TU WIEN Informatics

Neural Network Arena: Comparing Machine Learning Models using Long-Term Dependency and Physical System Time Series 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

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

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 practice the loss function is applied to a batch of inputs separately and the arithmetic mean of all scalar loss function return values of the individual input samples is used as loss function to differentiate. The size of this input batch is called batch size. 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 [RHW86]:

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

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. As at the differentiation of the loss function, which can be represented as a computational graph with lots of nested functions, with respect to the individual model parameters involves lots of applications of the chain rule, the machine learning term for repeatedly applying the chain rule is backpropagation. If these nested functions correspond to applying the same machine learning model function across multiple input time steps as done in recurrent neural networks, then this backpropagation procedure can also be called backpropagation through time as introduced in [RHW86]. The chain rule for differentiating $z(y(x_0))$ with respect to x for $x = x_0$ where z and y are both functions in a single variable is given by:

$$\frac{dz}{dx}\Big|_{x=x_0} = \frac{dz}{dy}\Big|_{y=y(x_0)} * \frac{dy}{dx}\Big|_{x=x_0} \tag{1.2}$$

This reveals that a machine learning framework has to compute all partial derivatives of all functions present in the above mentioned computational graph. Furthermore, it must keep track of the so called activations which are denoted by $y(x_0)$ in the above formula 1.2, as otherwise the gradient of the loss function with respect to the parameters cannot be computed. As this can use lots of memory, reversible layers were introduced by [GRUG17] where intermediate activations can be computed from the output vector of that layer which makes storing intermediate activations obsolete.

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:

$$\dot{x}(t) = f(x(t), u(t), t) \tag{1.3}$$

$$y(t) = h(x(t), u(t), t) \tag{1.4}$$

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 1.3 and 1.4 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:

$$x_{k+1} = f(x_k, u_k, k) \tag{1.5}$$

$$y_k = h(x_k, u_k, k) \tag{1.6}$$

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 1.5 and 1.6 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:

$$h_{t+1} = \sigma(W * h_t + V * x_{t+1}) \tag{1.7}$$

$$y_t = U * h_t + b_o \tag{1.8}$$

Without the time shift on the input in the next state equation 1.7, 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:

$$\left\| \frac{\partial L}{\partial h_t} \right\|_2 \leq \left\| \frac{\partial L}{\partial h_T} \right\|_2 * \|W\|_{2,ind}^T * \prod_{k=t}^{T-1} \left\| diag(\sigma'(W * h_k + V * x_{k+1})) \right\|_{2,ind} \tag{1.9}$$

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 1.9 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 Model Factory

As all the benchmarks require variants of the same models with different output vector sizes, a model factory function was implemented that is capable of producing an output tensor given the model's name, the output vector size and the input tensor tuple. This function was called `get_model_output_by_name` and can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. This mechanism of creating the output tensor including the internal computational graph of the Tensorflow library [AAB+15] is called the Functional API. Most of the models were parametrized such that they have roughly 20000 trainable parameters in the Walker Benchmark 3.4 as this benchmark features the largest input and output vector size of all benchmarks. The exceptions of the parameter cound are the Unitary RNN model 2.8, the NCP model 2.7, the Unitary NCP model 2.10, the Recurrent Network Augmented Transformer 2.12 and the Recurrent Network Attention Transformer 2.13. All these models' computational graphs lead to high computation cost during backpropagation which leads to training durations up to a whole day for a single benchmark. This was unnacceptable and therefore their parameter count was reduced, such that at least some results can be presented for these models.

2.2 LSTM

The LSTM (Long Short-Term Memory) recurrent neural network architecture is a discrete-time machine learning model as introduced in 1.4. The model not only has an ordinary (hidden) state vector, but also a cell state vector which should store information over a longer time horizon than the hidden state vector. This thesis uses the open-source LSTM implementation provided by the Keras library [C+15] which is based on the original LSTM paper [HS97] as well on its successor paper [GSC00] that introduces a forget

mechanism for the LSTM. The function the LSTM model is applying to its inputs to produce the outputs is given as follows with inputs denoted as x_t and outputs which equals the hidden states denoted as h_t [GSC00, p. 4-8]:

$$f_t = sigmoid(W_f * x_t + U_f * h_{t-1} + b_f) \tag{2.1}$$
$$i_t = sigmoid(W_i * x_t + U_i * h_{t-1} + b_i) \tag{2.2}$$
$$o_t = sigmoid(W_o * x_t + U_o * h_{t-1} + b_o) \tag{2.3}$$
$$\tilde{c}_t = tanh(W_c * x_t + U_c * h_{t-1} + b_c) \tag{2.4}$$
$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t \tag{2.5}$$
$$h_t = o_t * tanh(c_t) \tag{2.6}$$

The term f_t 2.1 is the forget gate's activation vector, i_t 2.2 is the input gate's activation vector, o_t 2.3 is the output gate's activation vector, $\tilde{c}_t$ 2.4 is the cell input activation vector, c_t 2.5 is the cell state vector and h_t 2.6 is the hidden state vector or also called output vector of the LSTM model. The initial hidden state h_0 and the initial cell state c_0 are picked to the all-zero vector. Matrices are denoted with capital letters and vectors are denoted with lower case letters. The LSTM model has a configurable state size. The multiplication sign between two vectors denotes a scalar product and it denotes matrix multiplication between matrices and vectors. This convention is used throughout this thesis. Dimensions of matrices are picked such that the resulting vector has the required state size which is configurable. The bias vectors denoted with b also have the required state dimension. The matrices denoted by W map the input vector in each time step and the matrices denoted by U map the hidden state vector at each time step to a resulting vector. The structure of the model allows it to capture long-term dependencies as by setting f_t equal to one and i_t equal to zero in some common vector indices i, only the previous cell state is used to build the next cell state in these next cell state vector entries. This will lead to $\frac{\partial c_{t,i}}{\partial c_{t-1,i}} = 1$, as this clearly approximates the identity function for a specific index i in the cell state vector. Backpropagation to activations in the distant past is feasible using this model function as gradients are not vanishing or exploding when the parameters of the model are learnt properly. This mechanism is called the constant error carrousel as described in [HS97, p. 7]. LSTMs can incorporate this mechanism to store essential information from the distant past, which may be useful to make accurate predictions in the future. Furthermore, the model can also decide to forget the previous cell state completely, if the current input vector makes the stored cell state obsolete in the corresponding application. This is done by learning to set the forget gate's activation vector close to zero and the cell input activation vector is then used to fill the cell state again if the input gate's activation vector is set accordingly. The output gate's activation vector determines which portion of the cell state is used to build the hidden state or output vector of the LSTM model. Throughout the thesis an LSTM model with a fixed state vector size of 64 was used. As mentioned in the benchmark framework section, each model must support an arbitrary output vector size. This is accomplished by postprocessing the hidden state outputs of the LSTM by a dense layer with the required amount of output neurons and without an activation function.

The output y of a dense layer without an activation function and input vector x can simply be given by: $y = W * x + b$. In this notation W is a matrix such that it maps the input vector x to the required output size and b is just a bias vector like in the functions describing the LSTM model. Training the LSTM model from the Keras library is fast as it uses an optimized cuDNN [CWV+14] implementation. The LSTM model implementation used in this thesis is exposed under the `get_lstm_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`.

2.3 GRU

The GRU (Gated Recurrent Unit) recurrent neural network architecture is a discrete-time machine learning model as introduced in 1.4. The model has only a single ordinary hidden state vector. This thesis uses the open-source GRU implementation provided by the Keras library [C+15] which is based on the original GRU paper [CGCB14]. The GRU model tries to simplify the LSTM architecture by removing the output gate for example without sacrificing expressivity. This leads to a smaller parameter count of a GRU model with the same hidden state vector size as an LSTM model. The function the GRU model is applying to its inputs to produce the outputs is given as follows with inputs denoted as x_t and outputs which equals the hidden states denoted as h_t [CGCB14, p. 4]:

$$z_t = sigmoid(W_z * x_t + U_z * h_{t-1} + b_z) \tag{2.7}$$

$$r_t = sigmoid(W_r * x_t + U_r * h_{t-1} + b_r) \tag{2.8}$$

$$\tilde{h}_t = tanh(W_h * x_t + U_h * (r_t * h_{t-1}) + b_h) \tag{2.9}$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \tag{2.10}$$

$$\tag{2.11}$$

The term z_t 2.7 is the update gate vector, r_t 2.8 is reset gate vector, $\tilde{h}_t$ 2.9 is the candidate activation vector and h_t is the hidden state or output vector of the GRU model. The initial hidden state h_0 is picked to the all-zero vector. The notation of operations, matrices and vectors stays the same as for the LSTM architecture 2.2. Subtraction in the output vector equation 2.10 is meant element-wise and the 1 should denote the all-one vector. As in the LSTM architecture the hidden state vector size is configurable and all matrices map their inputs to a vector of the corresponding hidden state vector size. The structure of the model allows it to capture long-term dependencies as by setting z_t equal to zero for some vector entries, only the previous hidden state vector is used to build the next hidden state vector for these indices i. This will lead to $\frac{\partial h_{t,i}}{\partial h_{t-1,i}} = 1$, as this clearly approximates the identity function for a specific index i in the hidden state vector. Backpropagation to activations in the distant past is feasible using this model function as gradients are not vanishing or exploding when the parameters of the model are learnt properly. As also mentioned in [CGCB14, p. 5], the LSTM architecture does not expose its full cell state in the output vector as the cell state is further processes using the output gate. The GRU architecture however exposes its full cell state at each

time step as it does not have an output gate as mentioned before. Another key difference between the LSTM and GRU architecture is that the LSTM architecture controls the portions of the previous cell state and the portions of the cell input activation that add up to the next step cell state separately using the forget gate's activation vector and the input gate's activation vector 2.5. The GRU model simplifies this mechanism by providing just a single update gate vector z. The other vector controlling the portion from the previous hidden state that is added together to build the next step hidden state vector is then determined by subtracting z from the all-one vector 2.10. This is feasible as the sigmoid activation function produces only outputs lying in the interval $[0, 1]$. Furthermore, also the reset mechanism works differently in the GRU architecture as the reset vector only operates on the previous step hidden state vector when computing the next state candidate activation vector. Throughout the thesis a GRU model with a fixed state vector size of 80 was used. As mentioned in the benchmark framework section, each model must support an arbitrary output vector size. This is accomplished by postprocessing the hidden state outputs with a dense layer just like in the case of the LSTM architecture 2.2. Training the GRU model from the Keras library is fast as it uses an optimized cuDNN [CWV+14] implementation. The LSTM model implementation used in this thesis is exposed under the `get_gru_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`.

