SWISS FEDERAL INSTITUE OF TECHNOLOGY

ELECTRICAL AND ELECTRONIC SECTION

Circuit forward Modeling Via GNNs

MASTER THESIS

Author:
Zyad HADDAD

Supervisors:
Prof. Volkan CEVHER
Igor KRAWCZUK

LABORATORY FOR INFORMATION AND INFERENCE SYSTEMS

June 25, 2021

1 Introduction

This works combines neural ODEs and graph networks for circuit forward modeling. The purpose is to model circuits as graphs and learn to predict the evolution of a circuit state (currents, unkown voltages, as well as graph-level attributes such as power consumption).

We first present the possible graph representations of a circuit that are explored in this work. We then present a modified interaction network (IN) framework that is used to learn the dynamics of circuits. In order to replicate the continuous nature of physical interactions occurring in circuits, we combine the IN framework with neural ODEs. We detail the combinations of implicit layers with an IN that we experimented. We list the experiments performed and describe their process. We finally analyse and compare the performances of our model.

1.1 Circuit forward modeling & Motivation

Dynamics of circuits are predicted using traditional simulators (spectre, ngspice, Hspice, etc). However this does not allow optimisation as the predicting function of a traditional simulator is not differentiable. A circuit forward model allows optimisation via reinforcement learning and gradient descent, since it is fully differentiable, and facilitates tasks that are not well automatized such as transistor sizing, component choice, transfer function fitting for filters and amplifiers or power consumption optimisation. Additionally, a neural network could speed up classical simulators.

Circuits are defined by voltages, currents, device internal charges and structure, omitting the influence of external factors as they are out of the scope of this work. A circuit dynamics is mainly defined by the interaction between devices via the nets connecting them. Relational reasoning is required to learn theses interactions. Thus we model circuits as graphs and use the graph structure to reason about the relation between nets. The interaction network framework [1] is designed for learning physical interactions between objects. Relational reasoning is at the heart of this framework as messages, in the sense of a message-passing network, are computed solely on the basis of objects states and the type of physical interaction between them. Messages are then applied to the receiver objects to update their states. In the context of circuit forward modeling, different interpretations and correspondences can be established between circuit elements and the physical objects of the IN framework. We explore two models for graph representation of circuits.

1.2 Graph neural networks

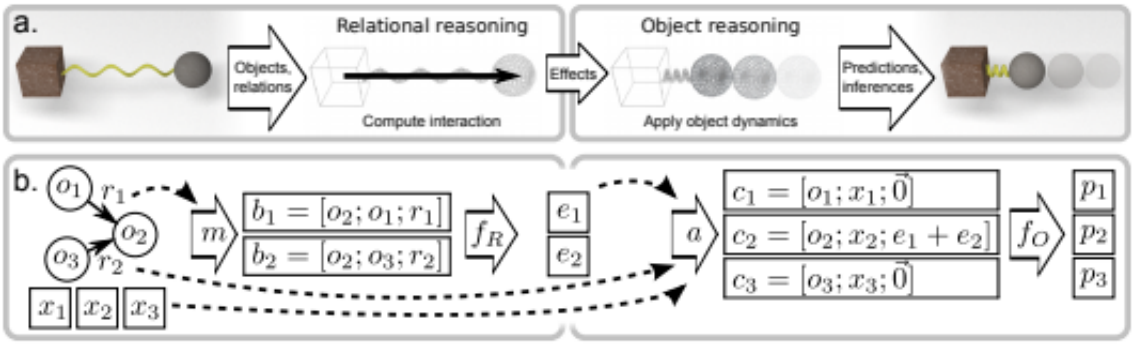


Figure 1: Illustration of IN and relational reasoning coupled to object reasoning for object state update, from Battaglia et al. [1]

Since 2009 graph neural networks (GNN) have seen an increase in interest to represent objects and their relationships in machine learning. A graph structure contains information on relations between objects that allows relational reasoning and inductive biases. Graph convolutional networks and other frameworks can handle different structures of information with varying richness (features). Graph neural networks have been introduced in 2005 [9] and 2009 [17].

In 2013 a clear separation between between spatial and spectral (or laplacian) based methods appeared with [3]. This work focuses mainly on spatial based methods here as our purpose is a physical simulation.

In 2016 graph neural networks frameworks have been developed and experimented to represent physical objects and learn their interactions to predict evolution of their physical states [16] as well as to control them [14]. The concept of relational inductive biases in 2018 [2] details different frameworks and generalize to a full graph network for learning relations between objects and relations between objects and their environment. Multi-dimensional edge features networks were introduced as message-passing networks in 2016 with interaction networks [1]. Other message-passing

networks use multi-dimensional edge features and have been introduced in 2017 [7] and later. IN have many advantages compared to other convolutional GNNs. First, they can readily handle multi-graphs. Second, message passing functions can be diverse, in opposition to matrix operations of other convolutional layers. For circuit forward modeling, we choose the interaction network implementation [1].

More recently there has been a focus on edge features to exploit the rich information about relations that they contain[4] [8] and to help compute messages [22] [19].

Recent reviews of graph network research show the fast development and adaptation to problems of very different natures with richer architectures [10] [23], including physics engine, which is similar to the focus of this work, circuit forward modeling.

1.3 Neural ODE

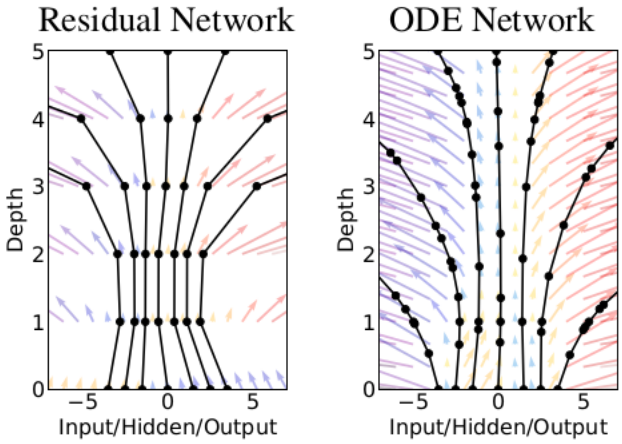


Figure 2: Illustration of adaptive step in ODE solvers, from Chen et al. [5]

Neural ordinary differential equations, or implicit layers, were formulated in 2017 (Weinan) and experimented in 2019[5] [11]. Neural ODE is the continuous version of residual networks in the sense that residual networks are defined by the equation:

$$h_{t+1} = h_t + f(h_t, \theta_t) \text{ or } h_{t+1} - h_t = f(h_t, \theta_t) \tag{1}$$

and ODE networks are defined by the equation:

$$\dot{h_t} = f(h_t, \theta) \tag{2}$$

or rather, Residual networks result from Euler's method with a fixed step size.

In neural ODEs, the step size is adaptive and managed by the ODE solver. Intermediate values within a chosen integration time are calculated by the ODE solver, depending on the derivative computed and the tolerance given to the solver. This supplementary computation usually causes an increased computation cost compared to residual networks. However ODE networks use the same parameters to compute derivatives at every solver call. This implies a reduced memory use compared to residual networks that usually have different parameters for every layer.

GNNs and implicit layers were combined in 2019[15] and 2020[13] [22]. Implicit layers can be used in the message-passing function (ODE in GNN) or the interaction network can be the function describing the ordinary differential equation of the implicit layer (GNN in ODE). A combination of both is possible as well.

In addition this work proposes an extension of interaction networks to hyperedges in order to model transistors, whose edges are interdependent. The proposition differs significantly from the usual hypergraph neural network framework[6].

2 Approach and Formulation

This section establishes the equations that describes behaviors of the device in the case of an RLC circuit and shows how neural ODEs has the potential to improve performance when combined to an IN.

2.1 Learning inductors and capacitors

The current-voltage relation of a capacitor is given by the equation:

$$I = C\frac{dV}{dt} \text{ or } V = \frac{1}{C}\int_0^t i \tag{3}$$

The analogous equation for an inductor is:

$$V = L\frac{dI}{dt} \text{ or } I = \frac{1}{L}\int_0^t v \tag{4}$$

For a transistor the equation is:

$$V = R \times I \tag{5}$$

Or in the integral form:

$$V = \frac{1}{C}\int_{t_0}^t i + V_0, \tag{6}$$

$$I = \frac{1}{L}\int_{t_0}^t v + I_0 \tag{7}$$

and

$$V = R \times I \tag{8}$$

The goal of learning a forward model is now to learn these device functions as well as responses of the circuits built from a composition of these elements from observations. Given the inputs and all internal voltages and current at time t at training, the model should learn to predict the evolution of the circuit given internal voltages and currents at time t_0 as well as inputs at time t. This work compares the performance of an IN using different f_r functions (MLP, implicit layer) and update mechanisms for this task, as well as exploring the use of a GNN to model the dynamics (i.e. placing a GNN inside the ODE solver). Since the device laws for capacitors and inductors are in fact ODEs (equations 6 and 7),[1] our hypothesis is that using a neural ODE or NODE for f_r will imbue the network with the inductive bias towards a state being modified according to some natural dynamics and benefit training, while using a GNN-in-ODE allows the mixing of information between nodes for the calculation of each residual.

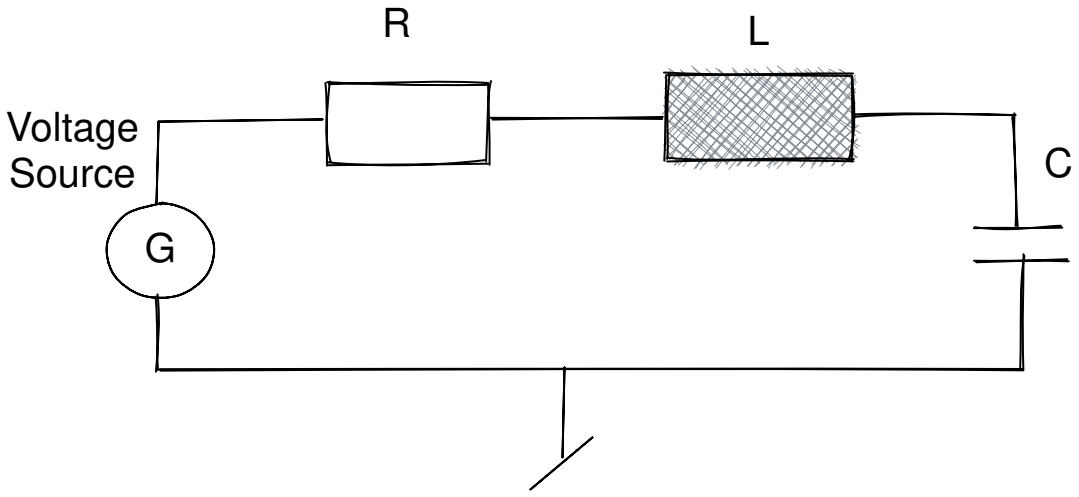


Figure 3: Serial RLC circuit

[1] as well as the resistor, although the dynamics in this case are trivial and an ODE form isn't required

2.2 Interaction Network

This section details the functioning of IN (figure 4) in the context of physical interactions between objects. However, in the case of circuit forward modeling, the IN is trimmed down to an IN without graph-features influence (i.e., using only ephemeral edge features and node states). For this reason, we omit external influences on node states to simplify the presentation. The inputs are the following:

1. $O \in \mathbb{R}^{Ds \times No}$ the objects represented as nodes with Ds the dimension of a node embedding, No the number of nodes in the graph
2. $Ra \in \mathbb{R}^{Dr \times Nr}$ the features of the edges with Dr the dimension of an edge embedding, Nr the number of edges in the graph
3. $Rs \in \mathbb{R}^{No \times Nr}$ and $Rr \in \mathbb{R}^{No,Nr}$ the sender and receiver nodes for every edges, respectively, and one-hot encoded.

First, the matrix $B = [ORr, ORs, Ra]$ is computed, which is a vertical concatenation of the matrix products ORs, ORr and device features Ra. This matrix contains all device features and the voltages at its ports.
Second, the effects $E = f_r(B)$ where f_r is typically an MLP.
Third, $\bar{E} = ERr$ and concatenate vertically with O to obtain the matrix $C = [O, \bar{E}]$.
Finally, C is input to f_o, another MLP, to obtain the new net voltages $f_o(C)$. In parallel $f_i(E)$ computes new currents for all edges.

2.3 Describing device as edge

The first model we explore represents circuit nets as nodes and circuit devices as edges. The graph describing the circuit at time t is composed of $[O_t, Rr, Rs, Ra_t]$ with Ra_t the edge features containing device type, magnitude and current flowing through and O_t the net voltages. Device internal charges are not expected to be explicitly stored, but only deduced from edge features and voltage. Current output from f_i is used to update Ra_t into Ra_{t+1}. A column in Ra_t is typically:

$$\begin{pmatrix} R \\ L \\ C \\ G \\ i \end{pmatrix} \tag{9}$$

$$\tag{10}$$

with one of the first four components being non-zero and the last one being the current. G is for voltage source. Other types of device can be one-hot encoded as well and the edge dimension extended.

Edge dimensions are expanded via an MLP:

$$\mathbf{e_{ij,k}} = \mathbf{f_{expand}}(\theta_{\mathbf{init}}, \begin{pmatrix} R \\ L \\ C \\ G \\ i \end{pmatrix} \tag{11}$$

at initialization.

It is uncertain whether this expansion improves performance. The idea is to help the network compute device state (charge, joule effect heating, varying port parasitic capacitance, etc) and work with higher level features.

2.4 Edge-featured model

This section introduces the implementation of an IN in the context of circuit modeling with devices described as edges. The classical IN framework restricts components to 2-ports devices. The circuit is modeled as an edge-featured graph $G = [O, R]$ with $R = (Rs, Rr, Ra)$ where :

1. $O \in \mathbb{R}^{Ds \times No}$ is the nets of the circuits represented as nodes with Ds the dimension of a node embedding, No the number of nodes in the graph

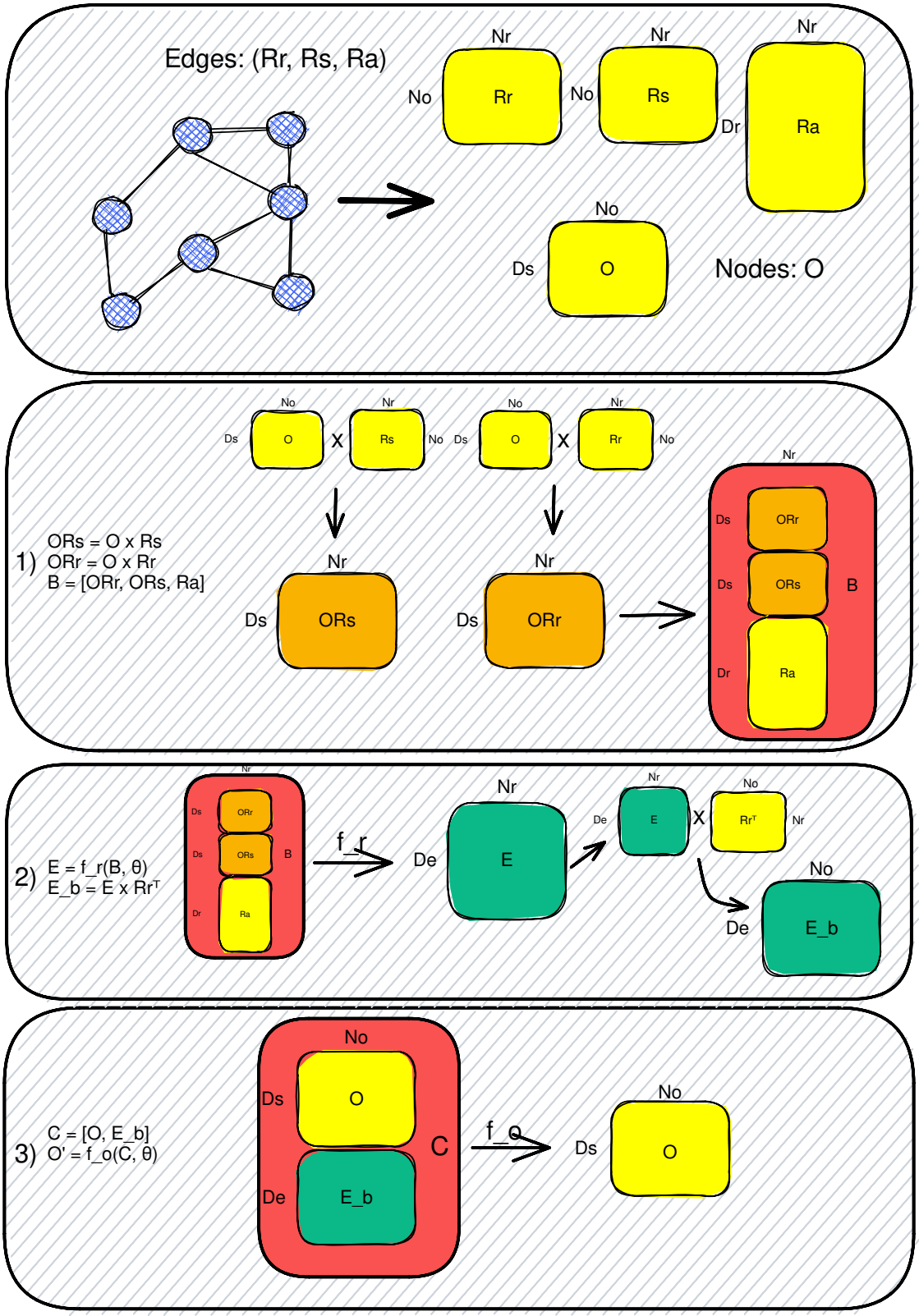


Figure 4: Interaction network visualization

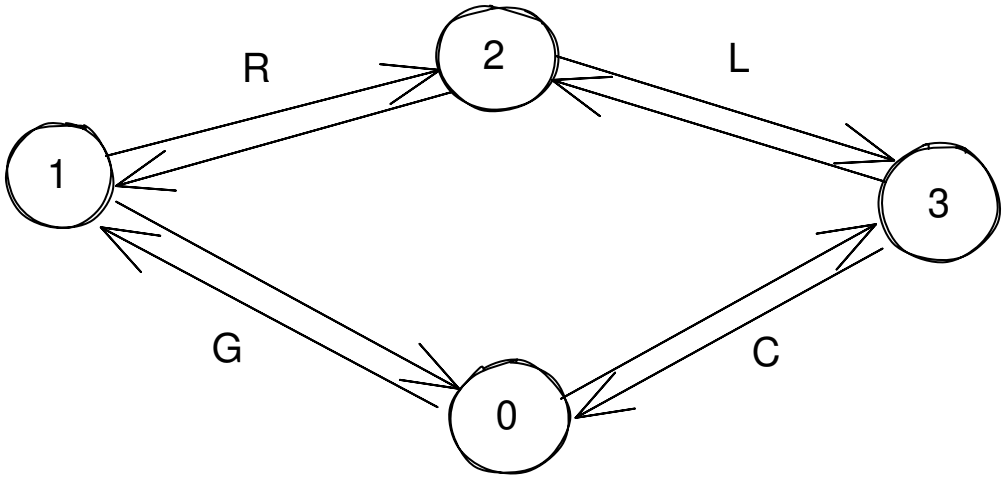


Figure 5: Edge-featured graph representation of serial RLC circuit

2. $Ra \in \mathbb{R}^{Dr \times Nr}$ is the features of devices of the circuits represented as edges and currents flowing through with Dr the dimension of an edge embedding, Nr the number of edges in the graph
3. $Rs \in \mathbb{R}^{No \times Nr}$ and $Rr \in \mathbb{R}^{No,Nr}$ are the sender and receiver nodes for every edges, respectively, and one-hot encoded.

For edge-featured model O contains only net voltages and is one-dimensional.

Additionally, we can extend the classical IN framework to interdependent relations between three objects. In the context of circuit modeling, these are 3-port devices. Effects are computed with $B' = [ORr', ORs', Ra']$ with Rr', Rs' the one-hot encoded receiver and sender nodes. Every edge here has 2 sender nodes and one receiver node, as the effects computed require one node to be applied to, but depend on three nodes (two sender nodes). Ra' contains edge features of three edges (receiver to first sender, receiver to second sender and sender one to sender two) stacked vertically. Effects $E' = f'_r(B')$ are concatenated horizontally to E and Rr' is concatenated horizontally with Rr in order to compute $\bar{E}'$. Some preliminary results are available in the annex. Deeper and further experiments are left for future works.

