role in the sentence. In the means of the ATIS dataset the tags are for example the departure/arrival airport or date/time information.

All the tests in this chapter will be evaluated based on the F1 score result that is computed as follows

$$F1 - Score = \frac{2 \times Recall \times Precision}{Recall + Precision}, \tag{5.1}$$

where Recall and Precision are computed as follows

$$Recall = \frac{\#\text{ correct slots found}}{\#\text{ true slots}}, \tag{5.2}$$

$$Precision = \frac{\#\text{ correct slots found}}{\#\text{ found slots}}. \tag{5.3}$$

More information about the ATIS dataset can possible be found in the paper[41] written by Tur et al.

book	a	flight	from	Hong Kong	to	Seattle
-	-	-	-	Dpt-city	-	Arv-city

Figure 5.2: An example of a sentence inside of the ATIS dataset. The assigned tags' names are shortened, with '-' meaning no tag assigned.

One ATIS dataset split contains 4978 training and 893 testing sentences. An example of an sentence and the correct tags can be seen on the Figure 5.1. I am dividing the training set into the training and validation ones, thus I am using 3983 training, 995 validation and 893 testing sentences. I am using the dataset splits available from the LISA laboratory webpage[42]. If not stated otherwise I will be using the dataset split 4 for all the tests performed in the later chapters.

For the work with the ATIS dataset I am, same as in the Pow's implementation, reusing the Grégoire Mesnil's code[36] available on GitHub. This code uses as the input not only the words of the sentences, but the words wrapped up with their context words. The number of the context words depends on the size of the context window, which is one of the input parameters, but if not stated otherwise the context window of length 7 will be used. Another important fact are the modified training epochs, where each epoch is extended to contain the same set multiple times. This means that the evaluation of the network is performed always after several appearances of the training set. For more information about the using of the code follow the tutorial[43] posted by the author of the code.

5.2 Tests of RNN-EM modifications

Before I will move onto the testing of the newly proposed RNM-EM model, I first have few modifications I would like to test on the original RNN-EM model. The Pow's implementation results can be seen in the Table 5.1 as the test number 1a. All the tests in this chapter will be performed on the parameter setting on which the tests in the original paper were performed. That particular setting is a network consisting out of 100 hidden layer neurons with an external memory of 8 slots with each of size 40. In all the upcoming test I will be training the network on the training set and observing the performance on the validation and testing sets.

There are two modifications in the Pow's implementation in compare to the original paper. The first modification is the use of "1 − cosine distance" instead of cosine distance in the equation 3.7. I have tested the use the cosine distance (test 1b) and it performs slightly better on the Validation set but at the same time worse on the Testing set. But since the performance is almost

similar (1% of the Testing set are 10 sentences), I am going to be staying with the, in the original paper proposed, cosine distance.

The second and probably more significant modification is the change of the activation function of the hidden layer activity in the equation 3.4. In the Pow's implementation the originally proposed tanh function is replaced by the sigmoid one. I have tested (test 2a) using the tanh function instead of the sigmoid one and the upgrade in the performance is significant, 4% on the Validation and 5% on the Testing set. Thus from now on I will be using only the tanh activation function.

Table 5.1: Table of the results of testing the modifications of the RNN-EM model. All the results are in a form of the F1 score. The particular tests are described in the section 5.2.

Test number	**Validation set**	**Testing set**
1a	91.98	88.97
1b	92.18	88.07
2a	96.14	92.99
3a	96.28	93.48

The first modification I am proposing on my own is the dependance of the scalar g_t on the memory weight w_t as described in the section 4.3 about the RNN-EM implementation. The test results of this modification (test 3a) are slightly better in both the Validation and Testing sets and thus I will stick with the modification for the rest of the testing.

The last possible modifications that needed to be tested in this section are the models proposed on the Figure 4.1. The test results can be found in the Table 5.2. From the test results it can be seen that the models A and D perform slightly better than the model C, but the upgrade in the performance is not so significant. Thus in the following section about the testing of the RNM-EM model I will still consider all these 3 models for testing. The model B is not considered in this section since it is almost the same as the model A, but will be part of the testing in the next section.

Table 5.2: Table of the results of testing the RNN-EM models from the Figure 4.1. All the results are in a form of the F1 score.