2.4 CT-RNN

The CT-RNN (continuous-time recurrent neural network) was first proposed in [iFN93] and is a continuous-time machine learning model 1.3. This thesis uses an implementation taken from the repository of the paper [LH20] which can be found under the url `https://github.com/mlech26l/ode-lstms`. The CT-RNN has a configurable hidden state vector size and its output vector is equal to its hidden state vector at each time step. The hidden state vector is parametrized as follows [iFN93, p. 2] with the same notation as introduced in 2.2:

$$\dot{h}(t) = -\frac{h(t)}{\tau} + W * sigmoid(h(t)) + i(t) \tag{2.12}$$

The division between $h(t)$ and the vector τ is understood element-wise. The vector τ is also called the time constant as it is the time constant of the exponential decay of the hidden state vector over time. As the input has not in general the same dimension as the hidden state vector, the input is preprocessed by mapping it to the right dimension with a matrix multiplication. Furthermore, the implementation used for benchmarking has a tanh activation function as it is applied at a different position in the formula to allow for negative activations. There is also an additional bias vector b and scaling vector α introduced whose multiplication is to understood element-wise. The derivative in the

CT-RNN implementation used for benchmarking is given by:

$$\dot{h}(t) = -\frac{h(t)}{\tau} + \alpha * tanh(W_h * h(t) + W_i * i(t) + b) \tag{2.13}$$

The idea of parameterizing the derivative or change of the hidden vector or activation rather than computing a completely new hidden vector or activation was extensively reused in recent research. For example ResNets [HZRS15] used the idea in a discrete-time model and Neural ODEs [CRBD19] reused it in a continuous-time model which features a similar model function as the CT-RNN. In discrete-time models residual connections are added which help backpropagation in a deep machine learning architecture as they are just representing the identity function which is easily differentiable. For more information on residual connections consult the corresponding paper [HZRS15]. As the benchmark input samples are only regularly sampled vectors and not a function $i(t)$ as needed by the CT-RNN model function 2.13, each input sample is held constantly for 1 time unit to form the input function. This mechanism is used for all continuous-time models throughout this thesis. Therefore, the input function is defined on the interval $[0, T]$ where T is the input sequence length. The output of the CT-RNN after consuming the whole input function $i(t)$ from time 0 to time T is then given by the hidden state vector $h(T)$ at time T. There is also the possibility to evaluate the hidden state at intermediate time points, for example at $T - 1$, which equals $h(T - 1)$. With this mechanism any continuous-time model can also map an input vector sequence to an output vector sequence. If additional timing information is available about the input vectors, for example the time interval between two input vector samples, it can be used as time to hold this specific input constantly in the input function. This leads to an irregularly sampled time-series where time-continuous models are exceptionally well suited as machine learning models, as discrete-time models 1.4 implicitly model a regularly sampled continuous-time system. This statement was also shown to be valid by [LH20]. The initial state of the CT-RNN is given by $h(0)$, which is picked to the all-zero vector. To compute the final hidden state $h(T)$, the ODE (ordinary differential equation) from 2.13 must be solved given the initial condition $h(0)$. This can be done by incorporating ODE solvers which simply compute $h(T)$ by approximately integrating $\dot{h}(t)$ with guarantees on the error bound. Then $h(T)$ is given by $h(0) + \int_0^T \dot{h}(t)\ dt$. In all continuous-time models implementations the ODE solver is called at each time step computing the next step hidden state $h(t + 1)$ as $h(t) + \int_t^{t+1} \dot{h}(t)\ dt$. Examples for ODE solvers are the explicit Euler method, the RK4 (Runge-Kutta 4^{th} order) method or the Dormand-Prince method. The Dormand-Prince method is the default ODE solver used in the `ode45` solver of MATLAB [MAT20]. All of these are members of explicit methods and the Runge-Kutta methods to solve ODEs. Explicit methods calculate the state at a later time only from the state at the current time. There are also implicit methods, which find a solution by solving an equation involving both the state at the current time and the state at the next time. Implicit methods are primarily used for stiff ODEs, which are characterized that small numerical deviations or errors can lead to a huge change in the output. For the CT-RNN implementation, the RK4 method was used to solve the ODE. The hidden state vector size was picked to 128 and the number of unfolds was set to 3. The number of unfolds determines how

often an ODE solver is called on a single input sample. This means that instead of integrating the whole interval of length 1 at each time step, the ODE solver integrates an interval of length $\frac{1}{3}$ three times, which yields more accurate results. Computing the loss gradient with respect to the model parameters is still possible for continuous-time models as the ODE solvers are just functions themselves which can be differentiated. The ODE solver can also be run as a black-box without knowing its internal operations as shown in [CRBD19]. Then the gradients with respect to the functions applied by the ODE solver can be computed by the adjoint sensitivity method [Pon62]. As pointed out by [HLA+20, p. 3] this memory-efficient procedure however comes with numerical errors as it forgets the forward-time computational trajectories. The CT-RNN model implementation used in this thesis is exposed under the `get_ct_rnn_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/ct_rnn.py`.

2.5 CT-GRU

The CT-GRU (continuous-time gated recurrent unit) recurrent neural network architecture is a continuous-time machine learning model firstly introduced in [MKL17]. The implementation of the CT-GRU architecture used in this thesis was taken from the repository of [LH20]. It shares many concepts with the GRU architecture 2.3, but the update gate 2.7 and reset gate 2.8 operate on multiple hidden state vectors stored across various time scales. This is done because some information may become obsolete very quickly, whereas some other information may also be very important in the longer term. These rates of information decay are referred to as time scales. The time scales are represented using time constants and the amount of time scales was fixed to 8 in this thesis. Therefore, the update gate is then called storage scale and the reset gate is then called retrieval scale as they operate not only on a single hidden vector, but across hidden vectors stored across multiple time scales. They can be thought of as multi-dimensional gates. As the amount of time scales is fixed, input data that matches a certain time scale not present in the fixed set must be approximated using a combination of the available time scales. This is indeed possible with small error when the time scale to approximate is in a certain range as pointed out in [MKL17, p. 5-6]. Then the half life of the combination of exponentials approximately matches the half life of the corresponding exponential to the correct time scale. A good match for time constants τ_i representing the various time scales is the set of constants where $\tau_0 = 1$ and $\tau_{i+1} = \sqrt{10} * \tau_i$. This set was also used in the benchmarked implementation. The explicit time input called Δt_k of this model was not used as interval to integrate an ODE, but instead as time duration of exponential decay between two input vectors. As all benchmarks do not provide time inputs and the input vectors of the benchmarks are regularly sampled, Δt_k was set to constant 1. The function the GRU model is applying to its input vectors to produce the output vectors or hidden state vectors is given as follows with inputs denoted as x_k and

outputs which equals the hidden states denoted as h_k [MKL17, p. 7]:

$$\ln \tau_k^R = W^R * x_k + U^R * h_{k-1} + b^R \tag{2.14}$$
$$r_{ki} = softmax_i(-(\ln \tau_k^R - \ln \tau_i)^2) \tag{2.15}$$
$$q_k = tanh(W^Q * x_k + U^Q * (\sum_i r_{ki} *_{ew} \tilde{h}_{k-1,i}) + b^Q) \tag{2.16}$$
$$\ln \tau_k^S = W^S * x_k + U^S * h_{k-1} + b^S \tag{2.17}$$
$$s_{ki} = softmax_i(-(\ln \tau_k^S - \ln \tau_i)^2) \tag{2.18}$$
$$\tilde{h}_{ki} = [(1 - s_{ki}) *_{ew} \tilde{h}_{k-1,i} + s_{ki} *_{ew} q_k] * e^{-\frac{\Delta t_k}{\tau_i}} \tag{2.19}$$
$$h_k = \sum_i \tilde{h}_{ki} \tag{2.20}$$

Multiplication which are meant to be understood element-wise are denoted with a subscript ew, otherwise the notation is kept the same as in the previous models. The equations 2.14 and 2.15 determine the retrieval scale and compute the weighting for each time scale. The equations 2.17 and 2.18 determine the storage scale and compute the weighting for each time scale. The retrival scale vector r_{ki} is the multi-dimensional equivalent to the reset vector of the GRU architecture and the storage scale vector s_{ki} is the multi-dimensional equivalent to the update vector of the GRU architecture. Equation 2.16 describes how the next candidate hidden state vector s_k is computed. Finally, equation 2.19 describes how the hidden state for each time scale is updated and equation 2.20 describes how the output vector h_k is computed out of the multi-dimensional hidden state vector. It can be said that the CT-GRU architecture is a GRU model with multi-dimensional state and exponential decay of its state between input vector observations with different time constants. Most of the features discussed for the GRU model are also applicable for the CT-GRU architecture. It should also be able to learn long-term dependencies as time scales featuring a large time constant have little decay on their corresponding hidden state and then simply the argument used in the GRU architecture 2.3 can also be applied here. As other models, the CT-GRU has a configurable hidden state vector size, which was picked to 32 throughout this thesis. The CT-GRU model implementation used in this thesis is exposed under the `get_ct_gru_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/ct_gru.py`.

