DECISION TREES THAT REMEMBER: GRADIENT-BASED LEARNING OF RECURRENT DECISION TREES WITH MEMORY

Sascha Marton[1]* **Moritz Schneider**[2]* **Jannik Brinkmann**[3] **Stefan Lüdtke**[4]
Christian Bartelt[3] **Heiner Stuckenschmidt**[1]

[1]University of Mannheim [2]Boehringer Ingelheim [3]Technical University of Clausthal
[4]University of Rostock
sascha.marton@uni-mannheim.de moritz.schneider@boehringer-ingelheim.com

ABSTRACT

Neural architectures such as Recurrent Neural Networks (RNNs), Transformers, and State-Space Models have shown great success in handling sequential data by learning temporal dependencies. Decision Trees (DTs), on the other hand, remain a widely used class of models for structured tabular data but are typically not designed to capture sequential patterns directly. Instead, DT-based approaches for time-series data often rely on feature engineering, such as manually incorporating lag features, which can be suboptimal for capturing complex temporal dependencies. To address this limitation, we introduce ReMeDe Trees, a novel recurrent decision tree architecture that integrates an internal memory mechanism, similar to RNNs, to learn long-term dependencies in sequential data. Our model learns hard, axis-aligned decision rules for both output generation and state updates, optimizing them efficiently via gradient descent. We provide a proof-of-concept study on synthetic benchmarks to demonstrate the effectiveness of our approach.

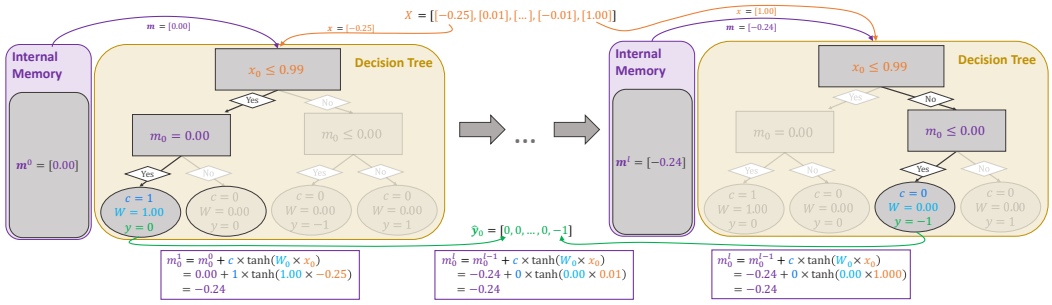

Figure 1: **Recurrent Decision Tree Example** This figure shows the minimal ReMeDe tree solving a sign recognition task. The task is to memorize the sign of $x \in (-0.5, 0.5)$ at the first position and predict it (-1 or 1) upon a trigger (1), while intermediate positions hold zeros plus small noise. At the root node, the ReMeDe tree checks whether the trigger occurs. If not (left branch), there are two cases: If the hidden state is zero, it updates based on input, adopting the sign of the entry; otherwise, it remains unchanged. If the trigger occurs (right branch), the tree splits on the hidden state to predict the sign of the first value: negative for a negative hidden state, positive otherwise.

1 INTRODUCTION

Dealing with sequential, i.e. time-dependent, tabular data is an important area of Machine Learning research. Besides forecasting, dynamic modeling is crucial for data-driven control methods. Both have many practical applications in science, finance, healthcare, and many industrial areas.

*Equal Contribution

Generally speaking, there are two distinct structural approaches to learn dependencies over time. The often employed *memory window* approach, that can be used with any type of regression or classification algorithm, reduces the temporal dependencies to a static prediction problem by collapsing the past L (input or output) values within a time series into a flat input to the model. This is also sometimes referred to as (Nonlinear) Autoregressive Exogenous Model ((N)ARX) (Nelles, 2020). The other approach are *recurrent* models, which deal with time dependency explicitly. Modern forms of recurrent architectures like Recurrent Neural Networks (RNNs) (Elman, 1990), Long Short-Term Memory networks (LSTMs) (Schmidhuber et al., 1997) define a hidden memory state, which is updated in each inference step together with the calculation of the model outputs.

