Informatics

Comparing Machine Learning Models using Long-Term Dependency and Physical System Benchmarks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Hannes Brantner, BSc
Registration Number 01614466

to the Faculty of Informatics

at the TU Wien

Advisor: Dipl.-Ing. Dr.rer.nat. Radu Grosu, BSc
Assistance: Dipl.-Ing. Dr. Ramin Hasani, BSc

Vienna, 31st April, 2021

Hannes Brantner

Radu Grosu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Declaration of Authorship

Hannes Brantner, BSc

I hereby declare that I have written this Master Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 31st April, 2021

Hannes Brantner

Acknowledgements

At first, I have to thank Ramin for providing me great support throughout my work on the thesis. He cared about me and was always pointing me to state-of-the-art literature, as he wanted to push me forward. I also have to thank Prof. Grosu for participating in numerous online meetings and for sharing his in-depth knowledge in the machine learning domain. Furthermore, I want to thank Mathias Lechner for giving me first-class support on questions I had regarding various machine learning models. I have to point out that he was always willing to help me and provided his responses incredibly fast. Last but not least, I have to thank my parents for providing me with mental and financial support throughout my whole study journey.

Abstract

The diversity of machine learning models has rapidly increased in recent years as research in the machine learning domain flourishes. This thesis tries to give an overview of machine learning models that are capable of dealing with regularly sampled time-series data without specifying a given history length that should be taken into account by the model. Therefore, all models presented in this thesis are either derivatives of the recurrent neural network or the transformer [VSP+17] architecture. Furthermore, new machine learning models are introduced that try to improve on the given transformer and unitary recurrent neural network [JSD+17] architecture. After the introduction of all models, they are all benchmarked against five benchmarks and compared thoroughly. These benchmarks try to determine the model's capabilities to capture long-term dependencies and the ability to model physical systems. Moreover, a time-continuous memory cell is introduced that is capable of storing a data bit over a large number of time steps without losing the stored information. This memory cell is built using the LTC network [HLA+20] architecture.

Contents

Abstract **vii**

Contents **ix**

1 Introduction **1**
1.1 Machine Learning Terms 1
1.2 Problem Statement 2
1.3 How to better model Physical Systems 2
1.4 Sampled Physical Systems 3
1.5 Why capturing Long-Term Dependencies is difficult 3
1.6 Aim of the Work 5
1.7 Methodological Approach 5
1.8 State of the Art 6

2 Models **7**
2.1 Differentiable Neural Computer 7
2.2 Memory Cell 7
2.3 Memory Cell NCP 8
2.4 Memory Layer 8
2.5 Memory Layer Attention 9
2.6 Transformer 9
2.7 Recurrent Transformer 9
2.8 Neural Circuit Policies 9
2.9 Unitary RNN 10
2.10 Unitary RNN 10
2.11 Enhanced Unitary RNN 10

3 Benchmarks **11**
3.1 Walker 11
3.2 Memory 12
3.3 Cell 12

4 Results **15**

5 Summary **17**

List of Figures **19**

List of Tables **21**

Bibliography **23**

CHAPTER 1

Introduction

1.1 Machine Learning Terms

A machine learning model is a mathematical parametrized function that gets an input and produces an output. For example, the machine learning model GPT-3 proposed in [BMR$^+$20] has 175 billion scalar parameters. This thesis will use imitation learning to optimally set the parameters of machine learning models. This means that for each input, there is an associative expected output provided that the model should return by applying its function to the input. Of course when the model's function is applied to the input with the initial state of the model's parameters, the returned model output will differ from the desired output in almost all cases. The measure that quantifies this error between model output and expected output is called a loss function and has a scalar return value. A sample loss function can be constructed as easy as computing the mean of all squared errors between the model output and the expected output. The model output is also often denoted as the prediction of the model. For each input sample, the loss function describes the error the model makes by applying its function and this error is only dependent on the parameters of the model. In the general case, a computer scientist wants to find the global minimum of that function with respect to all machine learning model parameters. As this is a problem that cannot be solved analytically in most cases, it is approximated by using gradient descent [RHW86]. This method incrementally changes each parameter depending on the gradient of the loss function with respect to each parameter in lock step. By denoting the loss function with L, the learning rate with α, the old whole parameter set with p, the old single scalar parameter with p_i and the new single scalar parameter with p'_i, the formula to update the individual parameters p_i in a single gradient descent step can be given as follows:

$$\forall p_i : \; p'_i = p_i - \alpha * \frac{\partial L}{\partial p_i}(p) \tag{1.1}$$