Test number	**Model**	**Validation set**	**Testing set**
3a	A	96.28	93.48
4b	C	95.73	93.04
4a	D	96.19	93.69

5.3 Tests of RNM-EM modifications and settings

The tests I will be performing on the RNM-EM model can be divided into two categories: the ones concerning the model's structure and the ones concerning the input parameter settings of the model. First I need to perform the tests that will be determining the structure I will be using for all the tests about the model's setting. If not state otherwise all the tests in this section will be performed on a RNM-EM network model with 3 modules, each containing 35 neurons with an external memory with 8 slots of size 14 each.

As I was already testing in the previous section, there are three basic models of the RNM-EM modules structure as stated in the Figure 4.1: A/B, C, D. But on the scope of the whole network, as I have already previously stated in the section 4.4, I have 3 options what to use as the recurrent context r_t that is propagated between the modules. Those 3 options are the hidden layer activity h_t, the memory context c_t from the current time step t or the memory context c_{t+1} from the next time step $t + 1$. Thus the 3 models together with the 3 options about the recurrent context give a total of 9 possible network structures worth testing.

Table 5.3: Table of the results of testing the RNM-EM models. All the results are in a form of the F1 score.

Test number	**Model**	**Validation set**	**Testing set**
5a	A, h_t	96.75	94.03
5b	A, c_t	95.90	93.41
5c	B, c_{t+1}	96.00	93.24
6a	C, h_t	96.36	93.67
6b	C, c_t	96.34	93.55
6c	C, c_{t+1}	96.48	93.87
7a	D, h_t	96.79	94.48
7b	D, c_t	96.35	93.93
7c	D, c_{t+1}	96.45	93.90

In the Table 5.3 there results of the tests for all the 9 possible network structures. As can be observed the models with the recurrent context $r_t = c_t$ perform the worst and thus will not be considered anymore. The best performing structures in each model section are the model A with $r_t = h_t$, the model C with $r_t = c_{t+1}$ and the model D with $r_t = h_t$. Since the difference in the performance of those 3 structures is not so significant, I have decided to use for the rest of this section the model C with $r_t = c_{t+1}$, even though it performs worse than the other two structures. The reason behind this decision is that this structure upgrades the original RNN-EM model but at the same time doesn't propagate back the hidden layer activity, which would result

into some sort of a hybrid model between the RNN-EM and the classic RNN. Another reason that influenced my decision in this matter is the fact that according to my opinion the model C acts the most naturally from all the models stated on the Figure 4.1. Note that this decision does mean that I will be using the selected structure for the rest of this section, but I may return to the other ones in the future.

After selecting the structure of the RNM-EM model I will be using, I can move to the part of testing the setting of the input parameters. The setting can be divided into the setting on the scope of the modules and on the scope of the network. Starting with the settings on the scope of the modules, there are two settings I would like to test within this section: the number of the memory slots and the ratio between the size of the memory slots and the number of the hidden layer neurons.

I will start with the number of the memory slots. In the original RNN-EM paper[8] the authors show that the best performing number of memory slots for the RNN-EM network are 8 slots and thus the capacity of the context the memory does not simply depend just on the number of memory slots. But the authors of the paper have been testing the values equal to the powers of 2 and thus there is the region between 4 and 16 where the network could actually perform better. I have thus decided to perform a test on the range between 2 and 14, focusing on the even number of slots. From the Table 5.4 it can be observed that the number of slots with the best performance, for the setting of the RNM-EM network I am using, lies somewhere around 12 memory slots.

Table 5.4: Table of the results of testing the influence of the number of memory slots in the RNM-EM model. All the results are in a form of the F1 score.

Test number	**M_slots**	**Validation set**	**Testing set**
8a	2	95.84	93.06
8b	4	96.41	93.45
8c	6	96.07	93.51
6c	8	96.48	93.87
8d	10	96.64	94.12
8e	12	96.65	94.13
8f	14	96.62	93.56

The ratio between the size of the memory slots (m_slots) and the size of the hidden layer (ms) is something that is not even discussed in the original RNN-EM paper. It is obvious that it is not worth to have the memory slots bigger than the hidden layer, but what is the ideal ratio? The authors just state that they are using the memory slots of size 40 and the hidden layer of 100 neurons and thus a 1:2.5 ratio, which is the setting I have been so

far following. For this test I have been selecting the ratios in the way so the number of trainable variables stays the same. The selected settings can be seen in the Table 5.5.

Table 5.5: Table of the settings for testing the influence of the ratio between hidden layer and memory slots sizes in the RNM-EM model.