Recurrent approaches are in principle more powerful, because the model can deal with long-term dependencies exceeding any practical choice of L for the memory window approach. Unfortunately, truly recurrent (neural) models are still challenging to train due to unstable dynamics of backpropagated gradients over long sequences (Hochreiter, 1998), even with advances like gated memory units as in LSTM networks (Schmidhuber et al., 1997). In addition, training neural networks can afford a large amount of data. In real-world applications with limited data availability, they are often outperformed by tree-based ensembles such as XGBoost (Chen & Guestrin, 2016) or CatBoost (Prokhorenkova et al., 2018). Unfortunately, for sequential data, such approaches have to be used with the limited memory window technique.

In this paper, we introduce a novel decision tree algorithm, **Re**current **Me**mory **De**cision (ReMeDe) Trees, that, for the first time, incorporates recurrence in DTs through an internal memory mechanism. Building on the techniques proposed by Marton et al. (2024a), our method enables efficient training of decision trees via gradient descent, resulting in hard, axis-aligned recurrent DTs capable of handling sequential data through a learnable internal memory. To the best of our knowledge, this is the first approach to learn a memory-augmented recurrent decision tree model using backpropagation through time (Werbos, 1990). Specifically, our contributions are:

- We extend Gradient-Based Decision Trees (Marton et al., 2024a) by incorporating an internal memory mechanism that can be learned using backpropagation through time.
- We modify the internal nodes of decision trees to enable splits based on internal memory values, allowing pathing decisions to be conditioned on past experiences.
- We propose a novel update procedure for the internal memory, leveraging the decision tree's output at each time step and incorporating a hard memory gating mechanism.

First experiments with synthetic problems indicate that, similar to RNNs, ReMeDe Trees can overcome the limitations of fixed-size memory windows by efficiently compressing information in their hidden state. This suggests that ReMeDe Trees could offer a promising approach for time series tasks involving long-term dependencies, potentially combining the benefits of recurrent models for sequential data with the interpretability and axis-aligned structure of decision trees.

2 Recurrent Memory Decision Trees

In this section, we introduce the core method of our paper, Recurrent Memory Decision Trees (ReMeDe Trees). Therefore, we build on GradTree proposed by Marton et al. (2024a) to learn hard, axis-aligned decision trees with gradient descent. This is achieved by applying backpropagation with a straight-through operator on a dense DT representation, to jointly optimize all tree parameters and due to space constraints is introduced in more detail in Appendix A. Here, we describe how we extended GradTree by incorporating an internal memory and a gating mechanism. This enhancement enables the model to effectively learn sequential patterns and capture temporal dependencies, thereby improving its ability to process time-dependent data. Therefore, we focus on time-series problems, where for each time step $k = 1, 2, ...$, a value $\boldsymbol{x^k} \in \mathbb{R}^{n_x}$ is observed. The outputs $\boldsymbol{y^k} \in Y$ may be either continuous, where $Y \subset \mathbb{R}^{n_y}$, or discrete in which $Y \subset \mathbb{N}^{n_y}$.

The Hidden Memory There are two approaches to deal with time-depencency in dynamic models: (1) NARX models, which use L past inputs $\boldsymbol{x^{k-1}}, ..., \boldsymbol{x^{k-L}}$ or outputs $\boldsymbol{y^{k-1}}, ..., \boldsymbol{y^{k-L}}$ as inputs at time step k. (2) Recurrent models, which incorporate an n_m-dimensional memory $\mathrm{M} \subset \mathbb{R}^{n_m}$ to store compressed past information. The memory state $\mathbf{m^k}$ serves as an input to determine $\boldsymbol{y^k}$ and is updated in each inference step.

It is obvious that NARX models can only effectively model Markov Processes up to order L, whereas recurrent models may be able to deal with much higher information lags. In order to use a hidden memory M in a decision tree, it has to be observed by internal nodes, similar to X, and modified by leaf nodes, similar to Y. This means that we can use the same equations as in GradTrees, but with $\tilde{X} = X \times \mathrm{M}$ and $\tilde{Y} = Y \times \mathrm{M}$. Since $\tilde{\boldsymbol{y}} = (\boldsymbol{y}, \boldsymbol{m})$, we may write

$$\begin{aligned} \boldsymbol{y^t} &= g(\tilde{\boldsymbol{x}}^{\boldsymbol{t}} | \boldsymbol{\lambda}, T, I)_y, \\ \boldsymbol{m^t} &= g(\tilde{\boldsymbol{x}}^{\boldsymbol{t}} | \boldsymbol{\lambda}, T, I)_m. \end{aligned} \tag{1}$$

Assuming that the memory is initially set to all zeros, i.e. $\mathbf{m^0} = \mathbf{0}_{n_m}$, we retain the general structure and methodology as in GradTree that can be trained by gradient descent (or, to be more precise, in this case: Backpropagation-Through-Time, see (Werbos, 1990)).

