**POLITECNICO DI TORINO**

**UNIVERSIDAD POLITECNICA DE CATALUNYA - FIB**

Department of Control and Computer Engineering

*Collegio di Ingegneria Informatica, del Cinema e Meccatronica*

*Master's degree programme*

*INGEGNERIA INFORMATICA (COMPUTER ENGINEERING)*

*LM-32(DM270)*


# *High-Performance Computing Techniques for Efficient Training and Inference of AI Models*

**Supervisors:**

*Professor Stefano Scanzio*

*Professor Josep Ramon Herrero Zaragoza*

*Professor Gianluca Cena*

*Professor Gabriele Formis*

**Candidate:**

*Mattia Gumina S323182*

*July 2025*

Academic year 2024-2025

Questo lavoro rappresenta il culmine di un percorso lungo e intenso, costellato di gioie e soddisfazioni, ma anche di ostacoli e momenti difficili. Non ho mai affrontato questo cammino da solo: al mio fianco ci sono sempre state tante persone che hanno reso il tragitto più leggero, che mi hanno aiutato a ritrovare la strada quando rischiavo di smarrirmi, e che non mi hanno mai fatto sentire solo. A tutti voi, grazie di cuore per aver reso questo percorso molto più bello.

Il mio primo pensiero va alla mia famiglia. A mamma e papà, che, pur non comprendendo sempre fino in fondo le mie scelte o ciò che stavo facendo, mi hanno sempre sostenuto, facendomi sentire il loro amore incondizionato e sopportando anche i miei numerosi, soprattutto ultimamente, momenti di crisi. A mio fratello Kevin, spirito guida e punto di riferimento da sempre: sei tu che, camminando davanti a me, mi hai aperto la strada, indicato la direzione e lasciato impronte da seguire. A mia sorella Katia, che, grazie alle nostre lunghissime videochiamate in cui ci raccontavamo ogni dettaglio delle nostre giornate, ha saputo rallegrare e colorare le giornate più grigie e buie di Torino, facendomi sentire un po' meno la sua mancanza. A mia zia Santina, sempre presente e disponibile in questi anni, che ha fatto di tutto per esaudire ogni mia richiesta e desiderio, senza farmi mai mancare la sua vicinanza e il suo affetto.

Dopo la famiglia, il pensiero non può che andare agli amici di "giù", dal gruppo Le Sapone's ai compagni del liceo, e in particolare a Fabri, Bizio e Basilio, per avermi fatto dimenticare tutto ciò che riguardava la "L-word" ogni volta che tornavo a casa, costringendomi a lasciare ansie e stress dall'altro lato dello Stretto e facendomi riscoprire la serenità e la spensieratezza della Sicilia, come se non me ne fossi mai andato.

Non posso poi non dedicare un pensiero speciale a tutte le persone conosciute durante questo percorso, tra Torino e Barcellona.

Al "socio" Simone, che è stato al mio fianco, letteralmente, fin dal primo giorno: ci siamo aiutati e supportati a vicenda, condividendo tantissime esperienze. Sei stato fondamentale per il mio cammino e so che lo sarai anche in futuro. Al "bomber" Patrick, anche tu arrivato all'inizio del percorso e mai più andato via: grazie per tutto il supporto e i tantissimi momenti belli vissuti insieme. Il trio è per sempre.

All'A4-2 della residenza Borsellino e a tutte le persone conosciute lì: grazie per avermi fatto sentire a casa. Agli "informatici" del Corso 2, con cui ho condiviso il percorso: grazie per le giornate di studio, le serate passate insieme tra Santa Giulia, le panche, il collegio e le varie case (e non le lezioni, visto che spesso a quelle non venivate). Avete reso più piacevole il tempo trascorso a Torino.

A Casa RGP e a tutte le persone conosciute a Barcellona nell'ultimo anno: grazie per avermi fatto vivere l'esperienza più bella della mia vita e per tutti i bei momenti trascorsi insieme, tra viaggi, aperitivi e karaoke.

Infine, un sentito ringraziamento ai miei due relatori, il Prof. Jose Ramon Herrero Zaragoza e il Prof. Stefano Scanzio, per avermi seguito con attenzione e per aver sempre incoraggiato le mie idee, offrendomi preziose indicazioni e suggerimenti.

Grazie a tutti, di cuore.

**Abstract**

*This thesis investigates the computational efficiency and scalability of biologically inspired neural network architectures—Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) models—under varying hardware and training configurations. While traditional sequence models such as RNNs, LSTMs, and Transformers have demonstrated strong performance in temporal tasks, they often suffer from high computational costs and limited scalability. LTC and CfC networks have emerged as promising alternatives due to their dynamic temporal modeling capabilities and lower parameter counts, but their real-world efficiency in training remains underexplored.*

*To evaluate the training behavior and parallelization potential of these models, experiments were conducted on four datasets of increasing complexity: a synthetic sine-cosine signal, the Human Activity Recognition (HAR) dataset, the Metro Interstate Traffic Volume, and the Individual Household Electric Power Consumption. Both LTC and CfC were trained under various conditions, including single-CPU, single-GPU, and distributed multi-GPU environments using PyTorch's Distributed Data Parallel (DDP) framework. Metrics such as training time, accuracy, loss, GPU utilization, and memory consumption were systematically collected and analyzed.*

*The results demonstrate that CfC models consistently train faster than LTC models across all scenarios, achieving comparable or higher accuracy, particularly in larger datasets. Moreover, distributed training significantly reduces training time for both models, with optimal gains observed at two GPUs, beyond which communication overheads begin to affect scalability. These findings highlight the practical trade-offs between architectural complexity and parallel efficiency, and support the use of biologically inspired models in high-performance AI pipelines.*

***Keywords***

*Liquid Neural Networks, Liquid Time-Constant (LTC), Closed-form Continuous-time (CfC), Distributed Training*

# Contents

# List of Figures

# List of Tables

# 1. Introduction

The rapid growth of deep learning has led to increasingly complex models that often demand significant computational resources, particularly in tasks involving sequential and temporal data. While new neural architectures continue to push the boundaries of accuracy and generalization, they also raise questions about scalability, efficiency, and deployability. This chapter introduces the core challenges motivating the thesis, formulates the problem being addressed, and outlines the motivation, objectives and structure of the work.

## 1.1 Problem statement

Recent advances in deep learning have led to increasingly complex neural architectures, particularly in domains involving sequential and temporal data. Models like classical RNNs, LSTMs [1], GRUs [2] and Transformers [3] have achieved high accuracy, but at the cost of substantial computational demands, both in terms of training time and memory usage. This poses a significant challenge when developing or scaling such models in real-world settings, especially when real-time performance or large datasets are involved.

As an alternative to these classical approaches, the field of biologically inspired architectures has introduced a new class of models known as *Liquid Neural Networks* (LNNs). LNNs are designed to adapt their internal dynamics over time, modelling information in a continuous-time framework. These networks aim to better handle streaming and asynchronous data, and have been shown to outperform traditional discrete-time models in several temporal tasks. LNNs (Figure 1) expresses their capacity to excel in scenarios requiring memory, context, and complex temporal reasoning [4].

Among LNNs, two notable architectures have emerged: the **Liquid Time-Constant (LTC)** network [5] and the **Closed-form Continuous-time (CfC)** network [6]. LTC simulates continuous-time behaviour using trainable ODE solvers, while CfC approximates neuron state evolution using analytical closed-form solutions.

However, despite their architectural advantages, LTC networks still require substantial computational resources during training, particularly on longer sequences or larger datasets. LTC models rely on ODE solvers that must update the internal dynamics at every time-step, resulting in high computational cost per sample. Therefore, the efficiency of training these models is highly dependent on how well parallel computing techniques are applied.

Moreover, while general-purpose parallelization strategies (e.g., data parallelism or model parallelism using multiple GPUs [7]) are well established in deep learning, their application and effectiveness in the context of models such as LTC and CfC remain under-explored. Understanding how these models behave under distributed training and whether they benefit

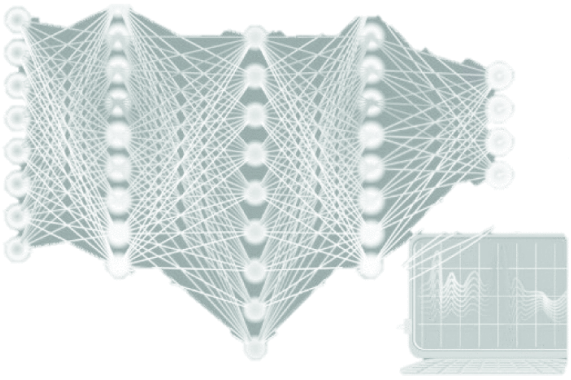


Figure 1: Liquid Neural Network [4]

from GPU-level optimization or require model-specific parallelization strategies is essential for making them scalable and practical in high-performance environments.

This thesis addresses the gap between the theoretical efficiency of these models and their practical computational behaviour by applying and evaluating high performance computing techniques, specifically `DistributedDataParallel` (DDP) [8], to train and optimize LTC and CfC architectures on multiple GPUs. The goal is to assess performance scaling, identify potential bottlenecks, and explore optimizations for efficient training and inference.

## 1.2 Motivation for optimizing LTC and CfC models

The growing scale of AI workloads, especially in temporal domains, makes training efficiency a critical concern in both research and real-world applications. Applications such as time-series forecasting or behaviour recognition demand not only accuracy but also fast, scalable models that can be trained on modern hardware.

To address these challenges, more compact and biologically inspired models such as the Liquid Time-Constant (LTC) network [5] and the Closed-form Continuous-time (CfC) network [6] have been proposed. These models offer strong temporal modelling capabilities with fewer parameters. For example, the LTC model in Hasani et al. [5] achieves competitive accuracy on temporal tasks using few parameters, while CfC achieves similar results with even fewer, and demonstrates faster training times [6].

Recent studies confirm that both LTC and CfC architectures outperform classical RNNs, LSTMs, and ODE-based models in temporal modeling tasks. For example, in the Walker2D benchmark task, which involves predicting the physical state of a simulated agent at irregular time intervals, the LTC model achieved a lower square error than all traditional baselines. CfC models further improved on these results, achieving both a lower prediction error and a faster training time per epoch. Table 1 summarizes these results from the original CfC paper [6], showing that CfC variants outperform even optimized LSTM and Transformer architectures in both accuracy and efficiency.

Table 1: Performance comparison on the Walker2D task (adapted from [6]). Lower square error and lower time per epoch are better.

| **Model** | **Square Error (↓)** | **Time per Epoch (min)** |
|---|---|---|
| ODE-RNN | 1.904 ± 0.061 | 0.79 |
| CT-RNN | 1.198 ± 0.004 | 0.91 |
| GRU-ODE | 1.051 ± 0.018 | 0.56 |
| CT-LSTM | 1.014 ± 0.014 | 0.31 |
| ODE-LSTM | 0.883 ± 0.011 | 0.56 |
| Transformer | 0.761 ± 0.032 | 0.80 |
| LTC | **0.662** ± 0.013 | 0.78 |
| CfC | **0.643** ± 0.006 | 0.08 |

Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) networks offer an alternative to traditional sequence models by modeling time as a continuous variable, allowing them to process sparse or asynchronous data more naturally. These models are particularly promising in domains where temporal precision, data sparsity, and biological plausibility are desired. Nonetheless, their real-world applicability depends not only on their theoretical elegance or accuracy but also on how efficiently they can be trained and deployed on modern hardware.

In practice, these models introduce unique computational patterns. LTC models, for example, require numerical ODE integration at each time-step, which creates challenges for GPU

utilization. CfC models improve efficiency using closed-form temporal updates, but their sequential structure and element-wise operations may still limit scalability on parallel architectures. This creates a compelling motivation to investigate whether general-purpose high-performance computing (HPC) techniques, such as multi-GPU training with DistributedDataParallel (DDP), can bridge the gap between model compactness and training efficiency.

By understanding and optimizing the training performance of LTC and CfC models, this work aims to support their adoption in scalable real-world AI systems and to provide insights into how modern HPC tools interact with biologically inspired neural architectures.

## 1.3 Objectives of the thesis

The main objective of this thesis is to evaluate and improve the computational efficiency of Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) neural networks by applying high-performance computing (HPC) techniques. This includes exploring how these biologically inspired models can be effectively parallelized for training on modern multi-GPU systems, and understanding the computational trade-offs involved in doing so. The work is guided by the following specific sub-objectives:

1. **To evaluate the training performance of LTC and CfC models across diverse datasets.**

   This includes conducting experiments on four distinct datasets: a synthetic sine-cosine sequence (to validate learning behaviour), the Human Activity Recognition (HAR) dataset [9] (for multivariate classification), the Metro Interstate Traffic dataset [10] (for time-series regression), and the Individual Household Electric Power Consumption (again for time-series regression). Each task is designed to reveal how model architecture interacts with data complexity and task requirements.

2. **To implement and validate multi-GPU training using PyTorch Distributed Data Parallel (DDP) [8].**

   Multi-GPU training is critical for accelerating model development at scale. The thesis includes the integration of DDP into both LTC and CfC pipelines, running synchronized training experiments on 2 and 4 GPUs. This helps evaluate DDP's effectiveness in handling different model structures, especially those with non-standard computational flows.

3. **To profile and analyse computational bottlenecks on CPU and GPU.**

   Using tools such as `cProfile` and `torch.profiler`, the thesis investigates where time and memory are spent during training. Key areas include the ODE solver loops in LTC and the sequential gating operations in CfC. Profiling allows for a deeper understanding of which operations scale well and which may hinder parallel execution.

4. **To examine how batch size, sequence length and learning rate affect training efficiency and model accuracy.**

   By systematically varying batch size and observing training time, loss convergence, and final accuracy, the thesis aims to understand the trade-off between computational throughput and model performance. These insights are essential to optimize training schedules and resource usage.

5. **To assess the overall scalability and practicality of LTC and CfC in high-performance computing environments.**

   Finally, the thesis provides a comprehensive assessment of how well these models scale in real-world HPC settings. This includes calculating speedup, scaling efficiency, and identifying limits where additional GPUs no longer provide significant gains. These findings contribute to future guidelines for deploying Liquid Neural Networks at scale.

## 1.4 Structure of the document

This thesis is organized into seven chapters, each contributing to a comprehensive understanding of efficient training and inference for LTC and CfC models using high-performance computing techniques.

- **Chapter 1 - Introduction** Introduces the problem of scaling temporal deep learning models, outlines the motivation, and defines the thesis objectives.
- **Chapter 2 - State of the Art** Reviews existing literature on temporal neural networks, including LTC and CfC models, and discusses current approaches to distributed training and model optimization using HPC tools such as DDP, Horovod, and DeepSpeed.
- **Chapter 3 - Methodology** Describes the implementation of the models, datasets used, preprocessing steps, and the experimental setup including hardware and training configurations.
- **Chapter 4 - Experimental Design** Details the experimental protocol, including how batch sizes, devices, and model configurations were varied. Defines the metrics used to evaluate performance.
- **Chapter 5 - Results and Analysis** Presents the outcomes of all training runs on CPU, single GPU, and multiple GPUs. Includes accuracy, loss trends, training time comparisons, and profiling insights.
- **Chapter 6 - Discussion** Interprets the results, reflects on the limitations, and discusses the broader implications of the findings in terms of model architecture and parallelization strategy.
- **Chapter 7: Sustainability and Ethical Implications** Discusses the environmental and social dimensions of the work, including energy consumption in distributed training, responsible use of computing resources, and the ethical considerations of applying temporal neural networks in real-world scenarios.
- **Chapter 8 - Conclusions and Future Work** Summarizes key takeaways, highliths the contributions of the thesis, and outlines possible directions for extending this work.
- **References and Appendices** Lists all cited works and provides supplementary material, including code snippets, training logs, and additional figures.

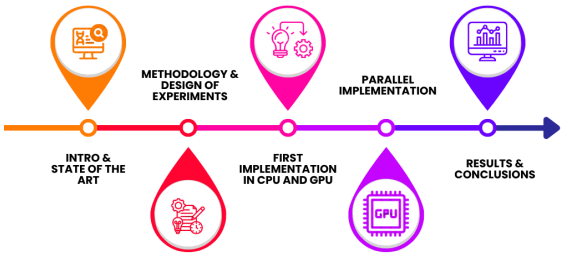



Figure 2: Structure of the document

# 2. State of the art

In recent years, the field of deep learning has witnessed significant advancements in models capable of handling temporal and sequential data. These models are essential for a wide range of applications, including time-series forecasting, natural language processing, human activity recognition, and autonomous control systems. The core challenge in these tasks lies in capturing dependencies over time, both short-term and long-term, while maintaining computational efficiency and model scalability.

This chapter presents a comprehensive overview of the state-of-the-art in neural models for temporal data. It starts by reviewing the foundational architectures that have shaped sequence modeling, including Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), Gated Recurrent Units (GRUs), and Transformer models. Then are presented evolutions of these networks such as continuous-time models and ODE based models. Finally, at the end, it is introduced the emerging family of *Liquid Neural Networks* (LNNs), which take inspiration from continuous-time systems and biological processes. These models offer new opportunities for efficient and adaptive sequence modeling, particularly in settings where time is irregular or sparsity is a concern.

The chapter also explores the computational challenges associated with training such models, particularly the Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) architectures. Special emphasis is placed on parallelization strategies, including data and model parallelism, as well as the use of PyTorch Distributed Data Parallel (DDP) and other HPC-oriented frameworks. The goal is to provide the necessary background for understanding both the modeling techniques and the high-performance computing tools employed in the thesis.

## 2.1 Background on Neural Models for Temporal Data

Modeling temporal data is a central problem in machine learning and artificial intelligence. Unlike static inputs such as images or tabular data, temporal sequences are characterized by their ordering in time, interdependence between elements, and potentially varying time intervals between observations. Examples of such data include sensor readings, speech signals, financial time series, and motion trajectories.

The core challenge in learning from temporal data lies in capturing the temporal dependencies between observations. In many real-world scenarios, the current output depends not only on the most recent input, but also on long-term historical context. This makes temporal modeling fundamentally different from traditional supervised learning tasks.

Early approaches to sequence modeling, such as autoregressive models and sliding window techniques, attempted to transform temporal problems into fixed-size input-output mappings. Although simple, these methods struggled to generalize across sequence lengths and failed to capture long-range dependencies.

To address these limitations, neural network architectures were adapted to incorporate memory and state. Recurrent Neural Networks (RNNs) were among the first to introduce a mechanism for maintaining internal state across time-steps, allowing information to persist through the sequence. Over time, more advanced variants such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) were developed to better manage memory and mitigate issues like vanishing gradients.

More recently, attention-based models like Transformers have shown remarkable success in handling sequence data, particularly in natural language processing. These models replace recurrence with self-attention mechanisms that allow for direct modeling of long-range dependencies.

However, while these architectures have achieved state-of-the-art performance, they come with substantial computational costs, especially when applied to long sequences or real-time systems. Additionally, they operate in discrete time, assuming fixed intervals between inputs, which limits their effectiveness in asynchronous or irregularly sampled environments.

In response to these limitations, a new class of models, Liquid Neural Networks, has emerged. These models treat time as a continuous variable and update internal states based on ordinary differential equations (ODEs) or closed-form functions. This continuous-time formulation enables them to adapt more naturally to time-varying inputs, operate efficiently on sparse or irregular data, and better mimic the dynamics of real biological systems.

The following sections explore these architectures in greater detail, beginning with classical sequence models and culminating in the biologically inspired liquid neural networks that form the focus of this thesis.

## 2.2 Classical Temporal Models

Understanding and modeling temporal data - time series, language, audio, video, sensor streams - is a central challenge in deep learning. Temporal data is unique in that observations are not independent but carry chronological dependencies: what happens *now* is often influenced by *what happened before*. Over time, a rich ecosystem of neural architectures has emerged to capture these dependencies with increasing sophistication.