It is essential to note that the model as well as the loss measure must be deterministic functions for the gradient to exist. This update rule ensures that if the loss function increases with increasing p_i, therefore if the computed gradient of the loss function is larger than zero, a decrease of the parameter will happen that leads to a decreasing loss function result. The opposite case holds as well and this is why there is a minus sign in 1.1. The learning rate α determines how large in magnitude the update to the parameters should be at each gradient descent step. A too small learning rate will lead to slow convergence, a too large learning rate will lead to divergence. Therefore, a too large learning rate is far more dangerous than a too small one. Convergence means that the parameter updates have led to a local minimum of the loss function. There are no guarantees that this is the global minimum. Divergence means that the loss function diverges towards infinity. A local minimum or convergence can be reached by applying the gradient descent update rule to as many inputs as needed to set the loss function derivative to nearly zero.

1.2 Problem Statement

As the sheer amount of different machine learning models can be overwhelming, the task was to fix a distinct application domain and compare the most influential machine learning models in this domain with suitable benchmarks. Benchmarks are just large input data sets with associative expected outputs. Additionally, ideas for possible improvements in existing architectures should be implemented and benchmarked against the already existing ones. All benchmarked models should be implemented in the same machine learning framework and the benchmark suite should be extensible and reusable for other machine learning research projects. The whole implementation work done for this thesis should be made accessible for everyone by open-sourcing all the code. As mentioned in the abstract, all the models covered in this thesis are either derivatives of the recurrent neural network or the transformer [VSP⁺17] architecture. The benchmarks used in this thesis either test the models for their capabilities to capture long-term dependencies or their ability to model physical systems.

1.3 How to better model Physical Systems

Physical systems are guided by differential equations. The relation between system state x, system input u and system output y is given by the state derivative function f and the output function h, both of which depend on the absolute time t, as follows:

$$\dot{x}(t) = f(x(t), u(t), t) \tag{1.2}$$

$$y(t) = h(x(t), u(t), t) \tag{1.3}$$

This form of system description is applicable to all physical systems in our daily surroundings, most of them are even time-invariant. This means the functions f and h do not depend on the absolute time t. For example, a mechanical pendulum will now approximately behave the same as in one year, as its dynamics do not depend on the absolute time t. The system description given in 1.2 and 1.3 proposes, that machine learning models that are built in a similar fashion and whose state is also determined by a differential equation, should be pretty capable of modelling the input-output relation of physical systems. When the benchmarked models are introduced in more detail, it can be seen that all continuous-time machine learning models use a comparable structure in terms of parameterizing the state derivative and the output function.

1.4 Sampled Physical Systems

As the evaluation of the current state x at point in time t' with initial state x_0 given the dynamics from 1.3 can be computationally very expensive or even infeasible, sampling was introduced to avoid solving a complex differential equation. Therefore, the whole system is only observed at equidistant successive time instants, values belonging to this time instant are denoted with a subscript index $k \in \mathbb{Z}$, and the system is now called discrete. Discrete systems are guided by difference equations. The relation between system state x, system input u and system output y is given by the next state function f and the output function h, both of which depend on the time instant k, as follows:

$$x_{k+1} = f(x_k, u_k, k) \tag{1.4}$$

$$y_k = h(x_k, u_k, k) \tag{1.5}$$

It must be noted that x and y are time series in discrete systems and no more functions like in the case of continuous-time physical systems. This slightly off-topic explanation is necessary, as vanilla recurrent neural networks are built using the same principle. The system equations 1.4 and 1.5 require a regularly (equidistantly) sampled input x. A similar argument as before in 1.3 proposes now that a machine learning model with a similar structure, which gets a regularly sampled input of a physical system, should also be pretty capable of modelling the input-output relation of this sampled physical system. The corresponding machine learning models are then called discrete-time machine learning models.