Internal Decision Nodes Similar to classical DTs and GradTree, we currently employ hard, axis-aligned splits. However, ReMeDe Trees operate in the combined input-state space $\tilde{X} = X \times M$. This means that at each internal decision node, a routing decision through the tree may be either based on a component of the input vector, or a particular dimension of the hidden memory state. Hence the inference logic within the tree may explicitly depend on stored past information.

Memory Gating Gating techniques have been introduced in RNNs to deal with unstable gradient dynamics during training (Hochreiter, 1998). Therein, additional input- or state-dependent gates determine write-operations to the hidden state (either update with new information, or even deletion of old information (Schmidhuber et al., 1997)), as formalized in the previous section. Augmenting the memory access operation in ReMeDe Trees with gating mechanisms is quite straightforward and should - similar to their effect in RNNs - allow the model to deal better with longer dependencies over time. We use a very simple form of non-smooth, i.e. binary gating which aligns very well with the overall DT model structure, and leave studying more intricate mechanisms for future work. This gating mechanism will be introduced in the next paragraph.

Output Representation For ReMeDe Trees, each leaf node prescribes an output value, but also an update to the n_m-dimensional memory state $\boldsymbol{m^t}$. Classical DTs use a zero-order output representation, i.e. the output value is explicitly prescribed in the leaf nodes. For classification tasks, such as those considered in the experiments within this article, this is of course reasonable. Hence, the output will be calculated as

$$\boldsymbol{y^t} = g(\tilde{\boldsymbol{x}}^{\boldsymbol{t}} | \boldsymbol{\lambda}, T, I)_y = y_j, \tag{2}$$

where $\boldsymbol{y_j} \in Y$ denotes the constant output prescribed in the leaf node j that was selected by the tree inference. However, for continuous output values - such as the memory updates - other variants have been considered. For applications in time-series prediction, it is often recommended in practice to use a first order output, i.e.

$$\boldsymbol{y^t} = \boldsymbol{y^{t-1}} + g(\tilde{\boldsymbol{x}}^{\boldsymbol{t}} | \boldsymbol{\lambda}, T, I)_y, \tag{3}$$

to be able to deal with trends effectively. Other, more sophisticated, approaches utilize a parametrized mapping in each leaf node for static or dynamic problems, such as linear model trees (Czajkowski & Kretowski, 2016; Ammari et al., 2023) or fuzzy weighted linear models in the LoLiMoT algorithm (Nelles & Isermann, 1996). Exploring different output representation variants, especially for memory updates, is a promising direction for future research. For outputs, we use a zero order formulation and for the memory update an RNN-inspired parametrized equation:

$$\begin{aligned} \boldsymbol{m^t} &= \boldsymbol{m^{t-1}} + \lfloor \psi_g(c_j) \rceil \, \psi(W_j^x \boldsymbol{x^t}), \\ c_j, W_j^x &= g(\tilde{\boldsymbol{x}}^{\boldsymbol{t}} | \boldsymbol{\lambda}, T, I)_m, \end{aligned} \tag{4}$$

where j denotes the leaf node selected by tree inference, ψ is $\tanh$, $W_j^x \in \mathbb{R}^{n_m \times n_x}$ is a learnable weight matrix, c_j is a zero order output prescribed by leaf node j, $\psi_g : \mathbb{R} \to [0, 1]$ is a sigmoid function, and $\lfloor \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ maps its argument to the nearest integer, i.e. applied componentwise to $\psi_g(c_j)$, we have $\lfloor \psi_g(c_j) \rceil \in \{0, 1\}^{n_m}$, representing the hard gating mechanism for the hidden state update. Similar to the split decision in GradTree, this is achieved by rounding the sigmoid output of a gating parameter c_j and using the ST operator to ensure a reasonable gradient flow.

3 EVALUATION

In our evaluation, we provide a proof of concept that our formulation allows learning a recurrent memory decision tree architecture in an RNN-like fashion solely with BPTT.

