Towards Commoditizing Simulations of System Models Using Recurrent Neural Networks

Ahmet Caner Yüzügüler*
Signal Processing Laboratory (LTS4)
Ecole Polytechnique Fédérale de Lausanne
Lausanne, Switzerland
ahmet.yuzuguler@epfl.ch

Alexandru Moga, Carsten Franke
Automation System Architecture
ABB Corporate Research Center
Baden-Dättwil, Switzerland
{firstname.lastname}@ch.abb.com

***Abstract*—System modeling and simulation plays a crucial role in the engineering of large and complex systems from various fields, such as industrial automation or power systems. In this paper, we propose a method that can be used to easily deploy high fidelity simulations at scale, onto various target platforms. Out method is to approximate the behavior of the modeled system using a recurrent neural network. We use artificial neural networks as they easily lend themselves to high performance execution, thus avoiding the need to (manually) translate system models (typically a system of differential equations) to specialized hardware architectures. Moreover, this approach is generic in the sense that it is decoupled from typical modeling and simulation tools, such as Matlab Simulink or Dymola. This paper presents a proof-of-concept neural network architecture including the methodology for training that we used to approximate the behavior of different example systems originating from the electrical power systems domain. We present our evaluation results mainly regarding accuracy and to a certain extent performance on a GPU-based testbed. Furthermore, we detail limitations of the used approach and outline potential directions for research regarding the general applicability of our method.**

***Index Terms*—recurrent neural networks, differential equations, learning, simulation**

I. Introduction

System modeling and simulation plays a crucial role in the engineering of large and complex systems from various fields, such as industrial automation or power systems. In such domains, engineers and researchers make use of simulation tools (such as Matlab [7], Dymola [8]) that simulate various models often including systems of differential equations. Depending on the complexity and the required accuracy of the models to be simulated, this process can be slow. Possibilities exist to implement the models on FPGAs or GPUs in order to accelerate the simulations and possibly even achieve real-time performance. However, the translation from high-level models into hardware-level programming is rather challenging, error prone, and time consuming. Moreover, existing tools that perform the translation are vendor specific and closed source.

In this paper, we propose a method that can be used to easily deploy high fidelity simulations at scale on various target platforms , e.g., [10], [11]. Our approach is thus generic and a step towards achieving efficient simulations without the need to invest into (mainly manual) model transformations.

In order to enable efficient simulations, we distinguish between the modeling of a system, typically done in feature-rich tools, such as Matlab Simulink or Dymola, and its actual simulation on a given computing platform that is not bound to a specific modeling tool. One way to implement such an approach would be to use Functional Mock-up Units (or FMUs, [9]), which allow a model to be compiled into a library and wrapped around a standard interface for interacting with it (i.e., the Functional Mock-up Interface, or FMI). Nevertheless, the FMU library offers rather limited options for efficient simulations, e.g., on parallel hardware.

Our approach is to commoditize simulation models by making them reusable and parameterizable. To this end, we propose to use *recurrent neural networks* to approximate the behavior of system models. This is beneficial for two reasons: 1) the resulting approximations of system models can easily run in parallel (e.g., on GPUs or FPGAs), thus allowing faster and more efficient simulations, and 2) the resulting computation time is more predictable, thus making such a method eligible even for real-time simulations.

With our method, the typical translation of a high-level model into a corresponding hardware-level implementation is replaced by the training phase of the neural network. The challenges that we address here are to find the right set of training data and the right hyper-parameter configuration for the neural network. Once the neural network is trained, the result of the actual simulation in each time-step is mimicked by the outcome of the neural network inference process. The challenge here is to have the right neural network architecture.

Lagaris et. al [1] presented a method to solve ordinary and partial differential equations using artificial neural networks for initial and boundary value problems. In their method, artificial neural networks are being trained to predict the solution of differential equations based on the independent variables, i.e., in most cases time. It is demonstrated that the proposed method produces accurate results, and the solution obtained by the resulting trained neural network is in differentiable and closed-form. However, this method would not be applicable in the presence of time-varying inputs in the models of physical system components. This is actually the case in smart grids

*This work has been performed as part of the first author's Master thesis at the ABB Corporate Research Center.

where we have control inputs to the actual equipment (e.g., generator) or at different levels in the grid. This is indicative of a *non-autonomous system*, whereas the proposed method targets autonomous systems, i.e., the neural networks trained with the proposed method are time-based (by taking the time as input), and their prediction is based solely on time rather than state values. Therefore, to be able to represent more completely the model behavior, we need a *time-independent solution* to approximate non-autonomous systems. To this end, we introduce a novel type of recurrent neural network that is based on a time-independent numerical solution, whereby the neural network tries to learn the function that allows a system to transition from one state to the next.