2.6 ODE-LSTM

The ODE-LSTM recurrent neural network architecture is a continuous-time machine learning model firstly introduced in [LH20]. The implementation of the ODE-LSTM architecture used in this thesis was taken from the repository of its original paper [LH20]. The idea behind this model is to combine the abilitiy of the LSTM architecture to capture

long-term dependencies and the ability of CT-RNNs to accurately model dynamical physical systems, even if an irregularly sampled time-series is provided to the model as input. As in this thesis, only regularly sampled time-series are used, the continous time model is always fed with the time input 1 as mentioned in the 2.4 section. This should be no problem as the ability to model dynamical physical systems generalizes to any time input very well. Similar to the LSTM architecture, the ODE-LSTM has two state vectors: one hidden state vector h_i and one cell state vector c_i. Both vectors are initialized to the all-zero vector. The function the ODE-LSTM model is applying to its input vectors to produce the output vectors or hidden state vectors is given as follows with inputs denoted as x_i and outputs denoted as h_i [LH20, p. 5]:

$$(c_i, h_i') = LSTM(x_i, (c_{i-1}, h_{i-1})) \tag{2.21}$$

$$h_i = CTRNN(h_i', (h_{i-1})) \tag{2.22}$$

The function $LSTM$ denotes one model function step of the LSTM model 2.2 starting from the given state (c_{i-1}, h_{i-1}) for input x_i. The function $CTRNN$ denotes one model function step of the CT-RNN model 2.4 starting from the given state (h_{i-1}) for input x_i, the input is set to 1 for each time step. Implementation-wise, the CTRNN model function call was done to the implementation as described in 2.4. The LSTM model function was implemented from scratch and no library modules were used. As only the hidden state vector of the LSTM architecture is postprocessed by the CT-RNN model, the cell state stays untouched, which enables the architecture to learn long-term dependencies by using the same argument as in 2.2. By the postprocessing of the hidden state vector which controls the LSTM's gates, the gating dynamics become dependent on the time input as well [LH20, p. 4]. Of course also the ODE-LSTM architecture has a configurable hidden state vector size which was picked to 64. The CT-RNN was initialized by the same hidden vector size, the number of unfolds were set to 4 and the explicit Euler method was used as an ODE solver. The ODE-LSTM model implementation used in this thesis is exposed under the `get_ode_lstm_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/ode_lstm.py`.

2.7 Neural Circuit Policies (NCP)

Neural Circuit Policies were used in the paper [LHA+20] which shows the high expressivity of the architecture in the domain of autonomous driving. The architecture is a subset of all LTC (Liquid Time-Constant) Networks that were introduced in [HLA+18] and further discussed in [HLA+20]. An LTC Network consists of biologically inspired neurons with leakage that are interconnected using chemical synapses with non-linear activations. LTC Networks model the cell membrane as integrator and are therefore a continuous-time machine learning model. Neural Circuit Policies were derived from the neuron interconnection structure of the Caenorhabditis elegans nematode [LHA+20, p. 3] which

trims the space of all possible LTC networks. The state of each neuron i with incoming chemical synapses from neurons j is given as its potential V_i and the ODE that describes the dynamics of a single neuron's potential is given by [HLA+18, p. 1-2]:

$$\dot{V}_i(t) = \frac{1}{C_i} * (I_{leak,i} + \sum_j I_{syn,ji}) \tag{2.23}$$

$$I_{leak,i} = G_{leak,i} * (E_{leak,i} - V_i(t)) \tag{2.24}$$

$$I_{syn,ji} = [G_{syn,ji} * sigmoid(\sigma_{ji} * (V_j(t) - \mu_{ji}))] * (E_{ji} - V_i(t)) \tag{2.25}$$

By reordering terms in equation 2.23, it can be shown that the time constant τ as used in the equation 2.12 in the CT-RNN architecture is varying with time. The capacitance of a neuron i is denoted as C_i and the whole equation will be more familiar when the capacitance is brought to the left hand side which yields $C_i * \dot{V}_i(t) = I_{leak,i} + \sum_j I_{syn,ji}$. This is just the differential equation describing the behavior of an electrical conductance. All the currents on the right hand side as given in the leakage current equation 2.24 and in the chemical synaptic current equation 2.25 are written according to Ohm's law $I = \frac{U}{R}$. By using the conductance G instead of the resistance R which is just the reciprocal value, the equation yields $I = G * U$, exactly the form both current equations are using. As the voltage U is given as the potential difference, all terms in equation 2.24 and 2.25 should be clear now. Worth mentioning is the non-linear conductance for chemical synaptic currents given as $G_{syn,ji} * sigmoid(\sigma_{ji} * (V_j(t) - \mu_{ji}))$, where the parameter $G_{syn,ji}$ controls the maximum conductance, the parameter μ_{ji} controls the mean conductance potential and the parameter σ_{ji} controls the steepness of the transition between conductance and non-conductance. Note that the non-linear synaptic conductance is only influenced by the presynaptic neuron potential $V_j(t)$. The potentials given by the capital letter E control the targeted potentials for the neuron i, therefore if the neuron has reached this potential the corresponding currents will vanish. The NCP architecture builds its output vector by determining output neurons in the same amount as the output vector size. These neurons are called motor neurons and their vectorized potentials then build the output vector. The input vector entries are fed to neurons as currents by using the chemical synaptic current equation as described in 2.25 and by setting the presynaptic potential equal to the input vector entry. Furthermore, before the input vector is provided to the NCP model and before the output vector is returned from the NCP model, an affine transformation is applied to the input and output vector by mapping both vectors with a dense layer as described in the LSTM section 2.2. Additionally to motor neurons, NCP models also have inter and command neurons. Inter neurons receive input vector entries as chemical synaptic currents and command neurons are the only neuron type where recurrent connections are allowed. Command neurons also are the only neuron type which has synaptic connections to motor neurons. Therefore, the input vector entries are processed using the inter neurons, which feed the processed information to the command neurons that control the motor neurons and therefore the output vector entries. The procedure to create the synaptic wiring is described in detail in [LHA+20, p. 3] and will not be covered in this thesis. The NCP implementation used for benchmarking uses the implementation provided in the repository of the paper [LHA+20] located under the

url `https://github.com/mlech26l/keras-ncp`. It was configured with 9 inter neurons and 7 command neurons. The amount of motor neurons was picked according to the required output vector size. There were two incoming synapses from input vector entries to inter neurons and two incoming synapses from inter neurons to command neurons. Each motor neuron receives two incoming synapses from command neurons and there were 14 recurrent synapses in all command neurons. The time input to solve the ODE was set to 1 per time step and the ODE was solved using the Fused Solver proposed in [HLA+20] that fuses explicit and implicit Euler methods. Per time step the ODE was unrolled 6 times, as there are at least 3 unrolls necessary until the currents from the input vector reach the command neurons via synapses in each time step. The initial potential of all neurons was picked to 0. The NCP model implementation used in this thesis is exposed under the `get_neural_circuit_policies_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/neural_circuit_policies.py`.

2.8 Unitary RNN

The Unitary RNN architecture was first introduced in [ASB16] and later refined in [JSD+17] and is a discrete-time machine learning model. It uses the same vanilla recurrent neural network model function as discrussed in section 1.5. The Unitary RNN implementation used in this thesis is a modified version of the implementetion provided with the original paper which can be found under the url `https://github.com/jingli9111/EUNN-tensorflow/blob/master/eunn.py`. The next hidden state vector of a Unitary RNN h_{t+1} which also equals its output vector is computed as given in [JSD+17, p. 2] by using a non-linear bias-parametrized activation function σ and two matrices W and V:

$$h_{t+1} = \sigma(W * h_t + V * x_{t+1}) \tag{2.26}$$

The bias-parametrized activation function σ was set to the modrelu function firstly introduced in [ASB16, p. 4]. The modrelu function applied to a complex vector z is defined as follows for each vector entry z_i: $moderelu(z_i) = max(0, |z_i| + b_i) * \frac{z_i}{|z_i|}$ with a real-valued bias parameter b_i per vector entry. The initial hidden state vector h_0 was picked to the all-zero vector. The difference with this model is that it does not use real parameters which is the standard in machine learning. It uses complex parameters which are represented by two single-precision floating-point parameters each. The parameter count for each model however is always given in terms of single-precision floating-point parameters. The matrices W and V are parametrized as complex matrices. Matrix V does not have to follow any particular restrictions, therefore it can be simply parametrized by two real matrices V_{real} and V_{imag} for the real and imaginary part. As explained in detail in section 1.5, a matrix W that fulfills $\|W\|_{2,ind} = 1$ and a suitable activation function σ would solve the vanishing and exploding gradient problem for the vanilla

recurrent neural network architecture and exactly this was done in the case of Unitary RNNs. Unitary matrices W fulfill the requirement $\|W\|_{2,ind} = 1$, as all eigenvalues of unitary matrices have a magnitude of 1 from which follows that 1 is always the largest singular value as unitary matrices are square. As the spectral norm is just the largest singular value, it is proven that unitary matrices fulfill the proposed requirement. The difficulty now is to parametrize unitary matrices efficiently as they are only a subset of all complex matrices and therefore cannot be as simply parametrized as the matrix V. The method to parametrize unitary matrices as used in [JSD+17, p. 3] was proposed by [CHM+17] and is called square decomposition method. The core statement is that any unitary matrix of dimension $N \times N$ can be represented by matrix multiplications involving a diagonal matrix D and rotational matrices R_{ij} as follows:

$$W = D \prod_{i=2}^{N} \prod_{j=1}^{i-1} R_{ij}. \tag{2.27}$$

The diagonal matrix D has only the entries e^{iw_j} on its diagonal which results in N parameters w_j. The matrices R_{ij} which are parameterized by two real parameters θ_{ij} and ϕ_{ij} are defined as N-dimensional identity matrices whose four entries at positions given as $(row, column)$ are replaced with given entries as follows:

$$\begin{bmatrix} (i,i) & (i,j) \\ (j,i) & (j,j) \end{bmatrix} \mapsto \begin{bmatrix} e^{i\phi_{ij}} \cos(\theta_{ij}) & -e^{i\phi_{ij}} \sin(\theta_{ij}) \\ \sin(\theta_{ij}) & \cos(\theta_{ij}) \end{bmatrix} \tag{2.28}$$