Table 1: **PoC Results.** We report the average test accuracy along with the standard deviation on our proof-of-concept datasets, computed over five independent random trials.

	PoC 1	PoC 2	PoC 3	PoC 4	PoC 5
ReMeDe (ours)	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**
LSTM	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**
Random Guess	0.334 ± 0.001	0.333 ± 0.001	0.332 ± 0.002	0.333 ± 0.001	0.333 ± 0.001
Naïve Baseline	0.930 ± 0.002	0.930 ± 0.002	0.889 ± 0.001	0.889 ± 0.002	0.877 ± 0.003

3.1 Experimental Setup

Datasets We evaluate our method on 5 different synthetic datasets with increasing complexity that are designed in a way that they can only be solved with an internal memory, whenever a competing memory window-based model faced hard limits with respect to window size. The datasets comprise different versions of a sign retrieval task, where the sign, e.g., of an initial number, has to be recognized and should be output after some delay if a trigger value occurs. The datasets include single- and two-dimensional dataset versions and include fixed and dynamic delays. The dataset generation is summarized in detail in Appendix B. Specifically, we set the fixed delay to 5 and defined the variable delay within the range $[3, 7]$. For each task, we generated a total of 10,000 sequences. Continuous values were sampled from the uniform distribution $\mathcal{U}(-0.5, 0.5)$, and the delay was perturbed with a small random noise drawn from the normal distribution $\mathcal{N}(-0.01, 0.01)$.

Methods We evaluate two recurrent architecture on our datasets: LSTMs and ReMeDe Trees aiming to show the viability of our method. We omit comparison with NARX models, as it is clear that given a fixed lookback size of the model, our experiments can always be configured such that these models cannot learn the necessary temporal dependencies. Furthermore, we compare against two baselines, a simple random guess and a naive baseline making an informed guess (i.e., predicting the most probable value for each element in the sequence) based on the task.

Hyperparameters To select suitable hyperparameters for each task, we used Optuna (Akiba et al., 2019) with 60 trials. Specifically, we optimized only the learning rates, while keeping all other hyperparameters fixed. In particular, we selected a small tree depth of 6 and a hidden state size of only 5, to demonstrate that even with a compact model architecture, meaningful patterns can still be learned effectively. For LSTM, we selected a basic architecture with two hidden layers of 32 and 16 neurons and dropout. Similar to ReMeDe, we optimized the learning rate using Optuna.

3.2 Results

We can learn recurrent decision trees with backpropagation through time Our results confirm that recurrent decision trees can be effectively trained end-to-end using backpropagation through time. As shown in Table 1, ReMeDe trees achieve perfect test accuracy across all PoC datasets, matching the performance of LSTM baselines. This demonstrates that gradient-based optimization using the proposed method is a viable approach for learning decision trees with temporal dependencies, enabling both structured decision-making and sequence modeling within a single method.

ReMeDe Trees have a small tree size Table 2 presents the average tree size, measured in terms of the number of nodes, including both internal and leaf nodes, across our proof-of-concept datasets. The decision trees are pruned by removing all redundant paths, ensuring a more compact representation. The results indicate that the learned decision trees remain compact, with an average size of 26.0 nodes. Notably, the tree learned on PoC5 exhibits a considerably larger size, averaging 43.8 nodes, whereas the trees for the remaining tasks are of similar size, ranging between 20 and 23 nodes. This observation underscores the efficiency of the proposed method in capturing underlying dependencies while maintaining a moderate tree size. The com-

Table 2: **Average Tree Size.** We report the average tree size, measured in terms of the number of nodes.