The interaction network for circuit forward modeling is then composed of the functions:

1. $f_r(B, \Theta_r)$ to compute the effects of edges and voltages
2. $f_o(C, \Theta_o)$ to compute new voltage for every node
3. $f_i(E, \Theta_i)$ to compute new current for every edge
4. $f_p(E, \Theta_p)$ to compute instantaneous power consumption as a global feature
5. $f_f(O, \Theta_f)$ to extract voltages from node states (for tripartite only)

2.5 Tripartite model

This approach differs from the standard interaction network as it uses featureless edges. With this model, devices, device ports and nets are modeled as nodes (tripartite). This implies that devices and nets can be connected only to ports and ports cannot be connected directly to other ports.

The interaction network, compared to described previously, is adapted in the following way:

$$G = (O, Rs, Rr),\ B = [ORr, ORs] \tag{12}$$

The graph describing the circuit at time t is composed of $[O_t, Rr, Rs]$ where port nodes contain currents, net ports contain voltages and device nodes contain device state such as internal charges. Here f_i is only used to explicitly extract currents but is not used to update the circuit as new currents (in ports) should be contained in O_{t+1}. Similarly an MLP function f_o extracts voltages from the hidden states of the nodes representing nets.

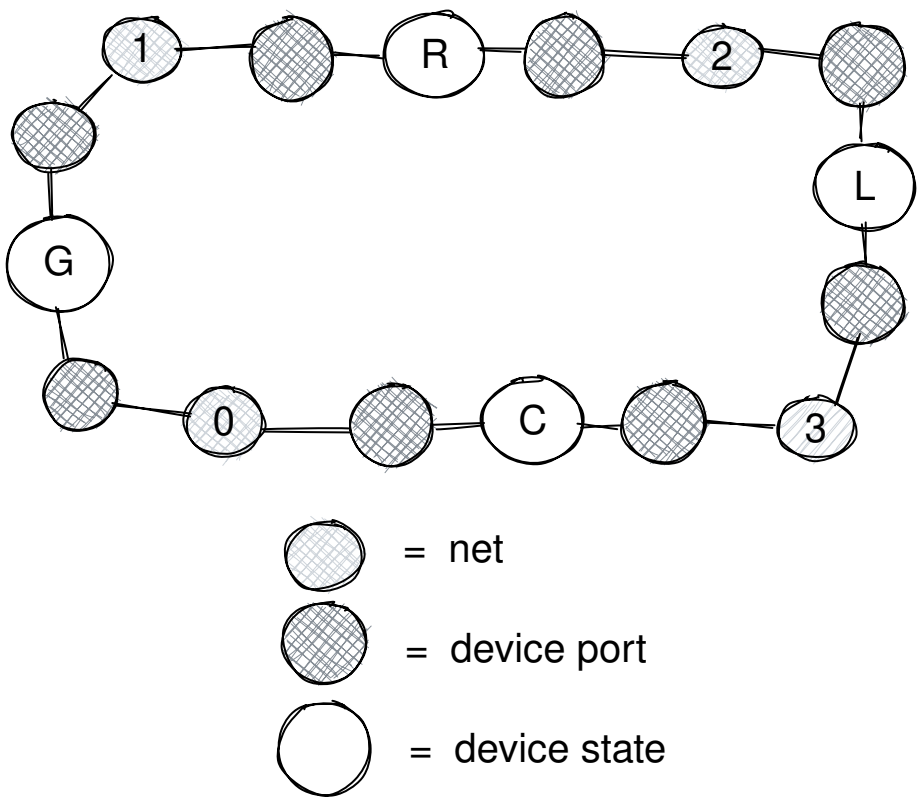


Figure 6: Tripartite graph representation of serial RLC circuit

2.6 Architectures

IN is essentially defined by f_r and f_o as other operations (matrix concatenations or transpositions) are fixed by design (f_i can be modified but is not part of the original IN framework). The most basic implementation consists in a simple IN. Experiments evaluate whether the IN can learn the electronic dynamics in a discrete way with a fixed time step. In this case f_r and f_o are MLPs. A function f_p is also an MLP and evaluates power consumption. A variation with an additional implicit layer to the basic implementation to compute effects E (ODE in GNN) is experimented and compared. This means that f_r is composed of an MLP and an implicit layer (INODE). In both cases the update rule is given by equation 13.

Additionally, a residual network (ResNet) version of these architectures (equation 14) is explored, which should be relevant as explained in the section 2.1 *Learning inductors and capacitors*. The ResNet version of IN is named RESIN. The ResNet version of INODE is named RESINODE.

The last architectures consists in a basic IN architecture inside an implicit layer (GNN in ODE). IN is then computing the voltage derivatives. f_r and f_o are MLPs, the difference between this architecture and the original IN resides in the graph update method from equation 15. This architecture is named GODEIN.

Currents and power consumption can be predicted based on matrix E or matrix B. Once new voltages and currents have been computed the graph can be updated, following equation 16 for edge-featured model and equation 17 for tripartite model.

The classical IN framework uses static edges (constant edge features). However currents in edges vary over time. In the case of GODEIN models edges need to be updated. Nodes also require an update strategy.

Node states can be static for every solver call of the implicit layer. This implies that the derivatives will be constant and the architecture becomes similar to RESIN. Node states are used to compute node derivatives for B and C matrices. This work experiments a "static" model (where gradient computation is always based on the initial nodes states of the first solver call, GODEIN-STATIC). However the implicit layer should be leveraged as much as possible to compute the most

accurate gradients with dynamic edges (updated at every solver call). The "static" version is named GODEIN-STATIC and provides a reference for comparison with more dynamic GODEIN architectures.

Edge states have varying currents that need to be updated for every solver call to fully leverage implicit layer dynamics. A first approach consists in keeping currents constant (values of the original call) while computing gradients based on updated nodes states for every solver call. This network is named GODEIN. This model might actually approach simulation results as single time step predictions might be based on constant current approximation with ngspice. For a potentially more accurate approach a variation with edge currents update for every solver call is also experimented. Similar to node updates, currents are updated by the MLP f_i that computes the current derivatives that are then integrated by the ODE solver.

Different strategies to compute node and edge states over multiple time steps are possible. The first one consists in calling the forward function multiple times while always integrating from 0 to 1 (GODEIN-ITER). The second one consists in integrating from 0 to the number of time steps requested (GODEIN-TIME). The GODEIN model, which gives the best results on a single time step, is used for multiple step prediction. Preliminary results for voltage prediction on multiple steps can be found in annex.

$$O_{t+1}, I_{t+1} = \mathbf{IN}(G_t, \Theta) \tag{13}$$

$$O_{t+1}, I_{t+1} = \mathbf{IN}(G_t, \Theta) + O_t, I_t \tag{14}$$

$$O_{t+1}, I_{t+1} = \int_{t_0}^{t_0+1} \mathbf{IN}(G_t, \Theta) + O_t, I_t \tag{15}$$

$$G_{t+1} = (O_{t+1}, Rr, Rs, Ra_{t+1}) \tag{16}$$

$$G_{t+1} = (O_{t+1}, Rr, Rs) \tag{17}$$

With $O_{t+1} = V_{t+1}$ for edge-featured model (one-dimensional hidden state).

3 Experiments

A serial RLC circuit is our first choice because of the dynamic it offers and the independence of 2-port devices (when represented as edges) that makes the IN implementation trivial. The diversity of components in the circuit is important as well to verify that IN can handle different devices (edge types for edge-featured model, node types for tripartite model) and their behaviors. Given an RLC circuit topology, the circuit state at time t is defined by:

1. R, L and C the resistor, inductor and capacitor values respectively
2. V_r, V_l, V_c and V_{in} the device voltages (or equivalently the voltage of every node with respect to the ground)
3. Device currents I in every edge.

To evaluate the performance of the different models on learning the dynamics of an RLC circuit they are trained to predict the next node and edge states or evaluate a global graph feature in 3 separate experiments:

1. Predict $V_R(t + \delta t), V_L(t + \delta t), V_C(t + \delta t)$ using MSE on the values as a loss
2. Predict $I(t + \delta t)$ using MSE on $I(t + \delta t)$, which is a dimension of the devices edge features. Although the current is equal in all devices of a serial circuit, an IN cannot learn that as it reasons only on interaction between two objects. So MSE loss is applied on every device current.
3. Evaluate instantaneous power consumption $P(t)$ using MSE on $P(t)$ and representing P as a graph level property

ResNets and GODEIN models are expected to outperform other models, since a circuit state at time $t + \delta t$ is strongly correlated to the circuit state at time t. GODEIN and DYN-GODEIN models are expected to outperform GODEIN-STATIC model as well as other models. The ability to update nodes multiple times within a single time step to compute more accurate predictions is expected to surpass other mechanisms. INODE and RESINODE are expected to perform better than their MLP equivalents but it is unclear whether an implicit layer can actually replicate the circuit dynamics from matrix E, without updating nodes within a time step.

3.1 Dataset preparation

Using Pyspice as a python interface with ngspice, a circuit is described element by element and a graph describing the circuit is constructed in parallel. Current probes to all 2-port devices are added to store edge currents at every time step.

Ngspice simulates an RLC circuit powered by a square voltage source. The amplitude of the voltage source varies from 2 to 14V with an interval of 4, resistors from 100 to 2100Ω with an interval of 500, inductors of 1 to 21 mH with an interval of 5 and capacitors of 1 to 21μF with an interval of 5. The simulation time is 40ms for every circuit with two square pulses of 10ms and simulation time steps are 10us. The dataset contains 500 simulations of 4000 states (voltages and currents) each.

Simulation of the circuit returns voltages for every net over time, which are saved in an hdf5 file. All currents are probed for every device at every time step and stored in the dgl graph as edge features. Static edge features (device type) are saved separately in the dgl graph as edge features as well.

Following the same method, circuit dynamics of a single transistor amplifier with RLC filter is collected (figure 7. The base of the transistor is powered by a sinusoidal voltage source centered at 0V. The emitter is connected to the serial RLC filter. The collector is connected to a resistance connected to a constant voltage source of 10V. The R (both resistors), L, and C values vary with the same values as in the first RLC circuit. Sine voltage source has an amplitude between 1 and 5V with a step of 1V. Simulation time and time steps are the same as for the first RLC circuit. This amount to a dataset of 700 simulations of 4000 circuit states each.

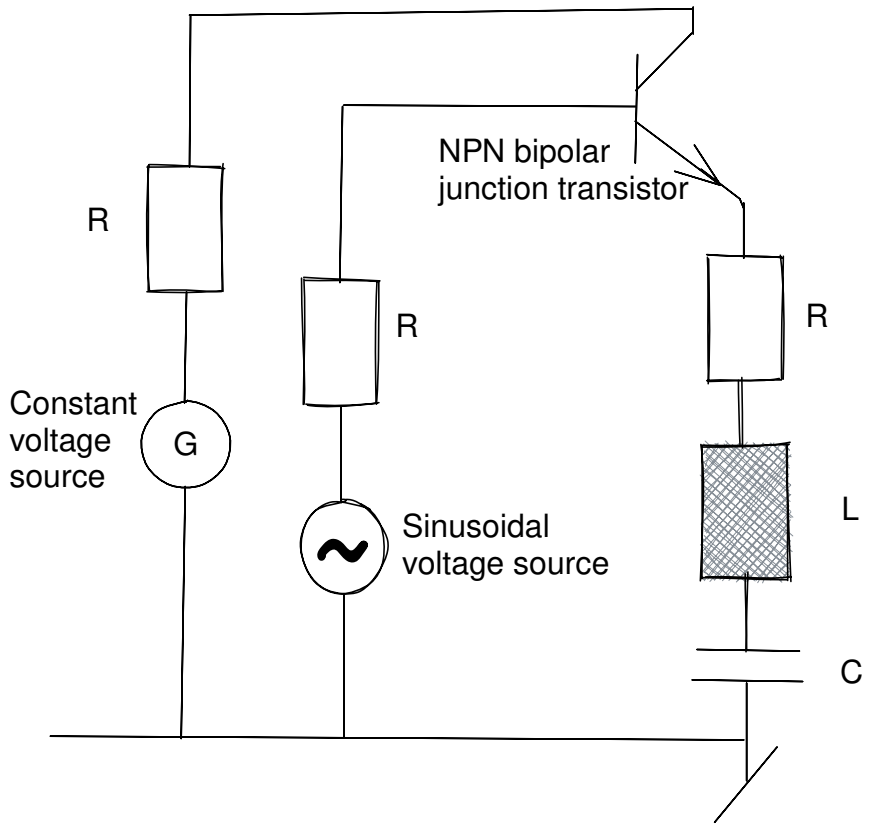


Figure 7: Single transistor amplifier with RLC filter

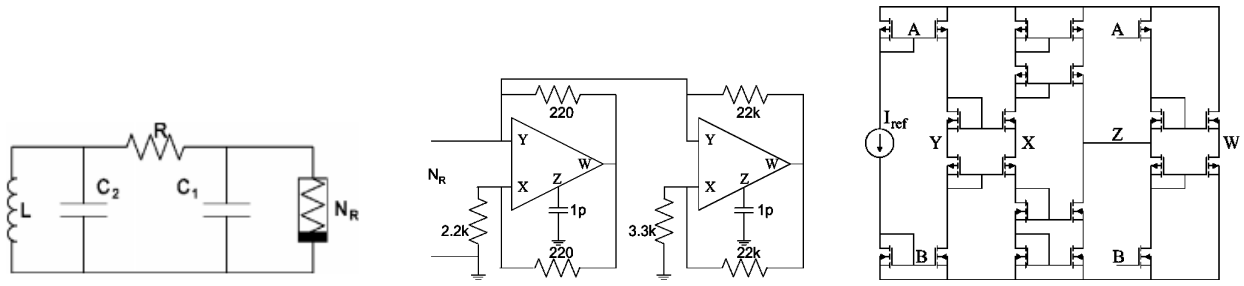


Figure 8: Schematic representation of chua's circuit. Leftmost is the high level schematic consisting of inductor, capacitors, resistors and an element called "Chua's diode", which can be implemented using OpAmps, OTAs or in this case, CFOAs following [20], using two CFOAs as shown in the middle schematic. Rightmost is the CMOS level implementation of a CFOA. Image from Tlelo-Cuautle et al. [20]

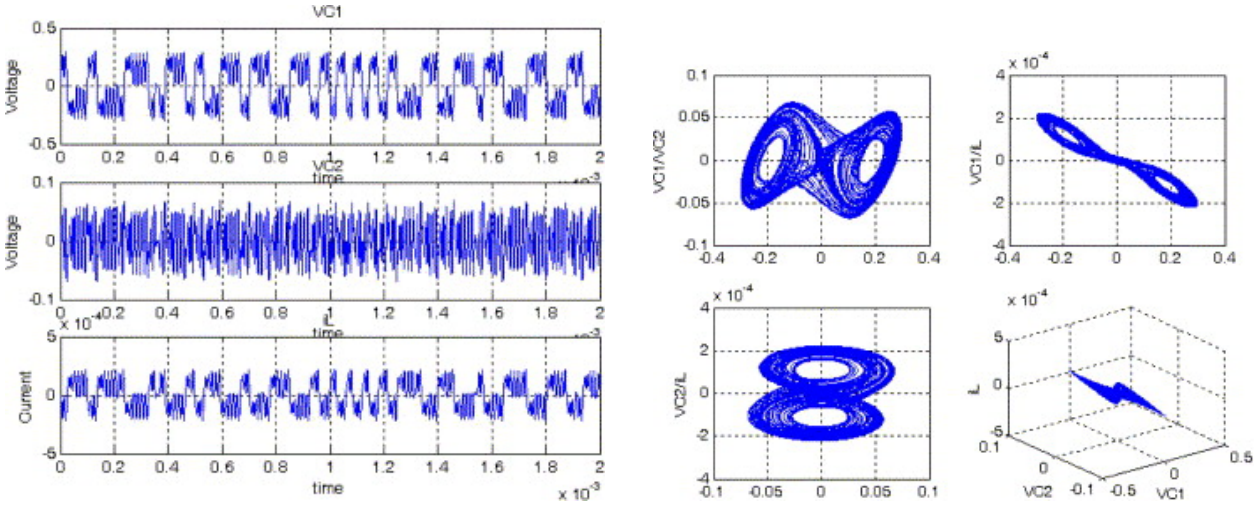


Figure 9: Sample dynamics of chua's circuit, left are the sate variables over time, right is the phase plot of the state variables showing chaotic behaviour. Taken from [21].

Finally, a Chua's circuit [20] is generated following the same methodology (figure 8). This circuit offers a rich and chaotic dynamic that is expected to be more challenging to learn for all networks. Figure 9 shows the hysteresis behavior of the circuit. This hysteresis allows to evaluate the interpretation of a circuit state for every model, which might be an indicator of generalization power.

Circuit	Model	# nodes	# edges	# inputs	# outputs
RLC	edge-featured	4	8	108	12
Transistor amplifier	edge-featured	8	17	433	25
RLC	tripartite	16	16	752	12
Transistor amplifier	tripartite	33	34	3036	26

Table 1: Graph representation summary of RLC and transistor amplifier for edge-featured tripartite model.

3.2 Training

Networks are trained on the collected RLC dataset. For the first experiment, networks are trained on 10000 iterations using ADAM or ADAM with weight regularization optimizers with an MSE loss. Learning rates between 10^{-3} and 10^{-6} have been experimented without improvement on final loss. A slower learning rate only resulted in a slower loss decrease before reaching a stable loss. Decaying rates have been tested between 0.9 and 0.5 and between 0.999 and 0.5 on GODEIN and DYN-GODEIN for b1 and b2, respectively. Loss decrease appeared faster before stabilizing during preliminary tests. Further experiments are required to establish whether learning parameters in this range could

significantly change final average loss. RESIN, GODEIN and DYN-GODEIN are trained with slower learning rates as they quickly reach smaller loss values than other networks (around 10^{-1} within the first 100 iterations). A time step corresponds to a simulation time step of $10\mu s$ as stated in *Dataset preparation*

For the second experiment we use the same training methods. For edge-featured model currents are computed from an "augmented" B matrix containing edge features, node voltages at initial time step t_0 (ORs_{t_0} and ORr_{t_0}) and, additionally, nodes voltages at the next time step $t_0 + 1$ (ORs_{t_0+1} and ORr_{t_0+1}). This matrix is passed to the MLP defining the function f_i.

Experiment 3) uses the same method as experiment 2) with a regular B matrix as described in the IN framework presentation.

For the single transistor amplifier circuit, inputs are composed of all net voltages and port currents at time t_0 and the networks are trained to predict their values at time $t_0 + \delta t$ with δt being a fixed time step. The difference with an RLC circuit is mainly in the interdependence of transistor edges, which require an extension of the IN framework as described above. This work focuses on voltage prediction only and leave current prediction for a future work. The preliminary results are available in annex.

Jax library is used for matrix operations and differentiation. Jax also offers a one-hot encoding function that is used in this work for edge encoding during the dataset collection. Using optax allows to implement optimizers such as adam. Haiku library is used to generate MLPs as pure functions for differentiation. Avoiding the use of graph neural network libraries offers great flexibility and a wide variety of implementations. Avoiding the use of pytorch-geometric or jraph makes implementations of variations easier as every matrix of the IN computation flow can be adapted to the needs of circuit forward modeling. A typical example is testing different B matrices to compute effects E. The IN framework combines matrix concatenation and MLP functions alternatively. Additionally, most models use implicit layers in different combinations or have residual network update mechanisms. Jax offers full control over gradient computation stochastic gradient descent algorithms, which is essential given the diversity of operations and variations required.

Architecture	Model	Optimizer	Learning rate	b1	b2
IN	Edge	ADAM	1e-3	0.9	0.999
RESIN	Edge	ADAM	1e-5	0.9	0.999
INODE	Edge	ADAM	1e-3	0.9	0.999
RESINODE	Edge	ADAM	1e-3	0.9	0.999
GODEIN-STATIC	Edge	ADAM	1e-3	0.9	0.999
GODEIN	Edge	ADAMW	5e-6	0.5	0.5
DYN-GODEIN	Edge	ADAMW	1e-5	0.5	0.5
IN	Tri	ADAM	1e-3	0.9	0.999
RESIN	Tri	ADAM	1e-3	0.9	0.999

Table 2: Training hyperparameters for every model

4 Results

Architecture	Model	Current loss	Constant loss	Voltage loss	Constant loss	Power loss	Constant loss
IN	Edge	8.9e-5	2.24e-8	4.06e-2	1.49e-2	3.34e-2	7.72e-3
RESIN	Edge	6.35e-06	2.93e-8	1.44e-3	1.435e-3	2.3e-2	1.58e-2
INODE	Edge	1.26e-4	1.02e-7	5.32e-2	2.41e-2	7.91e-2	7.69e-3
RESINODE	Edge	1.04e-5	2.2e-7	1.11e-2	1.05e-2	2.68e-2	8.5e-3
GODEIN-STATIC	Edge	-	-	1.98e-2	1.96e-2	-	-
GODEIN	Edge	1.91e-6	1.17e-7	9.59e-4	9.7e-4	-	-
DYN-GODEIN	Edge	7.19e-7	3.36e-8	3.41e-3	3.44e-3	-	-
IN	Tri	8.4e-5	1.03e-8	1.8e-2	1.07e-3	1.79e-2	3.17e-3
RESIN	Tri	1.46e-4	3.4e-8	4.15e-2	5.34e-3	4.87e-2	8.84e-3

Table 3: Average MSE losses for voltages, currents and power for every model on the 1000 last training iterations and comparison with constant function performances

A comparison of model losses evolution (figure 10 and 11 shows significantly improved performance for ResNets and GODEIN architectures.