Ratio	**Size of memory vectors**	**Size of hidden layers**	**Number of trainable variables**
1:1	33	33	$1.7 \cdot 10^5$
1:2	17	34	$1.7 \cdot 10^5$
1:2.5	14	35	$1.7 \cdot 10^5$
1:3	12	36	$1.7 \cdot 10^5$
1:4	9	36	$1.7 \cdot 10^5$

From the Table 5.6 it can be observed that for the ratio 1:2.5 the RNN-EM model performs the best and thus I will stick to that ratio. Another modification I have performed in all these tests is the fact that the memory weight w_t is propagated between the steps of the network. Comparing the results of the test 6c from the Table 5.4 and the test 9c from the Table 5.6, it can be observed that with this modification the network performs better.

Table 5.6: Table of the results of testing the influence of the ratio between hidden layer and memory slots sizes in the RNM-EM model. All the results are in a form of the F1 score.

Test number	**Ratio**	**Validation set**	**Testing set**
9a	1:1	95.72	93.64
9b	1:2	95.87	93.76
9c	1:2.5	96.61	94.03
9d	1:3	96.28	93.39
9e	1:4	96.16	93.58

One of the most important test of this section is how the performance of the model reacts to a change in the number of modules. So far I have been performing tests just on the 3 modules model, but splitting the hidden layer into even more modules offers the ability of a better parallelism and also a potential chance for the communication between the modules to make a bigger impact on the performance of the model. In the Table 5.7 can be seen all the settings of the network I have used for this testing. It is obvious that the sizes of the hidden layers and the memory vectors have been selected in order to keep the number of trainable variables approximately the same for all of the settings.

Table 5.7: Table of the settings for testing the influence of the number of modules in the RNM-EM model.

Number of modules	Size of hidden layers	Size of memory vectors	Number of trainable variables
2	55	22	$1.7 \cdot 10^5$
3	35	14	$1.7 \cdot 10^5$
4	25	10	$1.7 \cdot 10^5$
5	20	8	$1.7 \cdot 10^5$

Based on the settings from the Table 5.7 I have performed the testing, whose results can be found in the Table 5.8. A pleasing result is the fact that the network with multiple modules performs for the selected settings not only similar, but in some cases even better. An illustrious example can be the test results of the model with 4 modules, where the F1 score for both the Validation and Testing sets was increased by approximately 0.5% in compare to the ones for the model with 3 modules. The results of this test give hope that the scaling on the scope of number of modules might be one of the ways how to later improve this model.

Table 5.8: Table of the results of testing the influence of the number of modules in the RNM-EM model. All the results are in a form of the F1 score.

Test number	Modules	Validation set	Testing set
10a	2	96.2	93.90
9c	3	96.61	94.03
10b	4	97.15	94.52
10c	5	96.72	93.88

5.4 Comparison testing

In the previous section I have performed tests of the structure and settings of the newly proposed RNM-EM model. In this section I would like to compare this model to some of the other models. I have decided to compare the performance of the following models:

1. simple RNN,
2. LSTM,
3. RNN-EM and
4. RNM-EM.

For the RNN-EM network I will be using the same model that was used for the test 3a of the Table 5.1. As for the RNM-EM network I will be using the

best performing model I have come over during all the previous tests and that would be the model that was used for the test 10b of the Table 5.8. But since there are multiple implementations of the simple RNN and LSTM networks, I would like to describe their used models a little bit more in detail in the next paragraphs.

The term simple RNN is usually used for two of the recurrent neural network models and those are the Elman and Jordan networks. I have decided to use the Elman network for the testing since it is more similar to the RNN-EM model than the Jordan network. As already previously stated in the section 4.2.2, I have been using as the source for this implementation the Grégoire Mesnil's code[36]. Since I will need to compare networks with similar number of trainable variables TV, I have been using for this network the following equation:

$$TV = (ne + 1) \cdot de + (de \cdot cs) \cdot nh + nh^2 + nh \cdot nc + 2nh + nc, \tag{5.4}$$

where the explanation of the parameters can be found in the Table 4.1.