	Number of Nodes
PoC 1	22.2
PoC 2	20.2
PoC 3	21.0
PoC 4	23.0
PoC 5	43.8
Mean	26.0

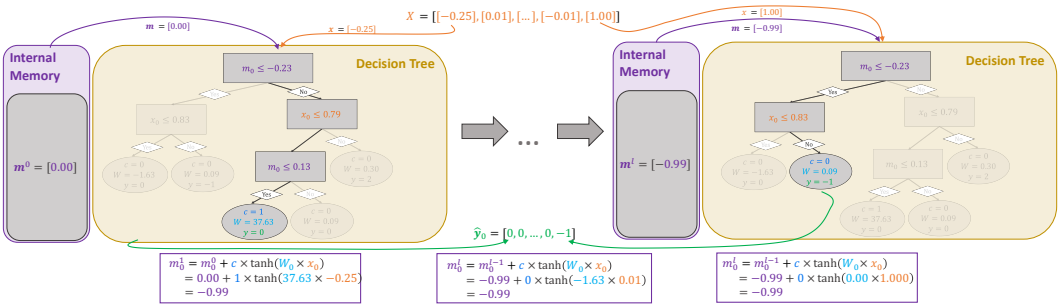

Figure 2: **ReMeDe Tree Update Visualization** This figure shows an ReMeDe tree trained to a sign recognition task. The task is to memorize the sign of $x \in (-0.5, 0.5)$ at the first position and predict it (-1 or 1) when a trigger value (1) appears, while intermediate positions hold zeros plus small noise.

pactness of ReMeDe Trees is particularly advantageous for interpretability on small datasets and enhances verifiability for more complex tasks.

ReMeDe Trees can effectively update and access the internal memory To illustrate how state updates operate in a compact ReMeDe tree, we present the example in Figure 2. The tree depicted in this figure was learned by our method in a simplified setting (using a tree of depth 4 with a single memory parameter) on PoC1. At the root node, the tree evaluates whether the hidden state is smaller than -0.23, effectively distinguishing whether the first entry in the sequence was negative (left branch) or positive (right branch). At the second level, the tree checks whether the trigger condition is met (> 0.5). If the trigger is activated, the tree makes the corresponding prediction for the sign. Otherwise, the hidden state may be updated. The update mechanism follows the left path in the diagram, where the hidden state is updated only if it falls within the interval $[-0.23, 0.13]$. This condition is typically satisfied only for the first element in the sequence, as subsequent updates are significantly amplified by a weight of 37.63. If the hidden state lies outside this interval, no update occurs, which corresponds to the delay phase. This example highlights how ReMeDe effectively captures and recalls sequential information, demonstrating its suitability for structured decision-making in temporal tasks.

4 Conclusion and Future Work

In this article, we have introduced a novel recurrent DT architecture using an internal hidden state, which is trained via Backpropagation-Through-Time to produce hard, axis-aligned recurrent DTs. The basic DT model used here is a standard, axis-aligned DT, or to be more precise, the GradTree model (Marton et al., 2024a). We have shown on synthetic test problems that our method is able to effectively compress past information into its hidden state to capture dependencies between inputs and outputs.

In the future, we would like to extend our method to more advanced base models, such as DTs with non-trivial output representations in leaf nodes and advanced memory gating techniques. Additionally, ReMeDe Trees can be readily introduced into tree ensembling approaches, such as GRANDE (Marton et al., 2024b). Combining the basic ReMeDe Tree model presented in this paper with the aforementioned extensions may hopefully show that recurrent DTs have the potential to yield competitive performance in time series learning tasks involving long-term dependencies, combining the advantages of recurrent models in time series tasks with the advantages of hard, axis-aligned DTs on tabular data.

References

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2623–2631, 2019.

Stephan Alaniz, Diego Marcos, Bernt Schiele, and Zeynep Akata. Learning decision trees recurrently through communication, 2021. URL https://arxiv.org/abs/1902.01780.

Bashar L Ammari, Emma S Johnson, Georgia Stinchfield, Taehun Kim, Michael Bynum, William E Hart, Joshua Pulsipher, and Carl D Laird. Linear model decision trees as surrogates in optimization of engineering applications. *Computers & Chemical Engineering*, 178:108347, 2023.

Leo Breiman. *Classification and regression trees*. Routledge, 2017.

Hossein Chegini and Caro Lucas. Prediction of financial time series with recurrent lolimot (locally linear model tree). In *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, volume 2, pp. 592–596, 2010.

Jianhui Chen, Hoang M Le, Peter Carr, Yisong Yue, and James J Little. Learning online smooth predictors for realtime camera planning using recurrent decision trees. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4688–4696, 2016.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794, 2016.

Marcin Czajkowski and Marek Kretowski. The role of decision tree representation in regression problems–an evolutionary perspective. *Applied soft computing*, 48:458–475, 2016.

Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02): 107–116, 1998.