In this paper, we present the benefits and trade-offs of approximating system models using neural networks in terms of simulation error vs. neural network size and training set size. We have chosen three types of use-cases to test our method for general applicability. The first focuses on solving nonlinear differential equations. The second is taken from the electrical domain and involves a transient model of a linear circuit. Finally, in the third use case our method tries to simulate a grid-connected PV (photo-voltaic) array. We have obtained very promising results in terms of the first two cases, while for the latter, the generalization potential of the neural network was found to be hindered by the scale of the model.

The remainder of the paper is structured as follows: Section II provides details of our time-independent numeric solution. The architecture of the recurrent neural network is described in Section III. Section IV presents details of the training method. The achieved results are shown in Section V using an initial GPU-based implementation, although our solution is general and applicable to various hardware setups. The paper ends with a summary and outlook in Section VI.

II. Time-independent Numeric Solution

The mathematical models of system components usually include time-varying input variables, which correspond to $u(t)$ in Equation 1. For example, the voltage level of a voltage source in a circuit, or the torque of a mechanical load attached to a motor are externally controlled variables. Their values do not depend on the solutions. Thus, the solutions to these differential equations vary with different values of $u(t)$.

$$\dot{\vec{x}} = A\vec{x} + \vec{u}(t) \tag{1}$$

In order to have a complete representation of a system model, our approximation needs to be input-agnostic; i.e for any set of inputs, the approximated behavior should be an accurate estimation of the actual solution. Supposing that a system model has a set of inputs, its approximation should produce an accurate solution regardless of the input values. This can be achieved by discretizing the solutions, and learning the state-update functions that predict the future state values based on the past state and input values, which is formulated in Equation 2.

$$\vec{x}[k] = h(\vec{x}[k-1], \vec{u}[k-1], \vec{u}[k]) \tag{2}$$

The purpose of this work is to train neural networks that estimate these state-update functions. As a state-update function works in a recurrent fashion, i.e it calculates the new state values based on their past values as well as the input values, it is best to utilize recurrent neural networks. Our proposed neural network is formulated in Equation 3, where $x[k] \in \mathbb{R}^N$ is a vector of N states, and $u[k] \in \mathbb{R}^M$ is a vector of M inputs at the step , and $\mathcal{N}_{w,b}$ is a multi-layer feedforward neural network with optimized weights w and biases b. Note that the output of the neural network corresponds to the difference in the state values between subsequent steps. During the training phase, this has proven to be numerically more stable in comparison to predicting the next values directly.

$$\vec{x}[k] = \vec{x}[k-1] + \mathcal{N}_{w,b}(\vec{x}[k-1], \vec{u}[k-1], \vec{u}[k]) \tag{3}$$

III. RNN Architecture

Our recurrent neural network architecture, which satisfies Equation 3, is depicted in Figure 1. The input nodes store the values of the n^{th} state in $(k-1)^{\text{th}}$ time-step, i.e., $x_n[k-1]$, and the values of the m^{th} external input in $k-1$ and k^{th} time-steps, i.e., $u_m[k-1]$ and $u_m[k]$, respectively. These nodes are connected to a set of fully-connected neural network layers with different number of neurons. The number of layers and their dimensions are problem-specific, and need to be determined case by case by taking the size and complexity of the model into account. The activation function of the neurons in these layers except the final one is the Leaky Rectified Linear Units (Leaky ReLu), whose output is $h(x) = \max(x, \alpha x)$, where x is the input of the activation function, and $\alpha \in (\mathbb{R} \setminus \{1\})$ is the slope of the activation function in the negative domain. For our experiments, we have arbitrarily chosen $\alpha = 0.2$. The output of the final layer, $\Delta x[k]$, corresponds to the difference of the state values between time-steps. The delta of the state values enable the amplification of the difference in the post-processing for the training.