At the foundation lies the **Recurrent Neural Network (RNN)**. Unlike feed-forward models, an RNN processes input step by step and maintains a hidden state that "remembers" past inputs, allowing the network to carry information across time steps. This makes RNNs well-suited for sequential tasks like speech recognition or language modeling, but traditional RNNs suffer from vanishing or exploding gradients, limiting their ability to capture long-term dependencies. [11]

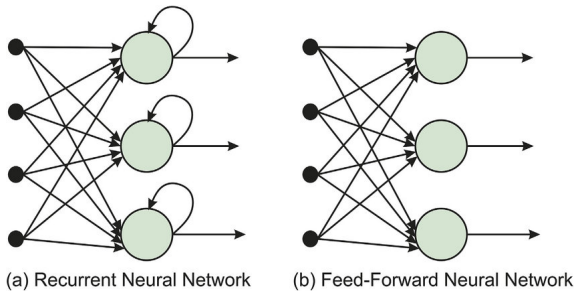



Figure 3: Comparison between Recurrent Neural Network and Feed Forward Neural Network [12]

To overcome these limitations, gated architectures were introduced:

- **LSTM (Long Short-Term Memory)** networks include special forget, input and output gates along with a "cell" state, carefully designed to regulate how much past information is retained or discarded. This enables LSTMs to model long-range dependencies, crucial tasks for translation, speech, and time-series forecasting. [13]
- **GRU (Gated Recurrent Unit)** is a streamlined alternative that merges some gates, offering comparable performance with fewer parameters and computational cost, a favoured option when efficiency matters. [14]

More recently, **Transformers** re-imagined sequence modeling by doing away with recurrence entirely. Instead, they compute self-attention to relate all positions in a sequence at once, enabling massive parallelism and capturing long term dependencies more directly. Transformers have since become the standard for language modeling and are making inroads into many other

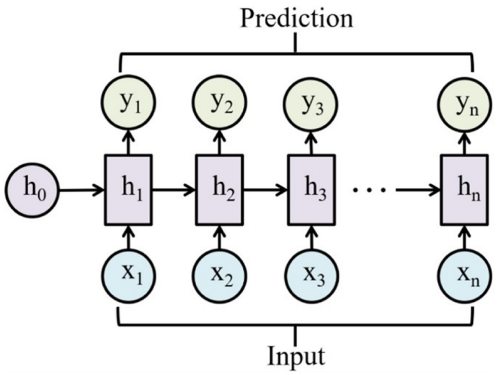



Figure 4: Basic RNN architecture [17]

temporal domains. [15]

Now, the next subsections analyse these models one by one in more depth, explaining how they work, their advantages, and disadvantages.

### 2.2.1 Recurrent Neural Networks (RNNs)

"Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence." [16] Unlike humans, traditional neural networks lack a mechanism for retaining contextual memory over time. Standard feed-forward networks process each input independently, without considering preceding elements in a sequence, and this limits their ability to model tasks where temporal coherence and historical context are essential. Recurrent Neural Networks (RNNs) were developed to overcome this limitation.

Recurrent Neural Networks (RNNs) are a specialized class of neural architectures designed to model patterns within sequential data. The basic architecture consists of an input layer, a hidden layer, and an output layer (Figure 4) [17]. Unlike traditional Feed-forward Neural Networks which process input in a strictly acyclic, one-dimensional manner, RNNs incorporate cyclical connections within their computational graph (Figure 5).

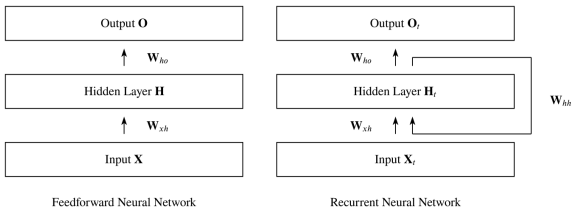



Figure 5: Visualization of differences between FFNs and RNNs [18]

This recurrence allows the network to maintain an internal memory of past inputs, effectively enabling it to consider not only the current input at time step $t$, but also contextual information from previous inputs $x_{0:t-1}$. This dynamic memory mechanism makes RNNs particularly well-suited for tasks where the order and temporal dependencies of the data are critical [18].

The mechanism by which Recurrent Neural Networks transmit information from previous time steps can be formally described using mathematical notation. Let $X_t \in \mathbb{R}^{n \times h}$ denote the corresponding hidden state, where $n$ is the number of hidden units. The transformation from input to hidden state involves a set of learnable parameters: the input-to-hidden weight matrix $W_{xh} \in \mathbb{R}^{d \times h}$, the hidden-to-hidden recurrent weight matrix $W_{hh} \in \mathbb{R}^{h \times h}$, and a bias vector $b_h \in \mathbb{R}^{1 \times h}$. The resulting hidden state is computed through a non-linear activation function,

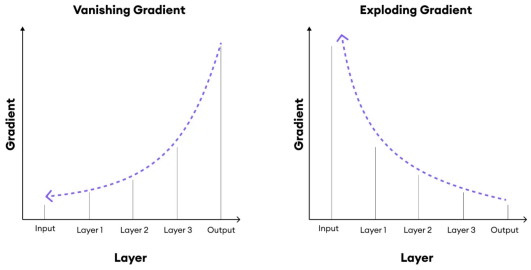



Figure 6: Vanishing and Exploding Gradient [19]

typically the hyperbolic tangent (tanh) or logistic sigmoid, to ensure gradient stability during back-propagation. This update rule is expressed as:

$$H_t = \sigma(X_t W_{xh} + H_{t-1} W_{hh} + b_h) \tag{1}$$

where $\sigma(\cdot)$ denotes the activation function. The output at time $t$ is computed as:

$$O_t = \phi(H_t W_{ho} + b_o) \tag{2}$$

where $W_{ho} \in \mathbb{R}^{h \times o}$ maps the hidden state to the output layer, $b_o$ is a bias term, and $\phi(\cdot)$ is typically a softmax or linear activation function, depending on the task. A key property of this formulation is that each hidden state $H_t$ recursively incorporates information from $H_{t-1}$, which in turn includes $H_{t-2}$, and so on. This recursive dependency allows RNNs to maintain a form of memory over arbitrary-length input sequences.

In contrast, Feed-forward Neural Networks compute their hidden states in a non-recurrent, one-pass manner. The equivalent transformations for a feed-forward network are:

$$H = \sigma(X W_{xh} + b_h) \tag{3}$$

$$O = \phi(H W_{ho} + b_o) \tag{4}$$

These expressions highlight the fundamental difference between the two architectures: RNNs carry information forward through time via recurrent connections, whereas feed-forward networks lack temporal dynamics and operate solely on static input [18].

RNNs are particularly well suited for applications involving structured sequences such as natural language, genomic data, handwritten text, speech signals, and time-series information generated by sensors, financial markets, or public systems. Unlike conventional neural networks, RNNs incorporate a temporal dimension, allowing them to capture the ordering and dependencies within sequences, making them inherently capable of reasoning over time-evolving data.

Training Recurrent Neural Networks involves learning the optimal parameters that minimize prediction error across sequences. This is typically achieved using gradient-based optimization techniques such as Stochastic Gradient Descent (SGD) or its variants. A crucial aspect of RNN training is the use of Back-propagation Through Time (BPTT), an extension of standard back-propagation that unfolds the recurrent network across time steps, allowing gradients to be computed for each parameter at every time step [17]. However, this procedure can lead to two critical numerical issues: the vanishing and exploding gradient problems (Figure 6). In the vanishing-gradient scenario, the gradients diminish exponentially as they traverse earlier time steps, ultimately becoming too small to meaningfully influence weight updates. As a result, the network has difficulty learning long-range dependencies, and although convergence may still occur, it is typically very slow and inefficient. In contrast, in the case of an exploding gradient, the gradients grow excessively large, which can cause instability in the optimization process. This often results in erratic weight updates, numerical overflow, and a failure to converge to a meaningful solution or global minimum. Both phenomena hinder the effective training of deep or

long-sequence RNNs and motivate the use of specialized architectures and mitigation strategies [20].

Another problem in RNNs is their limited ability to remember inputs from the distant past and use them effectively to influence future outputs, as observed in Hochreiter's 1991 thesis [21]. Standard RNN architectures tend to assign diminishing importance to earlier inputs as sequences progress. This leads to situations where inputs that are critical for predicting later outputs are effectively "forgotten" by the network, especially when the temporal gap between cause and effect is large. Even when the correct input-output relationship exists, the network often fails to detect and reinforce it during training, since the internal representations at later time steps retain little trace of early relevant information. This memory decay severely restricts the network's ability to model long-term dependencies, making it unsuitable for tasks where the context of earlier in the sequence plays a decisive role in determining future states.

To summarize, Table 2 highlights the main advantages and disadvantages of RNNs discussed in this section.

Table 2: Advantages and Disadvantages of Recurrent Neural Networks (RNNs)

| **Advantages** | **Disadvantages** |
|---|---|
| Accepts variable-length input and preserves temporal context through hidden state maintenance. | Prone to vanishing or exploding gradients during back-propagation through time (BPTT), which hampers learning of long-term dependencies. |
| Effective for tasks requiring immediate temporal information, such as real-time speech processing. | Computationally demanding for long sequences due to inherently sequential nature, limiting parallel execution. |
| Conceptually straightforward, with weight sharing across time steps reducing model complexity. | Retention of long-range information is poor without specialized architectural modifications. |

To address these intrinsic limitations, particularly the inability to capture long-range dependencies and the instability caused by gradient issues, enhanced architectures such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) were introduced. These models incorporate gating mechanisms to regulate information flow and mitigate the effects of vanishing and exploding gradients.

### 2.2.2 Long Short-Term Memory (LSTM)

Imagine you're reading online reviews to decide whether to buy a particular brand of cereal. As you go through the comments, your brain doesn't retain every single word. Instead, you subconsciously focus on the most important parts, phrases like "amazing" or "perfectly balanced breakfast" stand out, while common words such as "this", "gave" or "all" fade into the background. Later, if someone asks what the review said, you may not recall it word-for-word, but you'll remember the general sentiment: "will definitely be buying again". In essence, your mind keeps the relevant information and filters out the rest [22].

This is precisely the kind of behaviour that Long Short-Term Memory (LSTM) networks aim to replicate in machine learning. By incorporating mechanisms that allow the model to decide what information to retain and what to forget, LSTMs are able to learn long-term dependencies more effectively than standard RNNs.

Long Short-Term Memory networks (LSTM) are a special type of RNN, designed to overcome the limitations of traditional RNNs, particularly their difficulty in learning long-term dependencies and so they are ideal in tasks such as language translation, speech recognition, and time series forecasting. They were introduced by Hochreiter and Schmidhuber in 1997 [1]. They are capable of learning long-range dependencies across extended time intervals in excess of steps

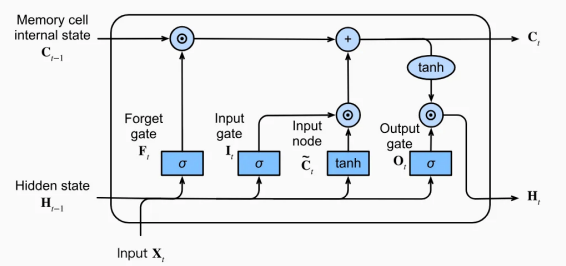



Figure 7: LSTM memory cell [23]

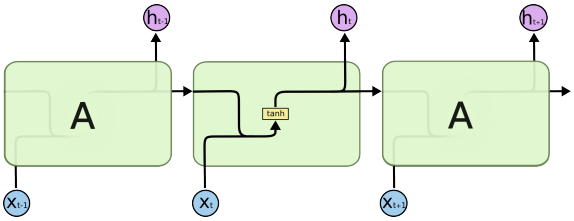



Figure 8: Repeating module RNN [16]

even when the input sequences are noisy or lack compressible structure while still preserving sensitivity to short-term patterns. This is made possible by an efficient gradient-based training algorithm applied to a specialized architecture that maintains a constant error flow through its internal memory units. By design, this architecture prevents gradients from either vanishing or exploding. Notably, even when gradient computation is truncated at certain architecture-specific points, the stability of long-term error propagation is preserved [24] [25] [26].

At their core, LSTM networks follow the same sequential, chain-like architecture as traditional RNNs, but with a more sophisticated internal structure. While standard RNNs use a simple repeating unit—often consisting of just a single layer with a non-linear activation like *tanh* (Figure 8), LSTMs replace this with a more complex module composed of four distinct components that interact in a coordinated manner (Figure 9). Central to this architecture is the *cell state*, a dedicated memory pathway that runs through the entire sequence, enabling the network to carry information forward with minimal modification. This pathway functions like a conveyor belt, allowing data to flow largely unchanged unless explicitly altered. The modification of the cell state is managed by structures known as *gates*, which regulate the addition and removal of information. These gates use sigmoid activations to produce values between 0 and 1, effectively controlling how much information is allowed to pass through. By combining these gating mechanisms with point-wise operations, LSTMs can dynamically update their memory and maintain long-term contextual information across sequences.

The functioning of an LSTM cell (Figure 7) involves a sequence of carefully orchestrated steps designed to manage and update its internal memory, the cell state, at each time-step. The process begins with the *forget gate*, a sigmoid-activated layer that determines which part of the previous

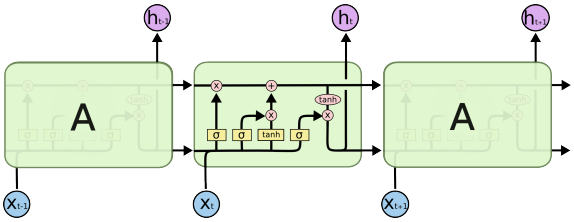



Figure 9: Repeating module LSTM [16]

cell state $C_{t-1}$ should be retained or discarded. It receives the current input $x_t$ and the previous hidden state $h_{t-1}$, and outputs a value between 0 and 1 for each element of the cell state, where 1 means "fully retain" and 0 means "completely forget".

Next, the model determines what new information to incorporate into the cell state. This involves two components: an *input gate*, which uses a sigmoid function to identify which elements should be updated, and a *candidate state* layer, typically using a *tanh* activation, which generates potential new values $\tilde{C}_t$ to be added. These two outputs are then combined to form the cell state update.

The updated cell state $C_t$ is computed by blending old and new information. Specifically, the previous cell state is multiplied element-wise by the forget gate output $f_t$, effectively discarding the designated components. Then, the result is incremented by the product of the input gate $i_t$ and the candidate values $\tilde{C}_t$, thus incorporating the selected new information.

Finally, the *output gate* determines what information to pass to the next hidden state $h_t$. This involves applying a sigmoid activation to decide which components of the cell state should influence the output. The cell state is then passed through a *tanh* function to scale its values between -1 and 1, and this is multiplied by the output gate's result. The final hidden state output thus reflects a filtered version of the internal memory, regulated by the learned gating mechanism [22] [16].

Although the gating mechanisms and memory cells of LSTMs greatly improve their ability to model long-range dependencies, they also introduce a level of computational complexity that makes training and inference resource-intensive, especially for long sequences or large datasets. To address these demands, modern implementations frequently rely on the parallel processing capabilities of Graphics Processing Units (GPUs), which significantly accelerate both training and inference. GPU-based execution has become the standard for LSTM workloads, offering up to a six-fold speedup during training and up to 140 times greater throughput during inference compared to CPU-based approaches. NVIDIA's CUDA Deep Neural Network library (cuDNN) provides optimized kernels specifically for training LSTM models in sequential learning tasks, while TensorRT, a high-performance inference engine, further improves runtime efficiency during deployment. Together, these tools, available through the NVIDIA Deep Learning SDK, make it feasible to apply LSTM architectures in real-time and large-scale deep learning systems [27].

Due to their ability to capture both short and long-term dependencies in sequential data, LSTM networks have been widely adopted across numerous domains. In natural language processing, they have powered tasks such as machine translation, text summarization, and sentiment analysis. In speech-related applications, LSTMs have been used for speech recognition, voice synthesis, and speaker identification. Their effectiveness in modeling temporal patterns also makes them suitable for time-series forecasting problems, including financial market prediction, weather modeling, and energy demand estimation. In healthcare, LSTMs have been used to model patient data over time, allowing early diagnosis and personalized treatment. Furthermore, their capacity for anomaly detection in streaming data has found use in cybersecurity, IoT monitoring, and predictive maintenance systems. These diverse applications highlight the versatility and practical value of LSTM architectures in real-world sequence learning problems.

To summarize, Table 3 highlights the main advantages and disadvantages of LSTM networks discussed in this section.

Despite LSTMs have proven to be highly effective in learning long-term dependencies, their architecture, comprising multiple gates and internal state vectors, introduces considerable computational overhead. To address this, the Gated Recurrent Unit (GRU) was introduced as a simplified alternative. GRUs retain the core idea of using gating mechanisms to control information flow but combine the forget and input gates into a single update gate and eliminate the separate memory cell. This results in a more compact and computationally efficient architecture, often achieving performance comparable to that of LSTMs on many sequence modeling tasks, with fewer parameters and faster training. The following section explores the GRU architecture in more detail, highlighting its structure, functionality, and practical advantages.

Table 3: Summary of Advantages and Disadvantages of LSTM Networks

| **Advantages** | **Disadvantages** |
|---|---|
| Capable of learning long-term dependencies in sequential data. | Higher computational complexity compared to simpler RNNs or GRUs. |
| Effectively mitigates vanishing/exploding gradient problems using gating mechanisms. | Slower training times and greater memory usage due to multi-gate architecture. |
| Widely applicable across diverse domains such as NLP, speech recognition, and time-series forecasting. | Less interpretable and harder to tune due to its intricate internal structure. |

### 2.2.3 Gated Recurrent Units (GRUs)

GRU cells, as like as LSTMs, were designed to extend the memory length of RNNs and address the gradient issues. They were introduced by Cho et al. in 2014 [28], that proposed the Gated Recurrent Unit as a simpler alternative that retains the benefits of gating while reducing complexity. The GRU streamlines the LSTM architecture, merging certain gates and eliminating the explicit memory cell, thereby using fewer parameters and achieving faster training without significant loss in performance. This combination of improved efficiency and maintained accuracy motivated the rapid adoption of GRUs in the mid-2010s as an attractive "state-of-the-art" RNN unit for sequential data. [2] [29] [30]

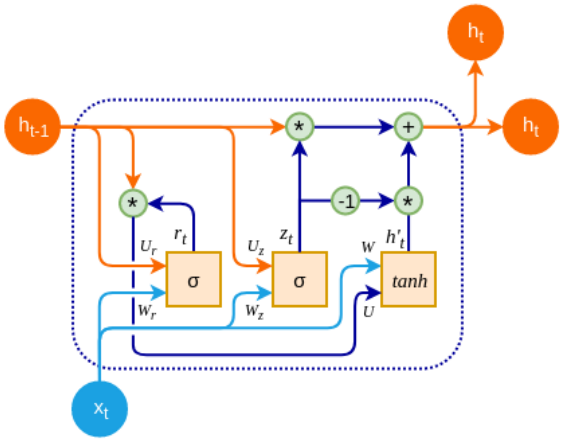



Figure 10: GRU memory cell [31]

A GRU is an RNN cell that relies on two primary gates (controls) to manage its hidden state: an *update gate* and a *reset gate* (Figure 10). At each time step $t$, the GRU takes the current input vector $x_t$ and the previous hidden state $h_{t-1}$ and computes these gates as sigmoids of linear combinations of the inputs and state. Intuitively, the *reset gate* $r_t$ determines how much of the past state to forget or reset before computing new content, while the *update gate* $z_t$ governs how much of the new candidate information will replace the old state. Formally, the update gate is computed as:

$$z_t = \sigma(W_z x_t + U_z h_t + b_z) \tag{5}$$

and the reset gate as:

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{6}$$

where $W_*$ and $U_*$ are learned weight matrices and $b_*$ are biases for the gates. Using the reset gate, the GRU computes a candidate hidden state $\hat{h}_t$ (with a *tanh* activation) by mixing the new input with the reset-modulated previous state. Finally, the new output hidden state is obtained via linear interpolation between the previous state and the candidate, controlled by the update gate:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t \tag{7}$$

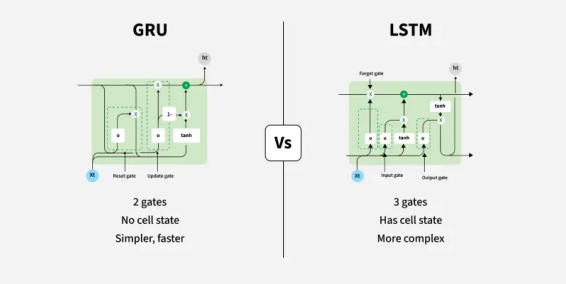



Figure 11: Comparison between LSTM and GRU cell [32]

In this way, if $z_t$ is close to 1 (update gate active), the new candidate $\hat{h}_t$ largely overwrites the old state, whereas if $z_t$ is 0, the state is carried over unchanged. This mechanism lets the GRU decide at each step how much past information to retain versus how much new information to write. [32]

The GRU can be viewed as a simplified LSTM (Figure 11). An LTMS cell has three gates (input, forget, output) and an explicit cell state $c_t$ in addition to the hidden state $h_t$. In contrast, a GRU cell has only two gates (update and reset) and no separate cell state: it merges the LSTM's cell and hidden state into a single unified hidden state. The update gate in a GRU plays a role analogous to the combination of LSTM's input and forget gates, deciding how much of the previous activation to keep versus overwrite. The GRU lacks an output gates: as a result, it exposes the full hidden state to the next layers at each time step, whereas LSTM's output gate can modulate the exposure of its cell state. This means a GRU has fewer control parameters than an LSTM. Indeed, by eliminating one gate and associated weight matrices, GRUs have fewer parameters for a given hidden size compared to LSTMs. This lighter architecture often translates to lower memory usage and less computation per time step. A simple summary is that *LSTM = 3 gates + cell state*, while *GRU = 2 gates, no separate cell.* Traditional "vanilla" RNNs, on the other hand, have no gating at all – only a single tanh or similar activation that fully replaces the state each step. Thus, vanilla RNNs are structurally simplest but unable to regulate memory, causing them to forget long-term information and suffer from unstable gradients. [33]

Empirically, GRUs and LSTMs have been found to achieve comparable performance on many sequence tasks, with no clear consensus on which is universally better. A study by Chung et al. (2014) [29] reported that GRUs and LSTMs yielded similar results, and the choice between them often did not significantly affect final performance. There are scenarios where one may edge out the other: some experiments showed that for a fixed model size (fixed number of parameters), GRUs trained faster and even outperformed LSTMs in terms of convergence speed and generalization on certain datasets [29]. Because GRUs have fewer parameters and slightly simpler computations, each training update can be less costly; one benchmark showed a GRU could reach convergence in fewer training epochs or less time than a similarly sized LSTM. In general, GRUs tend to train faster than LSTMs and can especially excel on smaller datasets or less complex tasks, where the extra capacity of LSTMs is not always needed. Vanilla RNNs, lacking gating, usually train the fastest per iteration but struggle to learn if long-term dependencies are present, often converging to worse results due to vanishing gradients. [34]

In terms of computational load, a GRU is somewhat lighter than an LSTM. Each LSTM cell update involves 4 weight matrix multiplications (for gates and candidate) and several element-wise operations, whereas a GRU involves 3 such multiplications (update gate, reset gate, candidate) and fewer internal operations. This reduction means that GRUs consume less memory and compute per time-step. In practice, it has been noted that "GRUs are computationally more efficient than LSTMs without compromising performance," which makes them attractive for real-time and high-performance settings [35]. For instance, Ravanelli et al. (2018) [34] observed that a simplified GRU model could reduce training time by more than 30% compared to a standard LSTM on a speech task. When sequences are long, this efficiency gain accumulates

significantly. Vanilla RNNs have the smallest computational footprint per step (only 1 matrix multiply), but they may require many more iterations or higher hidden dimensions to achieve the accuracy of gated models, potentially negating the raw per-step speed advantage. Importantly, all RNN variants (including GRU and LSTM) share a disadvantage relative to newer Transformer models: they perform sequential updates and thus cannot be parallelized across time steps. GRUs, like other RNNs, must process one timestamp after another, which limits throughput on modern parallel hardware for very long sequences. Transformers use parallel self-attention to examine entire sequences at once, so they scale better with long inputs. Nonetheless, for moderate sequence lengths, the GRU's simpler operations and fewer parameters give it a better balance of speed and performance than the LSTM in many cases.