Ozan Irsoy, Olcay Taner Yıldız, and Ethem Alpaydın. Soft decision trees. In *Proceedings of the 21st international conference on pattern recognition (ICPR2012)*, pp. 1819–1822. IEEE, 2012.

Haoran Luo, Fan Cheng, Heng Yu, and Yuqi Yi. Sdtr: Soft decision tree regressor for tabular data. *IEEE Access*, 9:55999–56011, 2021.

Sascha Marton, Stefan Lüdtke, Christian Bartelt, and Heiner Stuckenschmidt. Gradtree: Learning axis-aligned decision trees with gradient descent. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 14323–14331, 2024a.

Sascha Marton, Stefan Lüdtke, Christian Bartelt, and Heiner Stuckenschmidt. Grande: Gradient-based decision tree ensembles for tabular data. In *The Twelfth International Conference on Learning Representations*, 2024b.

Josue Nassar, Scott W Linderman, Monica Bugallo, and Il Memming Park. Tree-structured recurrent switching linear dynamical systems for multi-scale modeling. *arXiv preprint arXiv:1811.12386*, 2018.

Oliver Nelles. *Nonlinear dynamic system identification*. Springer, 2020.

Oliver Nelles and Rolf Isermann. Basis function networks for interpolation of local linear models. In *Proceedings of 35th IEEE conference on decision and control*, volume 1, pp. 470–475. IEEE, 1996.

Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. *Advances in neural information processing systems*, 31, 2018.

J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

Xinming Ren, Huaxi Gu, and Wenting Wei. Tree-rnn: Tree structural recurrent neural network for network traffic classification. *Expert Systems with Applications*, 167:114363, 2021.

Jürgen Schmidhuber, Sepp Hochreiter, et al. Long short-term memory. *Neural Comput*, 9(8):1735–1780, 1997.

Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley Osher, Yingyong Qi, and Jack Xin. Understanding straight-through estimator in training activation quantized neural nets, 2019. URL `https://arxiv.org/abs/1903.05662`.

A GRADTREE: GRADIENT-BASED DECISION TREES

This section introduces the core principles and notation of Gradient-Based Decision Trees (GradTree), which serve as the foundation for learning decision trees through gradient-based optimization. For a comprehensive overview, we refer to Marton et al. (2024a).

Traditional DTs rely on a hierarchical structure of nested decision rules. GradTree reformulates DTs into arithmetic functions based on addition and multiplication, enabling efficient gradient-based learning. Specifically, GradTree focuses on learning fully-grown (i.e., complete and balanced) decision trees, which can later be pruned if necessary. This means that every node has either zero or two successors and all nodes with zero successors have the same depth.

Such a tree of depth d can be expressed in terms of its parameters as:

$$\boldsymbol{y} = t(\boldsymbol{x}|\boldsymbol{\lambda}, \boldsymbol{\tau}, \boldsymbol{\iota}) = \sum_{l=0}^{2^d-1} \lambda_l \, \mathbb{L}(\boldsymbol{x}|l, \boldsymbol{\tau}, \boldsymbol{\iota}) \tag{5}$$

Here, $\mathbb{L}$ is an indicator function that determines whether a data point $\boldsymbol{x} \in \mathbb{R}^n$ reaches leaf node l, $\boldsymbol{\lambda} \in \mathcal{C}^{2^d}$ assigns class labels $y \in Y$ to each leaf, $\boldsymbol{\tau} \in \mathbb{R}^{2^d-1}$ contains the split thresholds, and $\boldsymbol{\iota} \in \mathbb{N}^{2^d-1}$ specifies the feature index for each internal node. The output space Y may be a set of discrete class labels, in which $Y \subset \mathbb{N}^{n_y}$, or some continuous space $Y \subset \mathbb{R}^{n_y}$ for application to regression problems.