The simulation of a system model with the proposed architecture starts from the initial conditions of the states and inputs, based on which we conduct one forward propagation with the set of our fully-connected layers to predict the delta values of the states. For the next time step, we first post-process the state delta values, and write them back to the input nodes. Then we shift the input values $u[k]$ into $u[k-1]$, and fetch the new $u[k]$ values from the external source. Finally, we repeat the forward propagation for the new time step. The process repeats until the simulation is stopped.

IV. Training Methodology

A. Data Generation and Processing

First of all, we make an assumption that the model has already been programmed in a modeling language such as Modelica, or built in a software such as Dymola or Simulink. This equivalent implementation of the models is used for generating training and testing data. Also, in the scope of this

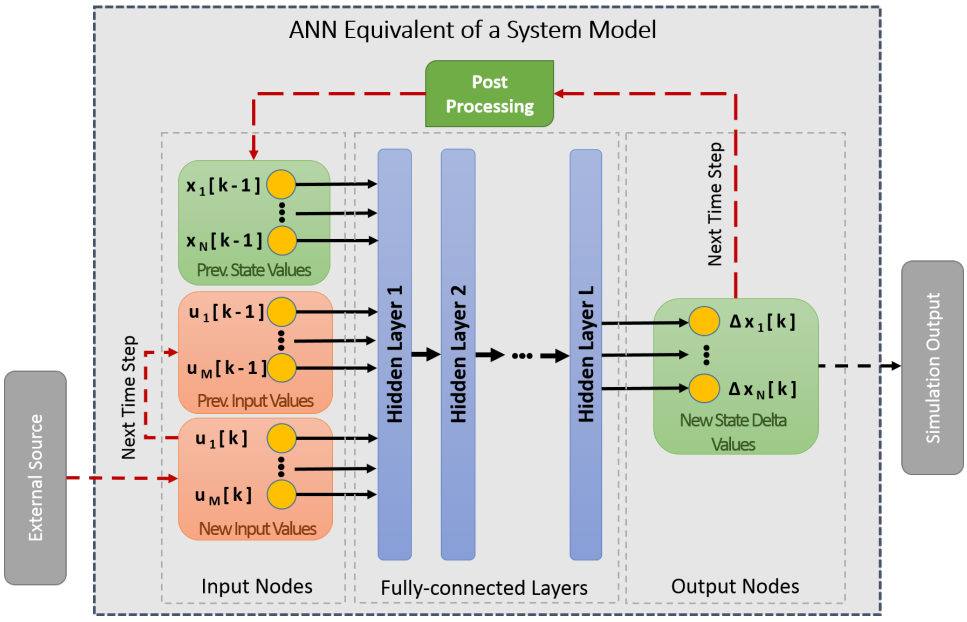


Fig. 1: Proposed RNN Architecture for a system model that consists of N states and M external inputs. $x[k]$ and $u[k]$ represent the state and input values at the time step k. Black solid lines show the data flow within a simulation step, whereas the red dashed lines depict the data transfer between the simulation steps.

paper, we assume that all the states of a model can be accessed, and their values can be recorded during a simulation.

The system models usually take a number of input data from external sources. For instance, a power grid model can take the voltage level of a generator, or a circuit breaker model can take a binary signal to simulate switching on/off a line. These inputs affect the behavior of the system, and change the values of the states. Thus, every different combination of the input sequence results in a different simulation outcome. Consequently, as many input combinations as possible need to be included in the training data.

Another important factor for the training is the simulation step size. Choosing a large step size would reduce the accuracy of the simulation, and the result may become unstable. On the other hand, a large step size will reduce the size of the training data set, and will make both the training and the simulations faster. In contrast, choosing a small step size will increase the accuracy, at the expense of slower training and simulation. Therefore, it is important to choose a problem specific step size for a model while generating the training data.

Once the simulation data is generated, we process the state and input values to mitigate some numerical issues that might happen in the training phase. We first linearly map the states and input values to the range of $[0, 1]$. Then, for the state values in the training dataset, we take the difference in consecutive samples, normalize them with their *Frobenius norm*, and finally amplify them with a scalar factor. The purpose of mapping the values between 0 and 1 is to prevent the 'dying ReLu' problem during the training, in which the gradient of a neuron is always stuck to zero due to the negative output value in the beginning. The reason for taking the difference of the state values is that, when the difference in two consecutive values is near zero, the neural network takes the two values as identical, and does not infer the actual state-update function. This is why we also need to amplify the difference after they are being normalized.