Gated Recurrent Unit (GRU) networks have been successfully applied across a wide range of sequential and temporal modeling tasks due to their ability to capture long-term dependencies with reduced computational overhead. In natural language processing, GRUs have been used in machine translation, text classification, summarization, and sequence labelling, often serving as efficient alternatives to LSTMs. In speech and audio processing, GRUs have demonstrated competitive performance in tasks such as automatic speech recognition and keyword spotting, offering a favourable trade-off between accuracy and speed. They have also been widely adopted in time-series forecasting domains, including financial prediction, traffic flow estimation, and sensor data modeling, where their efficiency supports real-time deployment. Additionally, GRUs have found applications in video analysis, bioinformatics, and reinforcement learning, making them a versatile choice for resource-constrained environments and scenarios requiring lightweight recurrent models.

To summarize, Table 4 highlights the main advantages and disadvantages of GRU networks discussed in this section.

Table 4: Summary of Advantages and Disadvantages of GRU Networks

| **Advantages** | **Disadvantages** |
|---|---|
| Simpler architecture than LSTM with fewer gates and parameters. | Potentially less control over output/exposure of memory due to the missing output gate |
| Faster training and lower computational cost per step. | Coupled update/forget decisions which may reduce flexibility |
| Comparable performance to LSTM in many sequence modeling tasks, especially on small datasets or real-time systems. | Sequential processing requirements (limiting parallelization and hence efficiency on very long sequences) |

While gated recurrent networks like LSTM and GRU have significantly advanced sequence modeling by addressing the limitations of traditional RNNs, they still rely on sequential computation, which hinders parallelization and efficiency on modern hardware. To overcome these bottlenecks, the Transformer architecture was introduced, offering a fundamentally different approach based on self-attention mechanisms. In the next section, it was explored the Transformer model, which has redefined the state of the art in natural language processing and beyond through its scalability, parallelism, and superior performance on long-range dependencies.

### 2.2.4 Transformers

Recurrent models like RNNs, LSTMs, and GRUs dominated sequential data modeling for decades, offering a way to capture temporal dependencies in sequences. However, this kind of networks operate strictly sequentially: they process one token at a time, carrying a hidden state through the sequence. This inherently sequential nature limits their ability to capture very long-range dependencies and hinders parallelization during training. These limitations motivated a search for new architectures that could better handle long sequences and fully utilize high-performance computing resources. The solution emerged was the *Attention mechanism*.

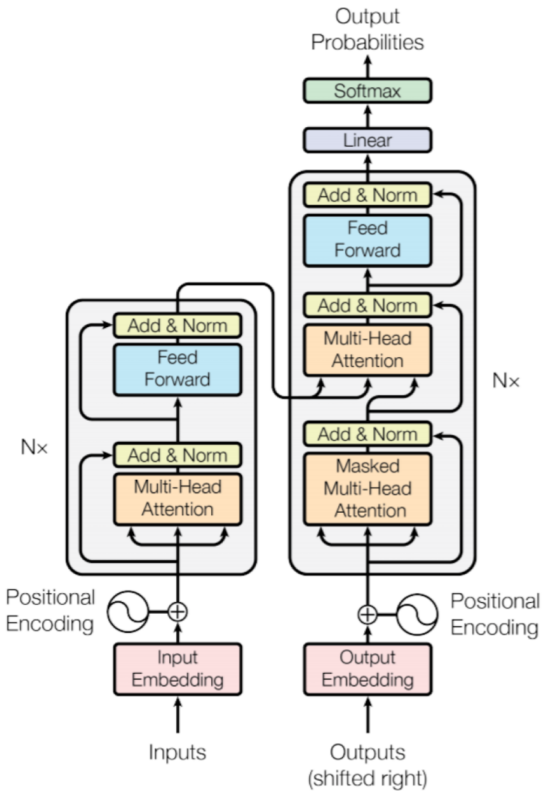



Figure 12: The Transformer - model architecture: consisting of an encoder (left) and decoder (right) stack of layers. [3]

Imagine being asked to extract information about categorical cross-entropy from an entire textbook on machine learning. One way would be to read the entire book cover to cover, that is a time-consuming and inefficient approach. A better method would be to consult the index, locate the chapter on loss functions, and go directly to the relevant section. This targeted focus dramatically improves both speed and precision. In much the same way, attention mechanisms in neural networks allow models to concentrate on the most relevant parts of the input data while de-emphasizing less important information. Attention answers the question of what part of the input we should focus on. [36]

This concept lies at the heart of the Transformer architecture, introduced by Vaswani et al. (2017) [3], which replaces the sequential nature of recurrent networks with highly parallel self-attention layers. In doing so, Transformers enable more efficient and scalable learning across long sequences, revolutionizing the field of deep learning in the process. In summary, Transformers arose as an evolution beyond RNNs to address the latter's shortcomings: limited long-range modeling capacity and poor scalability on modern hardware.

At the core of the Transformer architecture (Figure 12) is the concept of self-attention. Unlike RNNs that maintain a single evolving hidden state, self attention allows the model to look at all positions in the sequence at once and dynamically decide how much each other token should influence the representation of a given token. In other words, the model computes attention "weights" between every pair of positions, indicating the relevance of one token to another. Each token's representation is then updated as a weighted sum of the representations of all tokens, enabling the network to capture contextual relationships without any recurrence. Crucially, Transformers use multi-head self-attention, where the attention mechanism is replicated multiple times in parallel ("heads") (Figure 13). Each head can focus on different aspects of the sequence: for example, one head might attend strongly to the most recent tokens, while another

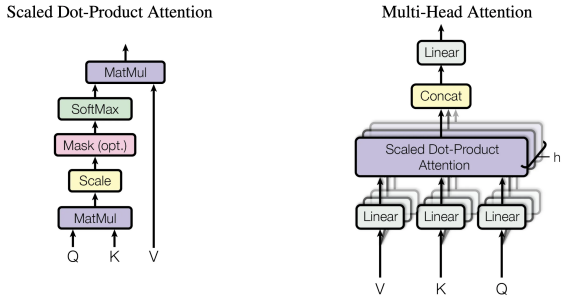



Figure 13: Self attention VS multi-head attention [3]

looks for a far-away relevant keyword. The results from all heads are concatenated and passed through a feed-forward layer, allowing the model to jointly consider various relationship in the data. This multi-head design ensures that important signals are not "averaged away" and gives the model a richer capacity to learn complex patterns. [3]

Another key component of Transformers is positional encoding. Since the model has no recurrent structure, it does not inherently know the order of tokens. Positional encodings solve this by adding an explicit position-dependent vector to each token's embedding at the input. The effect is to give the model a sense of sequence order, so that it can distinguish, for example, a token at the beginning from one at the end of a sequence. With positional encoding, Transformers can leverage sequence position information while still processing all tokens in parallel.

The Transformer follows the overall architecture showed in Figure 12 using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder. The encoder is a stack of $N$ identical layers; each layer consists of the multi-head self-attention sub-layer followed by a position-wise feed-forward network. The encoder takes the entire input sequence and produces a set of continuous vector representations (one for each input position) that are rich in contextual information. The decoder is another stack of $N$ layers of a similar structure, but with an extra sub-layer in each decoder layer for encoder-decoder attention. This sub-layer performs attention over the encoder's output vectors: it allows the decoder to consult the encoder's representation of the input sequence at every decoding step. Additionally, the decoder's self-attention sub-layer is masked so that each position can only attend to earlier positions in the output sequence (ensuring the model can generate outputs one token at a time, without looking into the "future" of the sequence). Through this architecture, the Transformer effectively decouples the sequence modeling problem: the encoder handles understanding of the input sequence, and the decoder handles generation of the output sequence, with attention links in between to bridge the two. Importantly, all these operations, attention and feed-forward transformations, are highly parallelizable matrix operations, which is a stark contrast to the step-by-step processing of RNNs.

The Transformer architecture introduces several significant improvements over traditional RNN-based models such as LSTMs and GRUs. The key differences are outlined below:

- **Long-Range Dependency Handling:** Transformers use self-attention mechanisms that allow each token to attend to all others in the sequence, capturing long-range dependencies directly. In contrast, RNNs rely on sequential updates, making it harder to retain information over long spans, even with gating mechanisms like those in LSTM and GRU.
- **Parallelization and Efficiency:** Unlike RNNs, which process sequences step-by-step, Transformers process all tokens simultaneously. This enables full parallelization of both training and inference, significantly accelerating computation on GPUs and TPUs and allowing for more efficient utilization of hardware resources.

- **Scalability and Model Capacity:** Transformers can be scaled to very large architectures (e.g., GPT-3 with 175 billion parameters) thanks to their stable training dynamics and ability to parallelize across layers and data. RNN-based models, by contrast, struggle with vanishing gradients and memory bottlenecks when scaled to similar sizes.
- **Performance on Long Sequences:** While LSTMs and GRUs include mechanisms to manage long-term memory, their effectiveness degrades with very long sequences. Transformers maintain context more effectively across hundreds or thousands of tokens, making them better suited for tasks like document-level translation or long-range time-series forecasting.
- **Training Speed and Data Utilization:** Due to parallelism and efficient computation, Transformers often train faster and generalize better when exposed to large datasets. They benefit significantly from large-batch training and distributed systems, unlocking performance gains not easily achievable with sequential RNN architectures.

The design of Transformers is particularly well-suited for high-performance computing (HPC) environments. Because the heavy computations (matrix multiplications for attention and feed-forward layers) can be done in parallel, Transformers can fully utilize the massive parallelism of GPUs, TPUs and distributed clusters. In training, one can distribute the workload of a Transformer across many processors:e.g. splitting batches across GPUs (data parallelism) or splitting the model's layers across devices (model parallelism). This has enabled training of models at scales previously unimaginable. For instance, the original Transformer achieved a milestone by training a high-quality translation model in only 12 hours on 8 GPUs, something infeasible with an RNN-based model under similar hardware [3].

In terms of inference, HPC techniques are equally important. Serving a gigantic Transformer-based model (with hundreds of billions of parameters) in real-time requires splitting the model and the data across multiple nodes. In other words, HPC infrastructures make it possible to deploy these models at scale, and the Transformer's inherently parallel computations make it possible to take advantage of such infrastructure. By contrast, a sequential model that processes one step at a time would become a bottleneck on large parallel systems, as it cannot easily divide the work among many processors.

Another HPC aspect is memory optimization. Transformers require substantial memory, especially for long input sequences, since self-attention uses $O(n^2)$ memory for sequence length $n$. High-performance hardware often comes with large aggregate memory (e.g. multiple GPUs each with high VRAM, plus possible CPU and disk offloading). Modern HPC techniques such as memory pooling, mixed precision, and batch re-computation are widely applied to Transformers to fit larger models and longer sequences into memory. In summary, the scalability of the Transformer is tightly coupled with HPC: the architecture thrives when given ample parallel resources, and conversely, the availability of HPC resources in recent years has fuelled the success of ever larger Transformer models. [37]

Transformer models have become the foundation of modern sequence modeling, initially revolutionizing natural language processing (NLP) and subsequently expanding into diverse domains. In NLP, Transformers outperformed previous RNN-based approaches in tasks like machine translation, question answering, and text generation. Models such as BERT, which captures bidirectional context using an encoder-only architecture, and GPT, a decoder-based model optimized for generative tasks, exemplify the architecture's flexibility and power. Today, Transformers are integral to a wide range of NLP applications including document summarization, dialogue systems, and named entity recognition. Their success extends beyond text: in time-series forecasting, Transformers effectively model long-range dependencies and complex temporal patterns, making them suitable for applications like electricity load prediction, financial forecasting, and anomaly detection. They have also been adapted for speech recognition, bioinformatics, reinforcement learning, and computer vision—with architectures like the Vision Transformer reinterpreting images as sequences of patches. This broad applicability underscores the Transformer's central role in advancing machine learning across domains where

understanding temporal or sequential relationships is essential.

To summarize, Table 5 highlights the main advantages and disadvantages of Transformer networks discussed in this section.

Table 5: Summary of Advantages and Disadvantages of Transformer Networks

| **Advantages** | **Disadvantages** |
|---|---|
| Captures long-range dependencies via self-attention across the full sequence. | Quadratic complexity ($O(n^2)$) with sequence length in both time and memory due to full self-attention. |
| Fully parallelizable computation enables faster training and better hardware utilization. | Requires significant computational resources for training and inference (e.g., high-end GPUs or TPUs). |
| Highly scalable architecture supports large models and datasets, achieving state-of-the-art results in NLP, time-series, vision, and more. | Limited generalization to sequence lengths longer than those seen during training unless specifically designed for. |
| Flexibility to be adapted to various domains (text, audio, vision, etc.) with encoder, decoder, or hybrid forms. | Interpretability challenges and lack of inductive biases like recurrence or locality seen in RNNs or CNNs. |

To conclude, Transformer models have revolutionized sequence modeling through their powerful self-attention mechanisms, parallelism, and scalability, outperforming RNN-based architectures in a wide range of domains. Nevertheless, their computational overhead, particularly with long sequences, and reliance on large-scale infrastructure have exposed certain limitations, especially in resource-constrained or real-time applications. These challenges have renewed interest in alternative modeling paradigms grounded in continuous-time dynamics and biological plausibility. In response, the research community has developed models that bridge the gap between discrete-time architectures and more adaptive, efficient systems—most notably through ODE-based neural networks, continuous-time recurrent models, and eventually Liquid Neural Networks (LNNs). The following section traces this progression, laying the foundation for understanding how these developments culminate in the Liquid Time-Constant and Closed-form Continuous-time models.

## 2.3 Continuous-Time and Biologically Inspired Models: Toward Liquid Neural Networks

While recurrent and attention-based models have dominated temporal learning tasks, their discrete-time structure and computational constraints have prompted the search for more flexible and biologically plausible alternatives. This section investigates the evolution toward continuous-time neural models, architectures that represent neural dynamics using differential equations and adaptive memory. Starting with Continuous-Time Recurrent models and Neural ODEs, and extending through ODE-based models, the section outlines how these approaches address temporal granularity, irregular sampling, and long-range dependencies. Finally, it introduces biologically inspired membrane models, which lay the conceptual groundwork for Liquid Neural Networks: adaptive, lightweight models capable of real-time temporal reasoning.

### 2.3.1 Continuous-Time Recurrent Neural Networks (CT-RNNs)

Continuous-Time Recurrent Neural Networks (CT-RNNs) [38] represent one of the earliest efforts to move beyond the limitations of discrete-time recurrent models by formulating neural dynamics in continuous time. Unlike standard RNNs that update their hidden states at fixed intervals, CT-RNNs define the evolution of neuron activations through differential equations, allowing the system to process inputs and internal dynamics in a temporally smooth and biologically plausible manner.

In CT-RNNs, the hidden state of each neuron is typically modeled by a first-order ordinary differential equation (ODE), often of the form:

$$\tau_i \frac{dh_i(t)}{dt} = -h_i(t) + \sigma\left(\sum_j W_{ij} h_j(t) + \sum_k U_{ik} x_k(t) + b_i\right)$$

where $h_i(t)$ is the hidden state of neuron $i$, $\tau_i$ is a learnable or fixed time constant that governs the decay rate, $\sigma(\cdot)$ is a non-linear activation function (e.g., *tanh* or *sigmoid*), $W$ and $U$ are weight matrices for recurrent and input connections respectively, and $b_i$ is the bias term.

The introduction of the time constant $\tau_i$ allows the network to capture the rate at which each neuron's state decays or "forgets" prior information. This enables the model to encode temporal dependencies over variable time scales—a property particularly beneficial for irregular or asynchronous input data. Such a formulation also provides a closer approximation to the behaviour of biological neurons, which integrate signals over continuous time rather than discrete steps.

CT-RNNs have been shown to possess strong theoretical properties, including the ability to approximate arbitrary dynamical systems under suitable conditions. However, in practice, they are limited by the rigidity of their fixed time constants and their reliance on numerical solvers for ODE integration, which can be computationally intensive and sensitive to solver configurations. As a result, CT-RNNs laid important theoretical groundwork but had limited practical adoption in large-scale machine learning tasks.

Nonetheless, they form a critical conceptual stepping stone in the development of more expressive and efficient continuous-time architectures, such as Neural ODEs and, eventually, the Liquid Time-Constant networks.

### 2.3.2 Neural Ordinary Differential Equations (Neural ODEs)

Neural Ordinary Differential Equations (Neural ODEs) represent a significant advancement in continuous-time modeling by generalizing the forward pass of deep networks as the solution to an ordinary differential equation. Introduced by Chen et al. [39], Neural ODEs reformulate the discrete sequence of transformations in a neural network as a continuous trajectory through latent space, governed by a learned differential equation.

Instead of computing hidden states through a fixed number of layers, a Neural ODE models the dynamics of the hidden state $h(t)$ using a differential equation of the form:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta)$$

where $f$ is a neural network parametrized by $\theta$ that defines the continuous dynamics of the hidden state. The solution to this ODE is computed using numerical integration techniques such as Euler, Runge-Kutta, or adaptive solvers. This continuous transformation replaces the traditional layer-by-layer computation with an initial value problem:

$$h(t_1) = h(t_0) + \int_{t_0}^{t_1} f(h(t), t, \theta) dt$$

This formulation provides several benefits: it allows for adaptive computation (the solver determines how many steps are needed), supports modeling of irregular or asynchronous time-series data, and enables memory-efficient back-propagation through the *adjoint method*. Moreover, Neural ODEs offer smooth hidden-state evolution, which can be crucial in scientific or physical systems where continuity is important.

However, Neural ODEs also introduce new challenges. The accuracy and efficiency of the model are strongly dependent on the choice of ODE solver, which can introduce significant computational overhead, especially in stiff or complex systems. Additionally, the expressivity of the model is ultimately limited by the architecture of the function $f$, which may not easily capture highly non-linear or multi-scale dynamics.

Despite these limitations, Neural ODEs laid the foundation for a new class of time-continuous models and directly influenced the development of more efficient and specialized variants, such as ODE-RNNs, CT-LSTMs, and ultimately, the Liquid Time-Constant networks.