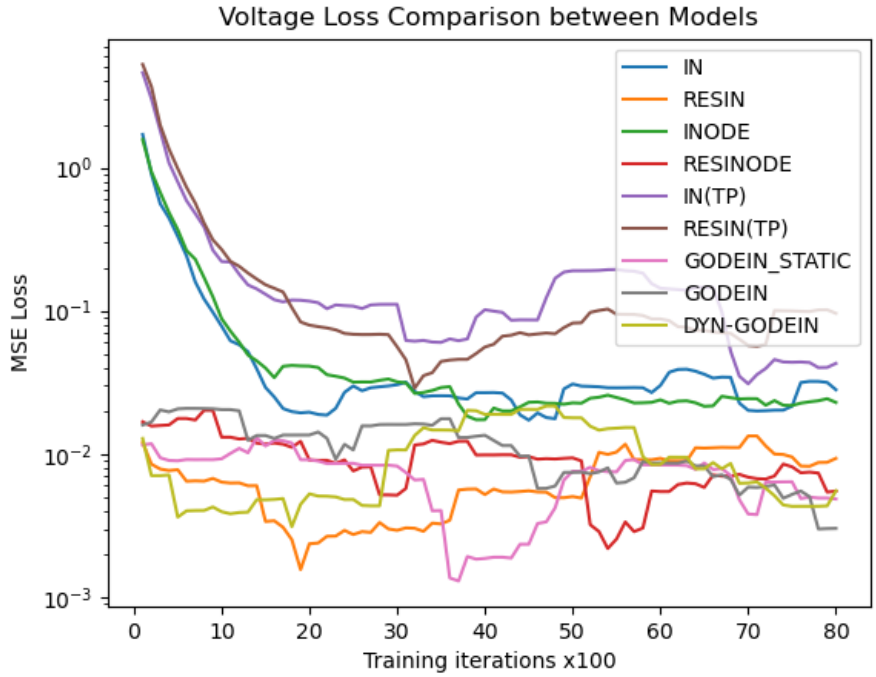


Figure 10: Comparison of average training voltage losses on 2000 iterations

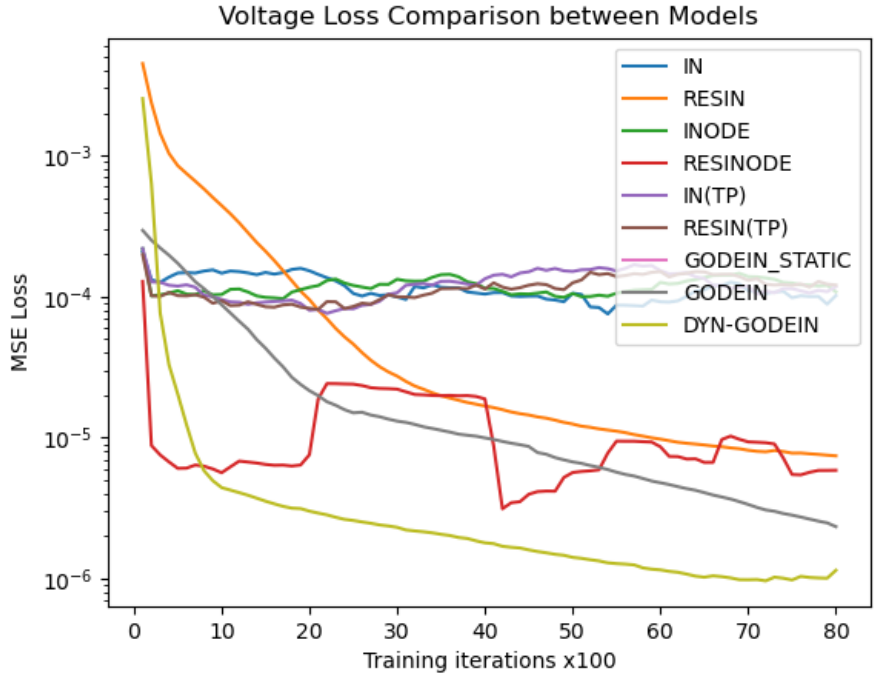


Figure 11: Comparison of average training current losses on 2000 iterations

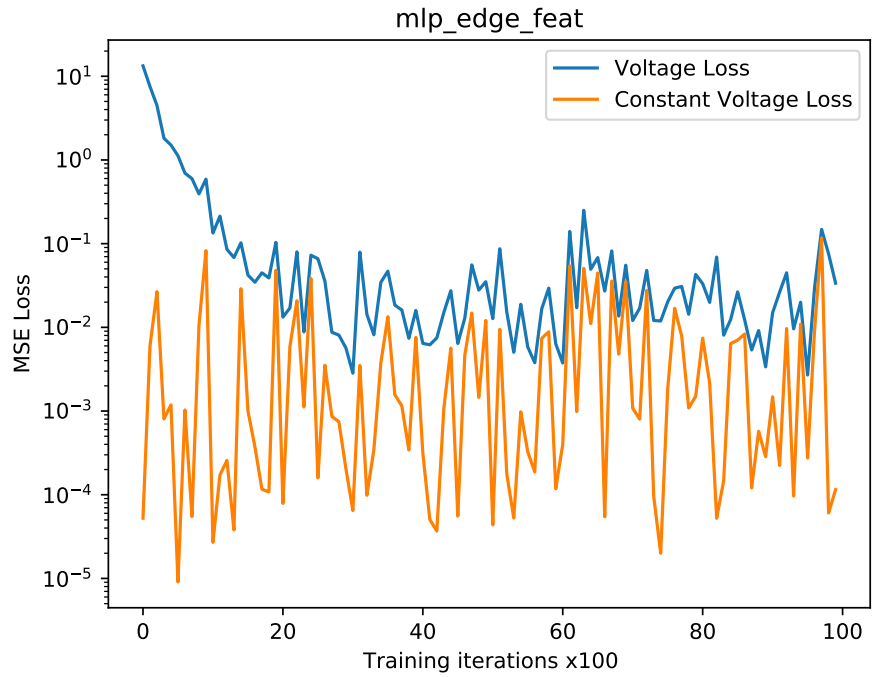
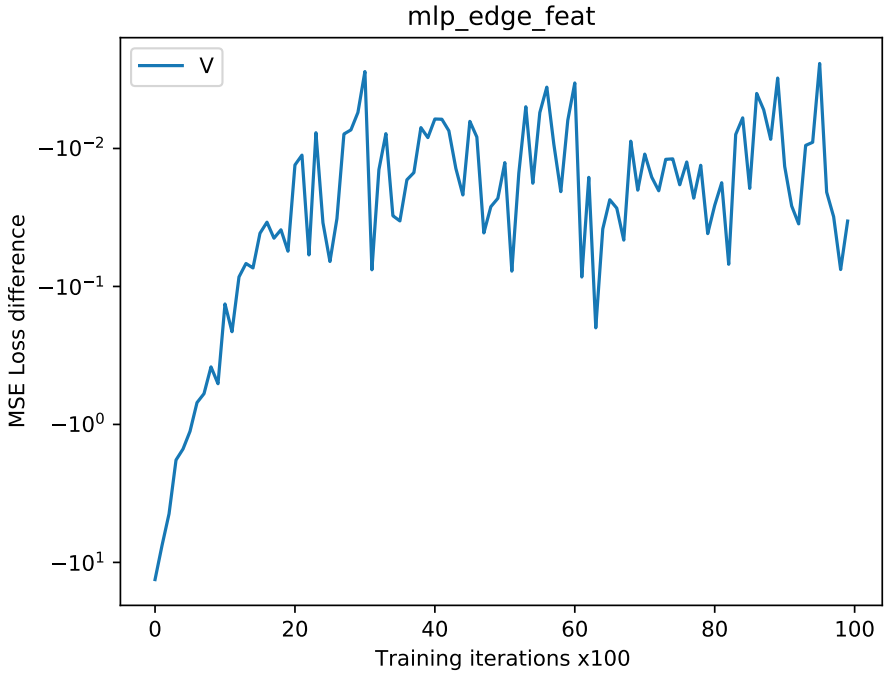


Figure 12: Voltage loss for constant function and for IN model with edge features



/

Figure 13: Voltage loss difference between constant function and IN model with edge features

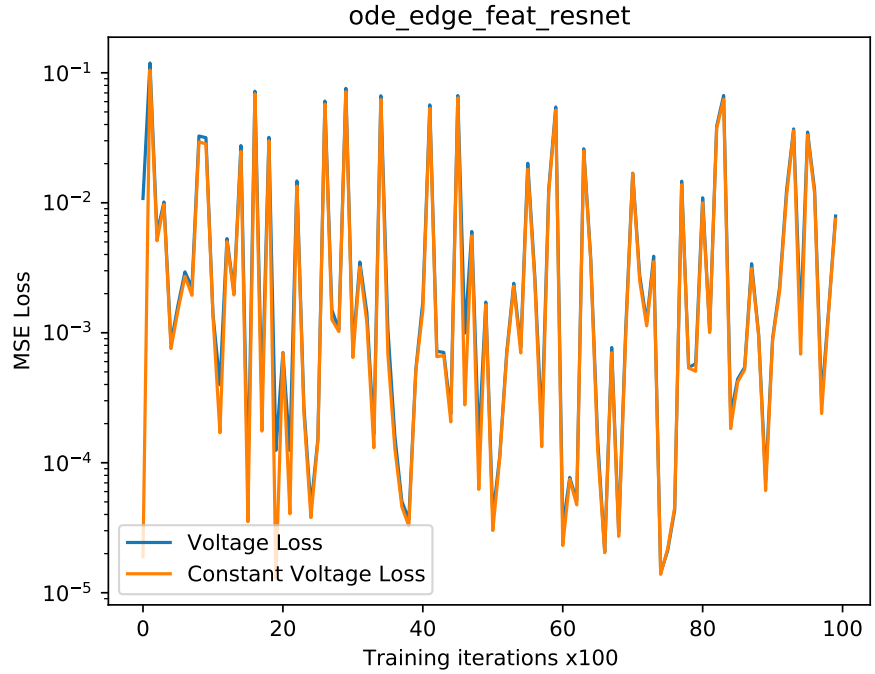


Figure 14: Voltage loss for RESINODE model with edge features

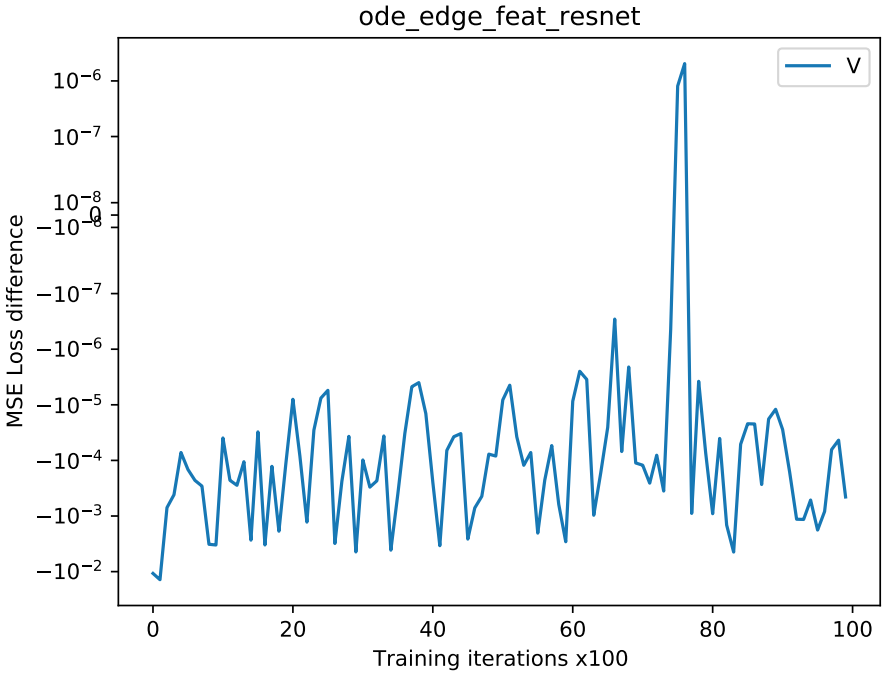


Figure 15: Voltage loss difference between constant function and RESINODE model with edge features

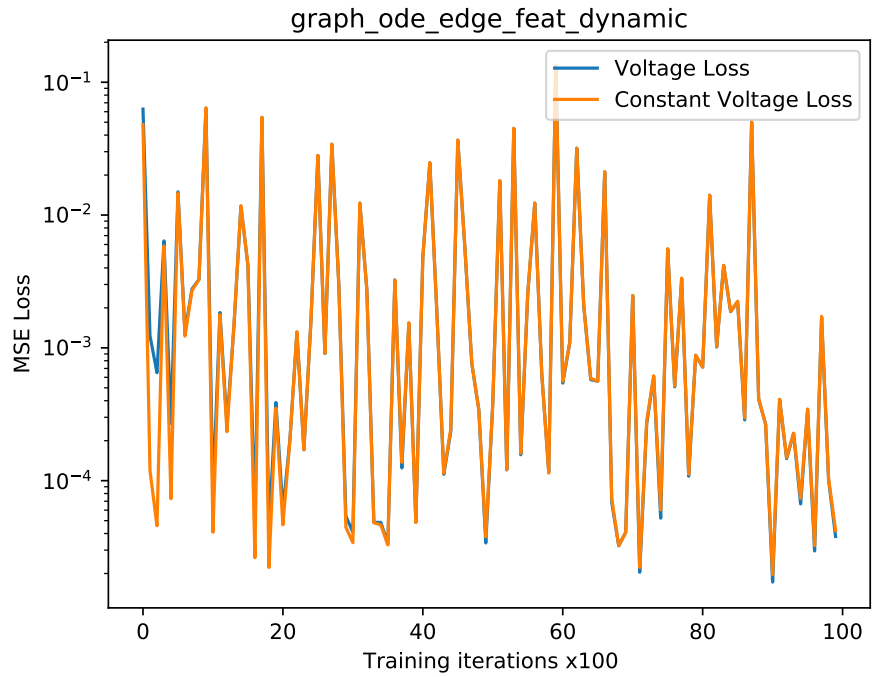


Figure 16: Voltage loss for constant function and GODEIN model with edge features

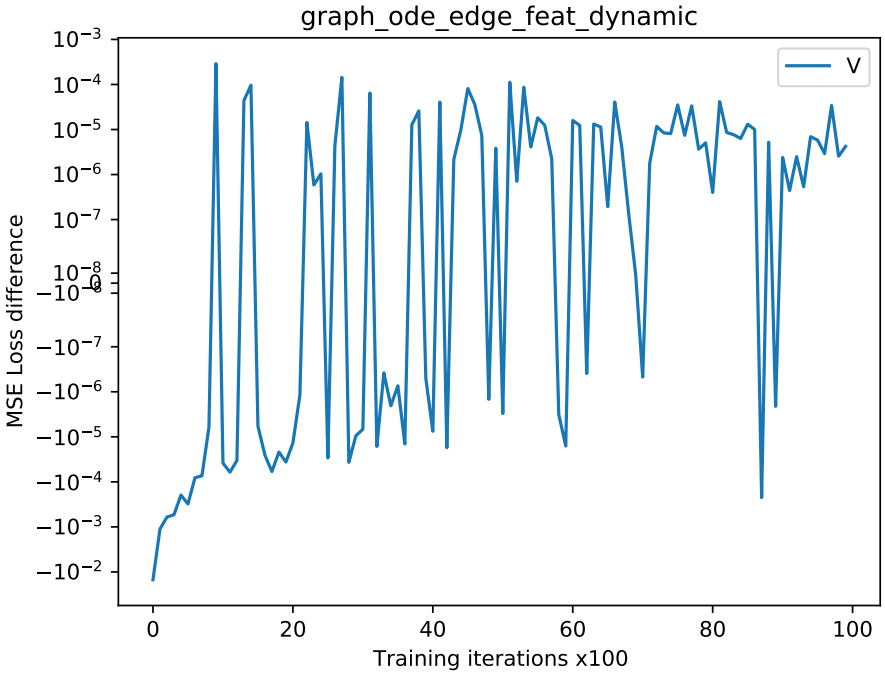


Figure 17: Voltage loss difference between constant function and GODEIN model with edge features

Architecture	Model	I Constant/Model loss ratio	V Constant/Model loss ratio	P Constant/Model loss ratio
IN	Edge	2.52e-4	3.67e-1	2.31e-1
RESIN	Edge	4.61e-3	9.97e-1	6.87e-1
INODE	Edge	8.1e-4	4.53e-1	9.72e-2
RESINODE	Edge	2.12e-2	9.46e-1	3.17e-1
GODEIN-STATIC	Edge	-	9.9e-1	-
GODEIN	Edge	6.13e-2	1.01	-
DYN-GODEIN	Edge	4.67e-2	1.01	-
IN	Tri	1.226e-4	5.94e-2	1.77e-1
RESIN	Tri	2.33e-4	1.29e-1	1.82e-1

Table 4: Constant/Model loss ratio for voltages, currents and power for every model on the 1000 last training iterations

Voltage and current plots compare MSE losses of the models to the MSE loss when returning the input voltages or currents to verify that the networks outperform a "constant" function (loss when returning input voltages or loss when returning input currents). Only a sample of model performances on voltage prediction is shown in this section to facilitate reading. The selected graphs shows the main performance tendencies. Omitted graphs can be found in annex. Power plots compare power consumption evaluation to a zero function to check that it outperforms a constant function as well.

For GODEIN-ITER and GODEIN-TIME voltages are predicted over multiple steps with both methods described above. Models are trained to predict voltages after a random number of time steps. The limit of the number of time steps is increased progressively during the first 2000 iterations and then maintained at a maximum of 50 steps. They are omitted here as only preliminary voltage results are available. They can be found in annex. This experiments, as well as other similar

Losses difference between model and constant function are also represented for better visualization and comparison. An increase in difference (increasing series) means that the model is improving. A negative difference means that the model is surpassed by the constant function. A positive difference means that the model predicts more accurately than a constant function

Power loss plots are omitted here because results do now show significant differences between models. RESIN tripartite plots are also omitted because they show little difference compared to IN tripartite. Omitted plots can be found in annex.

Training on other circuits such as single transistor amplifier or chua's circuit and on combinations of circuits are in process and will be available for the presentation. The dataset preparation scripts have been recently updated to handle a wide diversity of circuits and training scripts are partially ready to run on diverse circuits as well.

Another experiment, consisting in an n-stop loss across a roll-out and predicting multiple steps ahead after training in order to account for accumulating error, has been planned and is still in process. This experiment is similar to what is done in [12].

An ablation over batch size experiment with a comparison between models has been planned as well but we did not end up having time for it. It will be done for the presentation.

5 Analysis

Voltage losses show that the residual nature of a network improves performance significantly, whether using update mechanism of equation 14 (RESIN, RESINODE) or 15 (GODEIN models) of residual network models and IN in ODE models respectively. In contrast, MLP models cannot outperform or approach the performance of a constant function. The main hypothesis to explain the large result difference is that the B matrix does not contain all information required to effectively calculate next voltages. This could by improved by adding information such as previous voltages, possibly allowing the network to deduce voltage and current discrete derivatives. This could result in computing more meaningful effects E. Furthermore, as detailed above from equation 6 and 7, the residual (or continuous) nature of voltage dynamics gives residual models a significant advantage by design.