As for the LSTM network I am using the model containing the peephole connections, which were proposed by Gers and Schmidhuber[44] in 2000. The sources of the implementation, I am using, are described in the section 4.2.2. Same as for the other networks, also for the LSTM I will need to be able to compute the total number of trainable variables TV. In case of the LSTM I am using the following equation:

$$TV = (ne + 1) \cdot de + 4(de \cdot cs) \cdot nh + 7nh^2 + nh \cdot nc + 4nh + nc, \tag{5.5}$$

where the explanation of the parameters can again be found in the Table 4.1.

Table 5.9: Table of the networks settings for the comparison testing.

Model	**Setting**	**Number of trainable variables**
RNN	120 neurons	$1.7 \cdot 10^5$
LSTM	36 neurons	$1.7 \cdot 10^5$
RNN-EM	110 neurons 44 memory size 8 memory slots	$1.7 \cdot 10^5$
RNM-EM	4 modules 25 neurons 10 memory size 8 memory slot	$1.7 \cdot 10^5$

In the Table 5.9 can be seen the setting of particular networks that was used for the comparison testing. The setting for each network was chosen in order so all the networks would approximately have the same number of trainable variables. All the networks were trained using the Adadelta algorithm in order for them to have the same conditions. In the Table 5.10 can be observed the best F1 scores the networks were able to achieve within the first 25 training epochs. The reason, why I am measuring the performance only in the first 25 training epochs, is because the area around the 25th epoch is the place where majority of the tested networks start to overtrain themselves and thus their training is terminated.

Table 5.10: Table of the results of the comparison testing. All the results are in a form of the F1 score.

Model	**Validation set**	**Testing set**
RNN	96.89	94.04
LSTM	95.17	92.53
RNN-EM	96.58	93.54
RNM-EM	97.15	94.52

From the Table 5.10 it can be observed that the best performing model for the given problem is the newly proposed RNM-EM model. But what is actually a little bit surprising is the fact that the Elman network performed slightly better than the advanced RNN-EM model. My assumption is that the given problem doesn't, because the input is always also provided together with its context, require such a long-term dependencies and thus the simple Elman network's capabilities are enough to handle it. While the RNN-EM model compresses the hidden layer activity into the memory vectors and thus might loose some of the information. But since this matter is not so important for the evaluation of this testing I will not pursue to prove its correctness. Note that even though the RNM-EM model also compresses the hidden layer activity into the memory vector, it works with two different contexts, the one on the scope of the module c_t and the one on the scope of the whole network r_t, and thus is able to provide a better overall context than the RNN-EM model.

As can be observed from the table the worst performing network for the given problem is the LSTM network. Even though this might seem unexpected and surprising, it actually has a simple explanation. If you look on the Table 5.9 you can observe that for the LSTM I have used a significantly smaller number of hidden layer neurons than for the other networks. This is because unlike the other networks the LSTM model contains 4 different weights, 1 for the cell state and 3 for the gates, for the input vector x_t. Since in the given problem the input x_t is significantly bigger in compare to the other used vectors, dimension of the word embedding $de = 100$ multiplied by the size of

the context window $cs = 7$, most of the weights (trainable variables) in the LSTM model are assigned to the input and this is also what probably limits the model.

Even though the newly proposed model RNM-EM performed the best in both the validation and testing sets, I am not able to make any strong conclusions here since the tests were performed only on one specific problem. This means that I can't state here that the newly proposed model would be in general better than the other ones, but the fact that it performed the best for the given problem, makes me believe that this model has a potential to perform well also for some other problems. Another fact that I feel I need to mention here is that the RNM-EM model allows more different settings for the same approximate number of trainable of variables, thus I believe there still might exists a setting that would perform better than the best one I have been able to find.

Conclusion

Within this chapter I would like to summarize this thesis and conclude its results and contribution. In the end I would like to talk about the possible direction the research, performed within this thesis, could take in the future.

Summary of the thesis

The first part of this thesis is devoted to the overview of the family of the neural networks with memory. There are only few papers that cover more than one model I consider a part of the neural network with memory. Unfortunately there is no paper that would cover the topic in general and thus I decided to devote a significant part of this thesis to this purpose. In the Chapter 2 I am giving an overview of the most significant models, I consider as the advanced representatives of the family of the neural networks with memory, that were proposed over the last 25 years. Later in the chapter six of the selected models, I consider the most interesting, are covered in detail.