B. Training

We have used the *Adam* optimizer [2] to optimize the weights and biases due to its faster and better convergence. We found that different learning rates lead to the best results for different models. For the training, we have chosen our cost function as L_2-loss between the ground-truth and predicted state values.

The optimization starts with a randomly-selected batch of training samples. The batch size of 200 is chosen arbitrarily. First, we optimize the parameters with the selected samples using the Adam optimizer. Then, instead of taking a new batch of samples from the training dataset, we take the predictions for the next timestep made by the recently-optimized network, and use them to train the network in the next training iteration. This way, we train the networks not only with the ground-truth data obtained from another simulation tool, but also with the noisy data that accumulates the network's self-induced simulation error. Consequently, the network learns giving accurate results in the presence of its self-induced simulation errors. This is inspired by the unrolling concept of LSTM training [3]. We observed that this also helps to prevent overfitting during the training. After a number of training iterations, we take fresh ground-truth data from the training dataset, and repeat this sequence until the training is stopped. Algorithm 1 shows the training schema.

Algorithm 1: Training of the Proposed RNN Architecture

Parameter: Learning rate α, Batch size B, Max. number of iterations I, Number of unrollings U, Number of iterations per full-simulation P
Input : Training input x_{tr} and output $\hat{y}_{tr}$, Validation input x_{val} and output $\hat{y}_{val}$, Scale values s
Output : Optimized Weights w, Biases b

Initialize w and b ;
$g_{nn} \leftarrow$ Build the neural network graph from w, b, s ;
Worst-error: $e_{min} \leftarrow \infty$;
Iteration counter: $i \leftarrow 0$
while *Max. number of iterations has not been reached:* $i < I$ **do**
  if i *mod* U *is equal to 0* **then**
    $x_k, \hat{y}_k \leftarrow$ Randomly selected B samples from $x, \hat{y}$;
    Optimize g_{nn} with $x_k, \hat{y}_k$ w.r.t w and b ;
  else
    $x_{k+1} \leftarrow$ Calculate next input from g_{nn} ;
    $\hat{y}_{k+1} \leftarrow$ Retrieve next output from $\hat{y}$;
    Optimize g_{nn} with $x_{k+1}, \hat{y}_{k+1}$ w.r.t w and b ;
  end
  if $i + 1$ *mod* P *is equal to 0* **then**
    $y \leftarrow$ full-simulation with x_{val} ;
    $e \leftarrow$ compute error from $\hat{y}_{val}$ and y ;
    if $e < e_{min}$ **then**
      Save w, b ;
      $e_{min} \leftarrow e$;
    end
  end
  Increment i ;
end

V. Experimental Evaluation

We have evaluated our proposed method for three cases: First, solving nonlinear differential equations; then simulating two models provided by Simulink, namely Transient model of a linear circuit[1] (TLC), and 100-kW grid-connected PV array[2] (PVA). We have chosen these models to try different sizes and complexities, and cover as much variety as possible such as continuous/binary inputs, existence of discrete events, and nonlinear behavior. Also, the models *TLC* and *PVA* are included in microgrid models, which remain as a future goal.

A. Training Phase Implementation

We have chosen Tensorflow [4] for the training of the neural networks, due to its easy-to-use API, support for both multi-core CPU and GPU, and comprehensive documentation. For the training, we have chosen Adam optimizer settings as $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. We initialize the network's weights from a truncated normal distribution with a mean of 0 and a standard deviation of 0.1. These values are truncated from two times the standard deviation to prevent starting the training with saturated neurons. The biases are initialized to 0.1, which is chosen as an arbitrary positive number to help in preventing the *'dying ReLu'* problem. Experimentally, we found that 20 unrollings yield the best results.

B. GPU Kernel for the Inference Phase

In order to run the simulations with the trained network on a GPU, we developed our custom GPU kernel with the help of cuBLAS[3] and CUDA[4] functions. This kernel is optimized for our proposed RNN architecture and post-processing. It is designed to minimize the communication overhead by keeping neural network parameters on the GPU memory, and transferring only the necessary data between CPU and GPU across simulation time steps. The matrix multiplication required for each layer is implemented with cuBLAS *cublasSgemv* function, whereas the addition of bias as well as calculating the activations are implemented as a custom CUDA kernel in a vector operation fashion. All simulations mentioned in this section are carried out using this kernel.