Similar results are observed for ode models, as the power of an implicit layer may not be exploited if the B matrix is insufficient. Additionally, ODE-in-GNN architectures do not see their depth increased by the their implicit layers as columns of the E matrix are updated independently with each solver call. No information from other interactions or effects can be used to correct trajectories with every call. Increasing IN depth with an ODE layer requires updating voltage nodes on every solver call (as IN reasons only between two objects), which is actually the GNN-in-ODE

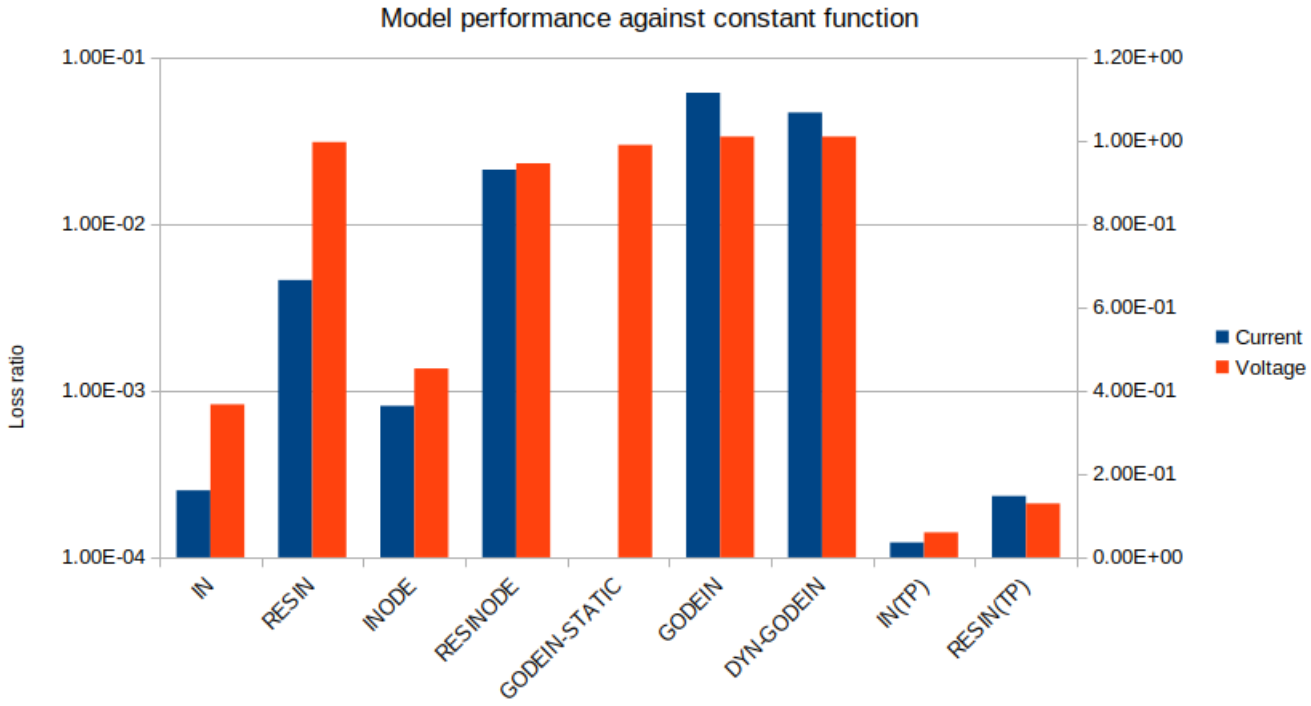


Figure 18: Model performances against constant function

variation. Furthermore, an implicit layer for effect computation might not be able to show improved results on a single step compared to MLP as the simulator used (ngspice) might use constant current and voltage to compute next step voltages.

IN in ODE models showed mixed results. As expected, GODEIN-STATIC gives the same performance as RESIN as it is essentially integrating a constant derivative over a fixed time and adding it to input voltages. GODEIN is the only model outperforming a constant function. This is probably due to the ability to adjust inaccurate messages (effects E) to find a stable output. This does not necessarily invalidate the hypothesis of a faulty B matrix as B could contain enough information for some device (but not all) that could compensate over solver calls for insufficient B on other device.

The performance difference between ODE models and GODEIN models highlights the importance of implicit layer proper integration in the architecture.

Another challenge consists computing messages (effects) in the transient regime. The equations 6 and 7 are not strictly respected in transient regime and it is not always possible to tell if device current or device voltage (or both) is going to change when considering a single device (edge) in the way of an IN. In other words, relational reasoning cannot take into consideration limitations or forced increase in current imposed by other devices in series. For this reason MLP function f_e might compute inaccurate messages. For this reason putting the IN in an implicit layer might be required to compensate through multiple solver calls the eventual faulty messages. Indeed implicit layers increase depth of graph convolution as nodes are actually updated on every solver call, allowing to take into account messages computed by some other device.

Performance in predicting next currents is overall worse than performance in voltage computation as no model was able to perform better than a constant function. However IN in ODE models require more training to correctly evaluate performance. Equivalent to the first voltage prediction hypothesis, the hypothesis of an insufficient B matrix is plausible once again for other models. The B matrix might not contain all required values and an augmented B matrix containing voltages at time t_0 and t_{-1} as well as currents might unlock better performance. In opposition, GODEIN models might manage to compute accurate predictions with further training and with limited information in the B matrix. This might be possible considering the increased network depth offered by the implicit layer with GNN-IN-ODE architectures. Overall, the analysis of voltage prediction performances holds for currents as well. Current and voltage prediction

performances are correlated. However, current training is more delicate due to numerical instability as the loss to achieve to surpass contant function performance is around 10^{-8}.

Power consumption is also not well evaluated. However the constant function here is a zero function and model loss is highly correlated with the zero function loss. This result is observed for all models and requires further investigation to be well understood.

6 Conclusion

In this work we developed nine different interaction networks with two different graph representations of a circuit. We constructed 3 datasets using Pyspice. We set up an environment to generate a wide diversity of circuits and compose a rich dataset containing different circuits. Most architectures can handle 3-port devices and all presented architectures will be completed and experimented for the presentation. We evaluated the architectures on 3 different experiments, testing the ability to predict node level and graph level properties varying over time.

Some experiments described in section 3 are left for the presentation due to lack of time. Even after adding these, this thesis is an initial exploration of this combination of architectures on only a few handcrafted datasets ans should be taken as proof of concepts, not a solid evaluation. Both the architecture and the optimisation procedure can surely be further improved.

A lot of time was spent on attempting to elaborate a graph network framework and explore possible implementation with Pytorch-geometric and Dgl. Otherwise all results of planned experiments would have been presented in this thesis.

Before this thesis, I had no knowledge in graph neural networks or implicit layers. My machine learning background was very limited and my experience non-existent. In fact I wrote my first lines of code in python in october 2020. I had only followed Professor Cevher's course *Maths Data: from theory to computation*. This is what motivated me to address the challenge of a thesis in machine learning. During the literature review I have learnt about graph neural networks and neural ODEs, from the first publications introducing the concepts to the most recent advances from the beginning of this year. During the second phase I experimented different machine learning libraries, especially those oriented towards graph networks. Finally I put into practice this newly acquired knowledge and propose a solution to circuit forward modeling with different models. More details about the time spent for this work can be found in annex.

I now intend to expand my knowledge by learning about different machine learning areas than the ones I have explored and solidify my fundamental knowledge in the domain. I would be glad to continue working on circuit forward modeling, as many ideas remain to be explored. Comparing performances with different B matrices and new circuits or comparing model trajectories on multiples steps are the next experiments to perform.

References

[1] Peter W. Battaglia et al. "Interaction Networks for Learning about Objects, Relations and Physics". In: *arXiv:1612.00222 [cs]* (Dec. 2016). arXiv: 1612.00222. URL: http://arxiv.org/abs/1612.00222 (visited on 03/09/2021).

[2] Peter W. Battaglia et al. "Relational inductive biases, deep learning, and graph networks". In: *arXiv:1806.01261 [cs, stat]* (Oct. 2018). arXiv: 1806.01261. URL: http://arxiv.org/abs/1806.01261 (visited on 03/15/2021).

[3] Joan Bruna et al. "Spectral Networks and Locally Connected Networks on Graphs". In: *arXiv:1312.6203 [cs]* (May 2014). arXiv: 1312.6203. URL: http://arxiv.org/abs/1312.6203 (visited on 06/22/2021).

[4] Jun Chen and Haopeng Chen. "Edge-Featured Graph Attention Network". In: *arXiv:2101.07671 [cs]* (Jan. 2021). arXiv: 2101.07671. URL: http://arxiv.org/abs/2101.07671 (visited on 04/10/2021).

[5] Ricky T. Q. Chen et al. "Neural Ordinary Differential Equations". In: *arXiv:1806.07366 [cs, stat]* (Dec. 2019). arXiv: 1806.07366. URL: http://arxiv.org/abs/1806.07366 (visited on 03/03/2021).

[6] Yifan Feng et al. "Hypergraph Neural Networks". In: *arXiv:1809.09401 [cs, stat]* (Feb. 2019). arXiv: 1809.09401. URL: http://arxiv.org/abs/1809.09401 (visited on 06/01/2021).

[7] Justin Gilmer et al. "Neural Message Passing for Quantum Chemistry". In: *arXiv:1704.01212 [cs]* (June 2017). arXiv: 1704.01212. URL: http://arxiv.org/abs/1704.01212 (visited on 06/22/2021).

[8] Liyu Gong and Qiang Cheng. "Exploiting Edge Features for Graph Neural Networks". en. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA: IEEE, June 2019, pp. 9203–9211. ISBN: 978-1-72813-293-8. DOI: 10.1109/CVPR.2019.00943. URL: https://ieeexplore.ieee.org/document/8954414/ (visited on 03/31/2021).

[9] M. Gori, G. Monfardini, and F. Scarselli. "A new model for learning in graph domains". In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* Vol. 2. ISSN: 2161-4407. July 2005, 729–734 vol. 2. DOI: 10.1109/IJCNN.2005.1555942.

[10] Guohao Li et al. "DeeperGCN: All You Need to Train Deeper GCNs". In: *arXiv:2006.07739 [cs, stat]* (June 2020). arXiv: 2006.07739. URL: http://arxiv.org/abs/2006.07739 (visited on 03/31/2021).

[11] Yiping Lu et al. "Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations". In: *arXiv:1710.10121 [cs, stat]* (Mar. 2020). arXiv: 1710.10121. URL: http://arxiv.org/abs/1710.10121 (visited on 03/08/2021).

[12] Yuping Luo et al. "Algorithmic Framework for Model-based Deep Reinforcement Learning with Theoretical Guarantees". In: (2021). arXiv: 1807.03858 [cs.LG].

[13] Michael Poli et al. "Graph Neural Ordinary Differential Equations". In: *arXiv:1911.07532 [cs, stat]* (June 2020). arXiv: 1911.07532. URL: http://arxiv.org/abs/1911.07532 (visited on 03/03/2021).

[14] Alvaro Sanchez-Gonzalez et al. "Graph Networks as Learnable Physics Engines for Inference and Control". en. In: (), p. 10.

[15] Alvaro Sanchez-Gonzalez et al. "Hamiltonian Graph Networks with ODE Integrators". In: *arXiv:1909.12790 [physics]* (Sept. 2019). arXiv: 1909.12790. URL: http://arxiv.org/abs/1909.12790 (visited on 03/03/2021).

[16] Alvaro Sanchez-Gonzalez et al. "Learning to Simulate Complex Physics with Graph Networks". en. In: (), p. 10.

[17] F. Scarselli et al. "The Graph Neural Network Model". en. In: *IEEE Transactions on Neural Networks* 20.1 (Jan. 2009), pp. 61–80. ISSN: 1045-9227, 1941-0093. DOI: 10.1109/TNN.2008.2005605. URL: http://ieeexplore.ieee.org/document/4700287/ (visited on 06/22/2021).

[18] Michael Schlichtkrull et al. "Modeling Relational Data with Graph Convolutional Networks". In: (2017). arXiv: 1703.06103 [stat.ML].

[19] Martin Simonovsky and Nikos Komodakis. "Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs". In: *arXiv:1704.02901 [cs]* (Aug. 2017). arXiv: 1704.02901. URL: http://arxiv.org/abs/1704.02901 (visited on 03/31/2021).

[20] E Tlelo-Cuautle, A Gaona-Hernández, and J Garcıa-Delgado. "Implementation of a chaotic oscillator by designing Chua's diode with CMOS CFOAs". In: *Analog integrated circuits and signal processing* 48.2 (2006), pp. 159–162.

[21] E. Tlelo-Cuautle and M.A. Duarte-Villaseñor. "Designing Chua's circuit from the behavioral to the transistor level of abstraction". In: *Applied Mathematics and Computation* 184.2 (2007), pp. 715–720. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2006.05.171. URL: https://www.sciencedirect.com/science/article/pii/S0096300306007478.

[22] Yue Wang et al. "Dynamic Graph CNN for Learning on Point Clouds". In: *arXiv:1801.07829 [cs]* (June 2019). arXiv: 1801.07829. URL: http://arxiv.org/abs/1801.07829 (visited on 03/31/2021).

[23] Zonghan Wu et al. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (Jan. 2021). arXiv: 1901.00596, pp. 4–24. ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2020.2978386. URL: http://arxiv.org/abs/1901.00596 (visited on 03/08/2021).

A Time spent & Lessons learned

I started this thesis by a literature review that compiled around 70 publications and articles focusing on graph networks and neural ODE. In parallel I learned about the basics of graph neural networks, reinforcement learning and implicit layers. Diversifying learning supports helped me to understand publications and the development on graphs in machine learning during the past ten years and implicit layers more recently. At this point I have read the publication [1] but did not realise that the IN framework was what I will need.

After about three weeks I began to explore different machine learning python libraries (pytorch, pytorch-geometric, torchdiffeq, torchdyn, jax, optax, haiku, dgl, sacred, jraph) and tested some of them to evaluate their flexibility, focusing on pytorch-geometric. At this point the modeling approach of circuits was not clear so I could not decide which library to choose based on a specific and already implemented convolutional layer.

I tested some graph networks from the examples of torch libraries that could resemble what I was actually going to use for circuits. I focused on convolutional layers that use edge features such as relational GCN [18] and checked the publications associated to these layers. This is how I found the publication [22]. In parallel I programmed my

first experiment, which was a combination of an example code for RGCN from the pytorch-geometric library with an implicit layer from the torchdyn library.

At the same time we discussed the possible graph representations of a circuit. We came up with the two models presented in the thesis: edge-featured and tripartite. I also wrote the first script generating a dataset of RLC simulations. It would build a dgl graph representing the circuit following the edge-featured model and store simulation results. Another three weeks have passed.

Then I spent about two weeks to improve the dataset script and thinking about handling multigraphs for edge-featured representation. This is required for elements in parallel in a circuit and oriented my search towards an "edge-focused" convolution. I tested dgl for message passing and update function but it was not satisfying. We decided to move to jax library. I performed a couple of simple tests with jax and jraph to become familiar with the new approach to graph networks and backpropagation methods. Indeed, graph networks in pytorch-geometric and in jax are totally different. However, programming at a lower level of abstraction, handling all matrix operations, simplifies IN implementation and experimenting variations.

Next, I read again physics-focused graph networks publications and found again [1]. I thought about using [4] but this does not handle multi-graphs as easily as IN. I learnt about haiku and optax and wrote the script of the first model for voltage prediction, edge-featured IN. I then added an implicit layer in the first model, which becomes INODE and GODEIN (voltage only). In three weeks I decided to model circuits as IN and wrote a first version of all models presented and added current prediction and power computation, as well as tripartite models.

I then spent about ten days trying to improve the GODEIN models. The main idea was to combine current and voltage predictions such that edge features (containing current) would be updated at every solver call of the implicit layer that predicts voltage. This gave the DYN-GODEIN model. I plan to keep working on this model as I believe it has the highest potential.

During this last month I prepared scripts to obtain preliminary results for the report and experimented a first version of an IN model that can handle 3-port devices with interdependent edges. This includes a new dataset with a transistor and an RLC filter and an extension of the IN framework. All scripts have been improved for better encapsulation with different abstraction levels. Working on 7 models and additional variations multiplies the time spent for every modification and the chances of encountering time-consuming bugs. I have spent a total of two weeks in this last month preparing scripts to handle a large variety of datasets and automating a part of the workflow, from circuit simulation to learning. Handling an end-to-end experiment, from dataset preparation to training results, with new architectures has been challenging given the limited amount of time after learning and reviewing the literature.

B The code

B.1 Dataset

Using ngspice we simulate a serial RLC circuit with a square voltage source and generate all net voltages and device currents. We use Pyspice for a python interface with the simulator. For the edge-featured models we model graphs using the DGL library to store simulation results (circuit states) as graphs with node and edge features. We have the choice between storing all circuit states as DGL graphs or storing only static components of the circuit states in DGL graphs. In the first option every node feature and edge feature is an array containing node and edge states at all time, respectively. Using the second approach edge and node features are single vectors and the dynamic part is concatenated during learning to form the complete edge state (the most typical example is the current component of edge features). Furthermore, only edge features are stored for edge-featured model, as the one-dimensional node features are purely dynamic. Tripartite model however requires node features only as it does not use edge features.

We choose to save static edge features and edge currents for all time steps as separate edge features. We store node voltage independently in an hdf5 file. This is close to the first possibility described above. We concatenate edge features (current and device) during learning.

For tripartite models we construct nodes depending on the type of the node. Nets take voltages, ports take currents and devices take component features.

B.2 Training

The first training step consists in generating haiku MLPs. We construct dummy matrices as entries to generate all MLPS and initialize random parameters. From this step we get MLPs and their parameters.

The second step consists in initializing optax optimizer states (and choosing the optimizer we want) and jax loss gradients with respect to every MLP's parameters.

Then starts the training loops:

We choose a random simulation and a random time step. We compute edge features Ra from the graph and chosen time step. We compute as well Rs and Rr from the dgl graph. We want an "undirected" graph to compute messages for both device ports so we concatenate horizontally Rr to Rs and Rs to Rr and concatenate Ra to Ra. Both nodes of every device is now sender and receiver node, for two different edges. We compute ORs, ORr. We now have edge features, node features and graph topology.

The forward function has all required elements to compute predictions. We can now compute loss gradients with respect to all parameters and backpropagate using apply updates using optax.

We repeat these last operations as many times as needed and observe the loss. We compare the loss to the loss of a constant function.