1.5 Why capturing Long-Term Dependencies is difficult

The difficulty will be outlined solely on the example of vanilla recurrent neural networks (RNNs). How transformer-based and advanced RNN architectures tackle the problem will be discussed later. Vanilla recurrent neural networks are discrete-time machine learning models. Its dynamics are given in a similar fashion to the equations that govern sampled physical systems 1.4. The current state vector h_t and the next input vector x_{t+1} determine the next state vector h_{t+1} and output vector y_{t+1} deterministically. In this model all the past inputs are implicitly encoded in the current state vector. This entails a big challenge for computer scientists, as computers only allow states of finite size and finite precision, unlike our physical environment, which results in an information bottleneck in the state vector. The next state of a vanilla recurrent neural network h_{t+1} and its output y_t is typically computed by equations like the two proposed in [ASB16, p. 2] by using a non-linear bias-parametrized activation function σ, three matrices (W, V and U) and the output bias vector b_o:

$$h_{t+1} = \sigma(W * h_t + V * x_{t+1}) \tag{1.6}$$

$$y_t = U * h_t + b_o \tag{1.7}$$

Without the time shift on the input in the next state equation 1.6, the equations are pretty similar to the ones describing sampled physical systems. The following inequality from [ASB16, p. 2] using norms shows the relation between the loss derivative, a recent state h_T and a state from the distant past h_t where $T \gg t$. The notation is kept similar to the examples before. A subscript 2 after a vector norm denotes the Euclidean norm and a subscript $2, ind$ after a matrix norm denotes the spectral norm:

$$\left\| \frac{\partial L}{\partial h_t} \right\|_2 \leq \left\| \frac{\partial L}{\partial h_T} \right\|_2 * \|W\|_{2,ind}^T * \prod_{k=t}^{T-1} \|diag(\sigma'(W * h_k + V * x_{k+1}))\|_{2,ind} \tag{1.8}$$

This inequality contains all essential parts to understand why capturing long-term dependencies with vanilla recurrent neural networks is difficult. Some problems that machine learning tries to solve require incorporating input data from the distant past to make good predictions in the present. As these inputs are implicitly encoded in the states of the distant past, $\left\| \frac{\partial L}{\partial h_t} \right\|_2$ should not decay to zero or grow unboundedly to effectively tune the parameters using the gradient descent update rule shown above in 1.1. This ensures that distant past inputs influence the loss function reasonably and makes it feasible to incorporate the knowledge to minimize the loss function. As known the spectral norm of the diagonal matrix in 1.8 is just the largest magnitude out of all diagonal entries. Therefore, if the norm of the diagonal matrix is close to zero over multiple time steps k, also the desired loss gradient will decay towards zero. Otherwise, if the norm of the diagonal matrix is much larger than one over multiple time steps k, the desired loss gradient may grow unboundedly. Using this knowledge it is now clear that a suitable activation function must have a derivative of one in almost all cases to counteract the above described problems. A good fit would be a rectified linear unit (relu) activation function with an added bias term. The relu activation function with a bias b can simply be discribed by the function $max(0, x + b)$. The max function should be applied element-wise. As the requirements for the activation function candidates are clear now, the next thing to discuss is the norm of the matrix W. If $\|W\|_{2,ind} > 1$, $\left\| \frac{\partial L}{\partial h_t} \right\|$ may grow unboundedly, making it difficult to apply the gradient descent technique to optimize parameters. If $\|W\|_{2,ind} < 1$, $\left\| \frac{\partial L}{\partial h_t} \right\|$ will decay to 0, making it impossible to apply the gradient descent technique to optimize parameters. These problems are identical to the problems regarding the norm of the diagonal matrix and also have the same implications. The first case is calles the exploding gradient problem and the second case is called the vanishing gradient problem for given reasons. Both phenomena are explained in more detail in [BSF94].

1.6 Aim of the Work

This work should help to objectively compare various machine learning models used to process regularly sampled time-series data. It should outline the weaknesses and strengths of the benchmarked models and determine their primary domain of use. Moreover, as there are many models benchmarked, their relative expressivity across various application domains can be compared reasonably well. Another aim is to provide an overview of what architectures are currently available and how they can be implemented. Furthermore, the implemented benchmark suite should be reusable for future projects in the machine learning domain.