### 2.3.3 ODE-Extended Recurrent Models

As the integration of differential equations into neural networks gained momentum, researchers began to adapt classical gated recurrent architectures such as RNNs, GRUs, and LSTMs into continuous-time formulations. These *ODE-extended recurrent models* aim to preserve the strengths of discrete-time memory gating while leveraging the temporal flexibility of continuous dynamics.

**CT-LSTM** (Continuous-Time LSTM) [40] introduces learnable decay parameters for the LSTM cell state, allowing the model to simulate how memory content fades over time. This enables more precise modeling of real-world scenarios in which information is gradually forgotten unless reinforced. The CT-LSTM modifies the LSTM equations by adding exponential decay governed by a time constant, providing a smooth and differentiable memory decay mechanism.

**ODE-RNN** [41] augments a standard RNN by using an ODE solver to evolve the hidden state continuously between input events. When an observation arrives, the model applies a discrete update; otherwise, the state evolves according to a learned ODE function. This architecture is particularly suited for irregularly sampled time series, where intermediate state updates are not tied to a fixed clock.

**ODE-GRU and ODE-LSTM** [41] [42] extend this idea further by incorporating GRU and LSTM-style gating mechanisms into the ODE evolution of the hidden state. These models combine the representational power of gated recurrent units with the capacity to model smooth temporal dynamics. Between inputs, the memory state evolves as the solution of an ODE, while updates at event times maintain temporal alignment and gating control.

These models bridge the gap between the rigid discretization of classical RNNs and the flexibility of Neural ODEs. By doing so, they offer a more fine-grained treatment of temporal sequences and enable learning from sparse or irregular data. However, they still inherit several limitations: their reliance on ODE solvers can incur significant computational cost, and they often use fixed or parametrized time constants, limiting adaptability across tasks with varying temporal dynamics.

As a result, researchers have continued to explore models that offer richer internal dynamics with improved computational efficiency. This pursuit led to biologically inspired formulations such as the Liquid Time-Constant networks.

### 2.3.4 Membrane Models and Biological Foundations

While continuous-time formulations like Neural ODEs and CT-LSTM introduce temporal flexibility, they are still governed by predefined solver mechanics and limited by static time constants. In contrast, *membrane models* take inspiration from biological neural systems, particularly the electro-tonic behaviour observed in the nervous systems of simple organisms such as *C. elegans* and *Ascaris* [5].

In these biological circuits, neurons transmit information not through discrete spikes but via graded, continuous changes in membrane potential—a form of analogue computation governed by

non-linear differential equations. A key property of such systems is the presence of *input-dependent, variable time constants*, which allow each neuron to dynamically adjust its response to stimuli based on input intensity and synaptic configuration.

This dynamic modulation enables more expressive temporal modeling: neurons can forget or retain information at rates that change in real time, offering robustness to noisy signals and varying temporal patterns. Additionally, biological neurons often exhibit complex excitatory and inhibitory interactions, modulating the direction and magnitude of signal propagation beyond simple weighted summation.

Inspired by these mechanisms, membrane-based neural models incorporate:

- Non-linear synaptic interactions beyond additive inputs,
- Input-dependent regulation of internal dynamics,
- Continuously evolving hidden states governed by time-varying ODEs.

Such principles pave the way for *Liquid Neural Networks*—a family of models that integrate continuous-time evolution, non-linear feedback, and biologically grounded adaptability. Among these, the Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) networks stand out for their theoretical innovation and practical efficiency. These models abandon fixed solvers and static time scales in favour of architectures where neurons evolve according to input-driven dynamics, enabling richer memory and real-time learning with compact representations.

In the next sections, these Liquid models in depth will be explored, beginning with the biologically inspired architecture of the LTC network.

### 2.3.5 Toward Liquid Neural Networks: Architecture and Principles

Liquid Neural Networks (LNNs) represent a novel paradigm in neural network design, emerging from the intersection of continuous-time modeling and biologically inspired computation. Unlike traditional RNNs, which rely on fixed-step updates, or Neural ODEs, which evolve hidden states through numerically integrated dynamics, LNNs define neuron behaviour through *input-dependent, non-linear differential equations* that adapt dynamically over time. This enables each neuron to continuously evolve its internal state based on the nature and strength of the input, offering both computational flexibility and temporal expressiveness. [43]

The core idea behind LNNs is that neural processing should not be governed by a fixed frequency or solver step size. Instead, neurons should evolve continuously with a *liquid* dynamic—changing their trajectory in response to stimuli, much like biological neurons. This is mathematically expressed by a differential equation of the form:

$$\frac{dh(t)}{dt} = F(h(t), x(t), t; \theta)$$

where $h(t)$ is the internal state, $x(t)$ is the input, and $F$ is a non-linear function parametrized by $\theta$, encoding neuron-specific dynamics. Crucially, the response of the neuron is not static; it depends on the current input and its own state, allowing for rich, non-linear time-dependent behaviour.

Key characteristics of Liquid architectures include:

- **Adaptive Dynamics:** Unlike fixed-weight or fixed-time networks, the behaviour of an LNN neuron changes depending on the input. This means the network can adapt in real time to new or evolving data distributions.
- **Minimal Parameter Footprint:** LNNs are often far more compact than conventional networks, achieving competitive performance with significantly fewer parameters. This enables both efficient training and low-memory deployment, making them well suited for edge computing or mobile applications.

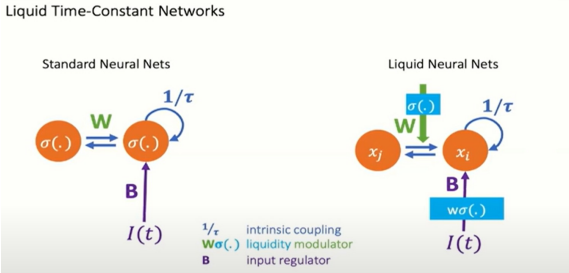



Figure 14: Standard Neural Network VS Liquid Neural Network [44]

- **Robust Generalization:** Due to their continuous and feedback-driven nature, LNNs often generalize better on out-of-distribution (OOD) data and are less prone to over-fitting. This property stems from the dynamic adjustment of internal state rather than reliance on static parameter configurations.
- **Biological Inspiration:** LNNs take cues from the non-linear signal propagation seen in small biological nervous systems, such as those of *C. elegans*. They mimic the graded and inhibitory control present in natural circuits, introducing biologically plausible time constants and feedback.

Figure 14 illustrates the fundamental architectural difference between standard neural networks and Liquid Neural Networks (LNNs). In standard models, neurons are connected through fixed weights and non-linear activation functions, with state evolution governed by simple update rules. By contrast, LNNs replace static activations with dynamic differential equations and introduce a non-linear modulation of interactions between neurons. This non-linearity, termed the *liquidity modulator*, can itself be a learnable function, such as a neural network, and governs how the internal dynamics of one neuron affect another. In this architecture, neuron states evolve over continuous time, and their coupling is regulated both intrinsically and through input-dependent mechanisms. As Hasani describes, the core innovation is that "activations are changed to differential equations," allowing for richer, more biologically plausible dynamics that adapt in real time.

These innovations position Liquid Neural Networks as a powerful solution to the temporal and computational limitations of earlier sequential models. They are especially valuable in environments where data arrives at irregular intervals or where the system must adapt continuously over time. Furthermore, their design aligns naturally with high-performance computing goals, as they reduce computational complexity without sacrificing expressivity.

The next sections examine two leading LNN architectures: the *Liquid Time-Constant (LTC)* network, which introduces input-controlled time constants, and the *Closed-form Continuous-time (CfC)* network, which simplifies temporal evolution through analytical expressions.

## 2.4 The Liquid Time-Constant (LTC) Network

Liquid Time-Constant (LTC) Networks [5] represent one of the first practical realizations of the Liquid Neural Network (LNN) paradigm. Building on the limitations of traditional and ODE-extended recurrent models, LTCs introduce a fundamentally different approach to temporal learning: one in which each neuron is modelled as a continuous-time dynamical system whose behaviour adapts in real time based on its input. Inspired by the graded signal transmission observed in simple biological nervous systems, LTCs replace static activations with non-linear differential equations, enabling neurons to evolve continuously and independently. This section explores the architecture, theoretical formulation, and dynamic properties of LTCs, highlighting their ability to model complex temporal patterns with fewer parameters and improved generalization. Through examples and performance benchmarks, it is illustrated how LTCs serve

as a compact, biologically grounded alternative to conventional neural architectures for sequential data.

### 2.4.1 Motivation and Background

Traditional neural network architectures, including RNNs and LSTMs, rely on discrete-time updates and fixed architectural rules for processing sequential data. While effective in many scenarios, these models often struggle to generalize in environments with continuous dynamics, sparse sampling, or high temporal variability. Moreover, their reliance on fixed or manually tuned time constants limits their ability to adapt to the temporal structure of the input in real time.

To overcome these limitations, recent research has drawn inspiration from biological systems, particularly the nervous systems of simple organisms like C. elegans. Unlike artificial networks with static update rules, biological neurons exhibit continuously evolving internal states modulated by both external stimuli and internal dynamics. These systems demonstrate remarkable flexibility and energy efficiency, processing sensory data with minimal computation and adapting their responses based on contextual input.

Liquid Time-Constant Networks (LTCs), introduced by Hasani et al. in 2020, were developed to mimic this biological principle by allowing each neuron to operate as a differential equation with input-dependent dynamics. Rather than applying a fixed update function at each time step, an LTC neuron updates its state based on the solution to an ordinary differential equation (ODE), with a time constant that varies as a function of its current input. This design enables the network to adaptively regulate how quickly or slowly it reacts to stimuli, depending on the context—providing a natural mechanism for temporal attention and stability.

The motivation behind LTCs is twofold: first, to create a neural architecture capable of learning from continuous-time data in a biologically plausible way; and second, to reduce the parameter footprint while maintaining or improving performance on temporal tasks. This makes LTCs particularly appealing for real-time systems, low-power devices, and scenarios requiring out-of-distribution generalization.

### 2.4.2 Architecture and Mathematical Formulation

At the core of Liquid Time-Constant (LTC) Networks lies a paradigm shift in how neural computation is conceptualized. Unlike conventional neural models where neurons apply a static activation function at each time step, LTC neurons are modelled as dynamic systems governed by ordinary differential equations (ODEs). This allows each neuron to exhibit continuous-time evolution, modulated by its inputs and internal connectivity.

The fundamental idea is to replace the fixed-time updates of traditional RNNs with input-modulated, time-continuous dynamics. Each LTC neuron maintains a membrane potential $x_i(t)$, which evolves over time based on both internal and external signals. The evolution of this state is described by the following differential equation:

$$\frac{dx_i(t)}{dt} = -\frac{1}{\tau_i(t)}x_i(t) + \sum_j W_{ij}\sigma(x_j(t)) + B_i u(t)$$

Here:

- $x_i(t)$ is the hidden state (or membrane potential) of neuron $i$,
- $\tau_i(t)$ is a time constant that is itself a function of the inputs, making the neuron's responsiveness dynamically adjustable,
- $W_{ij}$ are the synaptic weights between neurons,
- $\sigma(\cdot)$ is a non-linear activation function (e.g., tanh or sigmoid),

- $u(t)$ represents the external input to the system,
- $B_i$ is the input weight matrix connecting the input $u(t)$ to neuron $i$.

The key innovation is the input-dependent time constant $\tau_i(t)$, which modulates the decay and integration rate of each neuron. In practice, this means that some neurons may "liquefy" their behaviour (responding quickly to rapid input changes) while others may "solidify" (retaining longer-term memory when appropriate). This behaviour is what gives rise to the term “liquid” in the architecture’s name.

To model this dynamic $\tau_i(t)$, a small auxiliary neural network or a parametric function is often used, allowing each neuron to regulate its temporal response based on the magnitude and pattern of the incoming signals. This mechanism equips the LTC architecture with inherent flexibility: instead of applying a global temporal filter (as in fixed time-step RNNs), each neuron autonomously adjusts its processing timescale.

The interactions between these neurons are not simply scalar-weighted connections, but rather governed by non-linear transformations of the neurons’ evolving states. This introduces a form of rich, time-varying computation where the network’s behaviour depends on the interplay between dynamics, connectivity, and input signals. In effect, the system becomes a collection of coupled, non-linear differential equations, capable of approximating a wide range of dynamic systems with high expressiveness and compact parametrization.

The use of differential equations to model neurons also necessitates the use of numerical ODE solvers (e.g., Euler or Runge-Kutta methods) during training and inference. These solvers approximate the solution to the system over small time steps, allowing gradient-based optimization to be performed using back propagation through time.

Overall, the LTC architecture redefines what it means to be a "neuron" in a neural network: rather than a static, memoryless unit applying a non-linear function, it becomes a self-regulating, adaptive dynamic system that evolves continuously and responds flexibly to its environment.

### 2.4.3 Interpretability and Dynamic Behaviour

One of the defining features of Liquid Time-Constant (LTC) networks is their inherent interpretability as dynamical systems. Unlike standard RNNs or LSTMs, where neuron activations are computed via fixed recurrence relations and often behave as black boxes, LTC neurons evolve in time following explicitly defined differential equations. This continuous-time formulation allows researchers and practitioners to reason more transparently about the behaviour of each neuron, as well as the global system dynamics.

The adaptivity of LTCs comes primarily from the input-dependent time constants $\tau_i(t)$, which modulate how quickly or slowly a neuron’s internal state responds to changes in the environment. When the input signal is strong or highly informative, $\tau_i(t)$ can be reduced, making the neuron react quickly, effectively acting as a short-term memory element. Conversely, when input variation is minimal, the time constant can increase, allowing the neuron to retain information for longer periods and function as a stabilizing memory. This mechanism provides an elegant and biologically inspired form of temporal attention: neurons “decide” how long to remember based on their context.

This dynamic adjustment of the internal timescale enables neurons within the same network to operate at different temporal resolutions, improving the model’s expressiveness and allowing it to capture both fast-changing and slowly evolving patterns in the input sequence. For example, in a time-series classification task, some neurons may track high-frequency signal components like noise or rapid fluctuations, while others model longer-term dependencies or trends, all without needing to manually configure multiple timescales into the architecture.

Moreover, LTC neurons are structured to support both excitatory and inhibitory connections, just as in biological neural circuits. Through learned parameters and non-linear activation functions, these interactions contribute to rich internal dynamics that resemble the signal regulation observed in natural systems. These mechanisms allow the network to stabilize or amplify certain trajectories over time and offer better robustness to perturbations in the input signal.

Moreover, LTC neurons are structured to support both excitatory and inhibitory connections—just as in biological neural circuits. Through learned parameters and non-linear activation functions, these interactions contribute to rich internal dynamics that resemble the signal regulation observed in natural systems. These mechanisms allow the network to stabilize or amplify certain trajectories over time and offer better robustness to perturbations in the input signal.

Visualizing the evolution of neuron states over time also provides useful insights into how the network processes sequences. For instance, plotting the internal states of different LTC neurons during a prediction task can reveal which units are more responsive to fast-changing features and which ones maintain a steady memory of past information. Such visualization techniques have been used to analyse model robustness, interpret decision boundaries, and detect the onset of phenomena such as over-fitting or input drift.

In summary, the dynamic and interpretable behaviour of LTCs makes them not only powerful modeling tools for complex temporal systems but also appealing from the standpoint of explainability and theoretical analysis. Their continuous-time operation, biologically plausible adaptability, and input-sensitive responsiveness offer a novel perspective on how neural networks can learn from and respond to temporal data.

### 2.4.4 Performance and Applications

Liquid Time-Constant (LTC) networks have demonstrated compelling performance across a variety of temporal tasks, particularly where efficient learning from sparse or irregular time-series data is required. Their biologically inspired architecture allows for rich temporal expressivity with fewer parameters compared to traditional RNNs, LSTMs, and even many continuous-time models.

The original paper demonstrated LTC performance on a wide range of time-series modeling tasks: overall, LTC networks showed improved accuracy in 4 out of 7 benchmark experiments and comparable performance in the other 3, compared to the best baseline models. These results, showed in Table 6 underscore LTC's strength in handling complex temporal dependencies.

Table 6: Comparison of model performance across datasets. Best results are highlighted in bold. [5]

| Dataset | Metric | LSTM | CT-RNN | Neural ODE | CT-GRU | LTC |
|---|---|---|---|---|---|---|
| Gesture | (accuracy) | 64.57% ± 0.59 | 59.01% ± 1.22 | 46.97% ± 3.03 | 68.31% ± 1.78 | **69.55% ± 1.13** |
| Occupancy | (accuracy) | 93.18% ± 1.66 | 94.54% ± 0.54 | 90.15% ± 1.71 | 91.44% ± 1.67 | **94.63% ± 0.17** |
| Activity recognition | (accuracy) | 95.85% ± 0.29 | 95.73% ± 0.47 | **97.26% ± 0.10** | 96.16% ± 0.39 | 95.67% ± 0.575 |
| Sequential MNIST | (accuracy) | **98.41% ± 0.12** | 96.73% ± 0.19 | 97.61% ± 0.14 | 98.27% ± 0.14 | 97.57% ± 0.18 |
| Traffic | (squared error) | 0.169 ± 0.004 | 0.224 ± 0.008 | 1.512 ± 0.006 | 0.389 ± 0.076 | **0.099 ± 0.0095** |
| Power | (squared error) | 0.628 ± 0.003 | 0.742 ± 0.005 | 1.254 ± 0.149 | **0.586 ± 0.003** | 0.642 ± 0.021 |
| Ozone | (F1-score) | 0.284 ± 0.025 | 0.236 ± 0.011 | 0.168 ± 0.006 | 0.260 ± 0.024 | **0.302 ± 0.0155** |

A major strength of LTCs lies in their adaptability to real-world, time-continuous scenarios. In domains such as robotics and control, where agents must operate in dynamically changing environments with asynchronous signals, LTCs exhibit superior generalization and robustness. The model's continuous-time nature enables seamless interpolation across varying temporal scales, making it highly effective for handling variable-length input sequences and irregular sampling intervals—conditions under which traditional discrete-time models often degrade.

Despite these benefits, one limitation of LTCs is their higher training time compared to discrete recurrent models or Transformer architectures. This increased computational demand stems from the need to solve a system of ordinary differential equations (ODEs) during training, which must be integrated at every time step. Additionally, gradient computation through ODE solvers (e.g., using adjoint methods or automatic differentiation) introduces further overhead. While LTCs train more slowly, they often require fewer parameters, generalize well even on small datasets, and offer faster inference at deployment time. Thus, the trade-off is between initial training cost and downstream efficiency and robustness.

From a resource-efficiency standpoint, LTCs are compact and compute-efficient. Their architecture allows them to learn competitive representations with far fewer neurons, making them attractive for deployment on embedded systems and edge devices. Moreover, their performance gains often emerge even when trained on small datasets, addressing a common challenge in domains where large annotated datasets are scarce.

In summary, LTCs strike a promising balance between biological realism, computational efficiency, and predictive performance. Their ability to adapt dynamically to the structure of temporal input, combined with their lightweight nature, positions them as strong candidates for future research and real-world applications in dynamic, time-sensitive environments.

### 2.4.5 Strengths and Limitations

Liquid Time-Constant (LTC) networks offer a unique blend of biological plausibility and computational efficiency, positioning them as powerful alternatives to traditional sequence models in time-sensitive applications. Their key strength lies in their ability to adapt neuron dynamics over time through input-dependent, learnable time constants, enabling compact models with high temporal expressivity. This makes LTCs particularly effective in domains characterized by sparse, noisy, or irregular data such as robotics, neuroscience, and healthcare.

Another advantage is the model's parameter efficiency: LTC networks consistently achieve competitive or superior performance with fewer parameters than LSTMs, GRUs, or Transformers [5]. They also exhibit excellent generalization with small datasets and can maintain performance across varying sequence lengths, benefiting from their continuous-time formulation. During inference, LTCs are lightweight and fast to execute, making them suitable for real-time and edge applications.

However, LTCs also come with notable limitations. Chief among these is their high training cost. The need to numerically integrate a system of non-linear differential equations during both forward and backward passes significantly increases computational load and training time. This limits their scalability in large-batch or long-sequence scenarios. Furthermore, because their internal dynamics are governed by ODE solvers, LTC models are less naturally parallelizable than discrete-time models like Transformers.

Another challenge lies in model interpretability and reproducibility. Due to the complex, dynamic behaviour of each neuron—driven by non-linearly coupled ODEs—debugging or analysing learned representations can be more difficult than with simpler architectures. Training stability may also depend on careful selection of solver methods and hyper-parameters, adding tuning complexity for practitioners.

In conclusion, while LTC networks introduce certain computational challenges during training, their adaptability, efficiency, and robustness make them promising candidates for future research and deployment in environments where compactness and dynamic temporal reasoning are paramount.