C Results

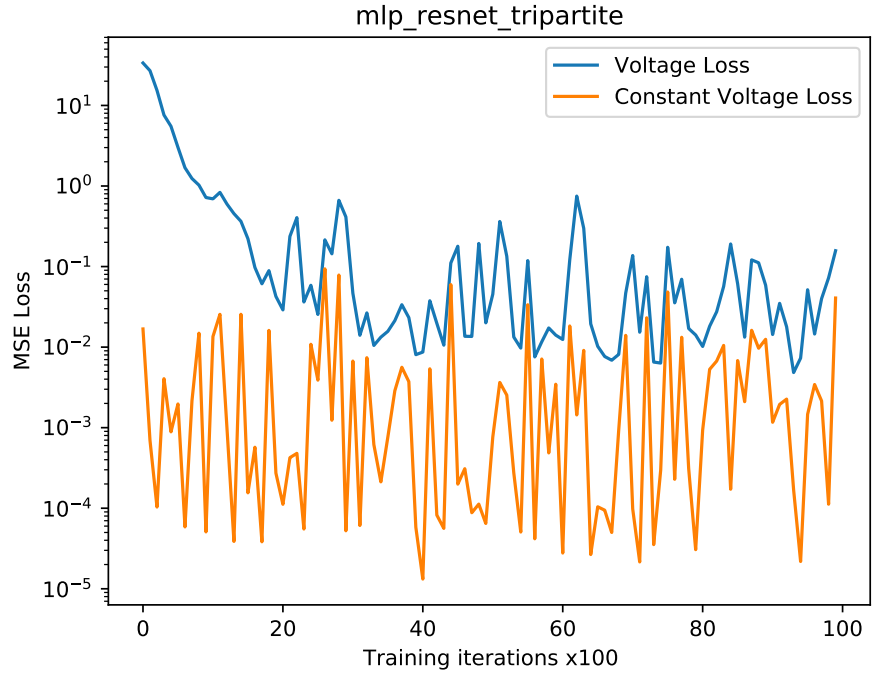


Figure 19: Voltage loss for constant function and RESIN model with tripartite representation

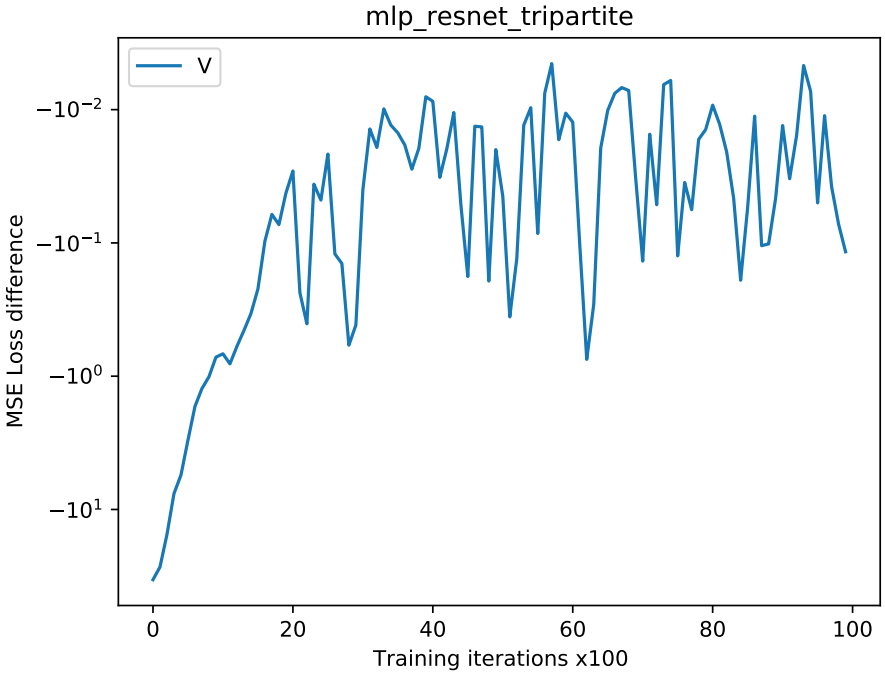


Figure 20: Voltage loss difference between constant function and RESIN model with tripartite representation

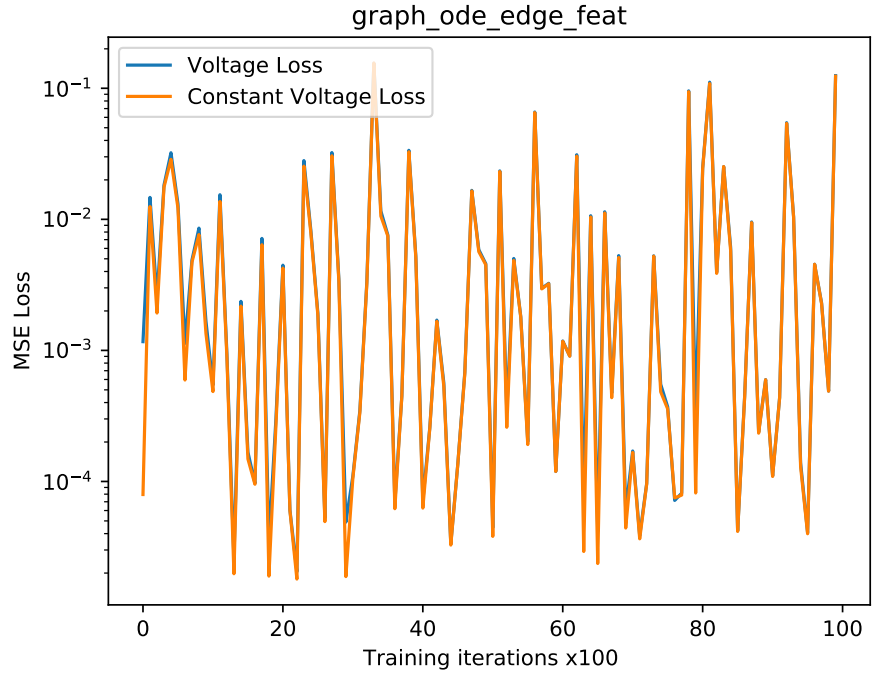


Figure 21: Voltage loss for constant function and GODEIN-static model with edge features

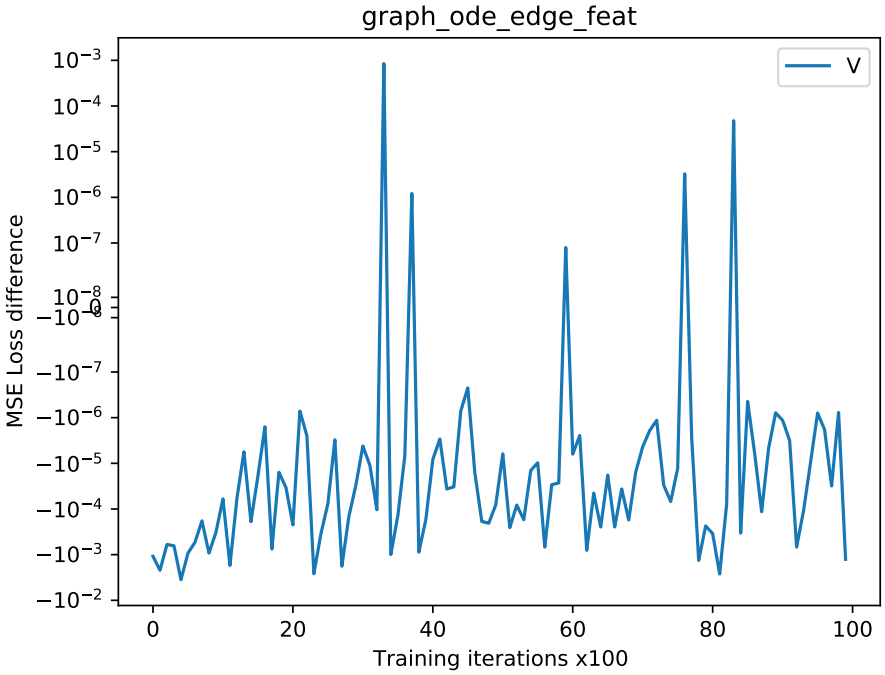


Figure 22: Voltage loss difference between constant function and GODEIN-static model with edge features

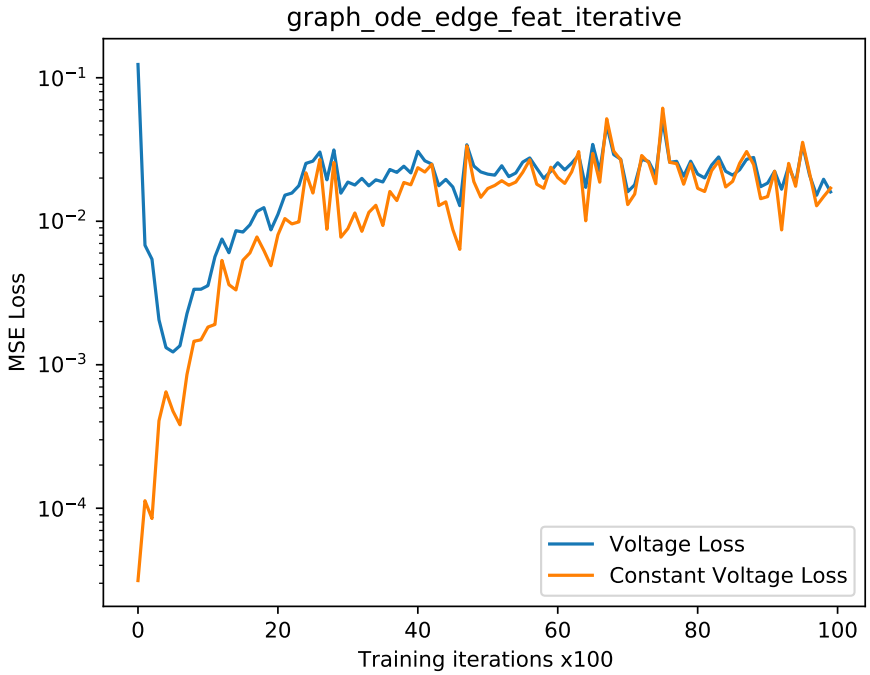


Figure 23: Voltage loss for GODEIN-ITER model with edge features

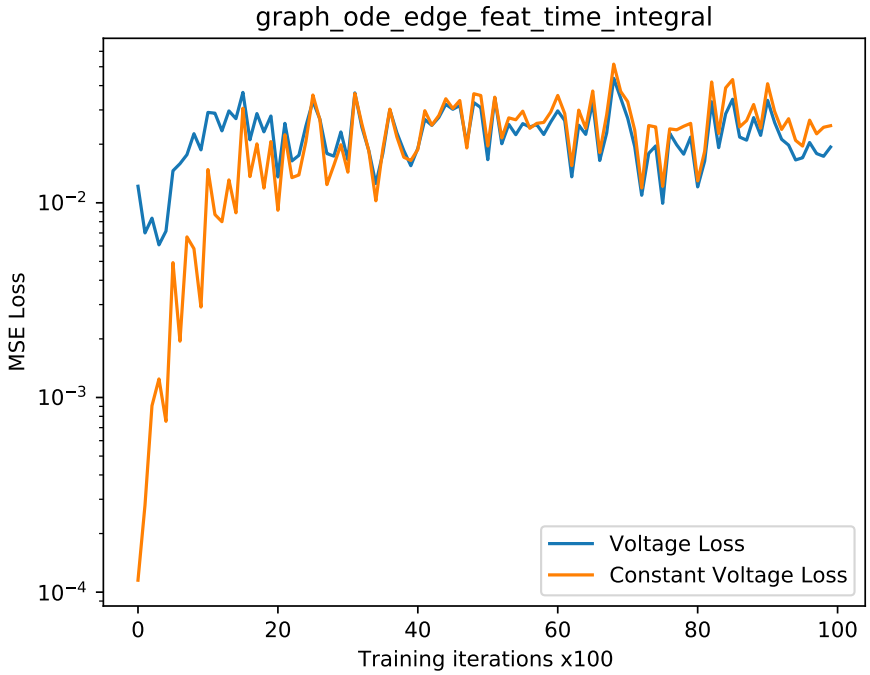


Figure 24: Voltage loss for GODEIN-TIME model with edge features

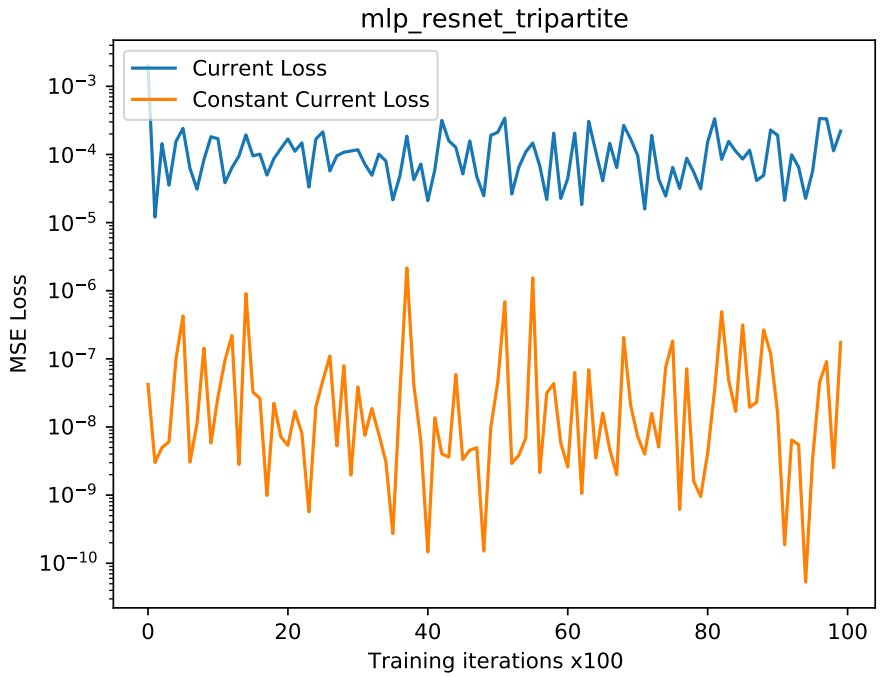


Figure 25: Current loss for constant function and RESIN model with tripartite representation

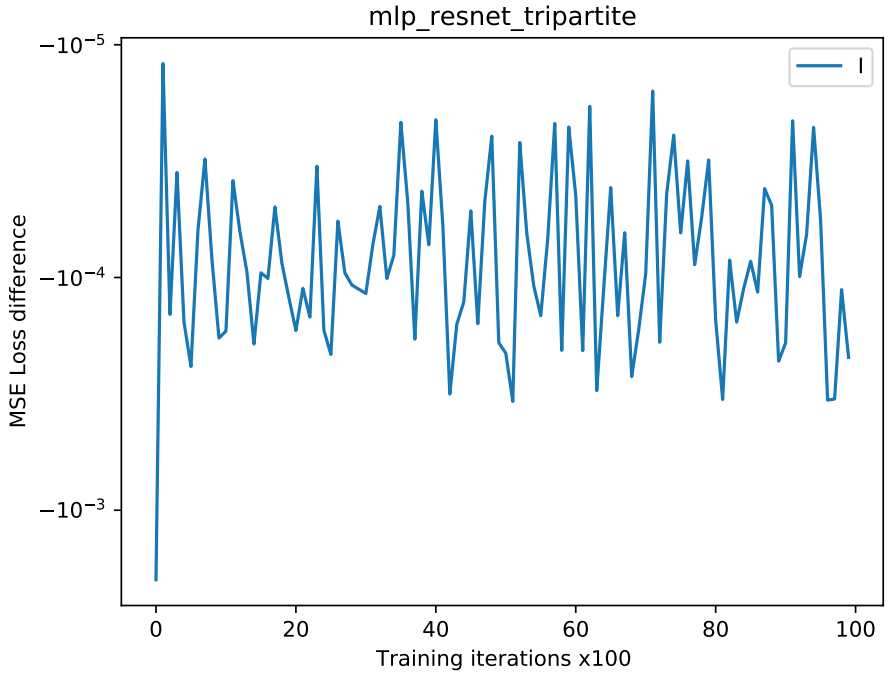


Figure 26: Current loss difference between constant function and RESIN model with tripartite representation

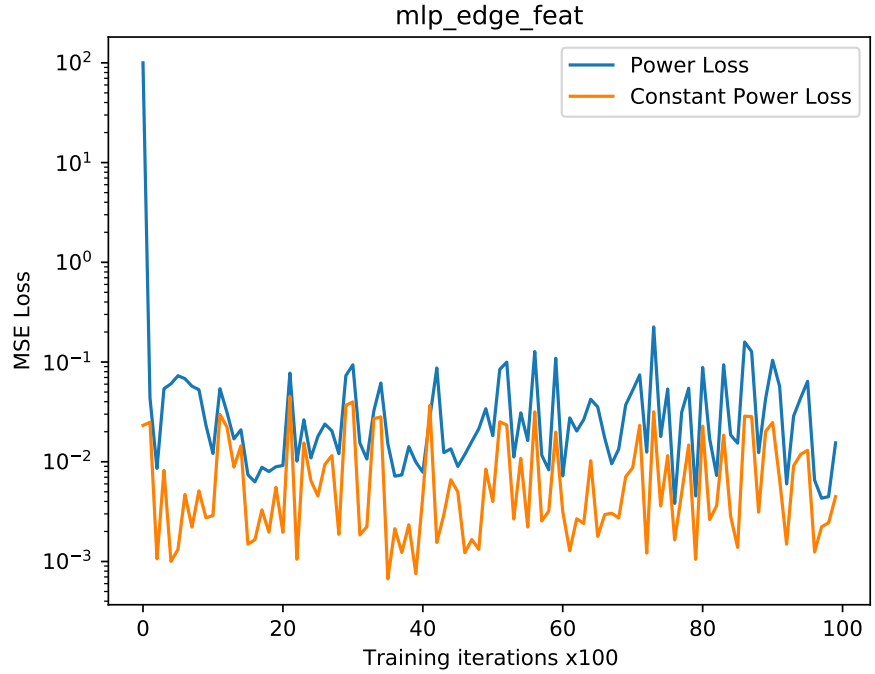
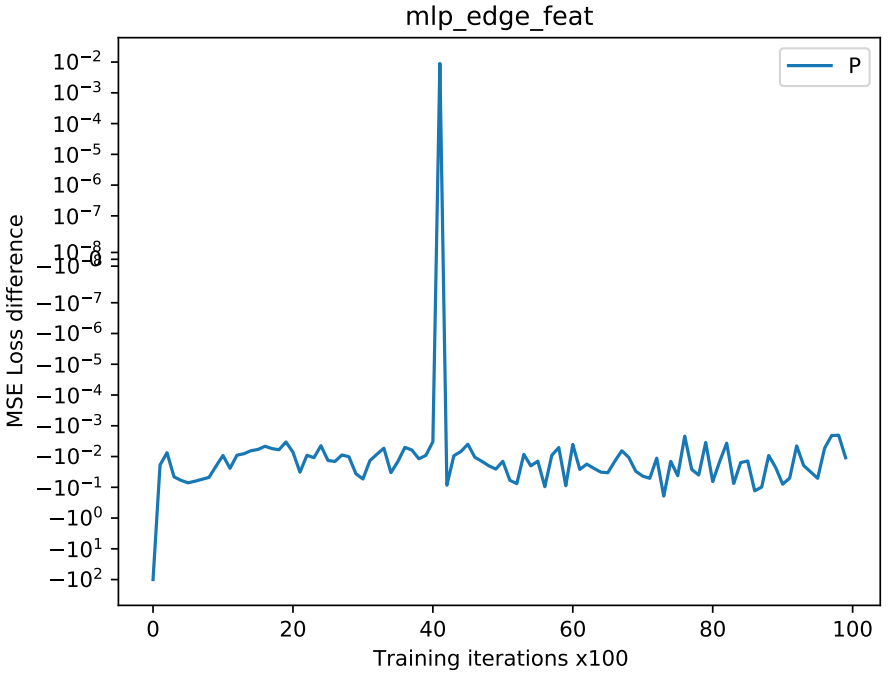


Figure 27: Power loss for constant function and for IN model with edge features



/

Figure 28: Power loss difference between constant function and IN model with edge features

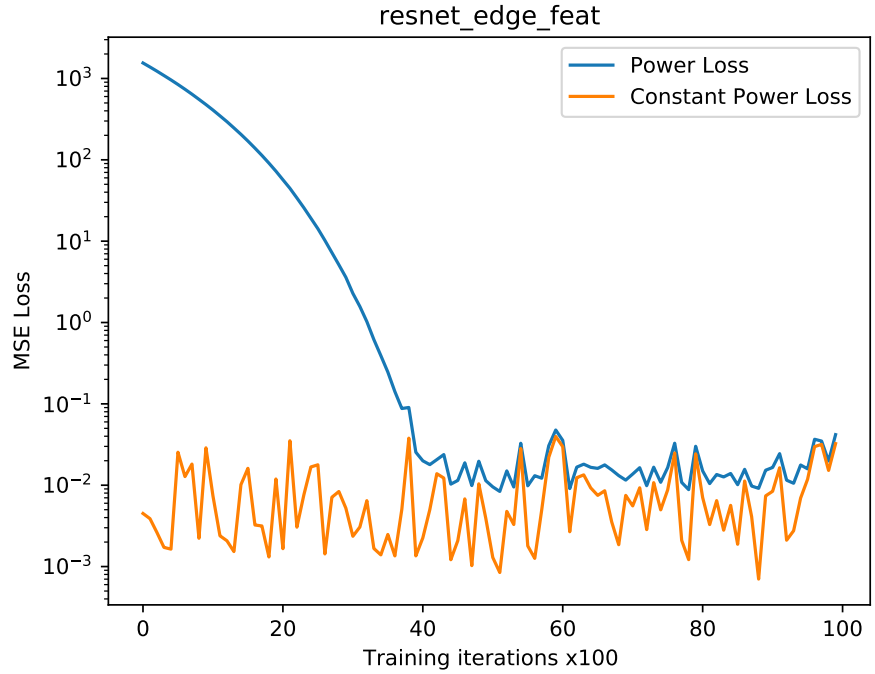


Figure 29: Power loss for RESIN model with edge features

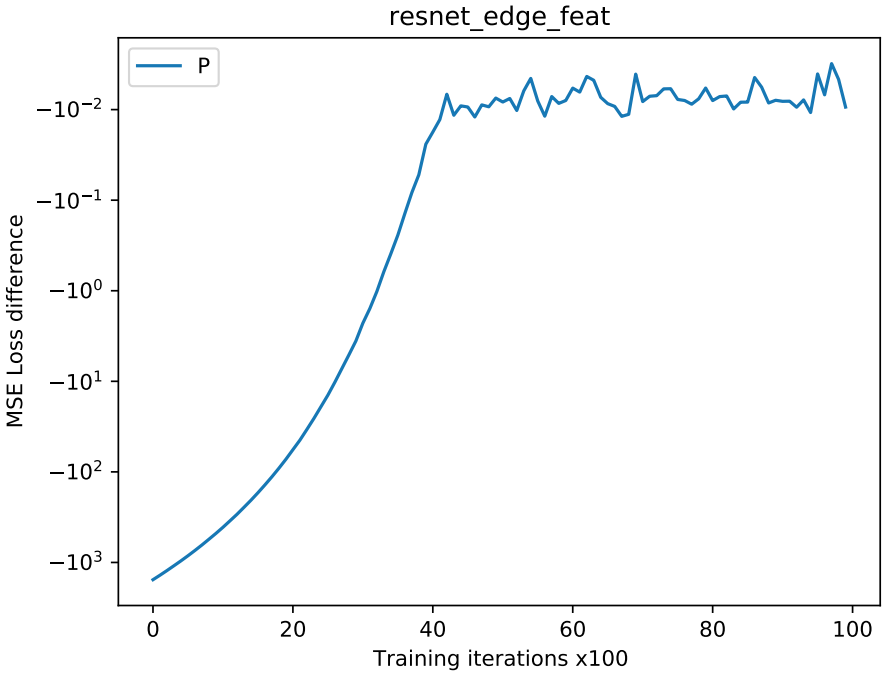


Figure 30: Power loss difference between constant function and RESIN model with edge features