1.7 Methodological Approach

The first part of this thesis was to determine the most influential models for processing time-series data. Some of the models that were benchmarked against each other in this thesis were taken from [LH20], even though this paper focuses primarily on irregularly sampled time-series. The other models were implemented according to the following architectures: Long Short-Term Memory [HS97], Differentiable Neural Computer [GWR$^+$16], Unitary Recurrent Neural Network [JSD$^+$17], Transformer [VSP$^+$17] and Neural Circuit Policies [LHA$^+$20]. These nine models are then complemented by five models that were newly introduced. All these models are benchmarked against each other. Additionally, a time-continuous memory cell architecture should be introduced. This architecture must have its own benchmark test and should not be benchmarked against all other fully-fledged machine learning models as it is only a proof-of-concept implementation. All mentioned models should be implemented in the machine learning framework Tensorflow [AAB$^+$15]. After the implementation of all models, an extensible benchmark suite had to be implemented to compare all implemented models. A basic benchmark framework should be implemented, which automatically trains a given model and saves all relevant information regarding the training process including generating plots to visualize the data. All that should be needed to implement a new benchmark is to specify the input, the expected output data, the loss function and the required output vector size of the model. The benchmarks regarding person activity classification, sequential MNIST classification and kinematic physics simulation were taken from [LH20] and were modified slightly to be compatible with the benchmark framework. The other two benchmark regarding the copying memory and the adding problem were taken from [ASB16], but were also slightly modified to fit the benchmark framework's needs. The sixth benchmark that had to be implemented was the cell benchmark that should check if the memory cell is able to store information over a large number of time steps. When this step is also done, all benchmarks should be run on all applicable models and then the results should be thoroughly compared to filter out the strengths and weaknesses of the diverse models. Only after that a summary should be written to concisely summarize the most important discoveries and fallacies that were made.

1.8 State of the Art

The whole field of sequence modeling started with recurrent neural networks. More and more modern machine learning architectures exploit the fact that continuous-time models are very well suited for tasks related to dynamical physical systems as explained in 1.3. A few examples for such models would be the CT-GRU [MKL17], the LTC network [HLA^{+}20] and the ODE-LSTM architecture [LH20]. But there are also some older architectures that exploit continuous-time dynamics in machine learning models like the CT-RNN architecture [iFN93]. The other problem described in the previous chapters is the hard task of capturing lon-term dependencies in time-series. One solution for the problem was proposed in [ASB16], which introduced the Unitary RNN architecture. This architecture in principle just uses the vanilla RNN architecture described above, but with the difference that the matrix W fulfills $\|W\|_{2,ind} = 1$ to tackle the vanishing and exploding gradient problem. This idea was later refined by [JSD^{+}17]. The vanishing gradient problem was also tackled by the LSTM architecture [HS97] using a mechanism called gating. This mechanism changes the next state computation of the vanilla RNN. Another possible mitigation to the vanishing gradient problem is the transformer architecture proposed in [VSP^{+}17] using a mechanism called attention. In principle the transformer architecture model has access to all past inputs at a single time-step and directs its attention to the inputs most relevant for solving the required task. This eliminates the need to backpropagate the error through multiple time-steps, which keeps the number of backpropagation steps low.

CHAPTER 2

Models

2.1 Differentiable Neural Computer

This model defines a memory-augmented neural network architecture, that consists of a controller, read and write heads and obviously an external memory that is not parameterized by the neural network parameter. Furthermore, the external memory was structured in rows, where each memory row has a specific length. The architecture was taken from [GWR$^+$16] and is an enhancement to the Neural Turing Machine firstly introduced in [GWD14]. The Neural Turing Machine introduced differentiable read and write functions that allow to access the memory by context or by location in both read and write mode. The access by context was implemented by comparing the cosine distance of an emitted key vector to all memory row contents and by applying the softmax function to that distance vector, which yields a weight vector. The access by location was implemented by adding a possibility to interleave the previous step weights with the current step content-based weights and adding a "blurry" shift operation on top of it. These weight vectors are then normalized using a softmax function that takes each argument to the power of an emitted number to sharpen the weights. Some improvements of the Differentiable Neural Computer include a memory management system that is able to allocate and free memory in the external memory to avoid overwriting of important information and a memory use link matrix that allows the model to track its memory operations through time.