Based on the detailed study I have performed an analysis of a modification on one of the selected models (see Chapter 3). I have discovered that there is one thing that all the neural networks deploying an external memory have in common and that is that they deploy the external memory on the scope of the whole network. But deploying the external memory on the scope of parts of the network might be useful, because thanks to that different parts of the network might be able to track contexts of different lengths. I have thus designed a model of the Recurrent Neural Modules with External Memory (RNM-EM), which is based on the RNN-EM model by Peng and Yao [8]. The new model proposes a deployment of the external memory on the scope of the modules, parts of the hidden layer, and thus deploys a multiple external memories within one neural network. To the study of this new RNM-EM model I have devoted most of the Chapters 3, 4 and 5. Note that this principle of deploying the external memory on the scope of the parts of the neural network might also be applied to other models containing the external memory. Thus I am

proposing not only a single new model, but a potentially new family of the neural networks with external memory.

The last part of this thesis I devoted to the evaluation of the new proposed model of the Recurrent Neural Modules with External Memory (RNM-EM) on a real task. I have decided to test the performance of the model on the Air Travel Information System (ATIS) dataset, which is a widely used language understanding dataset. From the results, described in detail in the section 5.4, it can be seen that the newly proposed RNM-EM model slightly outperformed all the other models it was tested against, including the Elman Network, LSTM or RNN-EM. Even though I am not able to make strong conclusions based on a testing on one dataset, the model shows a significant potential and thus I consider the thesis successful and its goals fulfilled.

Contribution of the thesis

The three major points of contribution of this thesis are the summarization of the topic of neural networks with memory, the idea of application of the external memory on the scope of parts of the network and the newly proposed model of the Recurrent Neural Modules with External Memory.

While doing the background research for this thesis I was trying to find any research paper that would cover the topic of what I call here the neural networks with memory. Unfortunately there is no such paper and thus I found only few papers that were focussing on the topic of the recurrent neural networks. But all these papers were always explaining the topic either too widely or strictly. Thus I have decided to devote a significant part of this thesis as an overview of the significant models proposed over the last 25 years in the field of the neural networks with memory. I believe this overview is unique and hopefully will serve some other researches as a source for their research.

While studying the different models of neural networks that are supplemented by additional external memory, I have come across an interesting fact that all the models I studied deploy the external memory on the scope of the whole network. Being surprised by this fact and feeling inspired by the models like for example the Clockwork RNN, that divides the network into multiple groups of neurons called modules, I proposed a completely new model of the Recurrent Neural Modules with External Memory (RNM-EM), that deploys the external memory on the scope of modules and thus the whole network contains multiple external memories, one for each module of the network.

I feel confident about that the innovation, the new RNM-EM model proposes, is something revolutionary and thus might be one of the possible ways how to make a progress in the field of the neural networks. Even though I am

not able to make so strong conclusions based on the testing I have performed, I believe that the newly proposed model sets important foundations for a further research and thus will hopefully leave its mark in the field of the neural networks.

Future work

This thesis is not just a standalone work, but it part of a long term research in the field of neural networks I am performing. The results of the research performed within this thesis sets important foundations for my further research in this area and determine the direction the further research will take. Based on the test results I have proven that it might be advantageous in some cases to deploy the external memory multiple times within a single neural network. This fact opens the door for similar modifications of several other neural networks with external memory.

But this doesn't mean that the research of the newly proposed model of the Recurrent Neural Modules with External Memory (RNM-EM) would be put aside. I consider the results of this thesis as the beginning of a long lasting research of the proposed model. I already have several ideas on further modifications of the proposed model. One of such possible modifications could for example be deploying the principle of the Clockwork RNN model on the RNM-EM model. The resulting model would consist out of multiple modules with external memory, where each module would be running on a different clock period. Determining, whether such a model could work, will actually be the goal of the next research.

Bibliography

[1] Neurons, Brain Chemistry, and Neurotransmission. online. Available from: https://science.education.nih.gov/supplements/nih2/addiction/guide/lesson2-1.html

[2] Lipton, Z. C. A Critical Review of Recurrent Neural Networks for Sequence Learning. *CoRR*, volume abs/1506.00019, 2015. Available from: http://arxiv.org/abs/1506.00019

[3] Hochreiter, S.; Schmidhuber, J. Long Short-Term Memory. *Neural Comput.*, volume 9, no. 8, Nov. 1997: pp. 1735–1780, ISSN 0899-7667, doi:10.1162/neco.1997.9.8.1735. Available from: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[4] Maass, W.; Natschläger, T.; Markram, H. Real-time Computing Without Stable States: A New Framework for Neural Computation Based on Perturbations. *Neural Comput.*, volume 14, no. 11, Nov. 2002: pp. 2531–2560, ISSN 0899-7667, doi:10.1162/089976602760407955. Available from: http://dx.doi.org/10.1162/089976602760407955