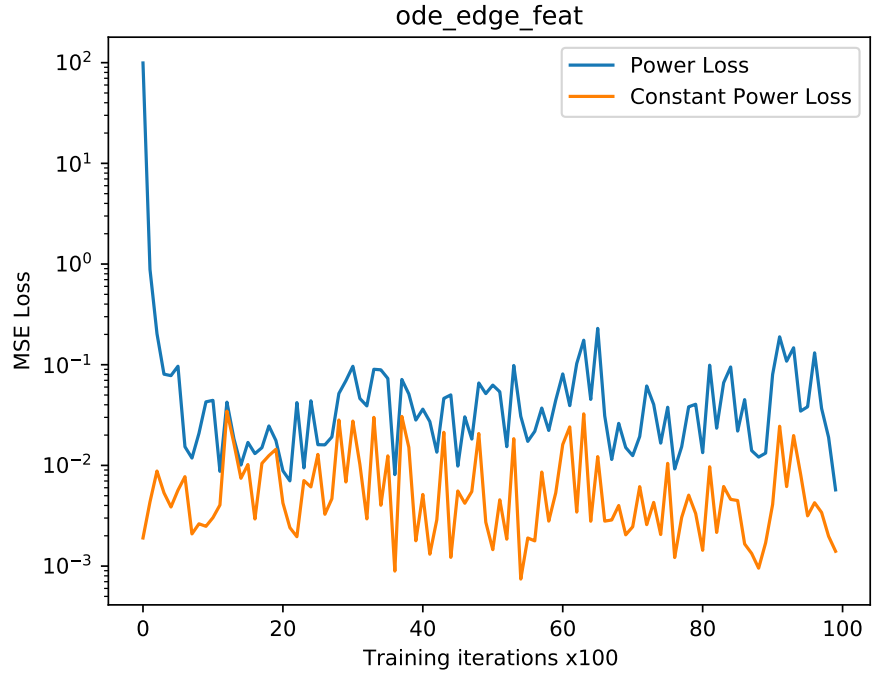


Figure 31: Power loss for constant function and INODE model with edge features

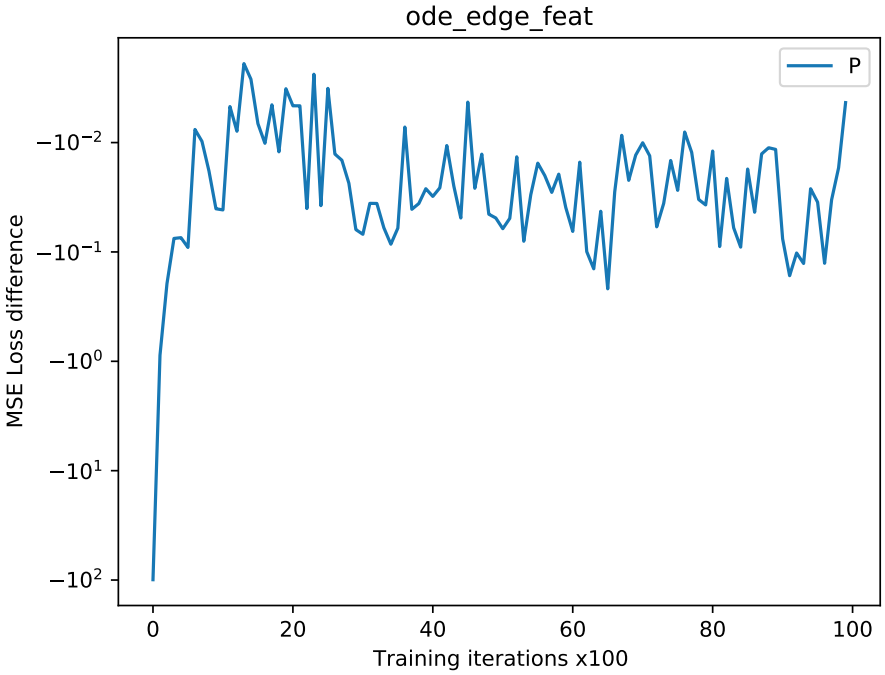


Figure 32: Power loss difference between constant function and INODE model with edge features

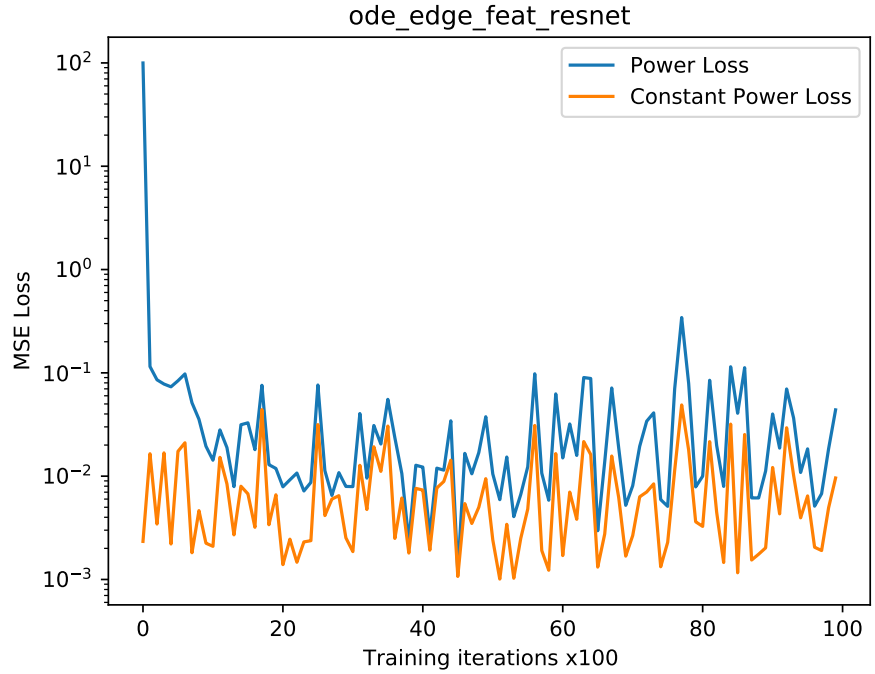


Figure 33: Power loss for RESINODE model with edge features

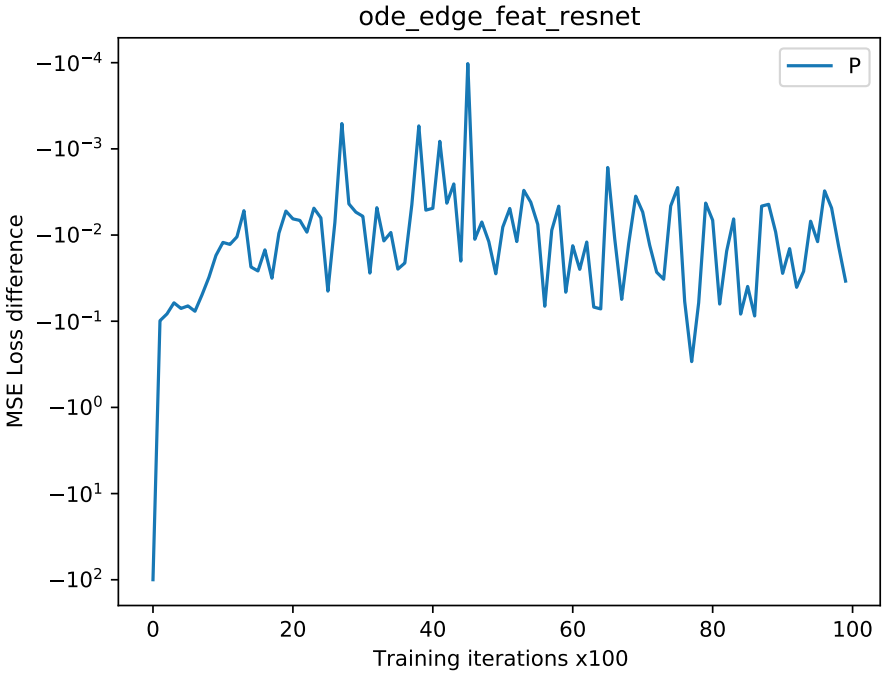


Figure 34: Power loss difference between constant function and RESINODE model with edge features

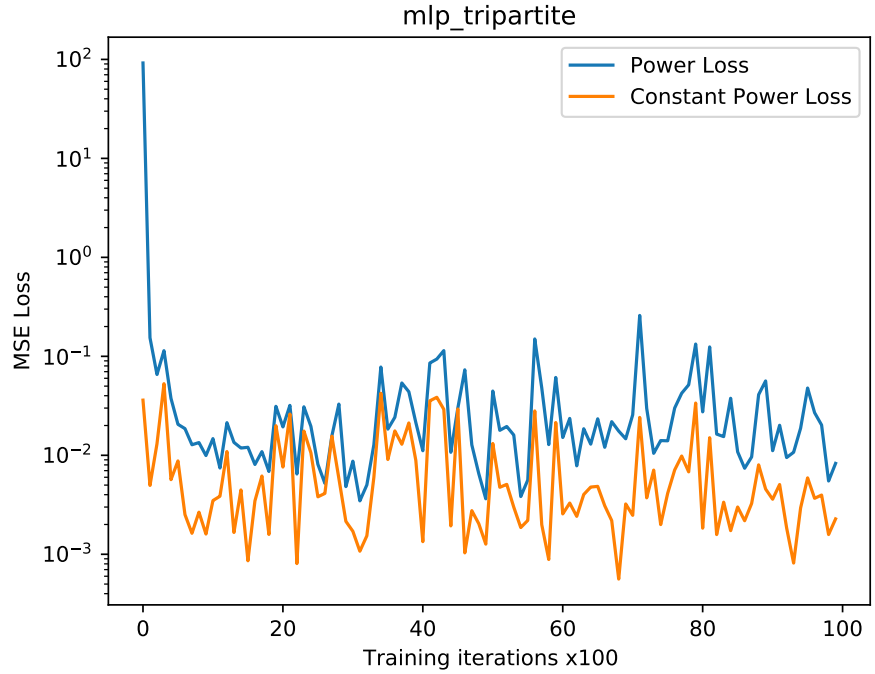


Figure 35: Power loss for IN model with tripartite representation

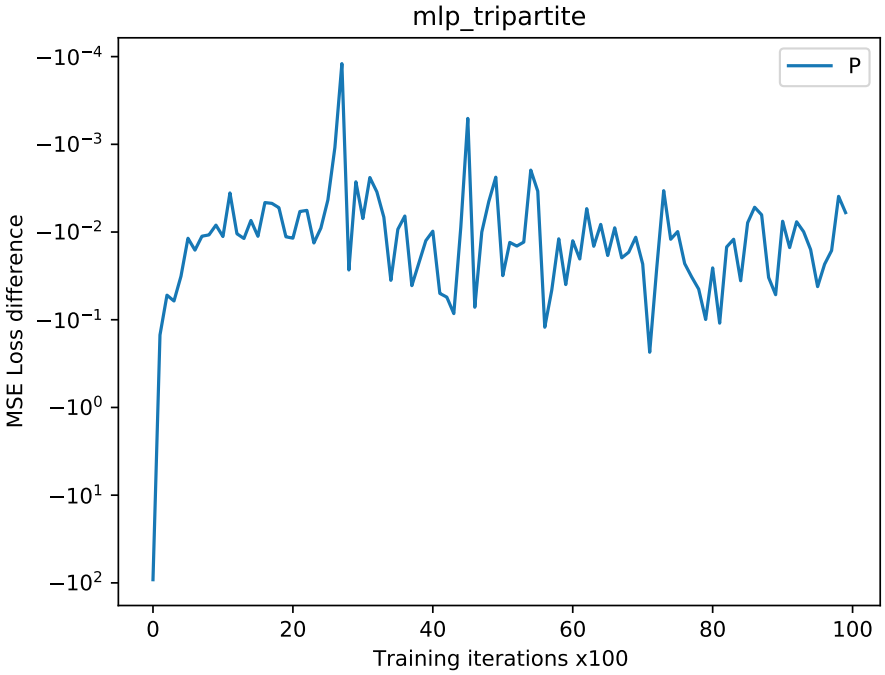


Figure 36: Power loss difference between constant function and IN model with tripartite representation

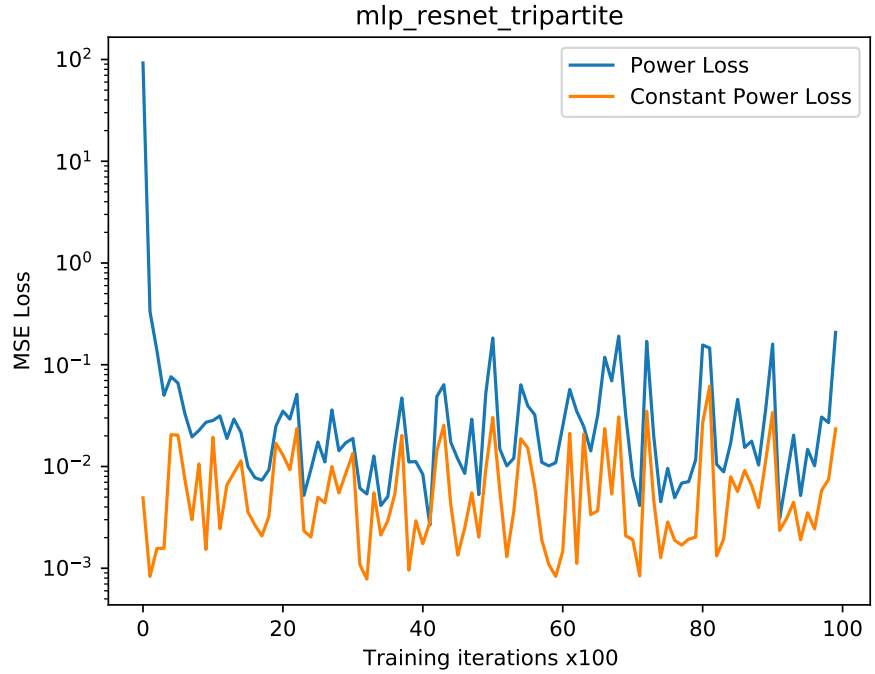


Figure 37: Power loss for constant function and RESIN model with tripartite representation

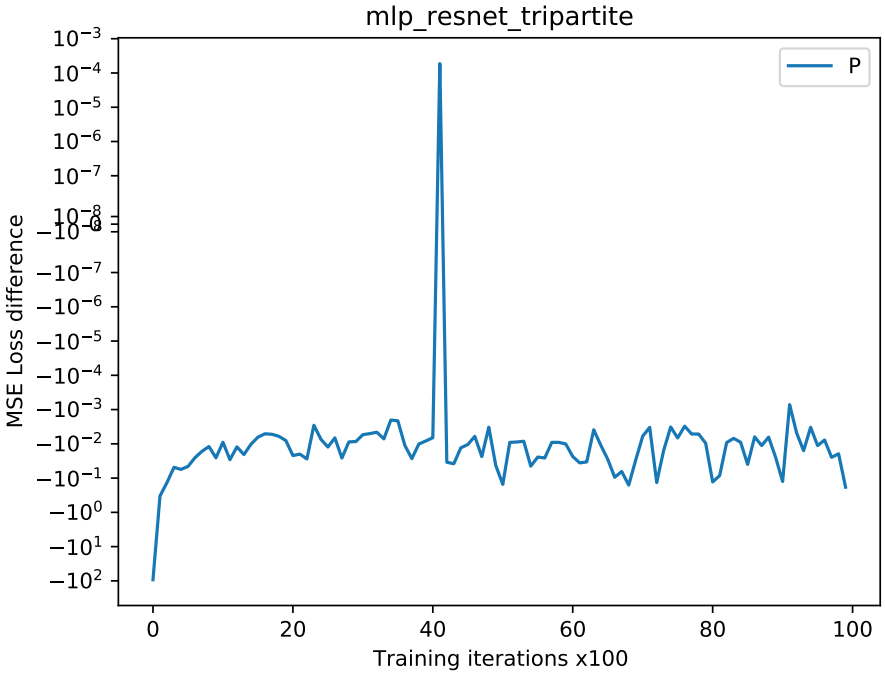


Figure 38: Power loss difference between constant function and RESIN model with tripartite representation

D Transistor and 3-ports device

We present here preliminary results of IN and RESIN models with edge features applied to hypergraph neural networks for 3-port device.

Results will be added here for the presentation.

E Plots

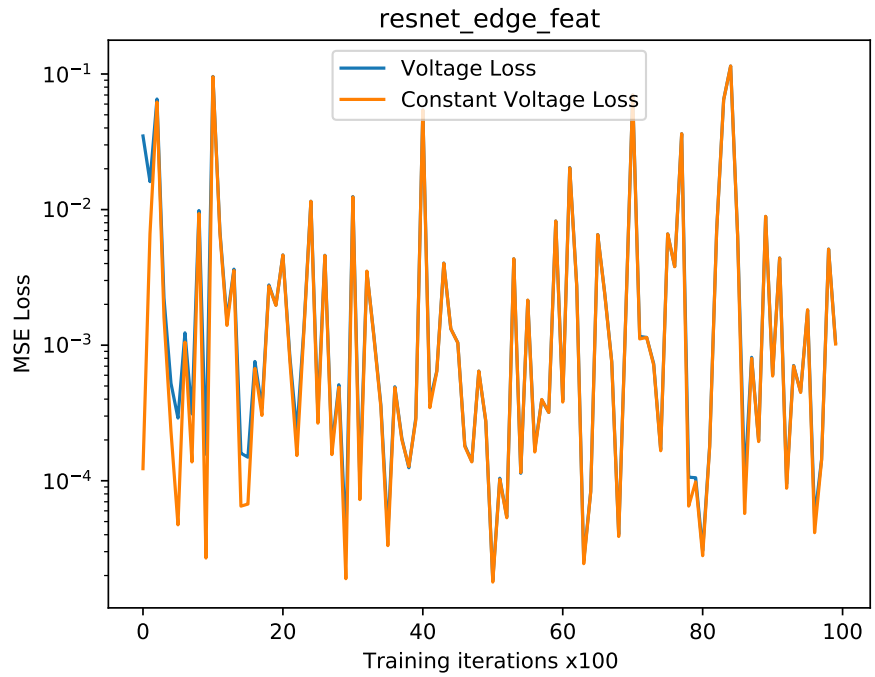


Figure 39: Voltage loss for RESIN model with edge features

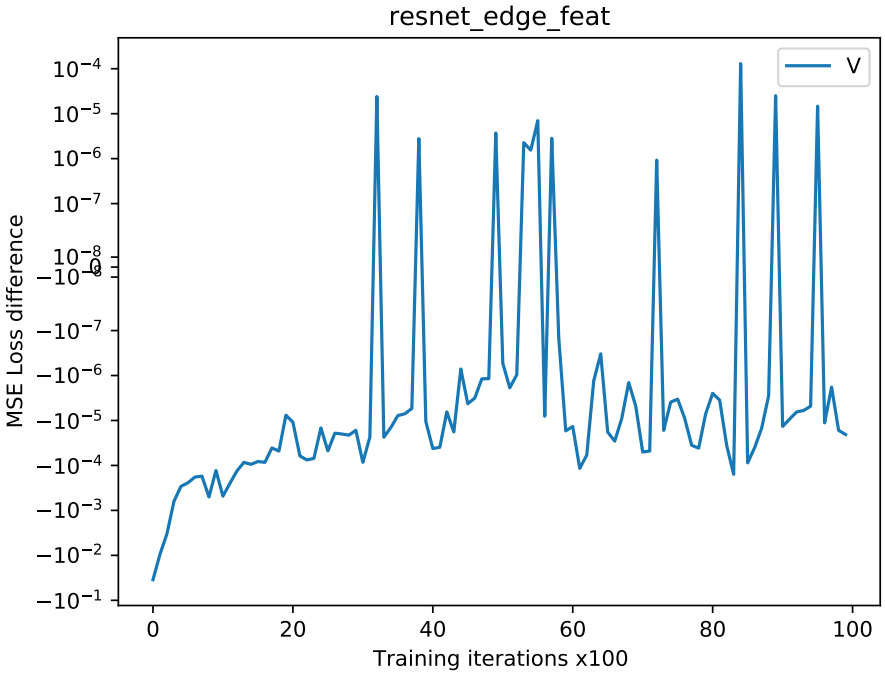


Figure 40: Voltage loss difference between constant function and RESIN model with edge features