2.2 Memory Cell

The Memory Cell is a simple RNN consisting of two LTC neurons as described in [HLA$^+$20] and used in [LHA$^+$20]. However the leakage term was removed from the ordinary differential equation describing the state dynamics, as a fading potential would mean losing information stored in the memory cell over time. It has a 2-dimensional input, the input voltage for each of the two neurons delivered by a synapse, and a 1-dimensional output, which is just the potential of the first neuron. This model was implemented to tune state dynamics and parameters for the suggested memory cell architecture. Each cell has three incoming synapses:

- an excitatory synapse that delivers the input voltage over a synapse to the neuron
- a recurrent excitatory synapse that connects each neuron with itself, which is useful to maintain an excitation in a single neuron
- an inhibitory synapse from the other neuron, to create mutual exclusive excitation

This simple memory cell should now be able to store information over a long time horizon, the input vectors are provided in the right way. Recent results have shown that this architecture is not capable of repeatedly storing information and I am not sure in what range the input values shall lie. As described in [HLA+20], the synaptic current is computed by multiplying a non-linear conductance with a potential difference. The potential difference is just a parameter E_{ij} minus the potential of the postsynaptic neuron V_j. However, if the potentials are not bounded, this would mean even an excitatory synapse can deliver a negative current or analogously an inhibitory synapse can deliver a positive current. The bounded dynamics of LTC networks is no longer valid, as the leakage term was removed from the state dynamics equation.

2.3 Memory Cell NCP

This is a wiring that was written for the keras-ncp Python package derived from [LHA+20] that implements the memory cell architecture from the previous section in a parallel manner, however with a leakage current term in the state dynamics. However results have shown that it is not really able to store information over a long time horizon.

2.4 Memory Layer

This model implements the architecture of the memory cell, in a parallelized manner. It is therefore also an RNN and includes an input and output control, which are simply multi-layer perceptrons. The input control gets the current step RNN inputs and the current memory state as input and transforms this information to an input current to each memory cell without passing it through a synaptic non-linearity. One neuron in each memory cell gets the unmodified input from the input control, the other neuron gets the negated input current as input. The output of the whole architecture is then computed by passing the potentials of every first neuron in each memory cell through the output control, that generated the final RNN output. This model also performs not as well as expected.

2.5 Memory Layer Attention

To possibly improve the Transformer architecture described in [VSP+17], a new mechanism instead of multi-head attention must be implemented. Therefore, memory layer attention was created which simply concatenates every query vector with each value vector and feeds all concatenated vectors per query as a sequence to an RNN architecture. As an RNN architecture, the predescribed memory layer RNN is used. The output of the RNN is then the new representation of the query vector at the input of the memory layer attention. This architecture does not use separate key and value vectors and combines them in a single representation. A similar approach was taken in the Reformer architecture [KuKL20], which is a computationally more efficient transformer. This approach was not thoroughly tested as the basic building block, the memory layer RNN does not work as expected.

2.6 Transformer

This model implements the transformer architecture firstly introduced in [VSP+17]. The input and output embedding were replaced with a fully connected layer, as this model should also work with vector sized input. The linear layer at the output of the transformer was also replaced with a fully connected layer, which emits an output token of the specified token size. Instead of stopping the transformer architecture when the stop token was emitted by the decoder layer, this architecture stops after a specified amount of tokens. Moreover, the positional input encoding was extended to also support non-equidistant positions at the input, occurring when sampling irregularly. Therefore, the input to the transformer is a tuple, which must include position intervals, between two successive inputs. These are then used to build a cumulative sum to generate the absolute positions. If you just want to use the regularly sampled mode, just provide only 1s for the intervals. This architecture performs very well if a reasonably sized model with enough parameters is chosen.

2.7 Recurrent Transformer

This is a slightly modified version of the transformer model 2.6. It does not simply add the weighted value vectors up in the multi-head attention mechanism, but instead passes the weighted value vectors through an RNN, whose final output vector represents the added weighted values vectors in the ordinary multi-head attention. This architecture also performs reasonably well in the benchmarks.

2.8 Neural Circuit Policies

This model is a specifically connected LTC network [HLA+20], firstly described in [LHA+20]. It is very demanding in computational resources and does not perform very well on the benchmarks.

2.9 Unitary RNN

Unitary RNNs were firstly introduced in [ASB16] and made learning long-term dependencies with RNNs possible by keeping gradient sizes stable, even if backpropagation occurs to a distant past state. It is achieved by parameterizing the weight matrix of the hidden state in the next state equation in a way, that it stays a unitary matrix. This matrix can be constructed out of real and complex numbers, the whole unitary matrix space of size N^2 was efficiently parameterized by [JSD$^+$17]. With this approach, it can be shown that distant past inputs have the same influence on the current state as current inputs in terms of their gradient size, which only differs by a constant.