By reordering and grouping rotational matrices as shown in [JSD+17, p. 4], the unitary matrix W with even capacity L can also be written as:

$$W = D * F_A^{(1)} * F_B^{(2)} * F_A^{(3)} * F_B^{(4)} * \ldots * F_B^{(L)} \tag{2.29}$$

Whenever the capacity L matches the dimension N of the unitary matrix W, this expression spans the entire space of all unitary matrices. Whenever the capacity L is smaller than the dimension N of the unitary matrix W, this expression spans a subspace of the space of all unitary matrices. The matrices $F_A^{(l)}$ and $F_B^{(l)}$ are constructed as follows where superscript (l) denotes different instances of the same type of rotational matrices when the subscript matches:

$$F_A^{(l)} = R_{1,2}^{(l)} * R_{3,4}^{(l)} * R_{5,6}^{(l)} * \ldots * R_{N/2-1,N/2}^{(l)} \tag{2.30}$$

$$F_B^{(l)} = R_{2,3}^{(l)} * R_{4,5}^{(l)} * R_{6,7}^{(l)} * \ldots * R_{N/2-2,N/2-1}^{(l)} \tag{2.31}$$

Furthermore, each matrix F of the above two types is a general rotational matrix and its mapping performend on a vector x can also be written as [JSD+17, p. 4]:

$$F * x = v_1 *_{ew} x + v_2 *_{ew} permute(x) \tag{2.32}$$

The vectors v_1 and v_2 are computable from the parameters θ_{ij} and ϕ_{ij} that are used to parameterize the rotational matrices R_{ij} that build the matrix F. The permutation given

by the function *permute* is fixed and set only at the first instantiation of the machine learning model. The formula used to generate both vectors v_1 and v_2 is given under [JSD+17, p. 4]. As this way of applying the mapping of the F matrices to the input vector avoids matrix multiplications and just uses element-wise multiplications and permutation operations, it is an efficient way to parameterize unitary matrices. As the output vector of this machine learning model is complex, the real part of the output was used for further processing as this was also done in benchmarks from the official repository of the paper [JSD+17] which can be found under the url `https://github.com/jingli9111/EUNN-tensorflow/blob/master/copying_task.py`. Also this model has a configurable hidden vector size which must be even and was picked to 128 throughout the thesis. The capacity L was always set to 16, therefore the matrix W is parameterized as a partial-space unitary matrix. As the output vector size has the be variable, the real part of the output vector of the model was then fed to a dense layer to achieve the right output vector dimension. The Unitary RNN model implementation used in this thesis is exposed under the `get_unitary_rnn_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/unitary_rnn.py`.

2.9 Matrix Exponential Unitary RNN

This machine learning model is an original contribution and a variant of the Unitary RNN 2.8 architecture. It is therefore also a discrete-time recurrent neural network model with the same model function as specified in 2.26 and augments the architecture in various ways. Only the differences between the two architectures will be listed. First, the option to use a trainable initial hidden state vector was added to the architecture which is initialized to the all-zero vector. Furthermore, there was an option added to use an augmented input for the model. This augmented input consists of the concatenation of the regular input vector x_k per time step and its 1D discrete Fourier transform as given by $FFT(x_k)$. As problems in the signal and system theory domain are either easier to solve in the time or the frequency domain, this feature may help to make better predictions in some tasks. Moreover, the DFT matrix used to convert a time-domain vector to the frequency domain is also a unitary matrix, which preserves the energy of the input vector, and is therefore a good fit for this architecture. Both described features are disabled during benchmarking of this model, as they showed no substantial decrease of the final test loss. Another difference to the Unitary RNN architecture 2.8 is the construction of the output vector of the required size. As the imaginary part of the hidden state vector may also convey useful information, the approach from [ASB16, p. 4] was used in the implementation to construct the output vector. With this method the final output vector is constructed by passing a concatenated vector consisting of the real and imaginary part of the hidden state vector which is now solely real through a dense layer to get the right output vector dimension. The last difference is the parametrization of the

unitary matrix W used in the model function equation 2.26. As presented in section 2.8 the parametrization is quite involved and therefore the new way of parameterizing the unitary matrix is using an approximated matrix exponential. As any unitary matrix W of dimension $N \times N$ can be written as the matrix exponential of a skew-Hermitian matrix A of dimension $N \times N$ as $W = e^A$, the problem is reduced to parameterizing a skew-Hermitian matrix A. This matrix exponential is the matrix generalization of $|e^j| = 1$ in the scalar case where j is any imaginary number. The approximated matrix exponential implementation used for this model is exposed under the function `tf.linalg.expm` in the Tensorflow library [AAB+15] which uses Padé approximation as desribed in [AMH09]. The fundamental idea is that $e^A = (e^{2^{-s}A})^{2^s} \approx (r_m(2^{-s}A))^{2^s}$ where $r_m(X)$ is the $[m/m]$ Padé approximant to e^X and the non-negative integers m and s are to be chosen [AMH09, p. 1]. An approximation is needed as the matrix exponential e^A is defined by an infinite sum as follows:

$$e^A = \sum_{k=0}^{\infty} \frac{A^k}{k!} \tag{2.33}$$

A skew-Hermitian matrix A fulfills $A^H = -A$ which implies that the individual matrix entries fulfill $a_{ij} = -\overline{a_{ji}}$. This further implies that the diagonal entries of A are purely imaginary. Therefore, the square skew-Hermitian matrix A can be parameterized by only a lower triangular matrix T with complex entries as all other entries follow by symmetry. The diagonal entries in this matrix T can be parametrized with a real parameters, therefore saving N parameters, but this optimization was not applied in the implementation. The skew-Hermitian matrix A can easily be constructed by the triangular matrix T by the following formula fulfilling all symmetry requirements:

$$A = T - T^H \tag{2.34}$$

As in this formula 2.34 only the diagonal entries overlap after the transposition, the diagonal entries will be purely imaginary as the real parts will cancel themselves. All other entries follow the predescribed symmetry. In this model's implementation the matrix T was parameterized by a vector v of size $N * (N + 1)/2$ which equals the number of all non-zero elements in T. This vector v was then converted to a triangular matrix by filling a triangular matrix with all the values from T. With this method any lower triangular matrix T can be constructed, from which any skew-hermitian matrix A can be constructed, from which any unitary matrix W can be computed by using the matrix exponential. This parameterization allows to parameterize the full-space of unitary matrices. If a partial-space parametrization is favored to reduce the parameter count of the model, there is a capacity measure c available in the model's implementation which should fulfill $0 \leq c \leq 1$. With this only the first $\lfloor c * N * (N + 1)/2 \rfloor$ entries of the vector v will be trainable and the remaining entries will be filled up with zeros. The model used for benchmarking had a hidden vector size of 128 and the capacity measure c set to 1, therefore the full space of unitary matrices was parameterizable. The Matrix Exponential Unitary RNN model implementation used in this thesis is exposed under the `get_matrix_exponential_unitary_rnn_output` function defined in

the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/matrix_exponential_unitary_rnn.py`.

2.10 Unitary NCP

The Unitary NCP model is a novel discrete-time machine learning model that combines the Unitary RNN model 2.8 and the Neural Circuit Policies model 2.7, just like the ODE-LSTM model 2.6 combines the LSTM 2.2 and the CT-RNN 2.4 architecture. This combination however is not as tightly coupled as the ODE-LSTM architecture. This architecture uses a Unitary RNN to preprocess all input vectors of the input sequence x_k to an intermediate sequence by storing the real part of the hidden state vector at each time step without feeding it through a dense layer afterwards. Then this intermediate sequence is fed to the Neural Circuit Policies model, which just treats it as its regular input sequence and maps it to the output vector sequence o_k. The Unitary NCP model function is given as follows where x_k is the input vector at time step k, $h_{k,unitary}$ is the hidden state vector of the Unitary RNN model, $h_{k,ncp}$ is the state vector of the Neural Circuit Policies model and o_k is the output vector at time step k:

$$h_{k+1,unitary} = UnitaryRNN(x_{k+1}, h_{k,unitary}) \tag{2.35}$$

$$(h_{k+1,ncp}, o_{k+1}) = NCP(\mathrm{Re}\{h_{k+1,unitary}\}, h_{k,ncp}) \tag{2.36}$$

The $UnitaryRNN$ function is just a pointer to the corresponding model function described in 2.26. How the NCP model maps the input sequence to an output sequence as meant by the function NCP is described in detail in section 2.7. The architecture should combine the great expressiveness of the NCP model with the ability to capture long-term dependencies of the Unitary RNN model. The Unitary RNN was configured with a hidden state vector size of 32 and the capacity was set to 4. The NCP model uses 4 inter and command neurons and no recurrent command synapses, as memory-related tasks should be handled by the Unitary RNN. For details on both architectures, consult their individual sections. The Unitary NCP model implementation used in this thesis is exposed under the `get_unitary_ncp_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/unitary_ncp.py`.

2.11 Transformer

2.11.1 Introduction

The Transformer architecture introduced in [VSP+17] is no recurrent neural network architecture in the strict sense like the LSTM 2.2 or the GRU 2.3 architecture. It encodes

its input sequence using an encoder, whose output is then decoded to the output sequence by a decoder. However, this model has recurrence in its decoder part as explained later. The problem of capturing long-term dependencies as described in 1.5 originates as the input time-series is provided as one input vector per time step to the machine learning models. This results in deep computational graphs for longer time-series which furthermore leads to problems like vanishing or exploding gradients. The Transformer architecture overcomes this issue by considering the whole input vector sequence at a single time step for prediction. Attention mechanisms were used to deal with that much input data at a single time step which means the model learns to weight the input data vectors according to their relevance in solving the required problem. By using this method the computational graph becomes much shallower and easier to backpropagate through, therefore overcoming the unwanted deep computational graphs.