Table 7: Summary of strengths and limitations of Liquid Time-Constant Networks.

| **Strengths** | **Limitations** |
|---|---|
| Biologically inspired, continuous-time formulation | High training time due to ODE solver integration |
| Compact architecture with fewer parameters | Limited parallelism during training |
| Effective with sparse and irregular data | Sensitive to solver and hyper-parameter choices |
| Good generalization with small datasets | More complex to interpret and debug |
| Fast and lightweight inference | Less mature ecosystem and tooling support |

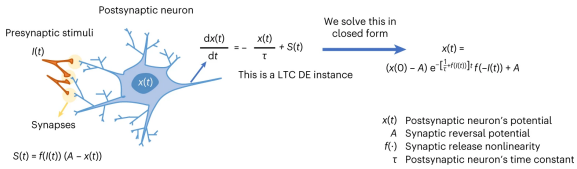



Figure 15: Neural and synapse Dynamics [6]

## 2.5 The Closed-form Continuous-time (CfC) Network

The limitations of numerically integrated continuous-time models, such as CT-RNNs, Neural ODEs, and even Liquid Time-Constant networks, have prompted the development of more efficient alternatives. Closed-form Continuous-time (CfC) Networks [6] represent a novel class of continuous-time neural architectures that eliminate the need for ODE solvers by modeling temporal evolution with closed-form equations. This innovation enables faster computation, better hardware compatibility, and improved scalability—without sacrificing temporal expressiveness. In this section, the foundations, architecture, and performance of CfC networks are explored, with a focus on their practical applications and their advantages over traditional recurrent and continuous-time models.

### 2.5.1 Introduction and Motivation

Closed-form Continuous-time (CfC) Networks were introduced to address fundamental computational and modeling challenges in previous continuous-time architectures. Models such as Continuous-Time RNNs (CT-RNNs), Neural ODEs, and Liquid Time-Constant networks (LTCs) are grounded in differential equation dynamics and have shown strong representational power for time-dependent data. However, they often rely on numerical integration during training and inference, which increases computational overhead, limits scalability, and complicates deployment on real-time or resource-constrained systems.

The motivation behind CfC networks lies in designing a recurrent architecture that maintains the expressive temporal modeling capability of ODE-based methods while enabling direct, efficient computation. Instead of requiring a numerical solver to approximate the evolution of neural states over time, CfC models leverage a *closed-form analytical solution* for this update. This eliminates the need for discretization and repeated function evaluations, reducing both computational cost and memory usage.

CfC networks operate by modeling the change in the hidden state over a time interval $\Delta t$ with a solution that is expressed explicitly as a function of the input and the previous state. This allows the model to reason over continuous time intervals without relying on iterative solvers. As a result, CfC is solver-free, more compatible with GPU acceleration, and better suited for

real-time applications compared to models like Neural ODEs and LTCs.

These properties make CfC particularly attractive in domains where timing is irregular or real-time inference is essential, such as physiological signal modeling, robot motion forecasting, or sensor fusion. Furthermore, CfC models retain biological plausibility and flexibility, making them an important step in bridging the gap between efficient machine learning and continuous-time dynamical systems inspired by neuroscience.

The following sections present an overview of the core architecture and formulation of CfC models, an analysis of their performance compared to competing baselines, and a summary of their strengths and limitations from both theoretical and practical perspectives.

### 2.5.2 Architecture and Mathematical Formulation

The CfC architecture was derived by analytically approximating the solution to a scalar LTC system, as discussed in Section 2.4. The resulting model avoids numerical integration by employing a closed-form expression to update hidden states. This analytical formulation makes the CfC model computationally efficient, expressive, and inherently compatible with hardware acceleration.

Starting from a simplified version of the LTC ODE, the solution for the hidden state $x(t)$ of a single neuron with no self-connections and piecewise constant input can be approximated as:

$$x(t) \approx (x_0 - A)e^{-[w_\tau + f(I(t),\theta)]t} f(-I(t), \theta) + A, \tag{8}$$

where $A$ and $w_\tau$ are system parameters, and $f(I(t), \theta)$ is a neural network-based function of the input. This formulation describes exponential decay modulated by the input-dependent dynamics. To make this form trainable at scale and expressive enough to serve as a neural network building block, several extensions are applied.

The CfC model generalizes Equation 8 for a full layer of $D$ hidden units, each receiving $m$-dimensional input vectors. The hidden state is now updated using the expression:

$$\mathbf{x}(t) = \mathbf{B} \odot e^{-[\mathbf{w}_\tau + f(\mathbf{x},\mathbf{I};\theta)]t} \odot f(-\mathbf{x}, -\mathbf{I}; \theta) + \mathbf{A}, \tag{9}$$

Here:

- $\mathbf{x}(t) \in \mathbb{R}^D$ is the hidden state,
- $\mathbf{B}$ collapses $(x_0 - A)$ into a trainable parameter vector,
- $\mathbf{I}(t)$ is an $m$-dimensional input at each time step $t$,
- $f$ is a neural network parametrized by $\theta$,
- $\odot$ represents element-wise multiplication.

The exponential decay in Equation 9 can introduce vanishing gradients during training. To address this, Hasani et al. (2022) [6] proposed replacing the exponential term with a *time-decaying sigmoid function* $\sigma(\cdot)$, which serves as a smoother gating mechanism. The gating equation is:

$$\mathbf{x}(t) = \sigma(-f(\mathbf{x}, \mathbf{I}; \theta_f)t) \odot g(\mathbf{x}, \mathbf{I}; \theta_g) + [1 - \sigma(-f(\mathbf{x}, \mathbf{I}; \theta_f)t)] \odot h(\mathbf{x}, \mathbf{I}; \theta_h), \tag{10}$$

In this form:

- $\sigma(\cdot)$ is a sigmoid gate that interpolates between two functional branches,
- $f(\cdot), g(\cdot), h(\cdot)$ are neural networks parametrized by $\theta_f, \theta_g, \theta_h$ respectively,
- Time $t$ can vary across tokens or instances, supporting irregular sequences.

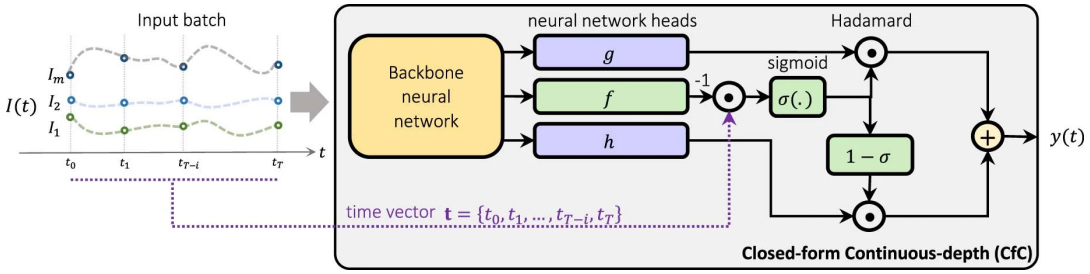



Figure 16: Closed-form Continuous-depth neural architecture [6]

To improve efficiency and stability, the CfC architecture shares the first layers of $f$, $g$, and $h$ through a *shared backbone* (Figure 16). This allows for learning a common representation while maintaining task-specific specialization in the head functions. This shared design improves training stability and fosters causal representation learning [6].

This time-continuous, gated formulation enables CfC models to reason over time with high efficiency and precision. By removing the need for ODE solvers and using analytically defined updates, CfC achieves an order-of-magnitude speedup over ODE-based networks while retaining temporal expressivity and differentiability.

### 2.5.3 Comparison with other Continuous-Time Models

Closed-form Continuous-time (CfC) Networks offer several distinct advantages over earlier continuous-time models, including CT-RNNs, Neural ODEs, and even the biologically inspired Liquid Time-Constant (LTC) networks. These advantages stem primarily from the CfC's use of analytical state updates rather than numerically integrated ODE solvers.

- **Solver-Free Computation:** Unlike Neural ODEs and LTCs, which rely on numerical integration schemes (e.g., Euler or Runge-Kutta) to compute the evolution of hidden states, CfC computes this evolution directly using a closed-form expression. This eliminates the need for iterative approximation, reducing runtime complexity and removing solver-specific tuning.

- **Hardware Efficiency:** Because CfC state updates are defined by standard algebraic operations and differentiable functions, they are naturally compatible with modern deep learning libraries and accelerators such as GPUs and TPUs. This enables CfC models to train and infer faster than ODE-based counterparts, which often suffer from poor hardware utilization due to sequential solver steps.

- **Time-Continuity with Irregular Inputs:** CfC models naturally handle irregularly sampled data by incorporating the time interval $\Delta t$ directly into their update equations. This is in contrast to fixed-step RNNs or LSTMs, which must be manually adjusted or padded to accommodate asynchronous events.

- **Improved Gradient behaviour:** The exponential term in early LTC-inspired formulations can lead to vanishing gradients when optimized via back-propagation. CfC addresses this by replacing the exponential decay with a smoother, time-decaying sigmoid function. This allows for better gradient flow and more stable training, particularly in the presence of recurrent connections.

- **Shared Backbone and Modular Design:** The CfC model architecture introduces a shared backbone that branches into multiple neural components responsible for different dynamic behaviours (e.g., interpolation, gating, non-linear state transformation). This modularity improves learning efficiency and promotes better causal representation learning [6].

- **Compact and Fast:** Empirical results from the CfC paper demonstrate that CfC models can outperform ODE-RNNs, CT-GRUs, and Transformers on several sequence modeling tasks while using fewer parameters and requiring less training time. On long-sequence tasks,

CfC models achieve up to an order-of-magnitude reduction in training and inference time, making them suitable for latency-sensitive applications.

Table 8: Comparison of CfC with prior continuous-time architectures.

| Model | Solver-Free | Time-Continuous | Hardware-Efficient | Gradient Stable |
|---|---|---|---|---|
| CT-RNN | No | Yes | No | No |
| Neural ODE | No | Yes | No | Partial |
| ODE-RNN | No | Yes | Partial | Partial |
| LTC | No | Yes | No | No |
| **CfC** | **Yes** | **Yes** | **Yes** | **Yes** |

In summary, CfC networks represent a breakthrough in continuous-time modeling by balancing expressive power with practical deployability. Their analytical nature, training stability, and runtime speed position them as a compelling alternative for sequence modeling tasks in real-world, latency-critical environments.

### 2.5.4 Applications and Empirical Results

Closed-form Continuous-time (CfC) Networks have been empirically validated across a wide range of real-world and synthetic benchmarks, demonstrating competitive performance and efficiency compared to both traditional RNNs and more complex continuous-time models. The evaluations presented by Hasani et al. [6] cover diverse domains including time-series forecasting, physiological signal classification, motion prediction, and long-sequence modeling.

In a series of comparative experiments, CfC models achieved state-of-the-art or highly competitive results across several datasets, notably:

- **Walker2D (MuJoCo):** A physics-based motion prediction task. CfC models achieved lower prediction error and faster training time than Transformer, LSTM, ODE-RNN, and LTC models.
- **ECG Classification:** CfC demonstrated strong temporal sensitivity and compactness in modeling multi-channel physiological signals.
- **Speech Commands and Human Activity Recognition:** CfC achieved near or better-than-Transformer accuracy with significantly reduced model size and training cost.
- **Sequential MNIST and Permuted MNIST:** CfC showed improved long-sequence memorization with low parameter count.

The following table, adapted from the original CfC paper, highlights CfC's performance compared to earlier architectures on the Walker2D dataset:

Compared to models such as Neural ODEs and LTCs, CfC networks provide a significant reduction in computational cost. Hasani et al. [6] report that CfC models are at least one order of magnitude faster in training time per epoch. This is particularly critical for real-time inference tasks such as robotics, autonomous vehicles, and edge computing.

CfC's architectural simplicity and temporal flexibility make it well-suited for:

- **Robotic control systems**, where decisions must be made at high frequencies under uncertain temporal dynamics.
- **Biomedical signal processing**, including electrocardiograms and wearable sensor data.
- **Autonomous navigation**, where sequential reasoning must scale to varying time steps and input delays.

Table 9: Performance comparison on the Walker2D task. Lower square error and training time per epoch are better. Adapted from [6].

| Model | Square Error ↓ | Time per Epoch (min) ↓ |
|---|---|---|
| ODE-RNN | 1.904 ± 0.061 | 0.79 |
| CT-RNN | 1.198 ± 0.004 | 0.91 |
| GRU-ODE | 1.051 ± 0.018 | 0.56 |
| CT-LSTM | 1.014 ± 0.014 | 0.31 |
| ODE-LSTM | 0.883 ± 0.011 | 0.56 |
| Transformer | 0.761 ± 0.032 | 0.80 |
| LTC | 0.662 ± 0.013 | 0.78 |
| **CfC** | **0.643 ± 0.006** | **0.08** |

- **Low-power embedded devices**, where model compactness and low inference latency are required.

In all these domains, CfC provides a compelling trade-off between expressiveness, efficiency, and hardware compatibility—qualities that are essential for next-generation time-series modeling in both academic and industrial settings.

### 2.5.5 Strengths and Limitations

Closed-form Continuous-time (CfC) Networks offer a compelling combination of theoretical elegance and practical performance. By eliminating the need for numerical integration, CfC models achieve solver-free continuous-time computation, enabling fast, stable, and hardware-efficient learning across a wide range of sequence modeling tasks. Nonetheless, like any architectural innovation, CfC comes with certain trade-offs.

To summarize, Table 10 highlights the main strengths and limitations of CfC networks discussed in this section.

Table 10: Summary of strengths and limitations of CfC networks.

| Strengths | Limitations |
|---|---|
| Solver-free, fast, closed-form state updates | Requires careful architectural tuning |
| Supports irregular time intervals | Less mature compared to LSTMs/Transformers |
| Hardware-friendly and scalable | Potentially sensitive to initialization and input scaling |
| Improved gradient flow and training stability | Reduced biological interpretability compared to LTC |
| Compact with fewer parameters | Complex interplay between gating functions |

To conclude, CfC models build on the theoretical depth of ODE-based neural networks while resolving key bottlenecks in training stability and computational efficiency. Their time-continuous design, combined with modern deep learning infrastructure compatibility, makes CfC a strong candidate for future deployment in scalable, low-latency, and adaptive temporal modeling systems.

## 2.6 Parallelization in Deep Learning

The increasing complexity and scale of deep learning models have made parallelization techniques essential for efficient training and deployment. As model sizes grow and datasets become larger, training on a single processing unit becomes computationally infeasible or prohibitively slow. To address these limitations, parallelization strategies have been developed to distribute both

computation and memory across multiple processing units—such as CPUs, GPUs, or entire nodes in a high-performance computing (HPC) cluster.

This section presents an overview of the foundational strategies and tools employed to parallelize deep learning workloads. It begins by outlining the distinction between data parallelism and model parallelism, the two primary paradigms for distributing computations. The focus then shifts to PyTorch's Distributed Data Parallel (DDP) framework, one of the most widely adopted methods for multi-GPU training in both research and production contexts. Subsequently, alternative parallelization frameworks such as Horovod are reviewed, with particular attention to their design philosophies and typical use cases. This foundational knowledge is essential for understanding the parallel execution environments examined in the experimental phase of this thesis.

### 2.6.1 Data Parallelism and Model Parallelism

Parallelization in deep learning primarily takes two forms: *data parallelism* and *model parallelism*. Both approaches aim to accelerate training by distributing the computational workload across multiple processors or machines, but they do so in fundamentally different ways.

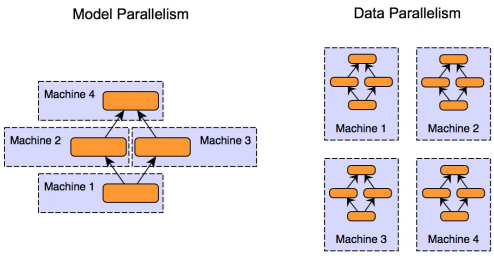



Figure 17: Model Parallelism VS Data Parallelism [45]

In **Data Parallelism**, the model architecture is replicated across all processing units (e.g., GPUs), and each replica is assigned a distinct subset of the input data. During each training step, all model replicas compute the forward and backward passes independently using their local data shard. Gradients from all replicas are then aggregated, typically via *all-reduce* operations, and averaged to update the model parameters synchronously. The updated parameters are broadcasted back to each device, ensuring consistent synchronization across the system.

Data parallelism is particularly effective when the model fits entirely within a single device's memory and the batch size can be divided evenly across devices. It is widely adopted due to its scalability and implementation simplicity, and it forms the foundation for many high-level distributed training frameworks such as PyTorch Distributed Data Parallel (DDP). [8]

**Model Parallelism**, in contrast, involves partitioning the model itself across multiple devices. Each device stores and processes only a portion of the model's parameters and is responsible for computing its part of the forward and backward passes. This is especially useful for training very large models that cannot fit into the memory of a single GPU. For example, the Transformer architecture in large language models like GPT-3 has been trained using pipeline or tensor model parallelism [46].

Model parallelism introduces additional communication overhead, as intermediate outputs (activations) must be transferred between devices. As such, it often requires more careful engineering and scheduling strategies (e.g., pipeline parallelism [47] or operator sharding). It is more complex to implement and tune than data parallelism, but it remains essential for training frontier-scale models.

In practice, hybrid approaches that combine both data and model parallelism are increasingly common. These methods divide the model across devices (model parallelism) and then replicate

that division across machines (data parallelism), enabling training of extremely large-scale networks in HPC environments [48].

Table 11 summarizes the two paradigms

Table 11: Comparison of Data and Model Parallelism.

| **Data Parallelism** | **Model Parallelism** |
|---|---|
| Model is replicated across devices | Model is split across devices |
| Input batch is divided among devices | Each device handles a segment of the model |
| Gradient averaging required | Activation communication required |
| Simpler to implement (e.g., DDP) | Complex scheduling and partitioning |
| Efficient for medium/large batch sizes | Required for extremely large models |

### 2.6.2 PyTorch DistributedDataParallel (DDP)

PyTorch Distributed Data Parallel (DDP) is one of the most widely adopted frameworks for training deep learning models across multiple GPUs and nodes. It builds on the concept of data parallelism, offering a scalable, efficient, and easy-to-use solution for distributed training. [8]

DDP works by replicating the model across multiple devices. Each device (GPU) processes a unique shard of the input data and computes the forward and backward passes independently. After back-propagation, gradients from all devices are automatically synchronized and averaged using `all-reduce` communication primitives (Figure 18). The model parameters are then updated locally on each device, ensuring that all model replicas remain in sync throughout training [8].

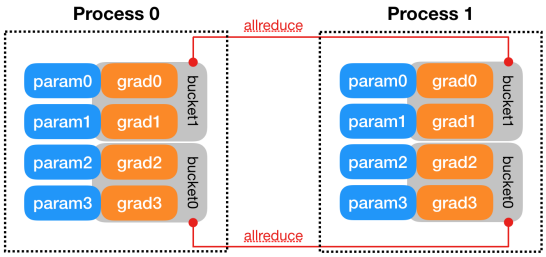



Figure 18: Gradient Synchronization with PyTorch DDP [49]

Traditional `DataParallel` in PyTorch performs gradient computation on multiple GPUs but updates the parameters only on a single device, introducing communication overhead and potential bottlenecks. In contrast, DDP performs gradient reduction during the backward pass itself, allowing updates to be fully parallel and eliminating the central bottleneck. This results in significantly better scaling efficiency, especially in multi-node environments. [49]

DDP supports multiple communication backends, including NCCL (optimized for NVIDIA GPUs), Gloo, and MPI. It also provides flexible initialization methods such as `env://`, shared file systems, and TCP-based rendezvous. Proper environment configuration (e.g., setting `MASTER_ADDR`, `MASTER_PORT`, and `WORLD_SIZE`) is essential to ensure correct group formation and communication among processes. [49]

DDP is used extensively in both academia and industry for large-scale model training, including training of Transformer-based architectures and ResNets. Its integration into high-performance computing pipelines makes it a powerful tool for accelerating deep learning research and deployment.

### 2.6.3 Other Parallelization Frameworks

While PyTorch's DistributedDataParallel (DDP) is widely adopted for multi-GPU training in PyTorch-based workflows, alternative frameworks have emerged to address different needs in distributed deep learning and performance optimization. Two prominent examples are **Horovod** and **DeepSpeed**, each offering distinct advantages in scalability, efficiency, and advanced training features.

**Horovod**, developed by Uber, is an open-source framework designed to facilitate scalable and efficient training of deep learning models across multiple GPUs and nodes [50]. Built on top of MPI (Message Passing Interface) and NCCL (NVIDIA Collective Communications Library), Horovod abstracts away much of the complexity of distributed training by providing a high-level API that integrates with popular frameworks such as TensorFlow, PyTorch, and MXNet.