[5] Koutník, J.; Greff, K.; Gomez, F. J.; et al. A Clockwork RNN. *CoRR*, volume abs/1402.3511, 2014. Available from: http://arxiv.org/abs/1402.3511

[6] Graves, A.; Wayne, G.; Danihelka, I. Neural Turing Machines. *CoRR*, volume abs/1410.5401, 2014. Available from: http://arxiv.org/abs/1410.5401

[7] Joulin, A.; Mikolov, T. Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. *CoRR*, volume abs/1503.01007, 2015. Available from: http://arxiv.org/abs/1503.01007

[8] Peng, B.; Yao, K. Recurrent Neural Networks with External Memory for Language Understanding. *CoRR*, volume abs/1506.00195, 2015. Available from: http://arxiv.org/abs/1506.00195

[9] Caudill, M. Neural Network Primer: Part I. *AI Expert*, volume 2, no. 12, Feb 1989: pp. 46–52.

[10] Patel, N. Artificial Neural Networks. online. Available from: `http://ocw.mit.edu/courses/sloan-school-of-management/15-062-data-mining-spring-2003/lecture-notes/NeuralNet2002.pdf`

[11] Burger, J. A Basic Introduction To Neural Networks. online. Available from: `http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html`

[12] Gers, F. *Long Short-Term Memory in Recurrent Neural Networks*. Dissertation thesis, Department of Computer Science, Swiss Federal Institute of Technology, Lausanne, EPFL, Switzerland, 2001.

[13] Bengio, Y.; Simard, P.; Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, volume 5, no. 2, Mar 1994: pp. 157–166, ISSN 1045-9227, doi:10.1109/72.279181.

[14] Weston, J.; Chopra, S.; Bordes, A. Memory Networks. *CoRR*, volume abs/1410.3916, 2014. Available from: `http://arxiv.org/abs/1410.3916`

[15] Jozefowicz, R.; Zaremba, W.; Sutskever, I. An Empirical Exploration of Recurrent Network Architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, edited by D. Blei; F. Bach, JMLR Workshop and Conference Proceedings, 2015, pp. 2342–2350. Available from: `http://jmlr.org/proceedings/papers/v37/jozefowicz15.pdf`

[16] Miikkulainen, R. *Text and Discourse Understanding: The DISCERN System*. New York, 2002, pp. 905–919. Available from: `http://nn.cs.utexas.edu/?miikkulainen:handbook00`

[17] Das, S.; Giles, C. L.; Sun, G.-Z. Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. *Proceedings of The Fourteenth Annual Conference of Cognitive Science Society*, 1992: pp. 791–795.

[18] Lin, T.; Horne, B. G.; Tino, P.; et al. Learning long-term dependencies in NARX recurrent neural networks. *IEEE Transactions on Neural Networks*, volume 7, no. 6, Nov 1996: pp. 1329–1338, ISSN 1045-9227, doi:10.1109/72.548162.

[19] Jaeger, H. The âĂIJecho stateâĂİ approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, volume 148, 2001.

[20] Grefenstette, E.; Hermann, K. M.; Suleyman, M.; et al. Learning to Transduce with Unbounded Memory. *CoRR*, volume abs/1506.02516, 2015. Available from: http://arxiv.org/abs/1506.02516

[21] Karpathy, A.; Johnson, J.; Li, F. Visualizing and Understanding Recurrent Networks. *CoRR*, volume abs/1506.02078, 2015. Available from: http://arxiv.org/abs/1506.02078

[22] Kužela, O. Liquid State Machine module for Brain Simulator. GitHub repository, https://github.com/GoodAI/SummerCamp/tree/master/LSMModule, 2015.