2.11.2 Encoder

At first the Transformer architecture passes its input vectors on to the encoder. The encoder embeds its input vectors into vectors of length d_{model} which is a hyperparameter of the architecture by passing them through a dense layer. As the embedded input vectors of the input time-series are not labelled with their corresponding time index k, the transformer architecture adds a positional encoding vector to each embedded input vector. This positional encoding vector is only dependent on the absolute position in the input time-series of the vector and the hyperparameter d_{model} as described in-detail in [VSP+17, p. 6] and should allow the architecture to infer its absolute position in the input time-series. After adding the positional embedding vectors dropout with a configurable architecture-wide dropout rate $0 \leq r \leq 1$ is applied to all embedded input vectors. Dropout was introduced in [SHK+14] and randomly sets vector entries to 0 with a frequency of r and vector entries not set to 0 are scaled up by $1/(1-r)$. According to [SHK+14, p. 1] this helps neural networks to prevent overfitting. The output vectors after dropout are then fed to a configurable amount of encoder layers. Each encoder layer consists of two sub-layers, one multi-head attention layer and one fully-connected feed-forward layer. Dropout is applied to each sub-layer output, whose result is then added to the input creating a residual connection [HZRS15]. Then layer normalization [BKH16] is applied to the sum of both which means the mean μ and variance σ^2 of all entries x in a vector are computed and these entries x are then mapped to $\frac{(x-\mu)}{\sigma+\varepsilon}$. The mapped vector entries are then normal distributed with mean 0 and variance 1, the ε in the formula is only added for numerical stability. This whole procedure of postprocessing the sub-layer output to the final outputs y given the inputs x can be also written in pseudocode [VSP+17, p. 3]:

$$y = LayerNormalization(x + Dropout(SubLayer(x))) \tag{2.37}$$

The multi-head attention layer requires three mandatory input arguments (queries, keys and values) and an optional attention mask. There must be as many keys as values as they are used as a key-value-pair. The number of heads h, the dimension of the projected

queries and keys d_k and the dimension of the projected values d_v can be configured. All queries, keys and values of the input arguments are mapped through three dense layers for queries, keys and values to the projected query, key and value vectors of the specified dimensions. This procedure is repeated h times with different dense layers but the same input. By writing the output vectors of this procedure in matrix form as queries Q, keys K and values V (vectors in rows), the scaled dot-product attention function output Y can be given as follows [VSP+17, p. 4]:

$$Y = softmax\left(\frac{Q * K^T}{\sqrt{d_k}}\right) * V \tag{2.38}$$

The matrix multiplication of Q and K^T corresponds to computing the scalar product of all combinations between query vectors and key vectors. The scalar product result of a single combination should describe how well the "question" or query matches the "answer" or key. If this result is high, it is said that the vectors attend to each other. These scalar products are then scaled, the attention mask is applied and after that the softmax function given as $\frac{e^{x_i}}{\sum_j e^{x_j}}$ is applied to each row and row entry x_i in the corresponding matrix. The attention mask is responsible for setting the scalar products of certain query-key combinations to $-\infty$ before the softmax function is applied to prohibit information flow from the corresponding value vector. This now results in a matrix where the entry in row i and column j corresponds to the attention weight between the query vector in row i and the key vector in row j. All attention weights of a single query vector to all possible key vectors add up to 1. The final output Y is then computed by doing a matrix multiplication of the attention weight matrix with the value matrix V, which equals to computing a new representation for each query according to a weighted sum of value vectors. Therefore, the corresponding key vector to a value vector describes how to access the information present in the value vector. This process can also be thought of as a continuous hash map where the key vector and value vector are the key-value-pairs and the indexing is done with a query vector. As the query and key vector may never be exactly equal, the values are weighted according to their relevance. The output Y of the individual heads are then concatenated together and projected back to output vectors of dimension d_{model} with a dense layer. The encoder layer uses this multi-head attention mechanism as self-attention which means query, key and value vectors are just the same input vectors each encoder layer gets as input. This process can be thought of exchanging information between all vectors. The second sub-layer in the encoder layer is the fully-connected feed-forward layer, which consists of a dense layer which maps the input vectors to size d_{ff} with a relu activation function and a second dense layer which maps the vector back to size d_{model} without an activation function. This process can be thought of exchanging information within all vectors. This concludes the functions present in the encoder layer.

2.11.3 Decoder

The output of the last encoder layer is then used in decoder layers of the same amount of encoder layers. The decoder gets a single start vector as input vector which was picked to the all-one vector. All input vectors to the decoder are then embedded, positional encoded and dropout is applied in the same fashion as for input vectors to the encoder. The self-attention sub-layer in a decoder layer sets the attention mask correspondingly such that input tokens to the decoder layer can only attend to other tokens up to the own token index which ensures the Transformer's auto-regressive property [VSP+17, p. 5]. The difference between decoder and encoder layers is that decoder layers have a third sub-layer function added between the two functions of the encoder layer. The third sub-layer function also uses multi-head attention, but the key and value vectors are provided by the output of the encoder whereas the query vectors are provided from the previous sub-layer output. This mechanism is called encoder-decoder attention and it is responsible for transferring information from the input vector sequence to the output vector sequence. The final output vectors of the last decoder layer are then passed through a dense layer which maps the outputs to the required dimension of *token_size*. The *token_amount* parameter determines how often the decoder architecture should be run. After a complete run of the decoder architecture, the output vector of size *token_size* corresponding to the last decoder input is concatenated to the list of all decoder inputs and the whole decoder architecture is run again, now with two or more input vectors for the decoder. This is the recurrence of the Transformer model present in the decoder as mentioned before. The output of the Transformer architecture is then a flattened version of all output vectors of the decoder excluding the first start vector.

2.11.4 Implementation

The implementation used for benchmarking had *token_amount* set to 1 and *token_size* set to the required output vector size. The hyperparameter d_{model} was set to 16, h was set to 2, d_{ff} was set to 64, there were 2 encoder and decoder layers used and the dropout rate was set to 0. The dimension d_k and d_v were always equal to d_{model} in the benchmarked implementation. The Transformer model implementation used in this thesis is exposed under the `get_transformer_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/transformer.py`.

2.12 Recurrent Network Augmented Transformer

The Recurrent Network Augmented Transformer architecture is a novel contribution and similiar to the Transformer architecture 2.11. The only difference is that it uses a slightly changed attention mechnism. As given in formula 2.38 the final output Y is constructed by a matrix multiplication of the attention weights and the value matrix V

which means that a new representation for the query vectors is computed by summing up the weighted value vectors per query vector. The idea now is that instead of summing up the weighted value vectors by ordinary summation, maybe the use of a recurrent neural network to accumulate the information present in the weighted value vectors can increase the expressivity of the Transformer architecture. Furthermore, the incorporated RNN can directly use positional information and the sum function is easy to learn for any RNN architecture which has a similar model function to 1.7. It just needs to learn that the W matrix should be identity matrix. A different RNN with different weights for each head was used. The implementaion used to benchmark the architecture uses the LSTM architecture for the described RNNs. The difference in hyperparmeters to the Transformer architecture are that this architecture sets d_{model} to 8, the number of heads h to 1, d_{ff} to 32 and the number of encoder and decoder layers to 1. The Recurrent Network Augmented Transformer model implementation used in this thesis is exposed under the `get_recurrent_network_augmented_transformer_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/recurrent_network_augmented_transformer.py`.

2.13 Recurrent Network Attention Transformer

The Recurrent Network Augmented Transformer architecture is a novel contribution and similiar to the Transformer architecture 2.11. The only difference is that it uses a completely new attention mechanism called recurrent network attention which uses recurrent neural networks. As in the Transformer architecture, this attention mechanism gets the four arguments queries, keys, values and an attention mask. The attention mask and keys argument is not used in this mechanism. The new representation of each query vector (the output of the attention mechanism) is computed by building a sequence of a single query vector concatenated with all value vectors. This sequence is as long as the amount of value vectors given in the values matrix from the argument and each concatenated vector in this sequence has size $2 * d_{model}$. Computing the new representation is then done by passing this sequence through an RNN and using the output after the last input vector for further processing. Of course also this attention mechanism supports multiple heads by passing the same sequence through multiple RNNs with different weights. The results are then concatenated together and projected back to vectors of size d_{model} with a dense layer to get the output vectors of this attention mechanism. The implementaion used to benchmark the architecture uses the Unitary RNN architecture for the described RNNs. The difference in hyperparmeters to the Transformer architecture are that this architecture sets d_{model} to 8, the number of heads h to 1, d_{ff} to 32 and the number of encoder and decoder layers to 1. The Recurrent Network Attention Transformer model implementation used in this thesis is exposed under the `get_recurrent_network_attention_transformer_output` function defined in

the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/recurrent_network_attention.py`.