C. Experiments

1) Use-case 1: Non-linear Differential Equations: We have chosen the Problem 4 from Lagaris et. al [1] as an example of non-linear ordinary differential equation, which is shown in Equation 4.

$$\begin{aligned} \frac{dx_1}{dt} &= \cos(t) + x_1^2 + x_2 - (1 + t^2 + \sin^2(t)) \\ \frac{dx_2}{dt} &= 2t - (1 + t^2)\sin(t) + x_1 x_2 \end{aligned} \tag{4}$$

To generate training data, we solved this equation with the default ODE solver *ode45* in Matlab. We found that a step size of 10^{-3} is sufficient for the given equation. Unlike the original work [1], we solved the equations for $t \in [0, 10]$ ($[0, 3]$ in the original work) to see more variance in the solution. The initial conditions are set to $x_1[0] = 0$ and $x_2[0] = 1$, as in the original work. Then, we trained a neural network with a single hidden layer of 100 neurons. Figure 2 shows the solutions obtained by both Matlab and the trained neural network. The solutions are almost identical: the maximum mean-squared error is as small as 4.49×10^{-7}.

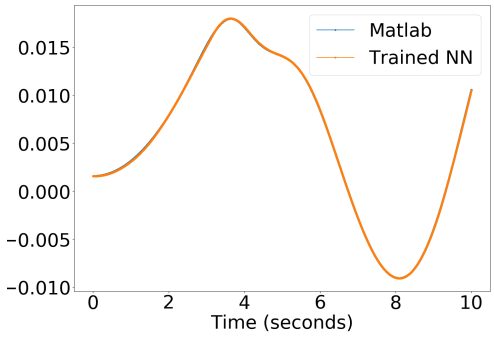
We trained several neural networks with varying sizes, and measured their accuracies and GPU runtimes, which are plotted in Figure 3. Larger networks result in more accurate simulations but longer simulation runtimes. This shows that we can trade-off accuracy and performance based on our simulations needs and computation budget. This is not possible with conventional ODE solvers.

[1] https://ch.mathworks.com/help/physmod/sps/examples/transient-analysis-of-a-linear-circuit.html

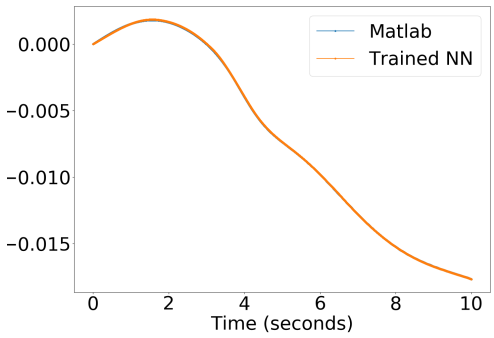
[2] https://ch.mathworks.com/help/physmod/sps/examples/average-model-of-a-100-kw-grid-connected-pv-array.html

[3] http://docs.nvidia.com/cuda/cublas/index.html

[4] https://developer.nvidia.com/gpu-accelerated-libraries



(a) Solution of x_1



(b) Solution of x_2

Fig. 2: Solution of the ODE example. The plots are almost identical, thus, they appear as single.

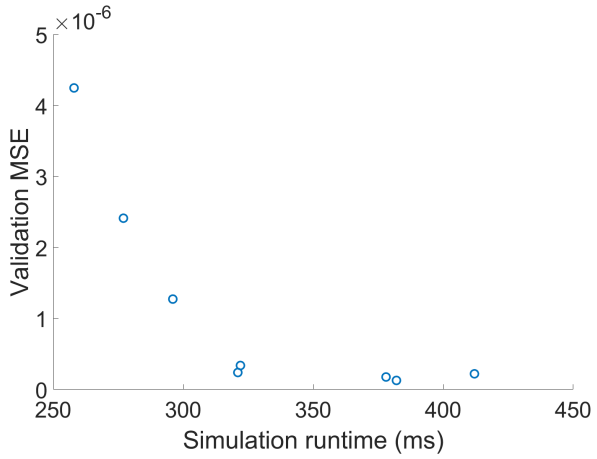


Fig. 3: Simulation error versus GPU run time. Each point in the graph represents a neural network of a different size.

2) Use-case 2: Transient Model of a Linear Circuit: In this experiment, we consider a model of an RLC circuit with a circuit breaker, which introduces non-linearity in the behavior of the model. It consists of five states, and two external inputs: one sinusoidal signal as the voltage source, and one binary signal for the circuit breaker to switch on or off. We trained several neural networks with a single hidden layer of varying number of neurons to approximate the behavior of this model. During the training, we validated the accuracy of our networks with an arbitrary binary waveform for the circuit breaker.