2.10 Unitary RNN

Unitary RNNs were firstly introduced in [ASB16] and made learning long-term dependencies with RNNs possible by keeping gradient sizes stable, even if backpropagation occurs to a distant past state. It is achieved by parameterizing the weight matrix of the hidden state in the next state equation in a way, that it stays a unitary matrix. This matrix can be constructed out of real numbers, which was done in [ASB16], but it can also be extended to the complex domain, which was done in [JSD$^+$17]. With this approach, it can be shown that distant past inputs have the same influence on the current state as current inputs in terms of their gradient size, which only differs by a constant.

2.11 Enhanced Unitary RNN

This model uses the time domain input to compute a frequency domain input, both are fed concatenated as input vector i_t to the RNN as input. My intuition was that some tasks are easier to solve in the frequency domain, whereas some tasks are easier to solve in the time domain. The state equation of the RNN can be expressed as follows, where W_1 and W_2 are unitary matrices and $\sigma(z) = max(0, |z| + b) * \frac{z}{|z|}$ (parameterized modrelu activation function defined in [ASB16]):

$$h_{t+1} = \sigma(W_2 * (\sigma(W_1 * h_t + V * i_{t+1}))) \tag{2.1}$$

The second matrix multiplication was added, such that the current hidden state has an influence where information about the current input is stored, which was not the case in the approach of [ASB16]. There exist other RNNs that use this state equations like the Fourier Recurrent Unit that was firstly introduced in [ZLSD18]. The parameterization of the unitary matrices was done with a complex square matrix A of the same dimension. This matrix A was then added to its adjoint A^H, which yields an arbitrary Hermitian matrix H. As each unitary matrix can be expressed by using the matrix exponential e^{iH}, each unitary matrix can be parameterized by A. However, the results were not as good as expected because gradients were not stable.

CHAPTER 3

Benchmarks

3.1 Walker

This benchmark evaluates how well a model can predict a dynamical, physical system's behavior and was taken from [?]. The training data is acquired simulation data of the `Walker2d-v2` OpenAI gym [BCP+16] controlled by a pre-trained policy. The objective was to learn the kinematic simulation of the physics engine in an auto-regressive fashion using imitation learning. To increase the task difficulty, the simulation data was acquired from different training stages of the pre-trained policy (between 500 and 1200 Proximal Policy Optimization iterations) and 1% of actions were overwritten by random actions. Furthermore, frame-skips were introduced by removing 10% of all time-steps to get irregularly sampled training data. Moreover, the training data was divided into equally long sequences. For non-recurrent models, the input per time-step was the whole training sequence, but future values were zero-padded to ensure the model is only able to derive future predictions from past values. The model needs to predict the simulation state in the next time-step. Mean squared error was used as loss function when training the models. Further results of recurrent neural network models doing this benchmark can be found in [RCD19]. The results are presented in the following table:

Model	Parameter Count	Mean Squared Error
Transformer	338257	0.793
Recurrent Transformer	439633	0.599
Recurrent Transformer	29617	1.102
Memory Layer Transformer	174529	7.624
Neural Circuit Policies	21889	2.297

3.2 Memory

This benchmark evaluates how well a model can be trained to incorporate long-term dependencies in its predictions. The structure of this benchmark was taken from [ASB16], who also tested their recurrent neural network with this long-term dependency benchmark. The training data only contains integers from 0 to $N-1$. The first ten input symbols of each input sequence are randomly sampled integers from 0 to $N-3$ (i.i.d. uniform). Then the following symbols are copies of the integer $N-2$ in the amount of the tested memory length T minus 1. After that follows a marker, which is a single integer $N-1$. Then the remaining ten symbols are again copies of the integer $N-2$. The expected output sequence starts with $T+10$ copies of $N-2$ followed by the ten randomly sampled symbols from the start of the input sequence. Each tested model must produce an output vector of size N per time-step to a apply a sparse categorical cross-entropy loss directly from the model output logits. The loss was weighted such that the loss given by each of the last ten output vectors has the same weight as all previous output vectors combined, as it is easily to optimize to predict only the same class for a long interval. Clearly, a model can only solve this task if it is able to store information over a time period in the same size of the memory length. The baseline for this benchmark's mean categorical entropy loss per batch can be set to $\frac{10\log(N-2)}{T+20}$, as a memory-less strategy can perfectly predict the first $T+10$ symbols of the required output sequence and then it predicts the remaining 10 symbols randomly from 0 to $N-3$ (i.i.d. uniform). The results are presented in the following table, where N was 10, T was 100 and the baseline therefore was 0.173:

