Master Thesis Proposal

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

Faculty Supervisor: Univ.-Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Direct Supervisor: Dr. Ramin Hasani
Student: Hannes Brantner (01614466)

17. Februar 2021

CHAPTER 1

Proposal

1.1 Machine Learning Terms

A machine learning model is a mathematical parametrized function that gets 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 set the parameters of machine learning models optimally. Imitation learning means an associative expected output provided for each input 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 model's parameters' initial state, the returned model output will differ from the desired output in almost all cases. The measure responsible for quantifying this error between model output and the expected output is called a loss function and has a scalar return value. A sample loss function can be constructed quickly by 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 model's parameters. 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 a loss function to differentiate. The size of this input batch is called batch size. A computer scientist wants to find the global minimum of that function concerning all machine learning model parameters in the general case. A visualized loss surface where the loss function return value is plotted in the z-axis and all possible model parameter combinations are given as points on the plane is given as follows:

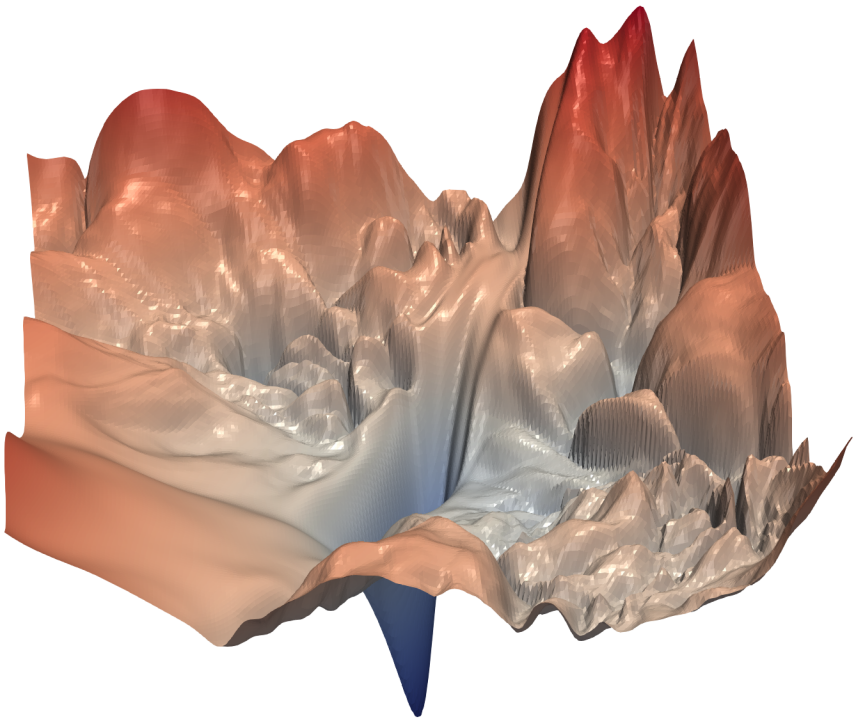

Figure 1.1: visualized loss surface [LXT+18, p. 1]

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 each parameter's loss function in a lockstep fashion. By denoting the loss function with L, the learning rate with α, the whole old 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 and 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, a decrease of the parameter will happen, leading 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 significant in magnitude the update to the parameters should be at each gradient descent step. A too-small learning rate will lead to slow convergence, and 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. The trajectory of the parameter set on the loss surface when repeatedly applying the gradient descent update rule was visualized in [CPGK19, p. 2] with the initial starting parameter set denoted as a black triangle as follows:

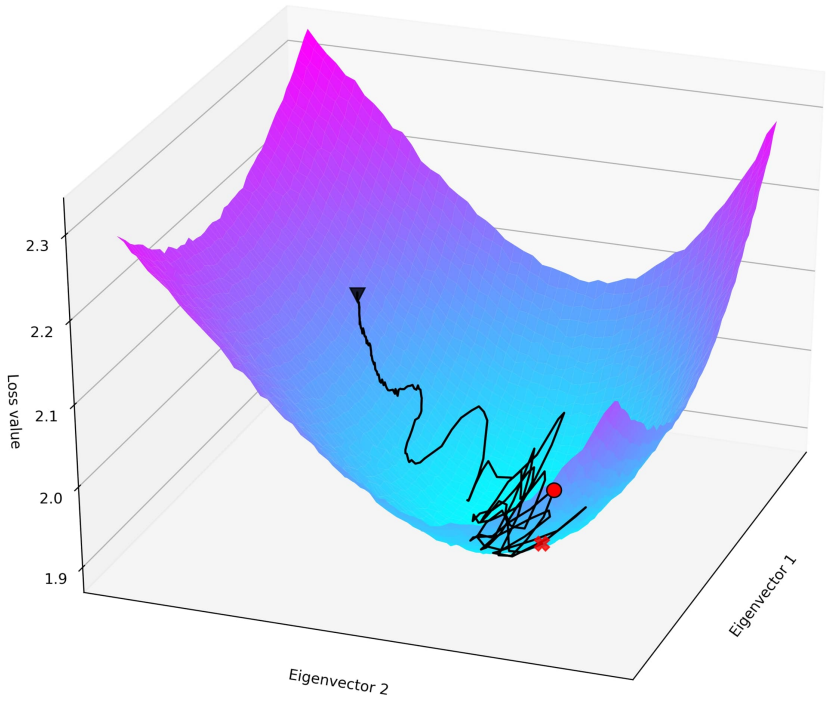

Figure 1.2: visualization of gradient descent

The differentiation of the loss function, which can be represented as a computational graph with lots of nested functions, involves many chain rule applications for the individual model parameter derivatives. 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:

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

The above equation 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 individual 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 layer's output vector, 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 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 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

Differential equations guide physical systems. 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 applies to all continuous physical systems in our daily surroundings. Most of these systems 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 presented in 1.3 and 1.4 proposes that machine learning models built similarly and whose state is also determined by a differential equation should be pretty capable of modeling 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. The key takeaway point is that continuous physical systems map an input function $x(t)$ to an output function $y(t)$ as visualized in [Smi97, p. 102]:

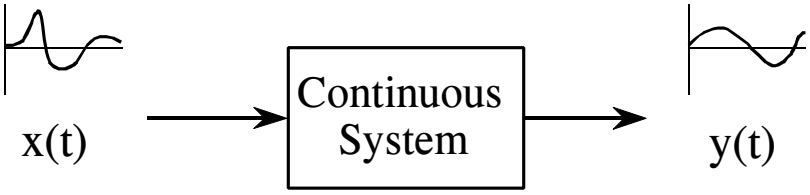

Figure 1.3: visualization of input-output relation of a continuous system

1.4 Sampled Physical Systems

As the current state's evaluation x at time point 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. Difference equations guide discrete systems. 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 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 modeling the input-output relation of this sampled physical system. The corresponding machine learning models are then called discrete-time machine learning models. The key takeaway point is that discrete physical systems map an input sequence $x[n]$ to an output sequence $y[t]$ as visualized in [Smi97, p. 102]:

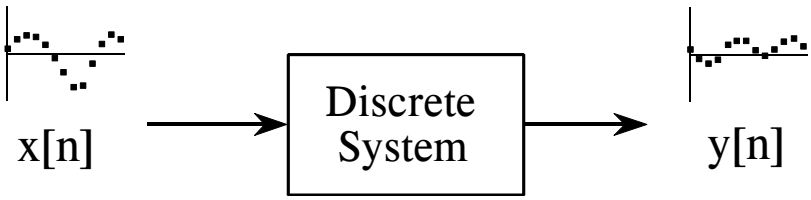

Figure 1.4: visualization of input-output relation of a discrete system

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 similar 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 implicit encoding 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 similar to those describing sampled physical systems. Equation 1.7 can be visualized by the following figure:

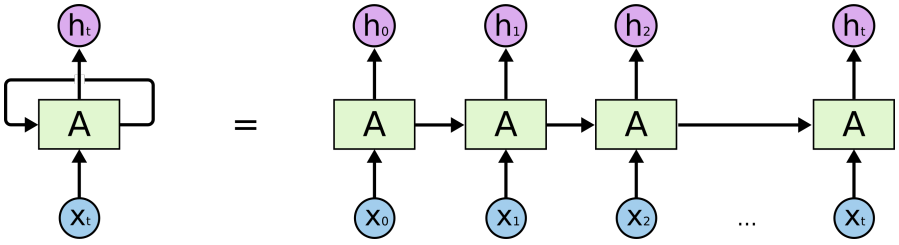

Figure 1.5: visualization of an RNN state update [Ma16]

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 persistence of the gradient 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 diagonal matrix's norm is close to zero over multiple time steps k, the desired loss gradient will decay towards zero. Otherwise, if the diagonal matrix's norm 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 precisely formulated 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 norm of the diagonal matrix and have the same implications. The first case is called 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 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 a dedicated 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 implementing 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 model's required output vector size. 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 benchmarks 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 can store information over many 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]. Nevertheless, some older architectures exploit continuous-time dynamics in machine learning models like the CT-RNN architecture [iFN93]. The other problem described in the previous chapters is the challenging task of capturing long-term dependencies in time-series. One solution for the problem was proposed in [ASB16], which introduced the Unitary RNN architecture. In principle, this architecture uses the vanilla RNN architecture described above. The difference is 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 gating mechanism. This mechanism changes the subsequent 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 simultaneously and directs its attention

to the inputs most relevant for solving the required task. This global access eliminates the need to backpropagate the error through multiple time-steps, which keeps the number of backpropagation steps low.

1.9 Structure of the Work

1. Introduction
2. Models
3. Benchmarks
4. Results
5. Summary
6. Appendix

1.10 Relevance to the Curricula of Computer Engineering

- 182.763 - Stochastic Foundations of Cyber-Physical Systems
- 186.844 - Introduction to Pattern Recognition
- 182.755 - Advanced Digital Design
- 191.105 - Advanced Computer Architecture
- 389.166 - Signal Processing 1
- 389.170 - Signal Processing 2
- 104.267 - Analysis 2
- 104.271 - Discrete Mathematics

List of Figures

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.

[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.

[CPGK19] Avraam Chatzimichailidis, Franz-Josef Pfreundt, Nicolas R. Gauger, and Janis Keuper. Gradvis: Visualization and second order analysis of optimization surfaces during the training of deep neural networks, 2019.

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

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

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

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

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

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

[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.

[LXT+18] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets, 2018.

[Ma16] Jianqiang Ma. All of recurrent neural networks. `https://medium.com/@jianqiangma/all-about-recurrent-neural-networks-9e5ae2936f6e`, 2016.

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

[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.

[Smi97] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, USA, 1997.

[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.