The key mechanism used by Horovod is *ring-allreduce*, a communication pattern that efficiently averages gradients across devices without requiring centralized parameter servers. This enables highly scalable synchronous training across many GPUs or even across compute clusters. Unlike PyTorch DDP, which is tightly integrated into PyTorch's autograd system, Horovod works at a higher abstraction level, which makes it more portable across different deep learning libraries. Horovod also supports features such as mixed precision training, gradient compression, and fault tolerance, making it a popular choice in production environments with high scalability requirements.

**DeepSpeed**, developed by Microsoft, is a deep learning optimization library that focuses on enabling large-scale model training while reducing memory consumption and improving throughput [51]. DeepSpeed supports model parallelism, optimizer offloading, and ZeRO (Zero Redundancy Optimizer) strategies that allow the training of models with billions of parameters on modest GPU resources.

Unlike Horovod and DDP, which are primarily focused on distributing standard training workloads, DeepSpeed provides end-to-end system-level optimizations that make it especially suitable for extreme-scale models and transformer-based architectures. It includes features such as:

- *ZeRO and ZeRO-Offload*: Efficient memory partitioning across devices to enable training of large models on limited hardware.
- *Sparse attention kernels*: Custom CUDA kernels for accelerating attention mechanisms in transformer models.
- *Advanced scheduling and mixed-precision support*: Allowing faster training with reduced memory usage and improved numerical stability.

DeepSpeed integrates tightly with PyTorch and is designed for seamless scalability across GPUs and nodes, making it a strong candidate for research and industrial-scale model training.

In summary, while Horovod is aimed at simplifying distributed training across large clusters and multiple deep learning frameworks, DeepSpeed focuses on optimizing memory and computation for training extremely large models within PyTorch. Both frameworks expand the landscape of parallelization strategies and could be valuable tools in future work aiming to scale or optimize the training of LTC and CfC models beyond the capabilities of DDP alone.

## 2.7 Conclusions

This chapter reviewed both foundational and advanced models in the domain of sequential data processing, along with the parallelization strategies essential for efficient training and inference.

The discussion began with classical architectures such as Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and Gated Recurrent Units (GRUs), highlighting their strengths in modeling temporal dependencies and their limitations related to vanishing gradients and the inherently sequential nature of their computations.

The emergence of Transformer architectures was subsequently examined, marking a paradigm shift through the introduction of self-attention mechanisms and parallel computation. These innovations enabled significant improvements in modeling long-range dependencies and scalability. Nevertheless, Transformers entail substantial computational and memory overhead, especially when processing very long sequences, which can constrain their applicability in resource-limited or latency-sensitive scenarios.

To address these challenges, it is introduced the class of biologically inspired *Liquid Neural Networks* (LNNs), with particular focus on Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) architectures. These models aim to provide a lightweight and adaptable alternative, capable of handling temporal dynamics with fewer parameters and more interpretable internal states. Their continuous-time formulation and theoretical grounding offer promising directions for real-time and adaptive AI systems.

The chapter concluded with an overview of high-performance computing techniques employed in deep learning, including data parallelism, model parallelism, and the PyTorch `DistributedDataParallel` framework. Alternative parallelization frameworks such as Horovod and DeepSpeed were also introduced, with an emphasis on their relevance for scalable AI workloads.

Altogether, this state-of-the-art review provides the conceptual and technical basis for the methodology adopted in this thesis. The choice of LTC and CfC as target architectures, and the implementation of distributed training using PyTorch DDP, are directly motivated by the limitations and opportunities discussed throughout this chapter. These insights lay the groundwork for the design, implementation, and optimization strategies presented in the following chapters.

# 3. Methodology

This chapter describes the methodological framework followed in this research, detailing the architectural, computational, and experimental strategies adopted to evaluate the performance and scalability of Liquid Neural Networks. The objective was to systematically study two recent biologically inspired continuous-time neural architectures—Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC)—in the context of temporal learning tasks and high-performance computing (HPC).

The methodology underlying this study is organized into four major components (Figure 19). First, a comprehensive analysis of the two neural models was conducted, focusing on their theoretical foundations, implementation details, and architectural differences. Second, a set of datasets was selected to represent a range of temporal learning challenges—ranging from toy to large-scale, real-world applications—and appropriate preprocessing steps were applied to format them for sequence learning. Third, distributed training techniques were employed to scale up the models across CPU and multi-GPU environments using PyTorch's Distributed Data Parallel (DDP) framework. This included parallelization setup, tuning of batch size and learning rate, and hardware deployment on the BOADA HPC cluster.

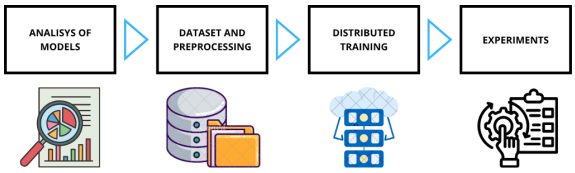



Figure 19: Steps of Methodology adopted

Finally, although the design of experiments and evaluation configurations form a central part of the methodology, they are presented separately in Chapter 4. to ensure clarity and modular organization. This division allows Chapter 3 to concentrate on the theoretical and technical foundations, while Chapter 4 offers a detailed account of the experimental configurations, training schedules, and parameter settings that guided the empirical evaluation.

Together, these methodological steps provide a structured approach for benchmarking LTC and CfC networks, with a focus not only on model accuracy but also on training efficiency, scalability, and parallel performance under real-world computational constraints.

## 3.1 Overview of the Approach

The research presented in this thesis adopts a model-driven empirical benchmarking approach to evaluate the training efficiency, scalability, and parallel performance of two biologically inspired neural network architectures: the Liquid Time-Constant (LTC) network and the Closed-form Continuous-time (CfC) network. These models belong to the emerging class of Liquid Neural Networks (LNNs), which are specifically designed for temporal and sequential learning tasks and offer promising architectural advantages such as dynamic time modeling and reduced parameter complexity.

From the outset, the objective was to investigate the behaviour of continuous-time neural models in high-performance computing environments, with a particular focus on how they respond to distributed training across CPUs and GPUs. To this end, an extensive review of literature on temporal modeling architectures was conducted to identify relevant models and trace the evolution of the field.

The first model considered was the Liquid Time-Constant (LTC) network, introduced by Hasani et al. in 2020 [5], which represents the first effective implementation of LNNs. LTCs simulate the dynamics of biological neurons through non-linear differential equations and

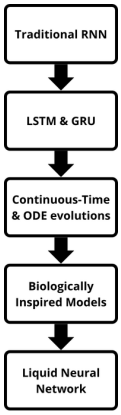



Figure 20: Overview of the models studied

input-dependent time constants, allowing them to process time-continuous data more naturally than traditional RNNs. During further investigation, a more recent work by the same authors (Hasani et al., 2022) [6] introduced the Closed-form Continuous-time (CfC) network, which builds upon LTC theory and derives an analytical solution that eliminates the need for numerical ODE solvers. This model further reduces computational overhead while maintaining strong performance on time-series benchmarks.

Given their complementary nature, one representing the original dynamic formulation and the other a more computationally efficient evolution, both LTC and CfC were selected as the central focus of this study. While their theoretical performance and predictive capabilities have been explored in literature, to date, no comprehensive evaluation of their training efficiency and behaviour under distributed computing settings has been conducted.

To fully understand these architectures, a historical analysis of related sequence models was first carried out, following the flow of Figure 20. This includes classical Recurrent Neural Networks (RNNs), which were the first to model sequential dependencies using internal recurrence, followed by their gated extensions such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU), which addressed the challenges of vanishing and exploding gradients and enabled learning of long-term dependencies. These were succeeded by continuous-time variants and ODE-based models, which treat time as a continuous variable and evolve the hidden state using neural differential equations. This progression paved the way for biologically inspired architectures like LTC and CfC, which aim to bridge the gap between temporal expressivity and computational tractability.

By narrowing the focus to LTC and CfC, this work aims to contribute a detailed comparative analysis of their behaviour in real-world training scenarios and under parallel, high-performance execution conditions.

## 3.2 Model Architectures

This work investigates and benchmarks two biologically inspired continuous-time neural architectures: the Liquid Time-Constant (LTC) network and the Closed-form Continuous-time (CfC) network. While their theoretical underpinnings and motivations were discussed in Chapter 2, this section focuses on how these models are defined, instantiated and deployed in the experimental setup.

The framework used for the entire work done is `PyTorch`. The code for the definition of the models is taken from the repository associated to the official papers by Hasani et al. that introduce the networks (LTC [5], CfC [6]).

### 3.2.1 Liquid Time-Constant (LTC) Model

The `LTC` model is implemented here using a PyTorch-compatible class structure. At the core of an LTC is the Liquid Time-Constant Cell (LTCCell), a modular recurrent unit that embodies this continuous-time modeling philosophy. An LTCCell simulates the behaviour of a neuron whose membrane potential evolves over time based on both external input and recurrent feedback. This evolution is governed by a discretized version of an ODE that integrates input-dependent currents and internal recurrent dynamics. A key feature of the LTCCell is its learnable time constant, which modulates how quickly the cell responds to stimuli. The dynamic equation that drives each LTCCell can be understood as a balance between decay, input influence, and recurrent feedback, where all components are parametrized and differentiable, enabling end-to-end training.

`LTCCell` defines the behaviour of a single recurrent unit processing one timestep. Its key parameters include:

- `wiring`: specifies the neuron connectivity pattern;
- `in_features`: dimensionality of the input;
- `input_mapping`, `output_mapping`: transformation types (typically "affine");
- `ode_unfolds`: number of iterations for solving the ODE (Euler approximation);
- `implicit_param_constraints`: if true, applies biologically inspired constraints.

Then, `LTC` is the sequence-processing wrapper built on `LTCCell` that applies a Liquid Time-Constant RNN to an input sequence. Below there is the definition:

```
class LTC(nn.Module):
    def __init__(
        self,
        input_size: int,
        units,
        return_sequences: bool = True,
        batch_first: bool = True,
        input_mapping="affine",
        output_mapping="affine",
        ode_unfolds=6,
        epsilon=1e-8,
        implicit_param_constraints=True,
    ):
```

The LTC Network is implemented as a subclass of PyTorch's `nn.Module` and is designed to be flexible in terms of architecture and connectivity. The number of input features is specified by the *input_size* parameter, while the internal structure of the network is determined by the *units* argument. This can either be an integer, indicating a fully connected hidden layer of that size, or a *Wiring* instance, allowing for the definition of sparse, structured connectivity inspired by neuroscience. The parameter *return_sequences* controls whether the cell outputs the full sequence of hidden states across time or just the final output. Additionally, the *batch_first* parameter determines whether the input tensor has its batch dimension first, which aligns with standard PyTorch conventions.

Input and output interactions with the cell are controlled by *input_mapping* and *output_mapping*, which define how signals are projected into and out of the internal state space. Typically, these mappings are set to "affine," meaning they use learnable linear transformations. The internal ODE is numerically integrated using a fixed number of sub-steps, specified by the *ode_unfolds* parameter. This allows for finer temporal resolution in modeling the evolution of the state variable, at the cost of increased computational effort. The epsilon parameter is a small constant added to prevent numerical instability during calculations, particularly in divisions involving time constants. Lastly, the *implicit_param_constraints* flag enforces biologically plausible bounds on the model's parameters, such as ensuring positivity of conductances or stability of time constants during optimization.

For this case study, a *Wiring* was used for defining the connectivity pattern of the neurons, and all the parameters were left with default value except for *return_sequences* that for some experiments was set to `False` because the interest was only on the prediction on the next step of the input sequences. Below the code that was used for build the model:

```
wiring = AutoNCP(hidden_size, out_features)
ltc_model = LTC(in_features, wiring, return_sequences=False)
```

Overall, the Liquid Time-Constant Cell and its network-level implementation provide a robust framework for modeling complex, time-varying phenomena. Their continuous-time formulation, learnable dynamics, and architectural flexibility make them especially suited for time-series learning tasks, where both short- and long-term temporal dependencies must be captured accurately and adaptively.

### 3.2.2 Continuous-time Closed-form (CfC) Model

The `CfC` model is implemented using a PyTorch-compatible class structure and serves as an efficient and interpretable alternative to traditional recurrent neural networks for modeling time-series data. Unlike standard RNNs that rely on discrete updates and numerical integration of hidden states, CfC models directly learn a closed-form approximation of continuous-time dynamics. This formulation makes them computationally efficient while maintaining temporal expressivity.

At the core of CfC is a dynamic equation that governs the evolution of the hidden state over time based on input-driven gating mechanisms and a backbone feed-forward network. The `CfCCell` does not numerically solve ODEs in the classical sense but instead approximates the continuous-time solution through a parametrized closed-form expression, thereby removing the need for iterative unfolding steps. This results in faster computation and greater interpretability, particularly useful for tasks requiring fast and stable long-horizon predictions.

Internally, the CfC cell uses a learned gating function to combine the current input with the previous hidden state, and passes this result through a feed-forward backbone. The resulting formulation approximates a continuous-time dynamic system in closed form, eliminating the need for step-based numerical integration. This approach offers improved efficiency and stability in time-series modeling.

The `CfC` class, whose definition is provided below, defines a recurrent unit capable of processing sequences through such closed-form updates.

```
class CfC(nn.Module):
    def __init__(
        self,
        input_size: Union[int, ncps.wirings.Wiring],
        units,
        proj_size: Optional[int] = None,
        return_sequences: bool = True,
        batch_first: bool = True,
        mixed_memory: bool = False,
        mode: str = "default",
        activation: str = "lecun_tanh",
        backbone_units: Optional[int] = None,
        backbone_layers: Optional[int] = None,
        backbone_dropout: Optional[int] = None,
    ):
```

Its key parameters includes:

- `input_size`: number of features in the input sequence;
- `units`: number of hidden units in the CfC cell;
- `proj_size`: if specified, adds a learnable linear projection to map the output to a desired dimension;

- `return_sequences`: controls whether the model outputs the full sequence or only the last time step;
- `batch_first`: defines whether the batch dimension is the first (0-th) in the input tensor;
- `mixed_memory`: when enabled, integrates an additional slow memory trace for capturing long-term dependencies;
- `mode`: defines the CfC variant: "default" (standard gate formulation), "pure" (direct gate-based solution), or "no_gate" (simplified model without gating);
- `activation`: sets the nonlinearity used in the backbone network (e.g., "lecun_tanh", "relu");
- `backbone_units`: number of hidden units in the feedforward backbone;
- `backbone_layers`: number of layers in the backbone;
- `backbone_dropout`: dropout probability applied to backbone layers.

For this study, the CfC model was initialized with default settings, except for `return_sequences`, which was set to `False` in experiments focusing on next-step prediction. Additionally, the activation function used was the default "lecun_tanh" due to its suitability for modeling smooth time dynamics. The model construction is shown below:

```
cfc_model = CfC(in_features, hidden_size, out_features, return_sequences=False)
```

Overall, the Closed-form Continuous-time model provides a computationally efficient and interpretable framework for sequence learning. Its ability to directly approximate continuous-time behaviour without iterative integration makes it particularly appealing for real-time and long-horizon prediction tasks. Furthermore, its gating structure and configurable backbone provide the flexibility needed for diverse sequence modeling applications.

## 3.3 Dataset and Preprocessing

To evaluate the performance and scalability of Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) neural network models under various computational and data conditions, four distinct datasets were selected: Sine-Cosine, Human Activity Recognition (HAR), Metro Interstate Traffic Volume (Traffic), and Individual Household Electric Power Consumption (IHEPC). The choice of these datasets was made to ensure comprehensive coverage across synthetic and real-world scenarios, as well as across classification and regression tasks of varying complexity and scale.

The **Sine-Cosine** dataset is a synthetic, low-dimensional toy dataset designed to serve as a sanity check for model behaviour. It contains a sequence composed of sine and cosine waveforms used as inputs, and a target output signal derived from their combination. Although simplistic, this dataset allows for fast experimentation and qualitative inspection of a model's capacity to learn basic temporal patterns. It was primarily employed to verify model correctness, training dynamics, and compatibility with the parallel training setup.

The **HAR (Human Activity Recognition)** dataset represents a real-world, medium-scale dataset collected from embedded sensors [9]. It involves a multivariate time-series classification task where each input sequence corresponds to body movement data, and the target is the type of human activity being performed. This dataset was chosen for its moderate complexity, well-defined structure, and widespread use in benchmarking temporal deep learning models. It provides a meaningful basis to assess model performance on practical classification problems and observe behaviour across different hardware settings and batch sizes.

The **Traffic** dataset, sourced from the UCI repository [10], is a large-scale, real-world time series dataset used for a regression task. It involves the prediction of vehicle volume based on various contextual features such as time of day, weather, and holiday flags. With its larger

number of samples and higher sequence length compared to the HAR dataset, it introduces challenges related to memory, scalability, and parallelization. The inclusion of this dataset was motivated by the need to test model robustness and efficiency in a high-throughput, regression-oriented setting under distributed execution conditions.

However, experimental results on the Traffic dataset revealed that performance gains when scaling from 2 to 4 GPUs were marginal or even regressive due to limited utilization and communication overheads. To further investigate this effect and better evaluate model behaviour under truly large-scale data conditions, a fourth dataset was introduced: the **Individual Household Electric Power Consumption (Power)** dataset from the UCI repository [52]. This dataset contains detailed measurements of electric power consumption from a single household, sampled every minute over nearly four years. It includes multiple electrical quantities and sub-metering values, making it a high-resolution, long-horizon, multivariate time-series regression task.

The IHEPC dataset offers a more computationally intensive scenario suitable for testing the scaling limits of distributed training. It helps assess whether additional GPUs can be effectively leveraged when the data volume and temporal context are sufficiently large, thereby providing a more stress-test-like condition for both LTC and CfC models.

Together, these four datasets provide a varied and balanced evaluation landscape, enabling the investigation of LTC and CfC architectures across toy and real-world domains, small to extremely large scales, and classification versus regression tasks. Detailed descriptions and preprocessing strategies for each dataset are presented in the following subsections.

### 3.3.1 Sine-Cosine Dataset

The sine-cosine dataset is a synthetically generated time series designed to verify the models' ability to learn basic temporal dependencies. It provides a controlled, low-noise environment for observing model convergence behaviour and prediction accuracy in a minimal setting.

**Data Generation.** The input sequence consists of two periodic signals:

- $\sin(t)$
- $\cos(t)$

These are sampled at $N = 48$ evenly spaced time steps over the interval $[0, 3\pi]$.

The target output is a sine wave with double the frequency of the input:

- $\sin(2t)$, sampled over $[0, 6\pi]$

and reshaped into a matching sequence length.

**Data Structure.**

- **Input shape:** $(1, 48, 2)$ — one sequence of 48 time steps and 2 input features.
- **Target shape:** $(1, 48, 1)$ — one sequence of 48 time steps and 1 target output.

**Preprocessing Pipeline.**

- Inputs and outputs are generated using NumPy and converted to `torch.Tensor` format.
- A batch dimension is manually added (batch size = 1).
- The dataset is loaded into a PyTorch `DataLoader` using `TensorDataset`, with shuffling enabled.

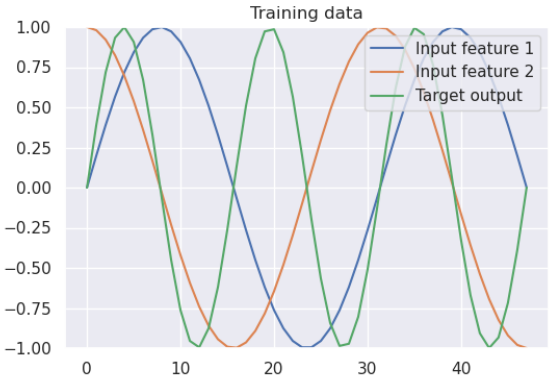



Figure 21: Input features and target output in the sine-cosine training dataset.

**Visualization.** Figure 21 shows the generated training sequence. The two input features exhibit a clear phase shift (sine and cosine), while the target output has a doubled frequency, allowing the model to learn more complex temporal relationships.

This dataset is primarily used to validate that both LTC and CfC architectures are correctly implemented and able to learn time-dependent patterns, even with minimal data and structure.

### 3.3.2 Human Activity Recognition (HAR) Dataset

The Human Activity Recognition (HAR) dataset is a widely used real-world benchmark for evaluating machine learning models on time-series classification tasks. It consists of sensor data collected from 30 volunteers performing six different activities (walking, walking upstairs, walking downstairs, sitting, standing, lying) while wearing a smartphone on their waist. The smartphone's embedded accelerometer and gyroscope were used to capture a range of motion signals at a constant rate of 50 Hz, generating a total of 561 features per time step after signal pre-processing and feature extraction.

