TU WIEN Informatics

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

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Hannes Brantner, BSc
Registration Number 01614466

to the Faculty of Informatics

at the TU Wien

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

Vienna, 31st April, 2021

Hannes Brantner | Radu Grosu

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 the general case, a computer scientist wants to find the global minimum of that function with respect to all machine learning model parameters. As this is a problem that cannot be solved analytically in most cases, it is approximated by using gradient descent [RHW86]. This method incrementally changes each parameter depending on the gradient of the loss function with respect to each parameter in lock step. By denoting the loss function with L, the learning rate with α, the old whole parameter set with p, the old single scalar parameter with p_i and the new single scalar parameter with p'_i, the formula to update the individual parameters p_i in a single gradient descent step can be given as follows:

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

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

1.2 Problem Statement

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

1.3 How to better model Physical Systems

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

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

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

This form of system description is applicable to all physical systems in our daily surroundings, most of them are even time-invariant. This means the functions f and h do not depend on the absolute time t. For example, a mechanical pendulum will now

approximately behave the same as in one year, as its dynamics do not depend on the absolute time t. The system description given in 1.2 and 1.3 proposes, that machine learning models that are built in a similar fashion and whose state is also determined by a differential equation, should be pretty capable of modelling the input-output relation of physical systems. When the benchmarked models are introduced in more detail, it can be seen that all continuous-time machine learning models use a comparable structure in terms of parameterizing the state derivative and the output function.

1.4 Sampled Physical Systems

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

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

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

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

1.5 Why capturing Long-Term Dependencies is difficult

The difficulty will be outlined solely on the example of vanilla recurrent neural networks (RNNs). How transformer-based and advanced RNN architectures tackle the problem will be discussed later. Vanilla recurrent neural networks are discrete-time machine learning models. Its dynamics are given in a similar fashion to the equations that govern sampled physical systems 1.4. The current state vector h_t and the next input vector x_{t+1} determine the next state vector h_{t+1} and output vector y_{t+1} deterministically. In this model all the past inputs are implicitly encoded in the current state vector. This entails a big challenge for computer scientists, as computers only allow states of finite size and finite precision, unlike our physical environment, which results in an information

bottleneck in the state vector. The next state of a vanilla recurrent neural network h_{t+1} and its output y_t is typically computed by equations like the two proposed in [ASB16, p. 2] by using a non-linear bias-parametrized activation function σ, three matrices (W, V and U) and the output bias vector b_o:

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

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

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

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

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

1.6 Aim of the Work

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

1.7 Methodological Approach

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

1.8 State of the Art

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

CHAPTER 2

Models

2.1 LSTM

2.2 GRU

2.3 CT-RNN

2.4 CT-GRU

2.5 ODE-LSTM

2.6 Neural Circuit Policies (NCP)

2.7 Unitary RNN

2.8 Matrix Exponential Unitary RNN

2.9 Unitary NCP

2.10 Transformer

2.11 Recurrent Network Augmented Transformer

2.12 Recurrent Network Attention Transformer

2.13 Memory Augmented Transformer

2.14 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.

2.15 Memory Cell

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

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

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

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. 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. 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 crossentropy 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 100000 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 physics engine in an auto-regressive fashion using imitation learning. To increase the task difficulty, the simulation data was acquired from different training stages of the pre-trained policy (between 500 and 1200 Proximal Policy Optimization iterations) and 1% of actions were overwritten by random actions. Furthermore, frame-skips were introduced by removing 10% of all time-steps to get irregularly sampled training data. Moreover, the training data was divided into equally long sequences. For non-recurrent models, the input per time-step was the whole training sequence, but future values were zero-padded to ensure the model is only able to derive future predictions from past values. The model needs to predict the simulation state in the next time-step. Mean squared error was used as loss function when training the models. Further results of recurrent neural network models doing this benchmark can be found in [RCD19]. The results are presented in the following table:

3.5 Memory Benchmark

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

3.6 MNIST Benchmark

3.7 Cell Benchmark

This is a very small benchmark that tries to find the right parameter and input ranges for the memory cell 2.15. This benchmark should make clear if a memory cell is able to switch its state multiple times and keep information over a long period of time. It is directly implemented in the same file as the memory cell itself. The memory cell in the current test setup has no more recurrent, excitatory synapses. As discussed in our meeting the state of a memory cell is now only changed through four synapses, two per neuron. One synapse per neuron delivers the input via a synaptic activation and the other synapse ensures the opposite potentials of both neurons. The synaptic current can be computed by $g_{syn}(V_{pre}) * (E_{pre,post} - V_{post})$. I have found out that the LTC equations are not suited at all to build a memory cell, as a 1 input should deliver a big current and a 0 input should lead to a big negative current, no matter what the potential of the postsynaptic neuron is. However with the current equations, this is not possible to achieve, since the non-linear synaptic conductance is always positive and therefore the sign of the current cannot be determined by the input. Therefore, the underlying equations must be changed. The same argument holds for the mutual inhibitory synapses.

The higher potential neuron should send a negative current to the other neuron, whereas the lower potential neuron should send a positive current to the higher potential neuron. These things are not possible with the current setup, and therefore need to be changed. The memory cell receives only a single input that will be passed untouched to the first neuron's synaptic activation, but the second neuron will get the negated input to its synaptic activation for the input. The negation is currently implemented using the function $f(x) = 1 - x$. This input is only passed at the specific time step, when the input current provider needs to save something, otherwise both neurons in the memory cell get no input and should therefore hold their state. All memory cells are initialized to all contain zeros. As the output of the memory cell is the potential of the first neuron, this means that all first neurons in the memory cells have starting potential 0 and all second neurons have starting potential 1. The benchmark now feeds sparse inputs as described before to the memory cell, and requires the cell to hold the state from the last non-sparse input. This time horizon is the memory length of the memory cell, the amount of time it can save information. The benchmark not only checks the potential of the first neuron, but also the potential of the second neuron to be in the required range. Furthermore, the benchmarks alternates the symbol that should be saved first, half of the time it is a 0, half of the time it is a 1. Moreover, the memory cell state is switched again two times after the first input by providing two additional inputs, which represent the negated current memory cell potential, to check if the cell is able to switch its state. This results can be easily viewed when executing the memory cell Python script, they are not interesting since the memory cell does not learn anything.

CHAPTER 4

Results

CHAPTER 5

Summary

List of Figures

List of Tables

Bibliography

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

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

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

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

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

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

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

[GWR+16] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

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

[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.

[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.

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

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

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

[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.