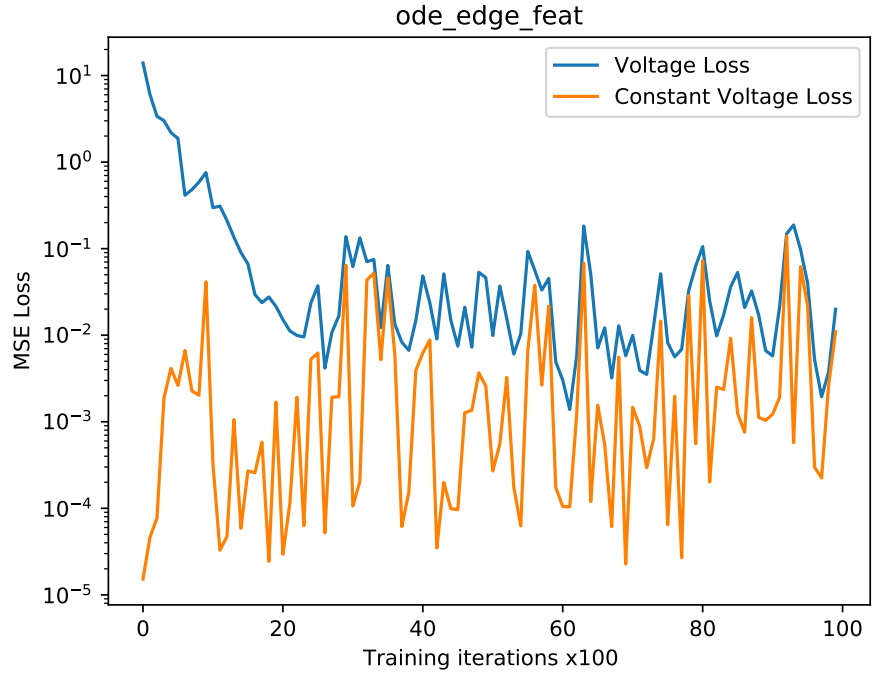


Figure 41: Voltage loss for constant function and INODE model with edge features

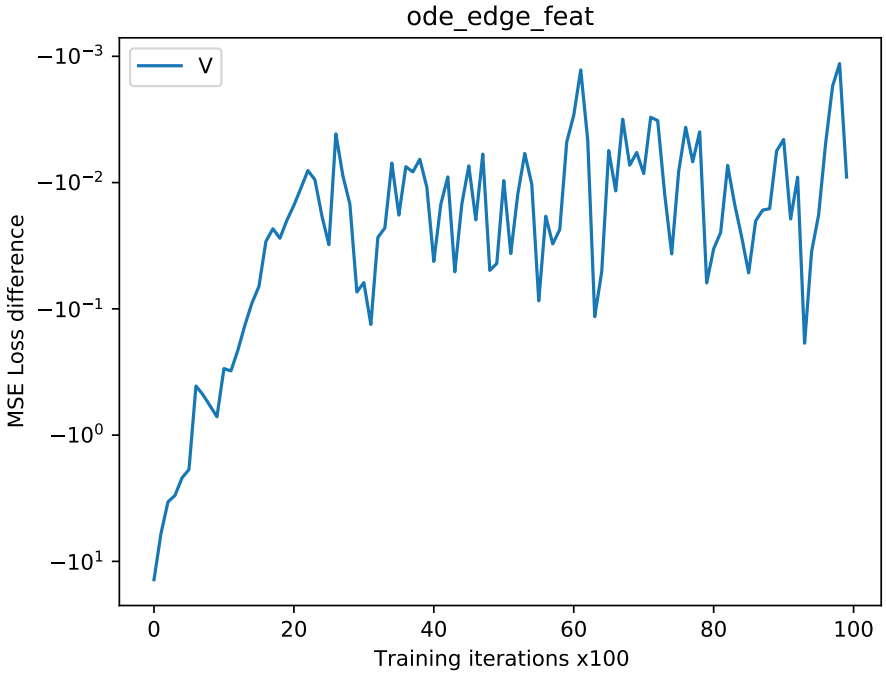


Figure 42: Voltage loss difference between constant function and INODE model with edge features

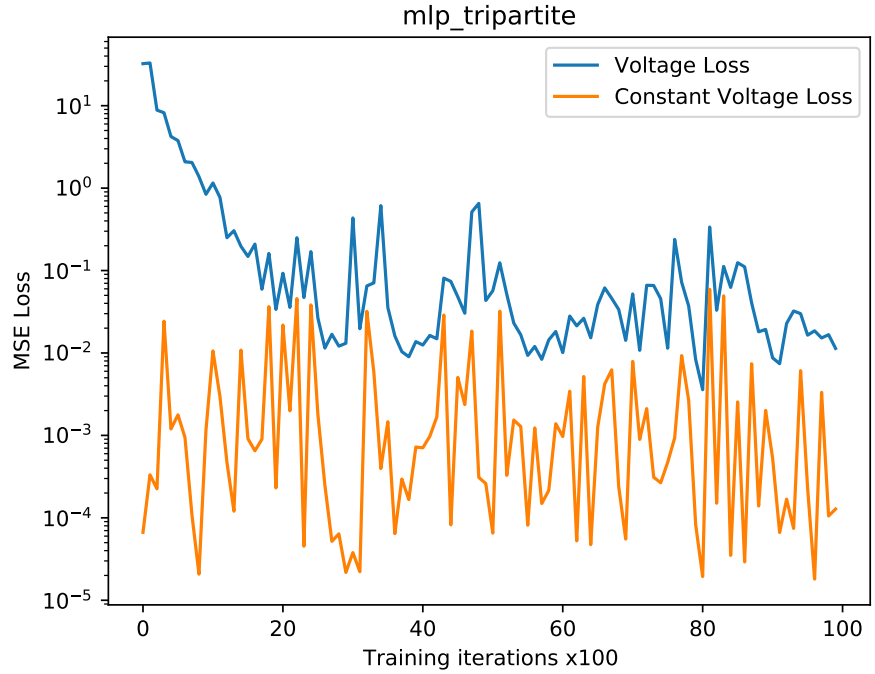


Figure 43: Voltage loss for IN model with tripartite representation

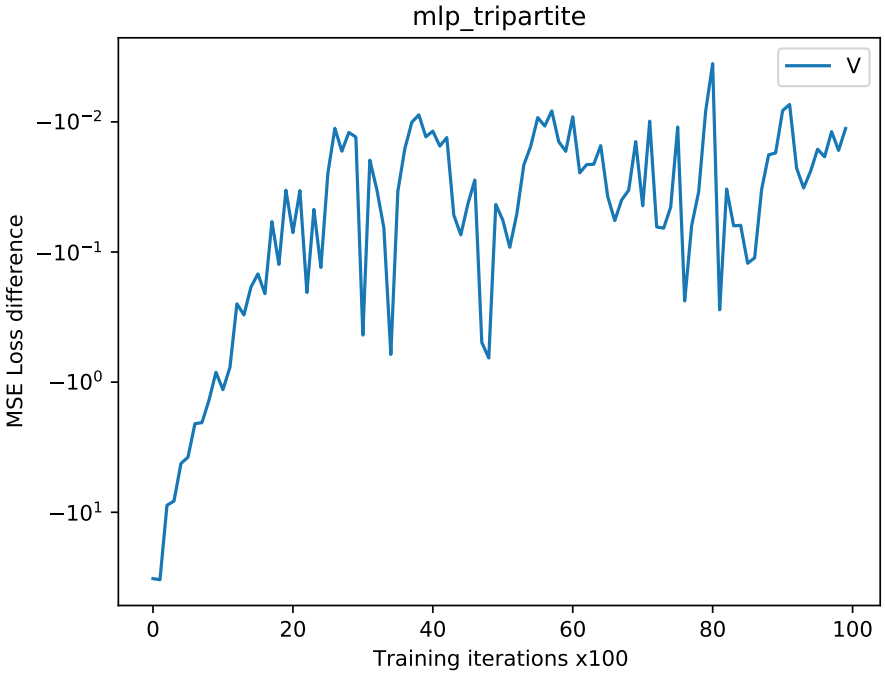


Figure 44: Voltage loss difference between constant function and IN model with tripartite representation

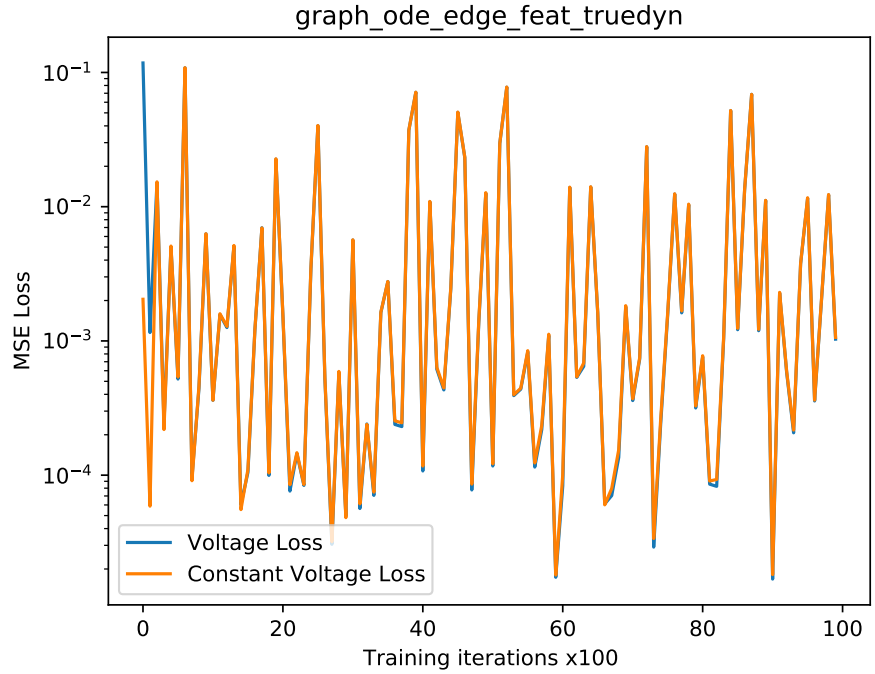


Figure 45: Voltage loss for constant function and GODEIN-DYN model with edge features

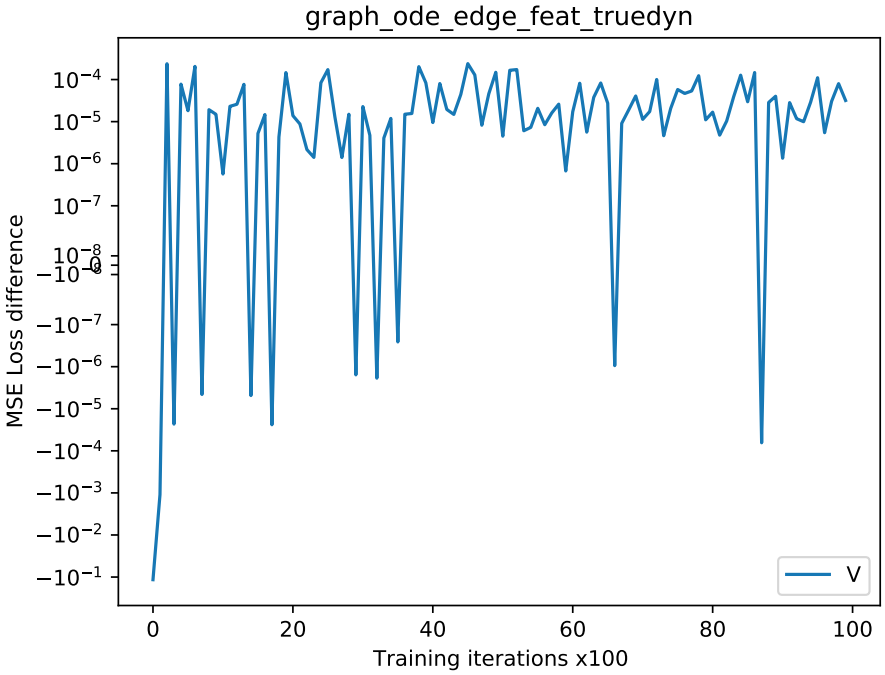


Figure 46: Voltage loss difference between constant function and GODEIN-DYN model with edge features

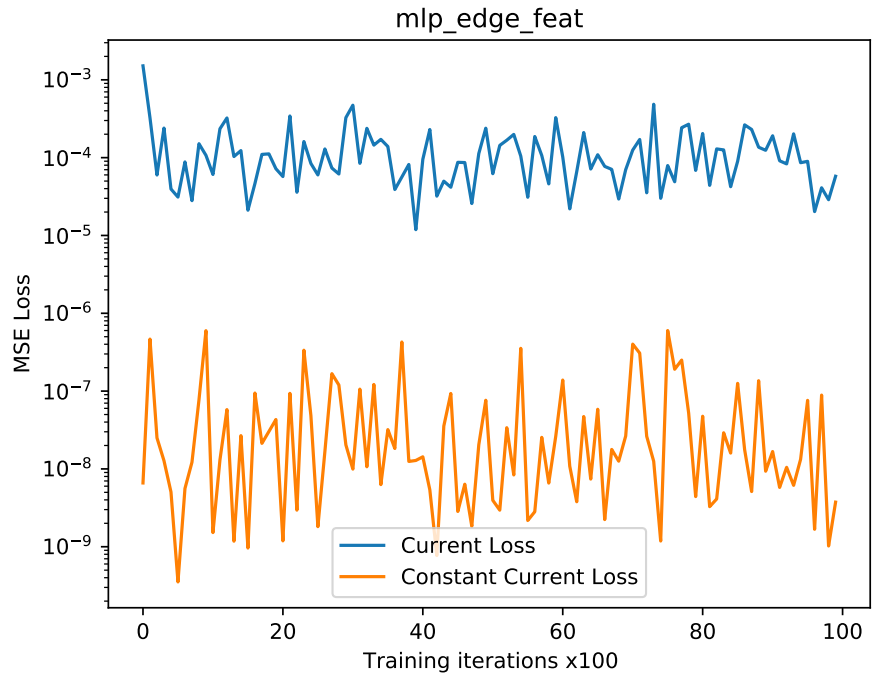
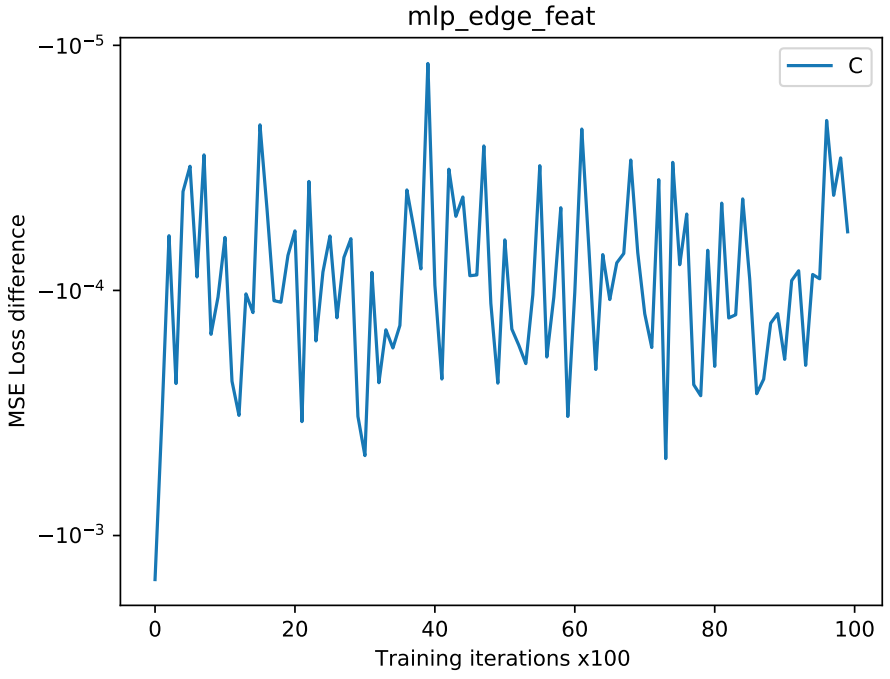


Figure 47: Current loss for constant function and for IN model with edge features



/

Figure 48: Current loss difference between constant function and IN model with edge features

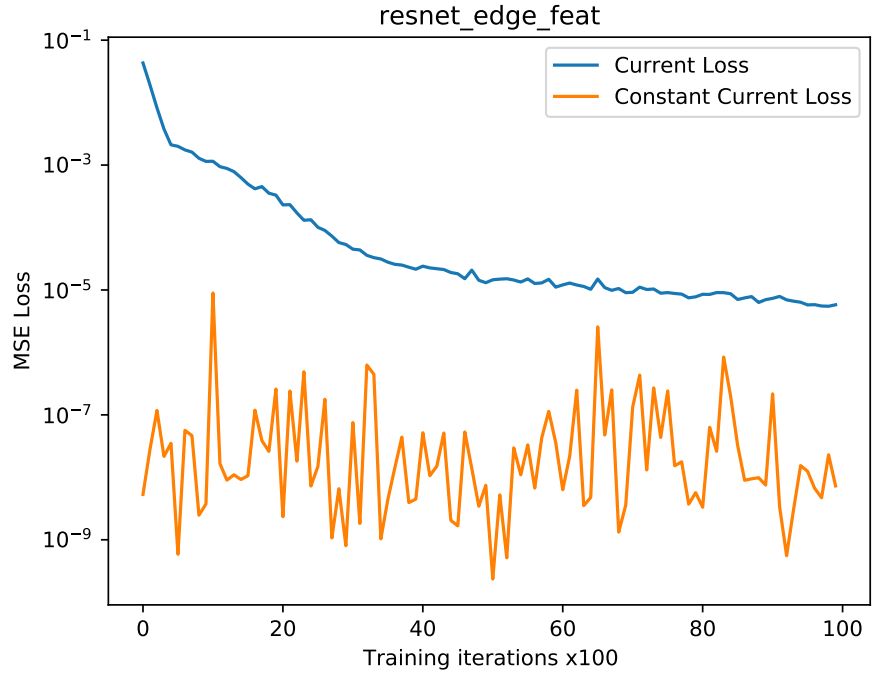


Figure 49: Current loss for RESIN model with edge features

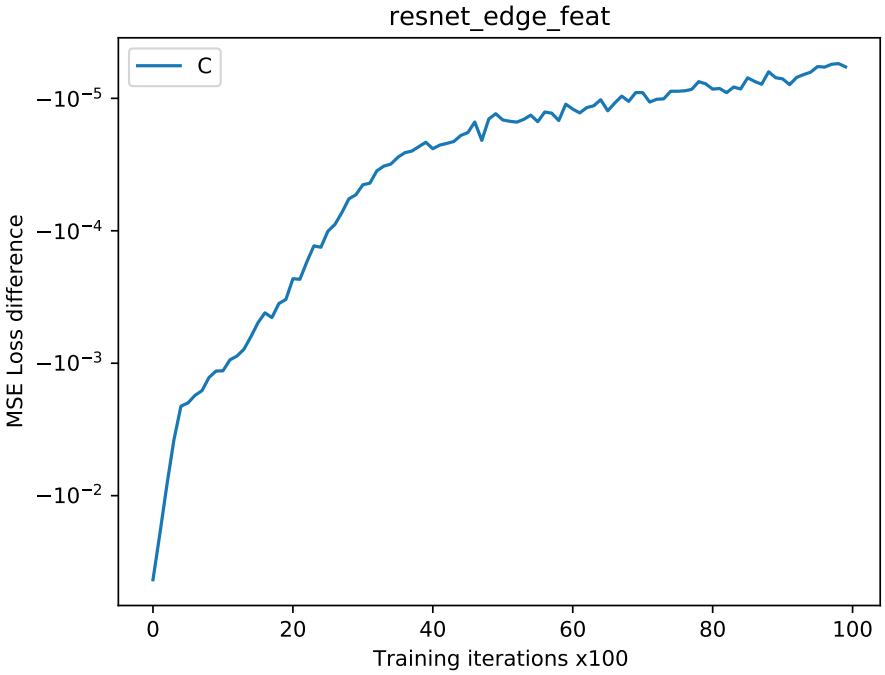


Figure 50: Current loss difference between constant function and RESIN model with edge features