For this study, the HAR dataset was employed as a medium-scale, realistic classification scenario to assess the ability of Liquid Neural Networks (LTC and CfC) to learn from multi-channel time-series data and generalize over human behavioural patterns. The original dataset is split into a training set with 7,352 samples and a test set containing 2,947 samples.

To make the data compatible with sequence-based models such as RNNs and Liquid Networks, the dataset was preprocessed by segmenting the time series into fixed-length sequences using a sliding window approach. Specifically, the input time-series were divided into overlapping sequences of length 16 (i.e., seq_len=16). For training data, the window was moved forward by one sample at a time (increment = 1), resulting in 7,336 sequences. For testing, a less granular stride of 8 was used (increment = 8), yielding 367 test sequences.

To further refine model evaluation, 10% of the training data was randomly selected and set aside as a validation set. After shuffling, this resulted in:

- 6,603 sequences for training,
- 733 sequences for validation,
- and 367 sequences for testing.

Each sequence has shape (16, 561) for the input features and (16,) for the corresponding activity labels, representing a continuous sequence of human activity labels aligned with the windowed input. All datasets were then converted into PyTorch tensors and wrapped in `DataLoader` objects for batched processing during training and evaluation.

This careful segmentation preserves temporal continuity and enables the models to learn transitions between different activities, making the HAR dataset a suitable choice for evaluating the performance of models designed for temporal pattern recognition.

### 3.3.3 Metro Interstate Traffic Volume (Traffic) Dataset

The Metro Interstate Traffic Volume dataset is a real-world, large-scale time-series dataset that contains hourly records of traffic volume in the Twin Cities metro area in Minnesota. The dataset, originally published on UCI Machine Learning Repository, consists of approximately 48,200 data points and includes various environmental and temporal features such as weather conditions, precipitation, temperature, holidays, and time-of-day, alongside the traffic volume target variable.

Given its continuous and high-resolution temporal nature, this dataset was selected to evaluate the models' scalability and forecasting capabilities in a regression setting. It represents a considerably larger data scale compared to the Human Activity Recognition (HAR) dataset and was thus particularly well-suited to test the models' performance under realistic deployment conditions, especially in a high-performance computing environment.

The preprocessing involved a number of domain-relevant transformations. The raw dataset was parsed to extract the following seven input features:

- Binary holiday indicator
- Normalized temperature (mean-centred)
- Rainfall and snowfall measurements
- Cloud coverage percentage
- Day of the week (as numeric)
- Hour of the day, encoded with a sinusoidal transformation to reflect cyclical daily patterns

The target variable, traffic volume, was normalized by subtracting the mean and dividing by the standard deviation. Once the input matrix was constructed, the full dataset was converted into sequences of fixed temporal length to accommodate the requirements of recurrent models. Specifically, a sequence length of 32 time steps was adopted, resulting in a three-dimensional input tensor of shape (48172, 32, 7) and a corresponding target tensor of shape (48172, 32).

These sequences were then split into training, validation, and test sets using an 70/20/10 split ratio, yielding:

- **Training set**: 33721 sequences
- **Validation set**: 9634 sequences
- **Test set**; 4817 sequences

The resulting dataset was wrapped into `DataLoader` objects to facilitate batch processing during model training and evaluation. Batches were shuffled during training and preserved in sequential order for validation and testing.

With over 48,000 multivariate sequences, this dataset offers a real-world benchmark to examine how efficiently and accurately LTC and CfC models generalize to high-volume, noisy, and irregular temporal patterns under constrained training time and memory usage.

### 3.3.4 Individual Household Electric Power Consumption (Power) Dataset

The Individual Household Electric Power Consumption dataset is a large-scale, real-world time-series dataset originally published by the UCI Machine Learning Repository. It records electric power usage in a single household over nearly four years, with a one-minute sampling rate. This fine-grained resolution results in a dataset of millions of entries, capturing variations in energy consumption due to daily routines, seasonal shifts, and appliance usage. Each entry includes multiple electrical features, such as active and reactive power, voltage, and current intensity, as well as energy sub-metering values.

This dataset was introduced into the thesis in response to the findings from earlier experiments with the Traffic dataset, which indicated limited benefits from using 4-GPU configurations due to relatively low utilization. By contrast, the Power dataset's greater volume and finer temporal resolution offer a more demanding computational challenge, better suited for assessing the full scalability of distributed training and parallel execution.

**Preprocessing:** The raw data is provided in a semicolon-separated format, with several missing values denoted by question marks. The preprocessing pipeline performs the following steps:

- Missing values are imputed using a memory-based forward fill.
- All input features are converted to floats and standardized using z-score normalization.
- The target variable is the *Global Active Power*, while the remaining six features are used as inputs.

**Sequence generation:** The preprocessed data is segmented into fixed-length sequences to accommodate the recurrent models. A sequence length of 32 time steps is used with non-overlapping windows (increment = 32), producing the following dataset sizes:

- **Training set**: 48,639 sequences
- **Validation set**: 6,485 sequences
- **Test set**: 9,727 sequences

All sequences are reshaped into tensors of shape $(\texttt{batch\_size}, 32, 6)$ for input and $(\texttt{batch\_size}, 32)$ for target output. These are wrapped in `DataLoader` objects to support efficient batched training and evaluation.

With its high-resolution temporal granularity, long recording period, and rich feature space, the Power dataset presents a rigorous benchmark for time-series forecasting models. It enables deeper evaluation of the LTC and CfC architectures under high-load scenarios and allows testing whether greater data complexity translates to improved multi-GPU efficiency.

## 3.4 Parallelization Strategy

The increasing scale and complexity of modern deep learning models, particularly those applied to temporal data such as Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) networks, demand substantial computational resources. Efficient training becomes a bottleneck, especially when experimenting across large datasets or running multiple configurations. In this context, parallel and distributed training strategies are essential to accelerate computation, improve throughput, and enable scalability.

This section outlines the high-performance computing techniques employed in this thesis to train the models under various hardware configurations. It begins with a discussion of different parallelization strategies, model parallelism, data parallelism, and hybrid approaches, justifying the use of data parallelism in our case. It then describes the implementation details of PyTorch's Distributed Data Parallel (DDP) module, including process setup, backend selection, and

integration with training pipelines. Subsequently, it discusses the critical role of batch size and learning rate scaling in distributed settings, providing rationale for the chosen hyper-parameters. Finally, it presents the hardware environment used to conduct the experiments, including a detailed overview of the university HPC cluster and its partitioned architecture.

### 3.4.1 Data Parallelism

To address the increasing computational demands of modern deep learning models and datasets, High Performance Computing (HPC) strategies have become essential. These techniques enable efficient model training by distributing workloads across multiple processing units, reducing time-to-solution and enabling scalability. In the context of neural networks, parallelization strategies fall into three primary categories: model parallelism, data parallelism, and hybrid approaches.

Model parallelism distributes the model's layers or parameters across different devices, allowing extremely large models to be trained when they cannot fit in a single device's memory. Data parallelism, on the other hand, involves replicating the model on each device and splitting the training data across these replicas. Each replica processes a different mini-batch and synchronizes gradients after back-propagation. Hybrid parallelism combines both techniques to handle large-scale models and massive datasets concurrently.

Given that the models investigated in this thesis, Liquid Time-Constant Networks (LTC) and Closed-form Continuous-time Networks (CfC), are not prohibitively large in terms of parameter size, data parallelism was selected as the most suitable strategy. This choice allows us to fully utilize the available hardware while maintaining simplicity in implementation and memory efficiency.

### 3.4.2 PyTorch Distributed Data Parallel (DDP)

Data parallelism was implemented using PyTorch's Distributed Data Parallel (DDP) module. DDP launches a separate process on each GPU, with each process maintaining its own replica of the model. At the beginning of training, model weights are broadcast from the process with rank 0 to all other processes to ensure consistent initialization. Each process then performs the forward and backward pass independently on its allocated subset of the data. After gradient computation, an all-reduce operation is used to synchronize gradients across all processes, thereby maintaining consistency among all model replicas.

The distributed setup requires initializing the process group and binding each process to its respective GPU:

```
# ======= DDP SETUP =======
rank = int(os.environ["RANK"])
local_rank = int(os.environ["LOCAL_RANK"])
world_size = int(os.environ["WORLD_SIZE"])

dist.init_process_group(backend="nccl")
device = torch.device(f"cuda:{local_rank}")
torch.cuda.set_device(device)
```

The model is then wrapped using `DistributedDataParallel`, specifying the device used:

```
model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[local_rank])
```

To ensure each process receives a distinct subset of the data, a DistributedSampler is used for each dataset loader. It partitions the data across GPUs and must be re-seeded at each epoch to enable proper shuffling:

```
sampler.set_epoch(epoch)  # Important for correct shuffling across epochs
```

**Backend Selection** PyTorch Distributed Data Parallel (DDP) supports multiple communication backends. In this work, the NCCL (NVIDIA Collective Communications Library) backend was employed, as it is highly optimized for GPU-based communication and is included

with PyTorch's CUDA-enabled builds. A brief comparison of available backends is provided below:

- **NCCL**: Best performance on CUDA tensors, with optimized collective operations. Recommended when all operations are on GPU.
- **Gloo**: More general-purpose; supports both CPU and GPU tensors and is cross-platform. It is suitable for debugging and CPU-only environments.
- **MPI**: Designed for advanced HPC environments but requires a pre-installed MPI library and additional setup. Less commonly used for deep learning tasks.

Given our reliance on multi-GPU setups and CUDA tensors, NCCL was chosen as the backend for its superior performance and ease of integration.

In summary, the use of PyTorch DDP with NCCL backend provided an efficient and scalable foundation for distributing training across multiple GPUs. This setup was crucial for reducing training times in large datasets while preserving reproducibility and consistency in gradient updates across devices.

### 3.4.3 Batch-size and Learning Rate Considerations

Batch size is a critical hyper-parameter in training deep neural networks, particularly when leveraging distributed data parallelism. It directly affects several aspects of model training, including convergence behaviour, generalization capacity, computational efficiency, and memory utilization.

Smaller batch sizes often result in noisier gradient estimates, which can be beneficial for escaping local minima and improving generalization. However, they require more frequent updates and tend to underutilize the hardware, leading to longer training times. In contrast, larger batch sizes offer better hardware efficiency and faster throughput per epoch by allowing greater parallel computation, but they can slow down convergence and risk poorer generalization if not properly tuned. [53]

In a distributed setting, batch size must be interpreted in two dimensions:

- **Local batch size**: the number of samples processed by each individual GPU.
- **Global batch size**: the total number of samples processed across all GPUs in a single forward/backward pass (i.e., local batch size $\times$ number of devices).

For example, if a batch size of 128 is used per GPU across 4 GPUs, the resulting global batch size is 512.

Different batch sizes were explored in order to evaluate their impact on model performance and scalability. The selected values aimed to balance training speed and model accuracy while remaining within the constraints of available memory. Furthermore, experiments conducted across CPU and multi-GPU configurations (1, 2, and 4 GPUs) allowed for the analysis of how increased parallelism interacts with batch size, influencing overall training efficiency and the overhead associated with gradient synchronization.

To maintain stable optimization behaviour and convergence properties across different device configurations, the learning rate must be adapted when increasing the global batch size. In this study, linear learning rate scaling is applied: the learning rate is adjusted proportionally to the number of devices used during training. If $\eta$ denotes the base learning rate for a single device, the scaled learning rate for $n$ devices is given by:

$$\eta_{\text{scaled}} = n \times \eta$$

This heuristic, commonly referred to as the linear scaling rule, compensates for the fact that more data is processed in each optimization step when multiple devices are used. Without this adjustment, the optimizer may take steps that are too small relative to the gradient magnitude derived from larger batch computations, leading to slower convergence. Empirical evidence from large-scale training also suggests that this adjustment preserves the effectiveness of the optimization process across scales. [54]
In this thesis, the learning rate was initially set to 0.001 for training with a single device. When training on 2 or 4 GPUs, this value was scaled to 0.002 and 0.004 respectively, in line with the number of processes/devices used in the Distributed Data Parallel (DDP) setting.

### 3.4.4 Hardware Configuration

All experiments in this work were conducted on the university high-performance computing (HPC) cluster **BOADA**, which consists of multiple partitions and compute nodes designed for different workloads. Specifically, three partitions were used for the execution of the experiments, based on the resource and time requirements of each setting:

- **Execution partition (boada[11–14])**: This partition provides access to CPUs with up to 20 cores per node but enforces a maximum execution time of 10 minutes. For this reason, it was used for CPU-based experiments that completed within the time limit.
- **Interactive partition (boada[18–20])**: This partition also provides CPU access, although limited to 2 cores per job. It has no strict runtime limits, making it suitable for longer CPU-based experiments that could not be completed on the execution partition.
- **CUDA partition (boada-10)**: This GPU-enabled node provides access to 5 GPUs in total, including **4 NVIDIA RTX 3080** units, which were used extensively for GPU and multi-GPU distributed training.

A summary of the available hardware resources and their usage in this thesis is provided in Table 12.

Table 12: Summary of BOADA cluster partitions used in the experiments

| Partition | Nodes | Resources | Time Limit | Usage in Experiments |
|---|---|---|---|---|
| Execution | boada[11–14] | CPU (up to 20 cores) | 10 min | CPU experiments with short execution time |
| Interactive | boada[18–20] | CPU (up to 2 cores) | 1 day | CPU experiments exceeding 10-minute limit |
| CUDA | boada-10 | 4 × RTX 3080 GPUs | 15 min (8 hours with cudabig) | GPU experiments, including single and multi-GPU configurations |

Table 13 represents the topology of the GPUs in BOADA-10 node. On the CUDA node (boada-10), the 4 RTX 3080 GPUs are arranged across two NUMA domains. GPUs 0 and 1 belong to NUMA node 0 and are directly connected via a `NODE` interconnect (within the same PCIe Host Bridge). Similarly, GPUs 2 and 3 reside on NUMA node 1 and are also linked through a fast `NODE` path. Cross-NUMA communication between GPUs on different nodes (e.g., GPU0-GPU2) must traverse the system interconnect (QPI/UPI), incurring higher latency.

Table 13: GPU Topology and Affinity on boada-10 Node

| | GPU0 | GPU1 | GPU2 | GPU3 | CPU Affinity | NUMA Affinity | GPU NUMA ID |
|---|---|---|---|---|---|---|---|
| GPU0 | X | NODE | SYS | SYS | 0–7,32–39 | 0 | N/A |
| GPU1 | NODE | X | SYS | SYS | 0–7,32–39 | 0 | N/A |
| GPU2 | SYS | SYS | X | NODE | 8–15,40–47 | 1 | N/A |
| GPU3 | SYS | SYS | NODE | X | 8–15,40–47 | 1 | N/A |

**Legend:**
**X** = Self
**NODE** = Connection traversing PCIe and interconnect within a NUMA node
**SYS** = Connection traversing PCIe and interconnect between NUMA nodes (e.g., QPI/UPI)

This topology has practical implications for performance in multi-GPU setups:

- **Intra-NUMA GPU pairs** (0–1 or 2–3) offer faster communication due to local PCIe links.
- **Inter-NUMA GPU setups** (e.g., 4-GPU training) introduce additional communication overhead due to data transfers across NUMA domains.

This architectural layout helps explain why certain multi-GPU experiments (especially 4-GPU runs) did not always yield the expected speedups: communication across NUMA boundaries introduces latency and bandwidth constraints that may offset the computational gains of adding more devices.

## 3.5 Profiling and Performance Diagnostics

To gain a deeper understanding of the computational characteristics of the LTC and CfC models, and to identify potential bottlenecks in their execution pipelines, a dedicated profiling phase was conducted. This diagnostic step was critical for validating theoretical assumptions about model complexity and for interpreting the performance differences observed in training time and resource usage.

**CPU Profiling.** For CPU-based executions, Python's built-in `cProfile` module was used to collect detailed statistics on function calls and execution time. This tool provided insight into where the most computational effort was concentrated during training.

In the case of the **LTC model**, the profiling results confirmed that the most computationally expensive component was the `ode_solver` routine, which is responsible for unfolding the differential equations over time. This function dominates the runtime following the backward pass, as expected due to LTC's numerical ODE integration performed at each time step.

For the **CfC model**, the profiling revealed a different pattern. The highest execution cost was incurred inside the forward pass, specifically within a linear transformation layer used in the model's backbone. This behavior is consistent with CfC's analytical closed-form solution, which avoids ODE integration but includes a dense feedforward network that is executed at every time step. The dominant cost, in this case, stems from matrix multiplications typical of fully connected layers.

**GPU Profiling.** To analyze GPU performance, two tools were employed: `nvidia-smi` and `torch.profiler`.

`nvidia-smi` was run at high frequency (every 0.2 seconds) during training to track GPU utilization and memory usage over time. This monitoring allowed us to observe GPU load distribution in distributed settings and detect underutilization or memory bottlenecks in specific configurations, such as when training across four GPUs with small batch sizes.

For a more granular breakdown of kernel-level activity and overhead, `torch.profiler` was used. This tool enabled inspection of CUDA operations, synchronization points, and time spent

in data transfer or collective operations (e.g., `all-reduce` in DDP). It provided valuable evidence of where latency accumulates in multi-GPU training, helping to explain why configurations with more GPUs did not always yield better speedups. Specifically, CfC exhibited minimal overhead, consistent with its lightweight structure, while LTC showed increased synchronization and compute cost, particularly during ODE-related operations.

**Summary.** The profiling analysis served to validate the computational intuition behind each model. LTC incurs higher per-time-step cost due to numerical integration, whereas CfC concentrates its computation in linear transformations within the forward pass. GPU profiling confirmed that distributed training benefits depend on workload granularity and model architecture, reinforcing the need for thoughtful batch size and hardware configuration choices. These insights complement the empirical findings and help guide optimization strategies in future implementations.

# 4. Experimental Design

The objective of this chapter is to detail the systematic approach adopted to empirically evaluate the training performance, scalability, and efficiency of the two Liquid Neural Network architectures, Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC). In particular, the chapter outlines the design choices behind the experimental protocol, the rationale for selected configurations, and the methodology used to ensure fair and reproducible comparisons across models, datasets, and hardware settings. Almost all the experiments designed were performed in all possible configurations of devices adopted: CPU, single-GPU, 2-GPUs, and 4-GPUs (Figure 22).

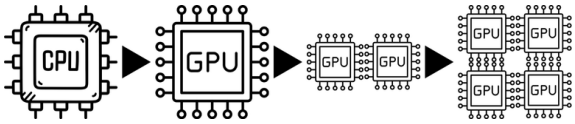



Figure 22: Configurations of devices adopted

Overall, this chapter forms the backbone of the empirical investigation, providing the structure and rigour necessary to interpret the results presented in the following chapter.

## 4.1 Objectives of the Experiments

The objective of this experimental design is to systematically evaluate the training behaviour, computational efficiency, and scalability of two state-of-the-art continuous-time neural architectures: Liquid Time-Constant (LTC) networks and Closed-form Continuous-time (CfC) networks. These models are assessed across diverse datasets of varying sizes and complexities, namely, a synthetic sine-cosine dataset (small), the Human Activity Recognition (HAR) dataset (medium), the Traffic dataset and the Power dataset (large-scale).

The primary focus of the experimentation is twofold. First, to determine how each model performs in terms of predictive accuracy and convergence when trained under different batch sizes and hardware configurations. Second, to measure the scalability and parallelization efficiency of the models when trained using PyTorch `DistributedDataParallel` (DDP) across multiple GPUs. Key computational metrics such as training time, speedup, GPU utilization, and memory consumption are also analysed to understand the trade-offs introduced by parallel training.

Through this design, the study aims to answer the following core questions:

- How do LTC and CfC compare in terms of predictive accuracy across datasets?
- How does training time evolve with increasing GPU parallelism?
- What is the impact of batch size on model convergence and efficiency?
- To what extent does DDP effectively distribute computation across GPUs?
- Do these biologically inspired architectures scale favourably on modern high-performance hardware?

The answers to these questions will serve as the empirical basis for the analysis and discussion in subsequent chapters, shedding light on the practical feasibility of deploying LTC and CfC models in real-world, resource-constrained, or time-sensitive environments.

## 4.2 Experimental Workflow

To ensure consistency, reproducibility, and clarity in the evaluation process, all experiments in this thesis followed a standardized workflow composed of well-defined stages. This section describes the complete pipeline adopted for running and evaluating the models, from dataset

preparation to metric collection. The workflow applies to both models under study (LTC and CfC) and across all datasets and hardware configurations.

The experimental process was divided into the following steps:

1. **Dataset Loading and Preprocessing** Raw datasets were loaded and transformed into time-series sequences with fixed lengths, normalized where needed, and split into training, validation, and test subsets. These datasets were wrapped into PyTorch `DataLoader` objects to support batched and shuffled access.

2. **Model Instantiation** For each experiment, either the LTC or CfC model was initialized using the appropriate input size and architecture configuration. For distributed runs, models were wrapped in `DistributedDataParallel (DDP)` with per-device replica initialization.