[23] Hazan, H.; Manevitz, L. M. Topological constraints and robustness in liquid state machines. *Expert Systems with Applications*, volume 39, no. 2, 2012: pp. 1597 – 1606, ISSN 0957-4174, doi:http://dx.doi.org/10.1016/j.eswa.2011.06.052. Available from: http://www.sciencedirect.com/science/article/pii/S0957417411009523

[24] Sukhbaatar, S.; Szlam, A.; Weston, J.; et al. Weakly Supervised Memory Networks. *CoRR*, volume abs/1503.08895, 2015. Available from: http://arxiv.org/abs/1503.08895

[25] Cho, K.; van Merrienboer, B.; Gülçehre, Ç.; et al. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR*, volume abs/1406.1078, 2014. Available from: http://arxiv.org/abs/1406.1078

[26] GoodAI. Brain Simulator. Available from: http://www.goodai.com/#!brain-simulator/c81c

[27] Laboratoire d'Informatique des Systèmes Adaptatifs. Theano. Available from: http://deeplearning.net/software/theano/

[28] Google Brain Team. TensorFlow. Available from: https://www.tensorflow.org/

[29] Collobert, R.; Farabet, C.; Kavukcuoglu, K.; et al. Torch. Available from: http://torch.ch/

[30] Bastien, F.; Lamblin, P.; Pascanu, R.; et al. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[31] Bergstra, J.; Breuleux, O.; Bastien, F.; et al. Theano: a CPU and GPU Math Expression Compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010, oral Presentation.

[32] Lasagne contributors. Lasagne, version 0.1. Available from: http://lasagne.readthedocs.io/

[33] Carrera, E. Pydot. GitHub repository, https://github.com/erocarrera/pydot, 2004-2012.

[34] Pow, N. Recurrent Neural Networks with External Memory implementation. GitHub repository, https://github.com/npow/RNN-EM, 2015.

[35] Yao, K. Recurrent Neural Networks with External Memory implementation. GitHub repository, https://github.com/kaishengyao/cnn/blob/master/cnn/rnnem.cc, 2015.

[36] Mesnil, G. RNN for Spoken Language Understanding. GitHub repository, https://github.com/mesnilgr/is13, 2015.

[37] Ollion, C. LSTM implementation. GitHub repository, https://github.com/deeplearningparis/dl-rnn/blob/master/lstm.py, 2015.

[38] Carrier, P. L.; Cho, K. LSTM Networks for Sentiment Analysis. Available from: http://deeplearning.net/tutorial/lstm.html

[39] Zeiler, M. D. ADADELTA: An Adaptive Learning Rate Method. *CoRR*, volume abs/1212.5701, 2012. Available from: http://arxiv.org/abs/1212.5701

[40] Karpathy, A. ConvNetJS Trainer demo on MNIST. Available from: https://cs.stanford.edu/people/karpathy/convnetjs/demo/trainers.html

[41] Tur, G.; Hakkani-TÃijr, D.; Heck, L. What is left to be understood in ATIS? In *Spoken Language Technology Workshop (SLT), 2010 IEEE*, Dec 2010, pp. 19–24, doi:10.1109/SLT.2010.5700816.

[42] Mesnil, G. ATIS Corpus dataset folds. Available from: http://lisaweb.iro.umontreal.ca/transfert/lisa/users/mesnilgr/atis/

[43] Mesnil, G. Recurrent Neural Networks with Word Embeddings. Available from: http://deeplearning.net/tutorial/rnnslu.html

[44] Gers, F. A.; Schmidhuber, J. Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, 2000, ISSN 1098-7576, pp. 189–194 vol.3, doi:10.1109/IJCNN.2000.861302.

Appendix A

Acronyms

ANN Artificial neural network

ATIS Air Travel Information System

CEC Constant Error Carousel

ESN Echo State Network

GRU Gated Recurrent Unit

LSM Liquid State Machine

LSTM Long Short Term Memory

MemNN Memory Networks

MemN2N End-To-End Memory Networks

NARX Nonlinear AutoRegressive model with eXogenous

NNPDA Recurrent Neural Network Pushdown Automaton

NTM Neural Turing Machine

QA Question answering

RNN Recurrent neural network

RNM-EM Recurrent Neural Modules with External Memory

RNN-EM Recurrent Neural Networks with External Memory

SRN Simple Recurrent Neural Network

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `impl` ..... implementation sources
    - `rnmem` ..... RNM-EM implementation
    - `other` ..... other networks implementations
    - `support` ..... supporting programs implementations
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
- `test` ..... the directory of test-related files
  - `atis` ..... ATIS dataset splits
  - `conlleval` ..... script for the evaluation of testing
  - `modifications` ..... modifications testing results
  - `comparison` ..... comparison testing results
- `other` ..... other supplements
  - `diagrams` ..... diagrams of the computational flow