Model	Parameter Count	Cross-Entropy Error	Correctly Memorized Sy
LSTM	7130	0.157	1.1
Memory Layer	49720	0.400	0.0
Differentiable Neural Computer	81685	0.145	2.0
Unitary RNN	1674	0.009	10.0
Enhanced Unitary RNN	101388	0.3151	1.3

3.3 Cell

This is a very small benchmark that tries to find the right parameter and input ranges for the memory cell 2.2. This benchmark should make clear if a memory cell is able to switch its state multiple times and keep information over a long period of time. It is directly implemented in the same file as the memory cell itself. The memory cell in the current test setup has no more recurrent, excitatory synapses. As discussed in our meeting the state of a memory cell is now only changed through four synapses, two per neuron. One synapse per neuron delivers the input via a synaptic activation and the other synapse ensures the opposite potentials of both neurons. The synaptic current can be computed by $g_{syn}(V_{pre}) * (E_{pre,post} - V_{post})$. I have found out that the LTC equations are not suited at all to build a memory cell, as a 1 input should deliver a big current and a 0 input should lead to a big negative current, no matter what the potential of the postsynaptic neuron is. However with the current equations, this is not possible to achieve, since the non-linear synaptic conductance is always positive and therefore the sign of the current cannot be determined by the input. Therefore, the underlying equations must be changed. The same argument holds for the mutual inhibitory synapses. The higher potential neuron should send a negative current to the other neuron, whereas the lower potential neuron should send a positive current to the higher potential neuron. These things are not possible with the current setup, and therefore need to be changed. The memory cell receives only a single input that will be passed untouched to the first neuron's synaptic activation, but the second neuron will get the negated input to its synaptic activation for the input. The negation is currently implemented using the function $f(x) = 1 - x$. This input is only passed at the specific time step, when the input current provider needs to save something, otherwise both neurons in the memory cell get no input and should therefore hold their state. All memory cells are initialized to all contain zeros. As the output of the memory cell is the potential of the first neuron, this means that all first neurons in the memory cells have starting potential 0 and all second neurons have starting potential 1. The benchmark now feeds sparse inputs as described before to the memory cell, and requires the cell to hold the state from the last non-sparse input. This time horizon is the memory length of the memory cell, the amount of time it can save information. The benchmark not only checks the potential of the first neuron, but also the potential of the second neuron to be in the required range. Furthermore, the benchmarks alternates the symbol that should be saved first, half of the time it is a 0, half of the time it is a 1. Moreover, the memory cell state is switched again two times after the first input by providing two additional inputs, which represent the negated current memory cell potential, to check if the cell is able to switch its state. This results can be easily viewed when executing the memory cell Python script, they are not interesting since the memory cell does not learn anything.

CHAPTER 4

Results

CHAPTER 5

Summary

List of Figures

List of Tables

Bibliography

[AAB$^+$15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[ASB16] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks, 2016.

[BCP$^+$16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[BMR$^+$20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[BSF94] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[GWD14] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014.

[GWR$^+$16] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward

Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[HLA$^+$20] Ramin Hasani, Mathias Lechner, Alexander Amini, Daniela Rus, and Radu Grosu. Liquid time-constant networks, 2020.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[iFN93] Ken ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks*, 6(6):801–806, 1993.

[JSD$^+$17] Li Jing, Yichen Shen, Tena Dubček, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Soljačić. Tunable efficient unitary neural networks (eunn) and their application to rnns, 2017.

[KuKL20] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer, 2020.

[LH20] Mathias Lechner and Ramin Hasani. Learning long-term dependencies in irregularly-sampled time series, 2020.

[LHA$^+$20] Mathias Lechner, Ramin Hasani, Alexander Amini, Thomas Henzinger, Daniela Rus, and Radu Grosu. Neural circuit policies enabling auditable autonomy. *Nature Machine Intelligence*, 2:642–652, 10 2020.

[MKL17] Michael C. Mozer, Denis Kazakov, and Robert V. Lindsey. Discrete event, continuous time rnns, 2017.

[RCD19] Yulia Rubanova, Ricky T. Q. Chen, and David Duvenaud. Latent odes for irregularly-sampled time series, 2019.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986.

[VSP$^+$17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[ZLSD18] Jiong Zhang, Yibo Lin, Zhao Song, and Inderjit S. Dhillon. Learning long term dependencies via fourier recurrent units, 2018.