2.14 Memory Augmented Transformer

The Memory Augmented Transformer architecture is a novel contribution and a discrete-time recurrent neural network architecture that incorporates a transformer model together with external memory. This model is therefore also a MANN (memory-augmented neural network). The external memory M represents the state of the model and has a configurable number of rows r and a configurable number of columns c. All memory fields are prefilled with a small value which was set to 10^{-6}. There are two embedding dense layers defined, the input embedding IE to embed the current step input and the memory embedding ME to embed each memory row of the external memory M. Then these embeddings which both project to a vector size of $embedding\_size$ are computed at each time step and the resulting vectors are concatenated together to a vector sequence of length $r + 1$. Then positional encoding vectors (denoted as PE in matrix form) are added to each vector in this sequence as described in 2.11.2. Moreover, dropout with rate dr is further applied on this vector sequence which is then fed through a single encoder layer with the functionality as described in 2.11.2. The first vector of the encoder layer vector outputs is then used to build the output y of the model by projecting it to the required output vector size through the output dense layer ODL. All other r output vectors of the encoder layers are then projected with the memory control dense layer $MCDL$ to a memory control signal vector per memory row of size $1 + c$ (denoted as MCS in matrix form). The first entry of this vector is called the enable signal and is used to activate the memory and write the remaining c entries to the memory. It can also deactivate the memory to mask the remaining c vector entries away and therefore resulting in keeping the current memory state. This was done by feeding the enable signal through a *sigmoid* function in the positive and negated form (both results add up to 1), which are then used to weigh the new and old memory state. At each time step, the following model function is executed with the input denoted as i_t and the output denoted as y_t:

$$z_t = Dropout(concat(ME(M_{k-1}), IE(i_k)) + PE) \tag{2.39}$$

$$e_t = EncoderLayer(z_t) \tag{2.40}$$

$$o_t = ODL(e_t[0]) \tag{2.41}$$

$$MCS_t = MCDL(e_t[1..r]) \tag{2.42}$$

$$M_k = sigmoid(-MCS_t[:, 0]) * M_{k-1} + sigmoid(MCS_t[:, 0]) * MSC_t[:, 1..r] \tag{2.43}$$

By incorporating the encoder layer, the architecture can freely choose how many memory rows it wants to read in a single time step, as this can be determined by the corresponding attention weights. Furthermore, by using the memory enable signal per memory row, the architecture may also freely determine how many memory rows it

wants to write in a single time step. This architecture tries to separate computation and memory just like personal computers do. The CPU equivalent in this architecture is the encoder layer, including all the dense layers, and the external memory is responsible for persisting information. In the implementation used for benchmarking r and c were set to 16, the embedding size was set to 32, the number of heads in the encoder layer was set to 2, the feed-forward size d_{ff} in the encoder layer was set to 128, and the dropout rate dr, as well as the dropout rate in the encoder layer, was set to 0. The Memory Augmented Transformer model implementation used in this thesis is exposed under the `get_memory_augmented_transformer_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/memory_augmented_transformer.py`.

2.15 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.

2.16 Memory Cell

The Memory Cell is a continuous-time recurrent neural network architecture consisting of two LTC neurons as described in the NCP section 2.7 without the input and output mapping using dense layers. It is a proof-of-concept implementation and tries to build an LTC network [HLA+20] that is able to capture long-term dependecies in time series, namely a single memory bit. For details on which inputs are provided to the model and what outputs are expected from the model, please consult the Cell Benchmark section 3.7.

The model has six synapses in total, each neuron had 3 incoming synapses. This model can only be used with input vectors and expected output vectors of size 2. The two entries of the input vector are the two scalar inputs that are passed on to the two neurons with a synaptic activation. As this model is built out of two neurons, the output vector size of this model is fixed to 2, and the output vector simply contains the potentials of both neurons. Each neuron has an input synapse, an inhibitory synapse and a recurrent synapse. The three synapses of each neuron share the same parameters as the Memory Cell model should employ the same mechanisms when storing a 0 or a 1 bit. The behaviour should be symmetric and the decision which bit to store should only be dependent on the current input. The current memory content of this architecture is encoded in potentials of both neurons. If the first neuron has high potential (≈ 1) and the second neuron low potential (≈ 0) the bit 1 is currently stored and if the second neuron has high potential (≈ 1) and the first neuron low potential (≈ 0) the bit 0 is currently stored. The purpose of the input synapse which connects the input vector entry with the synapse is to supply each neuron with a large input current to increase its potential to ≈ 1 if the input vector entry to the corresponding neuron is ≈ 1, too. It is only applicable that at most one neuron gets a large input vector entry (≈ 1) at a single time step. This either leads to a switch of the stored memory bit or to a persistence of the current memory state. It can also be the case that both neurons receive a small input vector entry (≈ 0), therefore both neurons receive little to no input current and the memory state is kept as ensured by the inhibitory and recurrent synapse. The inhibitory synapse which connects a neuron with the other neuron is responsible for suppressing the other neuron when a neuron itself has currently high potential, therefore it ensures that the second neuron's potential is kept low such that only one neuron can have a high potential. The recurrent synapse which connects a neuron with itself is responsible that a single neuron keeps its potential if it is not inhibited too much by the other neuron. Three synapses per neuron were at least necessary for a working Memory Cell architecture. The theoretical lower bound may be two synapses per neuron, as an input synapse and a communication synapse that handles communication between the two neurons is needed. The communication synapse would be connected from one neuron to the other and must fulfill the tasks of the recurrent and the inhibitory synapse. However, in this scenario with only two synapses assuming its proper functionality, at a memory switch, the communication synapse will have to supply a negative current to inhibit the other neuron. Furthermore, when there is no memory switch, the same communication synapse must provide a positive current to a neuron with high potential to keep its state as there is leakage current present. The sign of a synaptic current $I_{syn,ji}$ 2.25 is determined by the sign of $E_{ji} - V_i(t)$. The postsynaptic potential $V_i(t)$ may be ≈ 1 in both cases, therefore the different sign of $E_{ji} - V_i(t)$ cannot be determined by the parameter E_{ji} which yields a contradiction to the assumption of proper functionality. Each synapse from neuron j to neuron i has four parameters as described in 2.25: the maximum conductance $G_{syn,ji}$, the mean conductance potential μ_{ji}, the steepness of the conductance transition σ_{ji} and the target potential E_{ji}. The steepness of the conductance transition was fixed to 100 for all synapses in the implemented model. The conductances of all neurons including the leakage conductance G_{leak}

were parameters and therefore learnt. The target potentials for the the leakage current and the inhibitory synapse were fixed to 0, the target potentials of the recurrent synapse and the input synapse were parameters and threfore also learnt. The mean conductance potential of the input synapse was fixed to 0.5, the mean conductance potential of the recurrent and inhibitory synapse was a parameter of the model. The capacitance of all neurons was fixed to 1 and the fixed time input t per time step used to integrate the state derivative in a continuous-time model was also a learnt parameter. The ODE of the state was unrolled two times per time step. This is necessaey such that the input currents can propagate to each of the two neurons in the first unroll step and the inhibitory synpapse currents can propagate in the second unroll step in case of a memory switch. Therefore, this architecture has 9 learnable parameters. Validation of the model was performed using the Cell Benchmark 3.7. The initial state of the first neuron was 0 and the initial state of the second neuron was 1. The Memory Cell model implementation used in this thesis is exposed under the `get_memory_cell_output` function defined in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/model_factory.py`. The in-detail implementation is provided in the file `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/models/memory_cell.py`.

CHAPTER 3

Benchmarks

3.1 Benchmark Framework

3.1.1 Setup

A single code base to run and evaluate the diverse set of benchmarks and models was inevitable. Otherwise, the whole project would have been unmanageable. As the implementation of all models occurred in the `Python` programming language [VRD09] using the framework Tensorflow [AAB+15], also the benchmark framework used the same set of tools. Therefore, a benchmark base class was created in the file `benchmark.py`, which is available under the URL `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/benchmark.py`. The creation of a new benchmark is as easy as subclassing the benchmark base class `Benchmark`. For instructions how to call the newly created class, please consulate the `README.md` file given under the URL `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/README.md`. After subclassing the base class, the new class has to correctly call the superclass constructor and overwrite the abstract method `get_data_and_output_size`. Furthermore, the new benchmark's name should be added to the `BENCHMARK_NAMES` list constant. The superclass constructor only has two arguments: `name` and `parser_configs`. The first argument is just the name of the new benchmark passed as a string. The second argument should be a tuple of individual parser configs. A parser config is itself a tuple consisting of the argument name, the argument default value and the argument type. This argument determines which values should be settable and usable when calling the benchmark from the command line. There are at least three parser configs required that set the loss name, the loss config and the metric name. A sample `parser_configs` argument would be: `(('--loss_name', 'SparseCategoricalCrossentropy', str), ('--loss_config', 'from_logits': True, dict), ('--metric_name', 'SparseCategoricalAccuracy', str))`. If loss config or metric name is not applicable to the benchmark, simply set the default loss config to `{}` or the default metric

name to ". Furthermore, if the benchmark needs additional parameters, just extend the `parser_configs` parameter to also include the desired command line arguments. This feature is used by all individual benchmark implementations. After calling the superclass constructor all command line arguments configured through `parser_configs` will be available by their names as properties of `self.args` without the double hyphen. For example the loss name can be accessed by `self.args.loss_name`. If some parameters were set through the command line, they will have the corresponding value, otherwise the configured default values will be applied. After that the benchmark base class will create paths for some required directories. There are five directories required during benchmark execution: a saved model directory (will be created to save the models together with their best weights during training), a TensorBoard directory (will be created to save TensorBoard logs for eventual later evaluation), a supplementary data directory (already present in the repo to pass input data to the benchmark), a result directory (will be created to save csv files with relevant information about the training process) and a visualization directory (will be created to save visualizations created after each training of a model). All these paths start in the root folder of the repository called `NeuralNetworkArena`. The structure how these paths continue is the same for all five kinds of folders. For the next step in path creation, the individual name for the required folder kind will be appended to the root folder. These names can be passed as a command argument when calling the individual benchmark classes. For a more in detail description of these command line parameters just call an implemented benchmark class with the `--h` command line parameter as described in the `README.md` file. Then the name of the individual benchmarks is added to the path, such that each benchmark has its own five subfolders. Then the benchmark base class calls its `get_data_and_output_size` method that should have been implemented by the subclass. The function should return a tuple of inputs, a tuple of expected outputs and an output vector size of the machine learning model. The input and output tuple should only contain numpy arrays [HMvdW+20]. The output tuple must have size exactly one. The input tuple must have size at least one. The benchmark base class has also support for time inputs to the models. Please make sure that the time input is the last entry in the input tuple. There is also the command line argument called `use_time_input`. If you want to use time input, then make sure you have this argument set to true. Otherwise, if the input tuple has a dimension larger than one, the last entry will be discarded from the input tuple, as it is assumed to be the time input. The benchmark suite works currently only for benchmarks which provide time-series input data and only expect a model output after the last input data in the time-series. For people familiar with the Tensorflow framework [AAB+15] this is equivalent to setting `return_sequences=False` in an RNN model. All input arrays in the input tuple should have the shape `(SAMPLE_AMOUNT, SEQUENCE_LENGTH, INPUT_DIMENSION)`. Of course the input dimension can vary between different inputs. Time data should have an input dimension of one. The single output array present in the output tuple should have the shape `(SAMPLE_AMOUNT, OUTPUT_DIMENSION)`. The sample amount should match between input and output data to be valid input to the benchmark framework. All the constraints on the shapes will be checked by the