We tested the accuracy of these networks as follows. We switched off the circuit breaker at random times between $[0, 5]$ seconds for 100 times. We simulated the model with the Matlab *ode45* solver and with our trained neural networks, and calculated the mean-squared errors between them. Figure 4 shows these test errors as well as the validation error for different layer sizes. We can observe that both the training and generalization errors are very small, especially for layer sizes larger than 10. In fact, this number is not surprising, and it can also be obtained intuitively. Since the model is a linear circuit when the circuit breaker is on or off, one neuron for each state should be sufficient to estimate its behavior for each circuit breaker mode. Since there are five states, and two different modes (circuit breaker on/off), one can expect that a network with ten neurons is sufficiently large to estimate this model. Unfortunately, it is not always the case that we can find the optimal number of neurons intuitively as in this example, especially when the model is non-linear.

In order to quantify the performance gain of a GPU, we present the execution times of the Simulink model versus the GPU based implementation, see Figure 5. We do not observe any significant difference in runtimes between the neural networks for a single implementation that we trained. However, the GPU implementation leads to a 2.8x speedup.

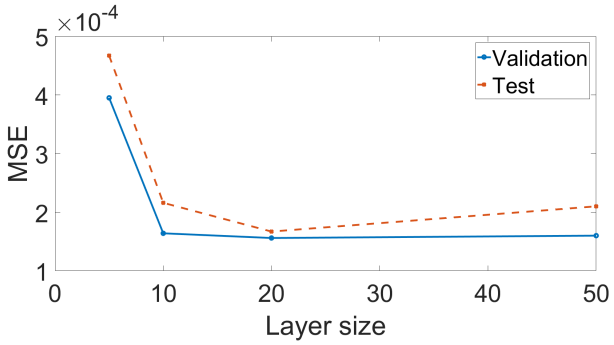


Fig. 4: Simulation error vs. the size of the single hidden layer.

3) Use-case 3: 100-kW Grid-connected PV Array Model: In this experiment, we considered a PV array model, which is connected to a 100-kW grid through DC/AC and AC/AC converters. The complete model consists of 63 states, and it takes 9 inputs, including sun irradiance, temperature signals and control signals for converters.

We trained several networks with different sizes, and measured their simulation accuracy on the training dataset. Figure 6 plots the maximum and average MSEs out of 63 states. We observe that, as we increase the network size, the simulation accuracy improves.

Although the trained networks perform well on the training dataset, they fail at simulating test cases in which the input waveforms are changed. We attribute this to the fact that the state-space may be too large to be covered with the current training data. When the simulation jumps to a value that is not

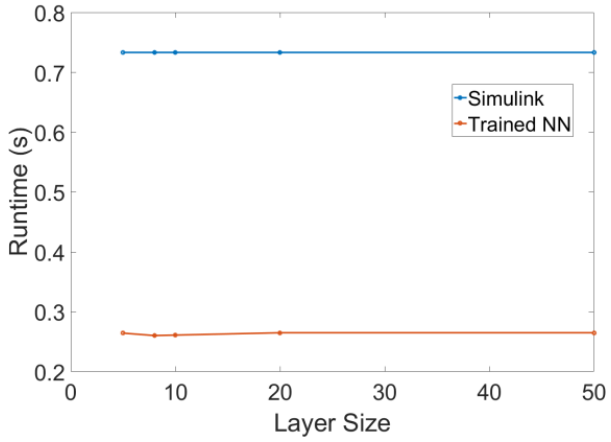


Fig. 5: Simulation runtime of single CPU (Simulink) and GPU implementation (Trained NN).