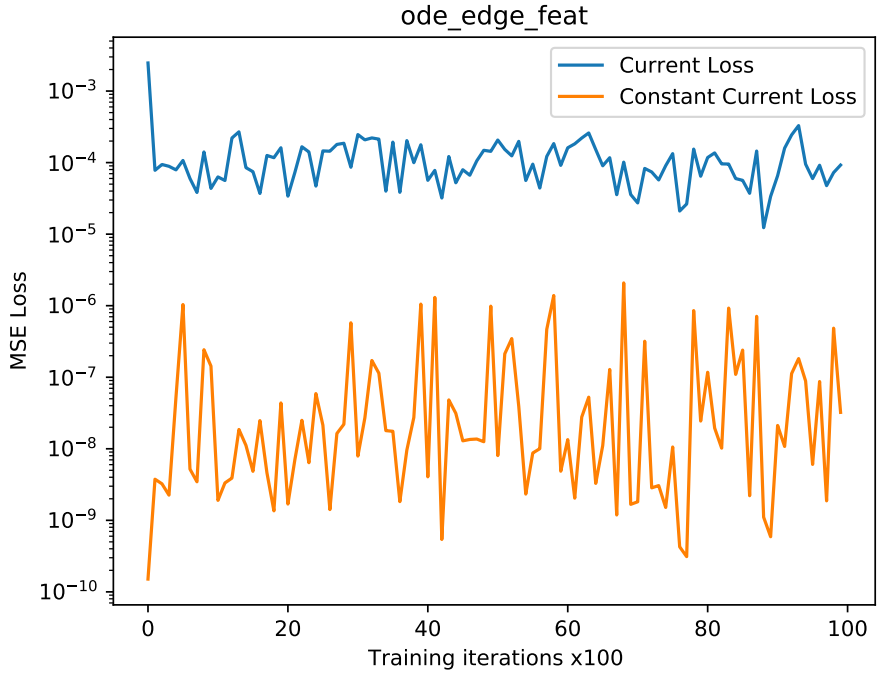


Figure 51: Current loss for constant function and INODE model with edge features

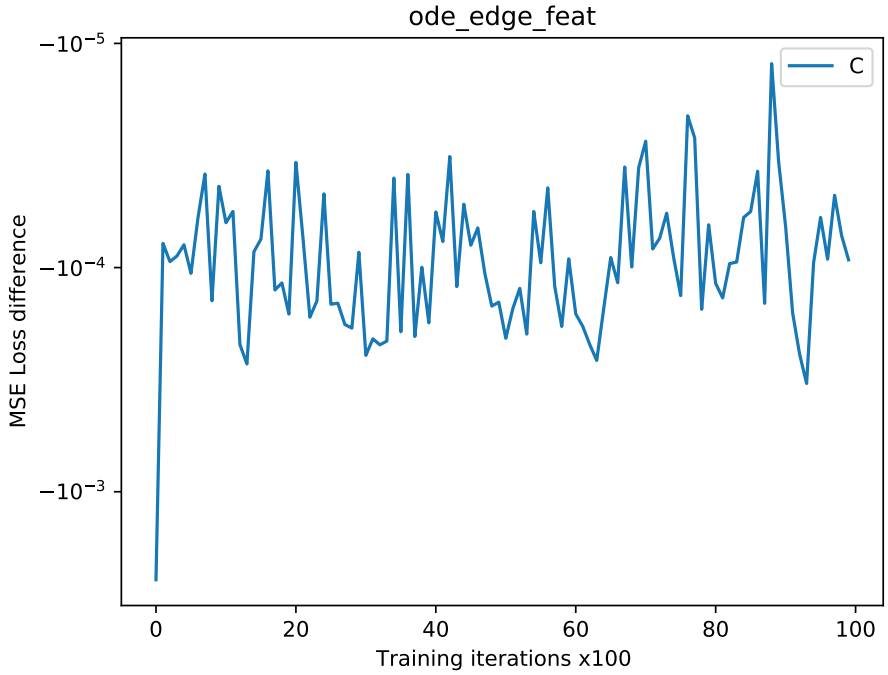


Figure 52: Current loss difference between constant function and INODE model with edge features

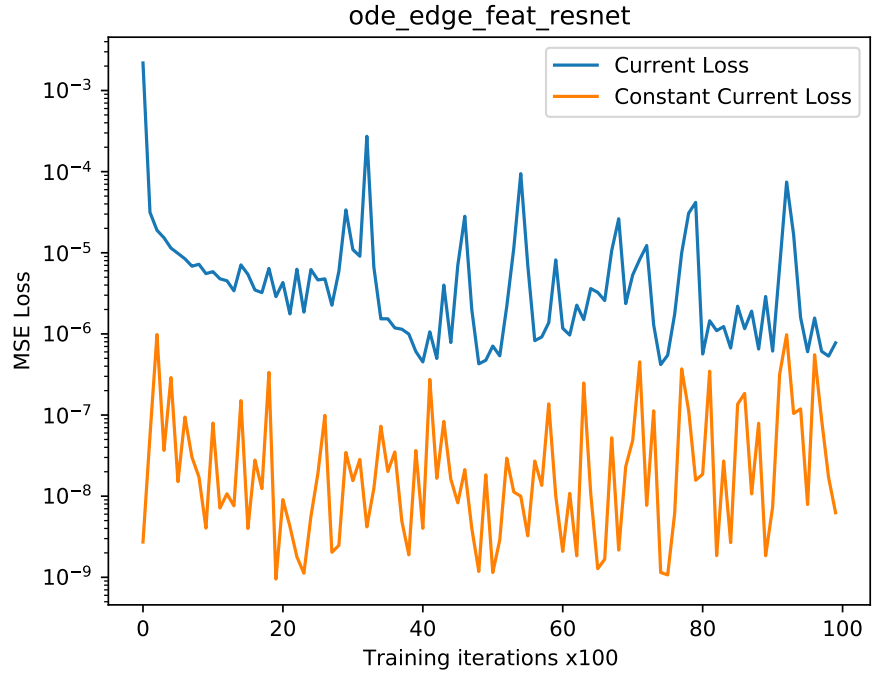


Figure 53: Current loss for RESINODE model with edge features

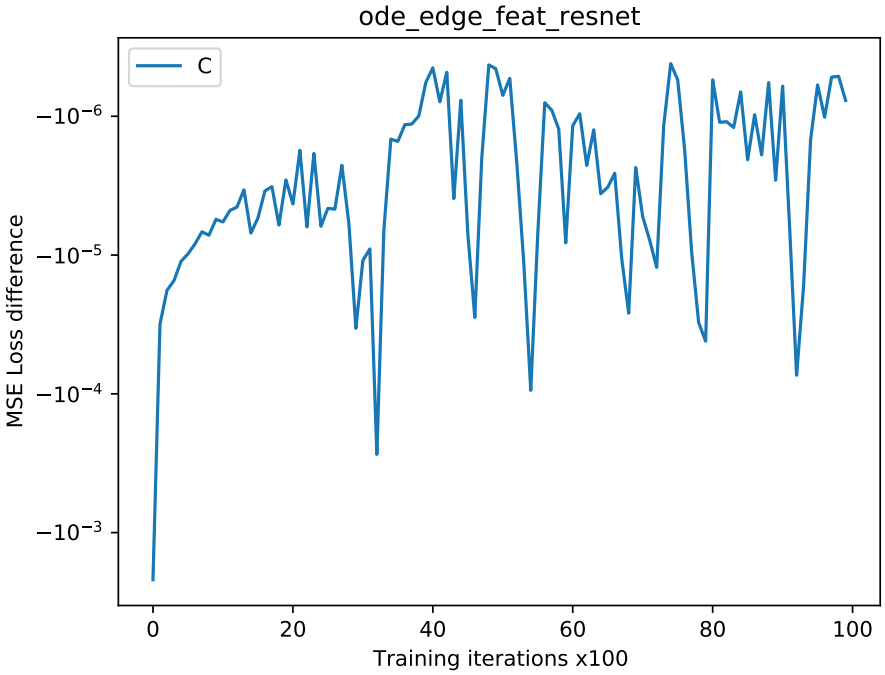


Figure 54: Current loss difference between constant function and RESINODE model with edge features

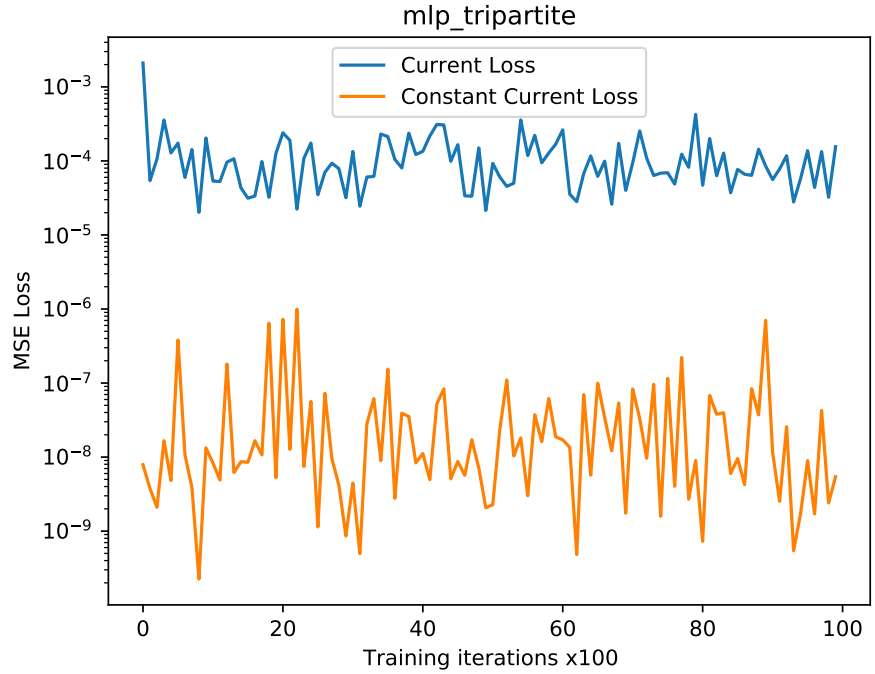


Figure 55: Current loss for IN model with tripartite representation

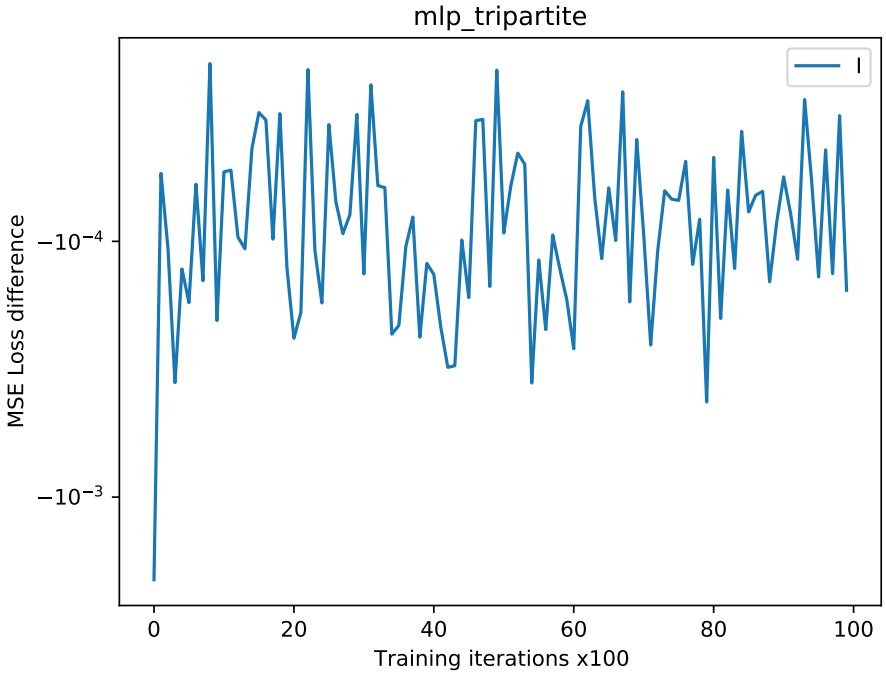


Figure 56: Current loss difference between constant function and IN model with tripartite representation

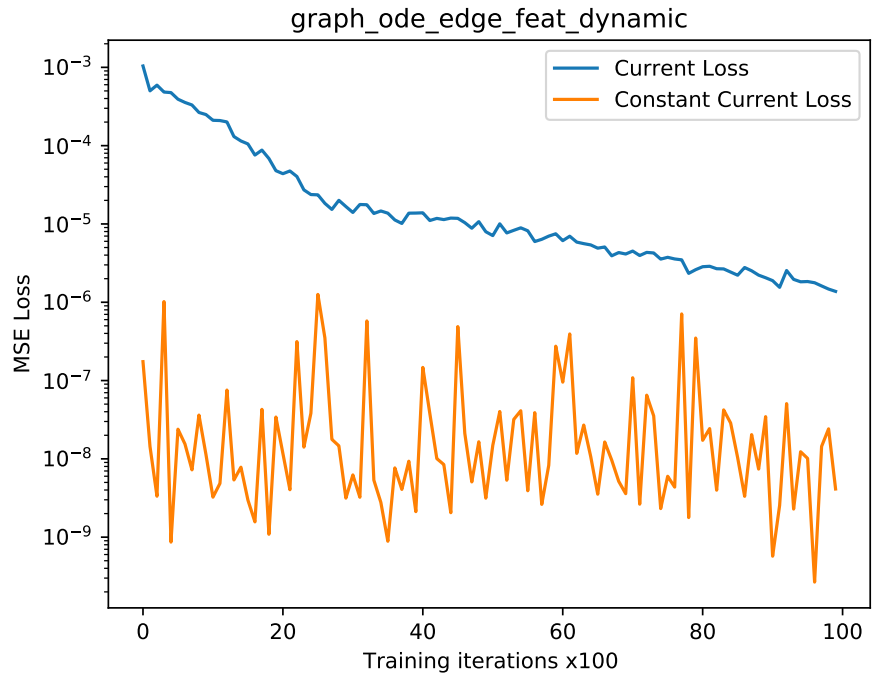


Figure 57: Current loss for constant function and GODEIN model with edge features

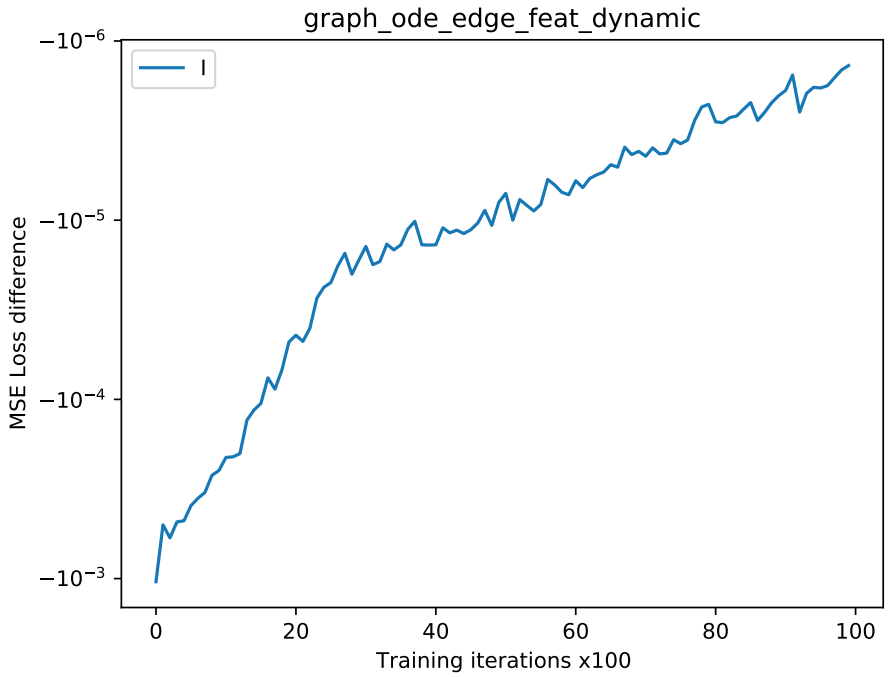


Figure 58: Current loss difference between constant function and GODEIN model with edge features

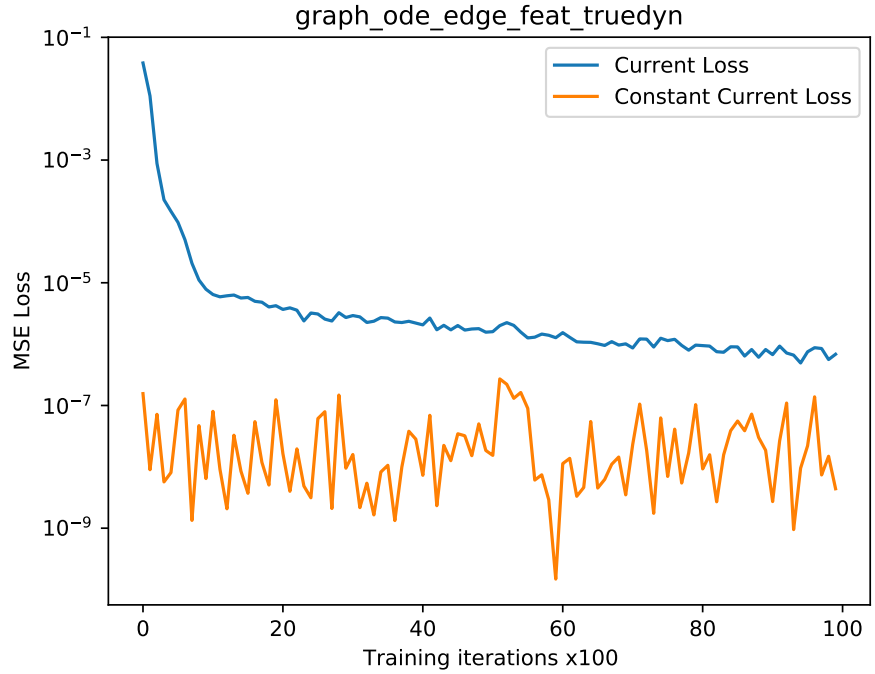


Figure 59: Current loss for constant function and GODEIN-DYN model with edge features

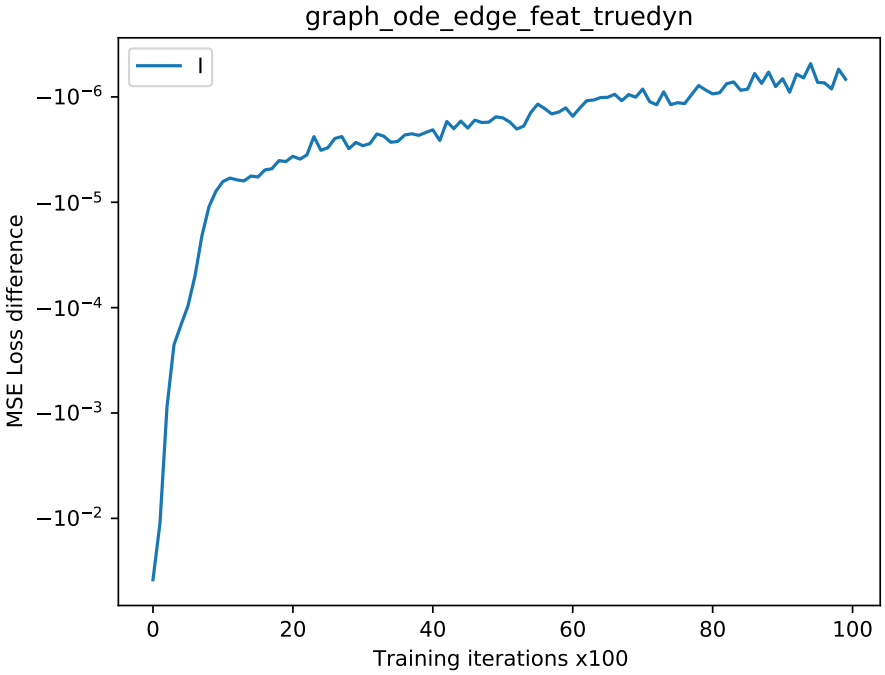


Figure 60: Current loss difference between constant function and GODEIN-DYN model with edge features