To enable gradient-based optimization and efficient computation using matrix operations, GradTree introduces a dense representation of decision trees. The traditional feature index vector $\boldsymbol{\iota}$ is expanded into a one-hot encoded matrix $\boldsymbol{I} \in \mathbb{R}^{(2^d-1)\times n}$, and the split thresholds are represented as a matrix $\boldsymbol{T} \in \mathbb{R}^{(2^d-1)\times n}$, allowing individual thresholds for each feature. With internal nodes ordered in a breadth-first manner, the tree function can be reformulated as:

$$g(\boldsymbol{x}|\boldsymbol{\lambda}, T, I) = \sum_{l=0}^{2^d-1} \lambda_l \, \mathbb{L}(\boldsymbol{x}|l, \boldsymbol{T}, \boldsymbol{I}) \tag{6}$$

The indicator function $\mathbb{L}$ for a leaf node l is defined as:

$$\mathbb{L}(\boldsymbol{x}|l, \boldsymbol{T}, \boldsymbol{I}) = \prod_{j=1}^{d} \left(1 - \mathfrak{p}(l,j)\right) \mathbb{S}(\boldsymbol{x}|\boldsymbol{I}_{\mathfrak{i}(l,j)}, \boldsymbol{T}_{\mathfrak{i}(l,j)}) + \mathfrak{p}(l,j) \left(1 - \mathbb{S}(\boldsymbol{x}|\boldsymbol{I}_{\mathfrak{i}(l,j)}, \boldsymbol{T}_{\mathfrak{i}(l,j)})\right) \tag{7}$$

In this formulation, $\mathfrak{i}(l,j)$ denotes the internal node on the path to leaf l at depth j, and $\mathfrak{p}(l,j)$ indicates whether the path follows the left ($\mathfrak{p} = 0$) or right ($\mathfrak{p} = 1$) child node.

Traditional DTs use the non-differentiable Heaviside step function for splits, which impedes gradient-based learning. GradTree replaces this with a smooth approximation using the logistic sigmoid function:

$$\mathbb{S}(\boldsymbol{x}|\boldsymbol{\iota}, \boldsymbol{\tau}) = \lfloor S\left(\boldsymbol{\iota} \cdot \boldsymbol{x} - \boldsymbol{\iota} \cdot \boldsymbol{\tau}\right) \rceil \tag{8}$$

where $S(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function, $\lfloor z \rceil$ rounds z to the nearest integer, and $\boldsymbol{\iota} \cdot \boldsymbol{x}$ denotes the dot product. To maintain axis-aligned splits, $\boldsymbol{\iota}$ is enforced as a one-hot encoded vector using a hardmax transformation.

Since both rounding and hardmax operations are non-differentiable, GradTree utilizes the straight-through (ST) estimator (Yin et al., 2019) during backpropagation. This allows non-differentiable

operations in the forward pass while enabling gradient flow in the backward pass. In contrast to many approaches to learn soft, differentiable DTs, e.g. (Irsoy et al., 2012; Luo et al., 2021), GradTrees are structurally (and also w.r.t. the inference process) equivalent to classical DTs without necessity for any postprocessing, which may degrade the performance of the final DT model.

B PROOF OF CONCEPT - SYNTHETIC DATA GENERATION PROCEDURES

This subsection introduces five synthetic data generation procedures designed to model temporal dependencies and delayed response behaviors in time series data. Each method simulates distinct patterns, including delayed reactions and memory effects, across one- and two-dimensional input spaces. The following subsections describe each method in detail with corresponding mathematical formalizations.

1. Delayed Sign Retrieval (Single-Dimensional, Fixed Delay) The first procedure generates a single-dimensional time series where the task is to recover the sign of the initial input after a fixed delay. A trigger signal appears at a specific timestep, prompting the output to reflect the sign of the initial value, while the output remains zero at all other timesteps. Let $x_0 \sim \mathcal{U}(-v, v)$ be the initial value, and d denote the fixed delay. The input sequence $\mathbf{x} \in \mathbb{R}^{d+2}$ and the target output $\mathbf{y} \in \mathbb{R}^{d+2}$ are defined as:

$$\mathbf{x} = [x_0, 0, 0, \ldots, 0, t], \tag{9}$$

$$\mathbf{y} = [0, 0, \ldots, 0, \text{sign}(x_0)], \tag{10}$$

where t is the trigger value and the sign function is defined as:

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ -1 & \text{if } x < 0. \end{cases} \tag{11}$$

2. Delayed Sign Retrieval (Two-Dimensional, Fixed Delay) This method extends the previous setup to a two-dimensional input. The first channel contains the initial value, and the second channel receives the trigger after a fixed delay. The model must output the sign of the first input upon the appearance of the trigger. Let $x_0 \sim \mathcal{U}(-v, v)$ and d be the fixed delay. The input matrix $\mathbf{X} \in \mathbb{R}^{(d+2)\times 2}$ and the output $\mathbf{y} \in \mathbb{R}^{d+2}$ are defined as:

$$\mathbf{X} = \begin{bmatrix} x_0 & 0 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & t \end{bmatrix}, \quad \mathbf{y} = [0, 0, \ldots, 0, \text{sign}(x_0)]. \tag{12}$$