framework and then all individual samples are shuffled such that corresponding input and output data are at the same indices in their arrays. Then tensors are created with the same shape as the inputs in the input tuple excluding the first dimension that denotes the sample amount. These are required to later use the Functional API of the Tensorflow framework [AAB+15]. They are created by specifying a fixed batch size, which helps the machine learning framework to better optimize the computational graph for the corresponding model. The default batch size is set to 128 and can be changed by a command line parameter. After that, the whole samples are divided into test, validation and training samples. The amount of test and validation samples can be set via command line parameters, which default to 10% each. It is ensured that each individual sample set is exactly divisible by the batch size, as the computational graph was optimized by only allowing inputs of a fixed batch size as described above. After all the setup work is done, the folder paths to the result, the saved model and the TensorBoard directory will be augmented with the model name that is currently under test and which was passed via a command line parameter. Then the TensorBoard directory for that model will be deleted, as each training run creates a significant amount of log files. After that, the TensorBoard, the result, the saved model and visualization directory will be created if they do not already exist. Then it will be checked if the passed model name is present in the list constant `MODEL_ARGUMENTS` in the file `model_factory.py`. When this check is passed, the benchmark framework either loads a saved model with the corresponding model name or it creates a new one using the model output functions in the predescribed model factory depending on the command line parameter `use_saved_model`. These output functions get an output vector size and the tensor inputs and create an output tensor that contains all the information about the operations in between. By knowing the input and the output tensors, the Tensorflow [AAB+15] Functional API can be incorporated to create a machine learning model. If the model is newly created and not loaded from a saved one, the model is also compiled using a customizable optimizer, learning rate, loss, loss config and metric. These can be changed by command line parameters. The default optimizer and learning rate used throughout all benchmarks in this thesis are the Adam optimizer [KB17] and a learning rate of 10^{-3}. The three remaining parameters also discussed in the previous subsection must be passed such that it is conforming with the requirements of the functions `tf.keras.optimizers.get` and `tf.keras.losses.get`. A debug mode can also be enabled via the command line which puts the newly created model in eager execution mode making it easier to debug the model. Furthermore, the model will be called on a single batch of inputs without invoking the model's `fit` method. This happens only in debug mode. In any case a model ready to train should now have been constructed and all the models characteristics including input and output shape will then be printed to the command line enabling to check if all the dimensions match the expectations.

3.1.2 Training

After printing available information of the model to the command line a unix timestamp is retrieved from the system to keep track of the total training duration. Then the training is

ultimately started by invoking the model's `fit` method. This method takes the training and validation sample set, the batch size, the number of epochs and a tuple of callbacks as arguments. The number of epochs can also be configured in the command line, but throughout the thesis it is left to the default value of 128. The `fit` method calls the machine learning model function for each batch of inputs in the training sample set. After that the model is validated on the validation sample set. This means the loss function is computed on the validation data, which is data that the model is not trained on. Validating the model should help to determine how well the model will perform on actual test data, which is also data that the model is not trained on. If the loss function results for training and validation data are similar, it is said that the model generalizes well. When the validation step is finished the training loop proceeds with the next epoch and starts the same cycle again by providing the first batch of inputs from the training sample set. This cycle is repeated as often as the set value of the epochs. The callbacks are invoked after each completed epoch. There were five callbacks added: a ModelCheckpoint callback (saves the model with the best validation loss), an EarlyStopping callback (terminates training if the validation loss has not improved for a configurable number of epochs), a TerminateOnNan callback (terminates the training when a nan loss is encountered), a ReduceLROnPlateau callback (multiplies the learning rate by a configurable factor after no improvement of the validation loss for a configurable number of epochs) and a TensorBoard callback (saves TensorBoard log data for eventual later inspection). The default number of epochs used in this thesis for the EarlyStopping callback is 5. Another important callback is the TerminateOnNan callback, which terminates the training loop if the loss evaluates to nan. This can for example happen when the loss function diverges towards infinity, therefore if the exploding gradient problem appears. It may also be the case that there is a division through zero somewhere in the computational graph, which may also lead to a nan loss. The term nan just stands for `not a number`. As all benchmarked models are trained until convergence in this thesis, the ReduceLROnPlateau callback is especially important. The corresponding default parameters are a learning rate factor of 10^{-1} and a default number of epochs equal to 2, both of which are used throughout all benchmark invocations. The EarlyStopping and the ReduceLROnPlateau do not see an improvement if the absolute change in the validation loss is less than 0.0001. This minimum delta can also be configured via the command line, but this thesis uses the default value throughout all benchmarks. Furthermore, all these parameters are configurable by passing alternative values in the command line. After the training loop has terminated, another unix timestamp is taken to compute the total training duration.

3.1.3 Evaluation

The model is then evaluated using the parameters that led to the smallest validation loss during the whole training loop. Evaluation means that the model function is applied to the test sample set inputs and the resulting loss function result on that inputs is saved. The created model also provides an `evaluate` function, which takes the test sample set a batch size and another callback tuple as arguments. The only callback passed in the tuple is the TensorBoard callback already used in the `fit` method invocation.

3.1.4 Data Processing

The return values of the `fit` and `evaluate` method invocations now contain information about the means of the loss function results and of the metric function results on training, validation and test sample set. The means for the training and validation sample set are available for each training epoch together with the currently applied learning rate. All of that information is automatically accumulated in a single csv file per model for the training and the testing process. The testing results of all models are also merged in a single csv containing all model results for a single benchmark. Data that was generated during training is automatically visualized by the benchmark base class and will be presented in a future chapter that discusses the benchmark results in more detail. Of course all generated files will be stored in their respective directories.

3.2 Activity Benchmark

As described in the benchmark base class, all benchmarks feature time-series data where the model output is only used after the last time step to compute the loss function. This benchmark uses a slightly modified person activity recognition dataset from the UCI repository [DG17]. The mentioned dataset was distributed under the `https://archive.ics.uci.edu/ml/machine-learning-databases/00196/ConfLongDemo_JSI.txt`. The target function to learn is to map a sequence of measurements from four inertial sensors worn on the person's arms and feet to an activity classification. This benchmark should test a model's capability to model dynamical physical systems and understand what motion patterns belong to which class. The ability to capture long-term dependencies is not tested with this benchmark as the most recent input vectors should be enough to make good predictions. At each time step only the measurement of a single inertial sensor is presented as input to the model. The model can differ between the individual sensors as the modified dataset of person activity has a one-hot encoding to mark the sensor from which the current measurement is coming. All benchmarks feature an additional time input, where the time interval since the last input is passed on to the model if the feature is activated. However, this thesis has not used an additional time input for any benchmark. All the measurements used for this dataset were stored in the file `activity.csv` located in the supplementary data folder described in the benchmark framework section. The dataset is annotated with an activity classification for each time step, this benchmark however only requires the model to predict the classification corresponding to the last measurement data received. As the benchmark is a classification task a categorical cross-entropy loss was used that was computed from the output logits of the model. A categorical accuracy metric is used in this benchmark better judge how accurate the model predicts the activity class annotation corresponding to the last measurement input. Each model had an output vector size of seven, as there were seven different activity classes with their respective indices in brackets: lying (0), sitting on a chair (1), standing up (2), walking (3), falling (4), on all fours (5) and sitting on the ground (6). The processing of the UCI dataset was similarly done as in [LH20]. The benchmark had

a configurable sequence length, maximum sample amount and sample distance. For this thesis, a sequence length of 64, a maximum sample amount of 40000 and a sample distance of 4 was used. This means that each model gets a history of 64 measurements before it has to predict the activity corresponding to the last measurement. The maximum sample amount should bound the number of samples and in the case of 40000 and a sample distance of 4, there were enough entries in the dataset file, so the benchmark was run with 40000 samples in total. The sample distance is the indices offset in the dataset file between two drawn sample sequences. A model will get a sequence of 64 input vectors of size seven that look like: $[0, 0, 0, 1, 4.3, 1.8, 0.9]$. The first four entries in that vector represent the one-hot encoding that describes from which one of the four sensors the measurement data was taken. The remaining three entries contain the x, y and z coordinate of the corresponding sensor. The required output vector has just one entry as it is just the index of the corresponding activity class with the mapping as described above. As this is a sparse class encoding, the framework has to extend this output value to a one-hot encoding to apply a cross-entropy loss between the extended one-hot encoding and the output vector of our model after a softmax function was applied. The softmax function is necessary to convert the so called output logits to an output probability for each class. The results of this benchmark are presented in a later chapter. The implementation of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/activity_benchmark.py`.