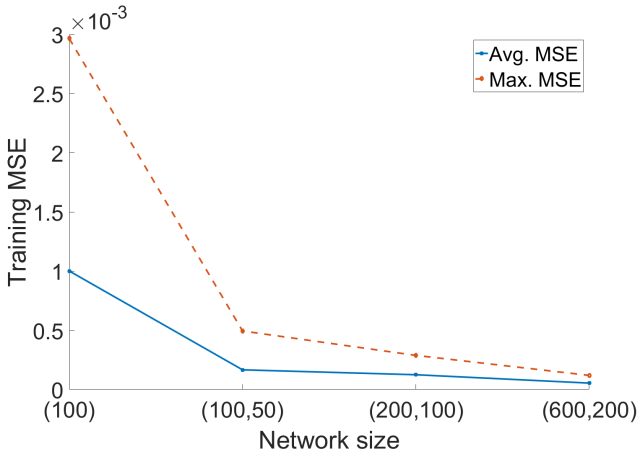


Fig. 6: Simulation error versus layer dimensions for the PV Array model.

included in the training dataset, the simulation diverges. Since the number of states, and the complexity of the model is higher compared to previous examples, it is not possible to cover the state-space with an exhaustive approach. Therefore, the trained networks cannot generalize the behavior of the model, without extending the training dataset.

VI. Conclusion and Future Work

We proposed a novel method to solve ordinary differential equations and simulate system models with recurrent neural networks, while leveraging parallel architectures such as GPUs. We tested our proposed method and showed its effectiveness in several use cases. We first showed that the proposed method is capable of solving ordinary differential equations with a very high accuracy. Then, we showed that our trained neural networks successfully approximate the behavior of two physical models, i.e., the transient model of an RLC circuit, and the average model of a PV array. In the former case, the trained networks showed excellent generalization capabilities. However, in the latter case, which has many more states and inputs, the trained networks were not able to produce accurate simulation results against test cases that were not included in the training dataset. That is, the state coverage is potentially a problem of our current approach as no state can be reached that was not part of the training data set. The same applies to the validation of the trained system as this would need to ensure that all possible system states can be reached. Therefore, we conclude that the scalability of this method for larger and more complex models should be further investigated and improved.

In general, we see several possible future directions to address the aforementioned limitations. First, an iterative training approach can be adopted. Once a neural network is trained, it can be tested with a number of test cases, and the network can be re-trained with the failed test cases. When this is repeated for a sufficient number of times, the accuracy and the stability of the trained networks should improve, although the reachability of all system states may still not be guaranteed.

Secondly, a more efficient approach for the generation of training data can be adopted to overcome the difficulties in covering as much state-space as possible. Our exhaustive search approach is not applicable for large systems such as the PV array example. A state-space explorer as presented in [12] can be integrated into the training data generation such that all reachable state values are included in the training dataset. Besides, the state-space explorer can be used for the verification of the trained networks as well.

Last but not least, the trained neural networks can be deployed onto an FPGA to run the simulations much faster. Since neural networks can be abstracted by only their weights and biases, a trivial register-transfer level (RTL) implementation of a fully-connected layer would be sufficient to translate highly complex simulations into hardware accelerators.

References

[1] I. E. Lagaris, A. Likas, and D. I. Fotiadis, Artificial neural networks for solving ordinary and partial differential equations, IEEE Transactions on Neural Networks, 1998.
[2] D. P. Kingma and J. Ba, Adam: A method for stochastic optimization, Proceedings of the3rd International Conference on Learning Representations (ICLR), 2014.
[3] S. Hochreiter and J. Schmidhuber, Long short-term memory, Neural Computation, 1997.
[4] M. Abadi et. al, TensorFlow: Large-scale machine learning on heterogeneous systems, 2015, Software available from tensorflow.org.
[5] T. Blochwitz et. al, The functional mockup interface for tool independent exchange of simulation models, 8th Intl. Modelica Conference, 2011.
[6] R. Eidenbenz, A. Moga, T. Sivanthi, and C. Franke, MARS: A flexible real-time streaming platform for testing automation systems, Design, Automation Test in Europe Conference Exhibition (DATE), 2017.
[7] MathWorks MATLAB, "https://www.mathworks.com/products/matlab.html", 2018.
[8] Dassault Systems, DYMOLA System Engineering, "https://www.3ds.com/products-services/catia/products/dymola/", 2018.
[9] Modelica Association Project, Functional Mockup Interface, "http://fmi-standard.org/", 2018.
[10] Microsoft FPGA-based configurable cloud, "https://azure.microsoft.com/en-us/resources/videos/build-2017-inside-the-microsoft-fpga-based-configurable-cloud/", 2018.
[11] NVIDIA GPU Cloud, "https://www.nvidia.com/en-us/gpu-cloud/", 2018.
[12] G. Frehse et. al, SpaceEx: Scalable Verification of Hybrid Systems, Proc. 23rd Int. Conf. on Computer Aided Verification (CAV), 2011