3. Delayed Sign Retrieval (Single-Dimensional, Variable Delay) This variant introduces a variable delay, randomly sampled from a uniform range $[d_{\text{min}}, d_{\text{max}}]$. The trigger appears at a random timestep, requiring the model to output the sign of the initial value. Let $\delta \sim \mathcal{U}(d_{\text{min}}, d_{\text{max}})$ and $x_0 \sim \mathcal{U}(-v, v)$. The input $\mathbf{x}$ and output $\mathbf{y}$ are defined as:

$$\mathbf{x} = [x_0, 0, \ldots, 0, t, 0, \ldots], \tag{13}$$

$$\mathbf{y} = [0, \ldots, 0, \text{sign}(x_0), 0, \ldots], \tag{14}$$

where the trigger t appears at timestep δ.

4. Delayed Sign Retrieval (Two-Dimensional, Variable Delay) This method generalizes the two-dimensional fixed delay scenario by allowing the trigger to appear at a randomly chosen

timestep within a predefined delay range. Let $\delta \sim \mathcal{U}(d_{\text{min}}, d_{\text{max}})$. The input matrix $\mathbf{X}$ and output $\mathbf{y}$ are:

$$\mathbf{X} = \begin{bmatrix} x_0 & 0 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & t \\ \vdots & 0 \end{bmatrix}, \quad \mathbf{y} = [0, \dots, 0, \text{sign}(x_0), 0, \dots]. \tag{15}$$

5. Sign Memory Task The final procedure generates sequences composed of alternating blocks of -1 and 1, interspersed with zero-valued delay blocks. The task is to reproduce the last non-zero block upon encountering a new non-zero block. Let $b_j \in \{-1, 1\}$ denote the j-th non-zero block of length l, and z represent a zero block of length d. The input $\mathbf{x}$ and target output $\mathbf{y}$ are constructed as:

$$\mathbf{x} = [b_1, z, b_2, z, \dots, b_n], \tag{16}$$

$$\mathbf{y} = [0, z, b_1, z, b_2, \dots]. \tag{17}$$

Each non-zero block b_j is randomly selected from $\{-1, 1\}$, and the model must recall and reproduce the previous block at the appropriate timestep.

C Related Work

Classical DT learning algorithms, such as C4.5 (Quinlan, 2014) or CART (Breiman, 2017), are based on growing a DT by greedily splitting the input space in a componentwise fashion to optimize the reduction in the chosen error metric at each step of building the tree. No method has been yet proposed to incorporate updates of an internal memory state based on these algorithms.

Nevertheless, the idea of using explicit time-dependency in the DT framework is not new. (Chen et al., 2016) propose a model which they call recurrent DT, for camera planning. In contrast to ReMeDe, no internal memory state is used but previous outputs are fed back into the model as inputs, which renders this approach a special case of NARX models in the terminology used here. The same holds for (Chegini & Lucas, 2010), who extend the LoLiMoT algorithm (Nelles & Isermann, 1996) to include output feedback for financial time series prediction. (Alaniz et al., 2021) propose an intricate scheme to learn a recurrent model, involving a DT, but also a combination between an LSTM and an Attribute-Learning System, where a DT uses the hidden state of an LSTM.

Others have taken the converse route and combine classical DT with recurrent models in leaf nodes, such as (Ren et al., 2021). Therein, first the input data is split using classical DT algorithms and then separate RNNs are trained for each leaf node, inheriting the potential suboptimality of the former. Also worth mentioning is a family of approaches that uses hierarchical, tree structured switching linear systems for dynamics modeling, such as (Nassar et al., 2018), which share some structural similarities with ReMeDe Trees, although the resulting models are quite different. In particular, the hidden state used there is discrete and some of the involved operations are soft, i.e. stochastic. In contrast, a ReMeDe Tree consists only of a single hard, axis-aligned DT which performs read and write operations on its own hidden memory state, enabled by training the complete model via gradient descent. To the best of our knowledge, no other recurrent DT using continuous hidden state feedback has been proposed yet.