3.3 Add Benchmark

This benchmark uses a the same structure as the add benchmark introduced used in [ASB16]. Data for this benchmark is generated randomly at each instantiation of the benchmark. The target function to learn is to simply add two numbers that are marked in a much longer stream of numbers. At each time step a number together with a marker bit is presented as input vector to the model. As in the Activity Benchmark 3.2, the sequence length and the sample amount are also configurable. For all models a sequence length of 100 and a sample amount of 40000 was used. As described above, the input vector has size two. The second entry is set to one only in one input vector of the first and last 50 input vectors. Their distribution is uniformly across the whole first and second half of the time-series. In all other input vectors this second entry is set to zero. The first entry of all input vectors is filled with random numbers taken independently and uniformly from the interval $[0, 1)$. A single input vector out of the 100 input vector each model gets during the benchmark looks like: $[0.5, 1]$. In this example the random number is 0.5 and it is marked, as the second entry is one. As described there are only two marked numbers and the expected output vector has size one and is simply the addition of both marked numbers. This benchmark simply uses the mean squared error loss function, as the smaller the mean square error is, the more similar the expected and the model output will be. Furthermore, there is no metric used in this benchmark. As this benchmark uses an increased sequence length of 100 and as described

the error signal is only provided after the last input vector, the model will be only able to learn this function when it is able to capture long-term dependencies. This means the model function must be designed in a way such that the gradient does not vanish or explode during backpropagation through the model's function. These problems was discussed in detail in chapter 1.5. When the model is not able to capture these long-term dependencies, therefore it is not able to store seen marked values in its state, the model will be forced to learn the naive memory-less strategy of always predicting one. This is the case as the expectation of each individual number out of the two marked ones is clearly 0.5, as they were drawn uniformly from the given interval. An addition of both expectation values reveals the output of the memory-less strategy. As also pointed out in [ASB16, p. 6], this naive strategy will lead to a mean squared error of $\frac{1}{6}$. This can be verified as the mean squared error when constantly predicting the mean is equal to the variance of the distribution. As both random numbers were picked independently of each other, the variance of the distribution of the sum of both random numbers is just the sum of their individual variances. The distribution from which the random numbers are drawn has variance $\frac{1}{12}$. Therefore, adding this value to itself proves the mean square error of the memory-less strategy. For this benchmark, the model output vector size is simply one, as it should just contain the sum of both marked numbers. The results of this benchmark are presented in a later chapter. The implementation of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/add_benchmark.py`.

3.4 Walker Benchmark

This benchmark evaluates how well a model can predict a dynamical, physical system's behavior and was taken from [LH20]. 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 MuJoCo physics engine [TET12] 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, the benchmark implements eventual frame-skips that would create an irregularly sampled time-series. This feature was not used in this thesis as it covers only regularly sampled time-series. Only if the model understands the dynamics that are guided by differential equations, it will be able to produce accurate predictions. The ability to capture long-term dependencies is not tested with this benchmark as the most recent input vectors should be enough to make good predictions. The benchmark had a configurable sequence length, a maximum sample amount and a sample distance just like the Activity Benchmark 3.2. Throughout the thesis a sample length of 64, a maximum sample amount of 40000 and a sample distance of 4 was used. All parameters have the same meaning as before. As there were enough training data provided in `.npy` files by the creators of [LH20], the benchmark had 40000 different samples available that are

partitioned in training, validation and test samples. The acquired simulation data can be downloaded from `https://pub.ist.ac.at/~mlechner/datasets/walker.zip`. The input sequence consists of input vectors of size 17, which contains the current state of the physics engine at this specific time step. These values represent the angles of the joints and the absolute position of the bipedal robot. The function to learn for this benchmark is to predict the physics engine's state in the next time step by giving the machine learning model a history of the past 64 physic engine's states. Therefore, the model output vector size was set to 17 and the expected output data were also vectors of size 17. As both vectors have the same size and the more similar they are, the better the prediction is, a mean squared error loss was used. There was no metric used for this benchmark. The results of this benchmark are presented in a later chapter. The implementation of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/walker_benchmark.py`.

3.5 Memory Benchmark

This benchmark evaluates how well a model can capture long-term dependencies by letting the model recall past seen categories exactly. It is a slightly changed version of the copying memory problem described in [ASB16]. Input data of the benchmark input is randomly created at each invocation of the benchmark. There is a configurable memory length to test for, a length of the sequence to memorize, an amount of categories and an amount of samples that are randomly generated. Throughout the thesis a memory length of 100, a sequence length of 1, a category amount of 10 and a sample amount of 40000 were used. Each single input vector sequence is created by concatenating three sub sequences. The first sequence is the sequence to memorize of length 1. It contains category indices sampled uniformly from 0 to 9. The second sequence is then just a sequence of the filler symbol 10 repeated 100 times. The third sequence is just the index of the category in the sequence to memorize that the model should recall, which is also sampled uniformly from all available indices in the sequence. This sequence is obviously of length 1 and always filled with 0 in the case of the predescribed setup. In total this makes up for a total sequence length of 102 and a vector size of 1 per time step. The expected output category is encoded sparsely as in the Activity Benchmark 3.2 and contains a category index from 0 to 9 that matches the category at the index the model got at the last time step in the sequence to memorize. The output vector size of the model is 10 and each output logit represents a single category. As this is a classification problem, a categorical cross-entropy loss was used between the output logits of the model passed through a softmax function and the one-hot encoding extension of the sparsely encoded expected category index. To better visualize how good a model can actually recall the category, a categorical accuracy metric was added to this benchmark. It must be pointed out that a model is only capable of recalling the category seen in the first input vector if the gradient does not vanish or explode, as the error signal is only provided after the last time step. The results of this benchmark are presented in a later chapter. The implementation

of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/memory_benchmark.py`.

3.6 MNIST Benchmark

This benchmark evaluates how well a model can capture long-term dependencies, as for correct classification the model needs to incorporate input vectors from the distant past, which will be described below. The idea to incorporate this benchmark was taken from [LH20], which also features an event-based sequential MNIST classification problem. Input sequences for this benchmark were constructed from the MNIST dataset of the Keras framework [C+15]. The MNIST dataset contains images of hand-drawn digits of size 28 by 28 pixels where each pixel is encoded by a single integer from 0 to 255. All images are in grey-scale and a higher integer represents a darker pixel. The images were vectorized to a vector of size 784 and then split up to a sequence of vector chunks of size 8, which results in an input sequence length of 98. The expected output class index is just the digit the current image is representing. Furthermore, the benchmark has a configurable maximum amount of samples, which was set to 40000. As the MNIST dataset had enough image samples, all specified 40000 samples were used. A long-term memory of seen input chunks is indeed necessary to produce an accurate category prediction, as digits like 1, 4 and 9 may be indistinguishable when only considering the most recent seen input chunks. This corresponds to classifying the image only based on a lower fraction of the image visible to the model, where the upper fraction was cut away. A model that yields accurate results must not suffer from the vanishing or exploding gradient problem, as only then the whole picture can be taken into account for classification. The model output vector size was set to 10, as each output logit should represent a single digit. As the expected output digit is encoded sparsely, the same procedure as in the Memory Benchmark 3.5 is applied to compute the categorical cross-entropy loss. The performance of the models was also measured by using a categorical accuracy metric, which produces a more human-interpretable result than the chosen loss function. The results of this benchmark are presented in a later chapter. The implementation of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/mnist_benchmark.py`.

3.7 Cell Benchmark

This benchmark evaluates if the newly introduced memory cell architecture is able to store a single bit of information repeatedly including switching the memory state. Furthermore, it should be checked if the memory state vanishes or if it successfully persists over a long time horizon. This requires the ability to capture long-term dependencies as the input is provided sparsely to the model as described below. The benchmark has a configurable memory high symbol, memory low symbol, memory length, amount of cell switches and

amount of samples that are generated at each invocation of the benchmark. The memory high and low symbol represent the expected output symbol when either memory state is active but the memory high symbol is also used as input symbol to sparsely activate a specific memory state, all other inputs are then set to the memory low symbol. The memory high symbol was picked to 1, the memory low symbol was picked to 0, the memory length was picked to 128, the amount of cell switches was set to 2 and the sample amount was set to 40000. As the memory cell is a bistable memory element, there are two memory states that can be activated sparsely. The input vector at each time step has size 2. If both entries are 0, the current memory state should be kept. Otherwise, if a single entry is 1 and the other entry is 0, the corresponding memory state should be activated. The first part of the input sequence is constructed by activating any of the memory states sparsely as described above and the succeeding 127 vectors are all-zero vectors. This sub sequence now has length 128. The next sequence is built like the first one, but it activates the opposite cell state at the first time step, which corresponds to a cell switch. There are 2 further sub sequences of this kind. The final input sequence is then the concatenation of all three sub sequences and has length 384. In half of the samples either memory state is activated first in the concatenated sequence. The required model output vector is also given as a sequence of vectors of size 2, therefore the error signal is provided at each time step. The output sequence can be easily built from the input sequence by continuing to set its entry to 1 at the corresponding index until a new sparsely input is provided to the model. Therefore, the model may get the input sequence consisting of the following vectors: $[1, 0], [0, 0], [0, 0], ..., [0, 1], [0, 0], [0, 0]$ and is required to produce the following vectors of the expected output sequence: $[1, 0], [1, 0], [1, 0], ..., [0, 1], [0, 1], [0, 1]$. The sparse activation of the memory cell should lead to a permanent storage of the activation, until a new sparse input is provided to the model. As described above the model output vector size is 2 and a mean squared error loss without a metric was used, as more similar vectors lead to a better prediction. The results of this benchmark are presented in a later chapter. The implementation of this benchmark can be found under `https://github.com/Oidlichtnwoada/NeuralNetworkArena/blob/master/experiments/benchmarks/cell_benchmark.py`.

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.

[AMH09] Awad Al-Mohy and Nicholas Higham. A new scaling and squaring algorithm for the matrix exponential. *SIAM Journal on Matrix Analysis and Applications*, 31, 01 2009.

[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.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 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.

[C+15] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[CGCB14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[CHM+17] William R. Clements, Peter C. Humphreys, Benjamin J. Metcalf, W. Steven Kolthammer, and Ian A. Walmsley. An optimal design for universal multiport interferometers, 2017.

[CRBD19] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2019.

[CWV+14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014.

[DG17] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[GRUG17] Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. The reversible residual network: Backpropagation without storing activations, 2017.

[GSC00] Felix Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12:2451–71, 10 2000.

[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+18] Ramin M. Hasani, Mathias Lechner, Alexander Amini, Daniela Rus, and Radu Grosu. Liquid time-constant recurrent neural networks as universal approximators, 2018.

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

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

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

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[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.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[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.

[MAT20] MATLAB. *R2020b*. The MathWorks Inc., Natick, Massachusetts, 2020.

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

[Pon62] Lev S Pontrjagin. *The mathematical theory of optimal processes*. Wiley, New York, NY [u.a.], 1962.

[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.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[TET12] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[VRD09] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[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.