3. **Pre-training Evaluation** Before any training occurred, the randomly initialized models were evaluated on the test set. This step served as a baseline to verify the learning capacity of the model during training and confirm that output values are initially unstructured.

4. **Training Loop Execution** Training was performed for a fixed number of epochs, that was 50 in this case. At each epoch, the following operations were carried out:

   - Forward pass on training data
   - Backward pass and gradient synchronization (in distributed mode)
   - Parameter updates using the Adam optimizer
   - Evaluation on validation and test sets, storing metrics computed

   In multi-GPU settings, a `DistributedSampler` was used to partition the training data across GPUs, with epoch-level reshuffling via `sampler.set_epoch(epoch)` to maintain randomness

5. **Post-training Evaluation** Once training completed, the model was evaluated again on the test set to assess improvements in performance. These results were logged for comparison across configurations.

6. **Resource Monitoring and Metric Logging** During distributed GPU training, GPU usage and memory utilization were tracked via periodic sampling of the `nvidia-smi` command (every 0.2 seconds). Training times were measured using wall-clock timing. All accuracy, loss, and other metrics were stored for later aggregation.

The diagram in Figure 23 summarizes the complete experimental flow. This standardized pipeline enabled consistent execution of all experimental configurations and allowed for scalable deployment across the BOADA HPC cluster.

This systematic approach ensured fair comparison between LTC and CfC models, across datasets and hardware configurations, while providing a robust foundation for the performance analysis discussed in later chapters.

## 4.3 Evaluation metrics

To assess the performance, scalability, and computational efficiency of the studied models (LTC and CfC), a diverse set of evaluation metrics was employed. These metrics were selected to cover both predictive performance (e.g., accuracy, loss) and computational behaviour (e.g., training time, hardware utilization). This section provides a description of each metric, its formulation, relevance to the task, and how it was applied in different experimental contexts.

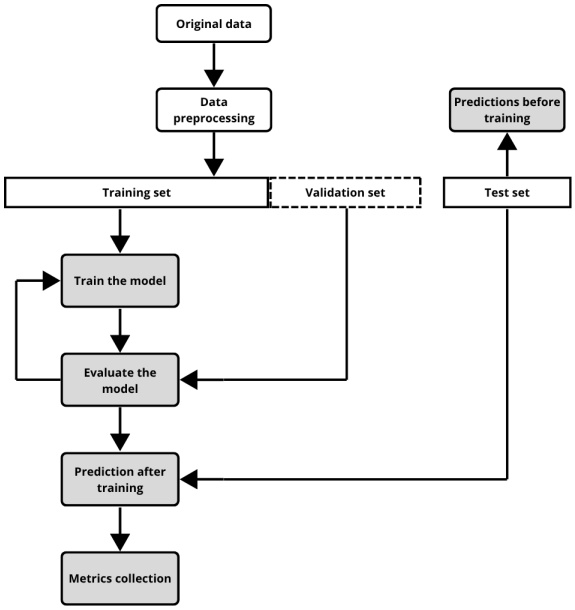



Figure 23: Experimental workflow followed across all experiments

**Loss Functions** Loss functions were used to quantify the difference between model predictions and ground-truth targets during training and evaluation. Depending on the nature of the task (classification or regression), different loss functions were adopted:

- **Cross Entropy Loss** Applied to the HAR dataset, a multi-class classification problem. The Cross Entropy Loss measures the dissimilarity between the predicted probability distribution and the actual class labels. It is defined as:

$$\mathcal{L}_{CE} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

  where $C$ is the number of classes, $y_i$ is the ground-truth label (one-hot encoded), and $\hat{y}_i$ is the predicted probability for class $i$

- **Mean Squared Error (MSE) Loss** Used for Sine-Cosine and Traffic datasets, both of which are regression problems. MSE computes the average squared difference between predicted and actual values:

$$\mathcal{L}_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  where $N$ is the number of predictions, $y_i$ is the true value, and $\hat{y}_i$ is the model prediction.

These loss values were recorded after each epoch on training, validation, and test sets to monitor convergence and generalization performance.

**Performance Metrics** To complement loss functions, additional metrics were employed to offer more interpretable insights into model behaviour and application-specific performance:

- **Accuracy** Used in the HAR classification task to measure the proportion of correctly classified activity sequences. It is defined as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

  This metric is intuitive and directly relates to model utility in real world classification scenarios.

- **Mean Absolute Error (MAE)** Used for the Traffic dataset as a regression accuracy metric. MAE (also known as L1 loss) captures the average absolute difference between predicted and actual values:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

  Compared to MSE, MAE is less sensitive to outliers and offers a more interpretable error scale.

**Training Time** Training time was measured for all experiments using wall-clock duration. It captures the total elapsed time required to complete the full number of training epochs for a given configuration. This metric is crucial for evaluating model efficiency and the impact of hardware acceleration and parallelism.

Training times were recorded for:

- Single CPU execution
- Single-GPU execution
- Multi-GPU execution (2 and 4 GPUs, using DDP)

Comparative analysis of training times across hardware configurations reveals the effectiveness of parallelism strategies and resource scaling.

**Hardware Utilization** In distributed training settings, two additional metrics were computed to evaluate hardware efficiency:

- **GPU Utilization** Represents the percentage of time each GPU was actively performing computations. It reflects how effectively the model and batch size are leveraging GPU resources. GPU usage was sampled every 0.2 seconds using the `nvidia-smi` utility and then averaged across time and devices.
- **Memory Utilization** Measures the percentage of available GPU memory occupied during training. Like GPU utilization, memory usage was sampled at 0.2-second intervals and averaged across devices. This helps determine if memory constraints are limiting scalability or model size.

Both metrics are valuable in identifying bottlenecks, under-utilized resources, or inefficient configurations (e.g., overhead from synchronization or small batch sizes).

**speedup** speedup quantifies the relative performance gain of using parallel hardware (GPUs) compared to a baseline single-CPU run. It is defined as:

$$\text{Speedup}(n) = \frac{T_{\text{CPU}}}{T_n}$$

where $T_{\text{CPU}}$ is the training time on CPU, and $T_n$ is the training time on $n$ GPUs. This metric is essential to evaluate the scalability of the models and the effectiveness of parallelism. Ideally, speedup grows linearly with the number of devices, but practical overheads may reduce this gain.

## 4.4 Experimental Matrix

To evaluate the scalability, performance, and parallel efficiency of the LTC and CfC models, a diverse and comprehensive set of experiments was conducted. These experiments span four datasets, Sine-Cosine (toy), HAR (medium-sized, classification), Traffic, and Power (large-scale, regression), and explore execution across different hardware setups including CPU, GPU, and multi-GPU configurations (2 and 4 GPUs). For each configuration, multiple batch sizes were tested to analyse the impact on training dynamics and performance metrics.

Table 14 summarizes the experimental settings. In all cases, 50 training epochs and an initial learning rate of 0.001 were used. For multi-GPU runs, the learning rate was scaled linearly with the number of devices to maintain optimization stability. This matrix provides a structured foundation for the results and analysis presented in Chapter 5.

Table 14: Summary of experiments across datasets, models, hardware settings, and batch sizes.

| **Dataset** | **Model** | **Hardware** | **Batch Sizes** | **Notes** |
|---|---|---|---|---|
| Sine-Cosine | LTC | CPU, GPU, 2-GPU, 4-GPU | 1 | Toy dataset for model sanity check |
| | CfC | CPU, GPU, 2-GPU, 4-GPU | 1 | Synthetic input/output wave patterns |
| HAR | LTC | CPU, GPU | 64, 128, 256, 512 | Classification task with 561 features |
| | LTC | 2-GPU (DDP) | 64, 128, 256, 512 | LR scaled to 0.002 |
| | LTC | 4-GPU (DDP) | 64, 128, 256, 512 | LR scaled to 0.004 |
| | CfC | CPU, GPU | 64, 128, 256, 512 | Same as above |
| | CfC | 2-GPU (DDP) | 64, 128, 256, 512 | LR scaled to 0.002 |
| | CfC | 4-GPU (DDP) | 64, 128, 256, 512 | LR scaled to 0.004 |
| Traffic | LTC | CPU, GPU | 64, 128, 256, 512 | Regression task, 7 features, seq_len=32 |
| | LTC | 2-GPU (DDP) | 512, 1024, 2048 | LR scaled to 0.002 |
| | LTC | 4-GPU (DDP) | 512, 1024, 2048 | LR scaled to 0.004 |
| | CfC | CPU, GPU | 64, 128, 256, 512 | Same as above |
| | CfC | 2-GPU (DDP) | 512, 1024, 2048 | LR scaled to 0.002 |
| | CfC | 4-GPU (DDP) | 512, 1024, 2048 | LR scaled to 0.004 |
| Power | LTC | 2-GPU (DDP) | 512, 1024, 2048 | LR scaled to 0.002 |
| | LTC | 4-GPU (DDP) | 512, 1024, 2048 | LR scaled to 0.004 |
| | CfC | 2-GPU (DDP) | 512, 1024, 2048 | LR scaled to 0.002 |
| | CfC | 4-GPU (DDP) | 512, 1024, 2048 | LR scaled to 0.004 |

# 5. Analysis of the Results

This chapter presents the empirical results obtained from evaluating the Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) neural network models under a variety of experimental configurations. The main goal of this analysis is to assess the performance, efficiency, and scalability of these models when applied to different datasets and deployed on different hardware platforms, including CPUs and GPUs in both single and distributed settings.

To address the research objectives outlined in the previous chapters, the experiments were designed to explore how each model behaves in terms of training speed, memory and GPU utilization, prediction accuracy, and scalability across batch sizes and hardware configurations. The evaluation includes both qualitative and quantitative comparisons, aimed at understanding not only which model performs better in a given context, but also why those differences emerge.

The results are organized by dataset to ensure clarity and relevance. The Sine-Cosine dataset, being synthetic and small, serves as a basic sanity check to verify the correct learning behaviour of the models. The Human Activity Recognition (HAR) dataset represents a realistic classification task on a medium-sized dataset, while the Metro Interstate Traffic Volume dataset and the Individual Household Electric Power Consumption datasets provides a large-scale regression scenario to test the models under more demanding computational conditions.

For each dataset, the chapter presents an analysis of prediction accuracy or regression loss, training time, scalability through speedup measurements, and hardware efficiency through GPU and memory utilization. Comparative insights between LTC and CfC are highlighted throughout the chapter to assess their respective strengths, limitations, and suitability for high-performance applications.

## 5.1 Sine-Cosine Dataset

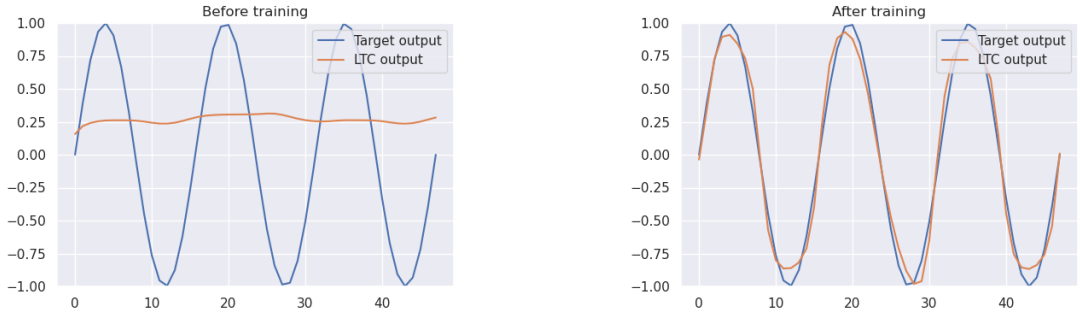
The sine-cosine dataset serves as a minimal synthetic benchmark for evaluating whether the LTC and CfC models are capable of learning a simple periodic mapping. It consists of two input channels (a sine and cosine wave with the same frequency but phase-shifted) and one output channel (a sine wave with double frequency). This setting is particularly useful for initial model verification and interpretability assessment, to check if the model can effectively learn something, rather than for evaluating scalability or generalization.

### 5.1.1 Prediction behaviour Before and After Training

Figures 24 and 25 show the outputs of the LTC and CfC models before and after training, compared to the target signal. Both models begin with flat, nearly constant predictions due to untrained weights. After training, they learn the underlying mapping: LTC captures the overall waveform, while CfC aligns nearly perfectly with the target, demonstrating a faster convergence and better approximation on this toy dataset.

### 5.1.2 Training Time Analysis

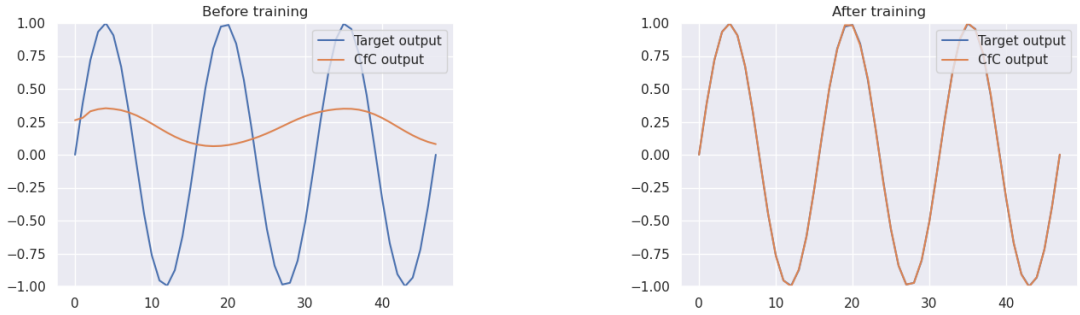
While the sine-cosine dataset is too small to benefit meaningfully from GPU acceleration, training experiments across CPU, 1 GPU, 2 GPUs, and 4 GPUs for both LTC and CfC models were conducted. The results are shown in Figure 26.




(a) LTC prediction before training

(b) LTC prediction after training

Figure 24: LTC model predictions on the sine-cosine dataset.




(a) CfC prediction before training

(b) CfC prediction after training

Figure 25: CfC model predictions on the sine-cosine dataset.

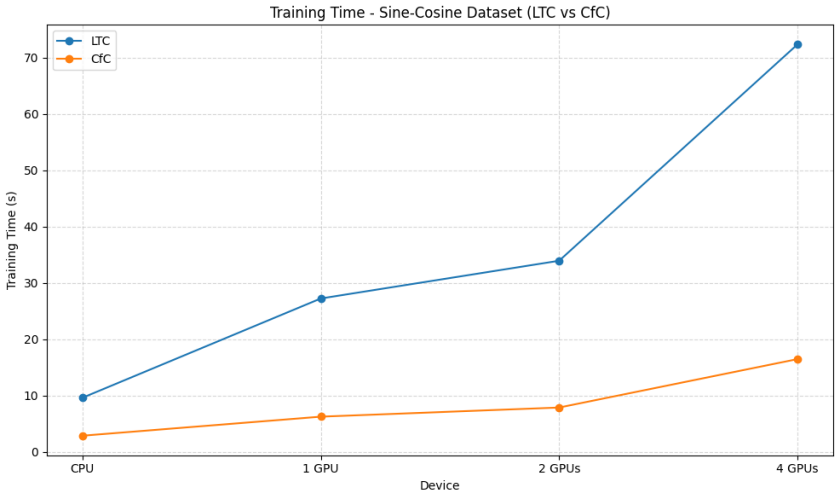



Figure 26: Training time for LTC and CfC on the sine-cosine dataset across devices.

It is observed that training time increases with more GPUs due to parallelization overhead dominating the runtime on such a small dataset. Notably, LTC is consistently slower than CfC across all hardware configurations. This highlights a key strength of CfC: its closed-form computations yield faster forward and backward passes, even in low-data regimes.

### 5.1.3 Summary and Insights

Despite the simplicity of the task, this experiment confirms that both LTC and CfC models are able to learn smooth periodic functions. However, CfC converges more quickly and with higher fidelity, as evidenced by the near-perfect output alignment and lower training time.

Moreover, the results demonstrate that GPU acceleration introduces unnecessary overhead for small-scale problems like sine-cosine. This motivates our use of larger, real-world datasets (e.g., HAR and Traffic) in subsequent sections to better evaluate the benefits of multi-GPU execution and distributed parallelism.

## 5.2 Human Activity Recognition (HAR) Dataset

This section presents the experimental results obtained from applying the Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) neural network models to the Human Activity Recognition (HAR) dataset. As a medium-sized benchmark composed of multivariate sensor time-series data, HAR provides a suitable testbed for evaluating the effectiveness of continuous-time models under moderate data and computational loads.

The objective of this section is twofold: first, to assess how model performance—measured in terms of classification accuracy and loss—varies with different batch sizes and hardware configurations; and second, to analyze the training efficiency and hardware utilization when scaling across multiple GPUs. The experiments are performed using both CPU and GPU-based training environments, including setups with 1, 2, and 4 GPUs using Distributed Data Parallel (DDP) training.

The section is structured to present a detailed comparison between LTC and CfC models. Results are organized into subsections that separately analyze test accuracy, test loss, training time, GPU/memory usage, and model scaling behaviour. Each model is examined in terms of its ability to maintain predictive quality across increasing batch sizes and to scale efficiently under distributed training. In addition to quantitative metrics, visualizations such as tables and plots are provided to support the comparative analysis and highlight key performance trends.

### 5.2.1 Accuracy and Loss Analysis

In this section, it is examined the classification performance of the LTC and CfC models on the HAR dataset in terms of test accuracy and test loss across various batch sizes and hardware configurations. The results help assess how scalable and consistent each model is under both resource-constrained (CPU) and parallelized (multi-GPU) environments.

**CPU Performance.** On CPU, both models show high classification accuracy, with CfC slightly outperforming LTC in general, as expected. LTC achieves a peak accuracy of 96% at a batch size of 64, but its performance drops to 92% at batch size 512, indicating that it may be more sensitive to large batch sizes (Table 15). Conversely, CfC demonstrates more robust behaviour, improving accuracy up to 97% at batch size 256 and maintaining 96% at batch size 512 (Table 16). Similarly, CfC consistently achieves lower test loss values, highlighting its superior convergence characteristics on this dataset.

Table 15: LTC CPU Accuracy and Loss on HAR Dataset

| Batch Size | Test Accuracy | Test Loss |
|---|---|---|
| 64 | 0.96 | 0.15 |
| 128 | 0.95 | 0.19 |
| 256 | 0.95 | 0.14 |
| 512 | 0.92 | 0.32 |

Table 16: CfC CPU Accuracy and Loss on HAR Dataset

| Batch Size | Test Accuracy | Test Loss |
|---|---|---|
| 64 | 0.94 | 0.22 |
| 128 | 0.96 | 0.17 |
| 256 | 0.97 | 0.13 |
| 512 | 0.96 | 0.14 |

**Multi-GPU Consistency.** Across all GPU configurations (1-GPU, 2-GPU, and 4-GPU), both models maintain stable accuracy and loss metrics, demonstrating that parallelization does not adversely affect predictive quality. Notably, CfC maintains accuracy above 95% for all batch sizes and GPU settings, showing better consistency than LTC. Loss values for both models remain within a small range of deviation, but CfC again shows slightly better stability, particularly at higher batch sizes, where LTC exhibits some performance degradation. Table 17 and Table 18 show the results.

Table 17: LTC Accuracy and Loss across Devices on HAR Dataset

(a) Test Accuracy

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 64 | 0.96 | 0.95 | 0.96 | 0.95 |
| 128 | 0.95 | 0.94 | 0.95 | 0.95 |
| 256 | 0.95 | 0.95 | 0.95 | 0.95 |
| 512 | 0.92 | 0.94 | 0.94 | 0.94 |

(b) Test Loss

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 64 | 0.15 | 0.20 | 0.15 | 0.17 |
| 128 | 0.19 | 0.18 | 0.16 | 0.16 |
| 256 | 0.14 | 0.15 | 0.15 | 0.14 |
| 512 | 0.32 | 0.30 | 0.28 | 0.22 |

Table 18: CfC Accuracy and Loss across Devices on HAR Dataset

(a) Test Accuracy

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 64 | 0.94 | 0.96 | 0.96 | 0.96 |
| 128 | 0.96 | 0.96 | 0.96 | 0.96 |
| 256 | 0.97 | 0.95 | 0.96 | 0.95 |
| 512 | 0.96 | 0.96 | 0.96 | 0.96 |

(b) Test Loss

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 64 | 0.22 | 0.15 | 0.19 | 0.18 |
| 128 | 0.17 | 0.18 | 0.20 | 0.17 |
| 256 | 0.13 | 0.16 | 0.14 | 0.14 |
| 512 | 0.14 | 0.14 | 0.13 | 0.16 |

**Observations** When increasing the batch size, CfC benefits more clearly from larger batches, especially on CPU, achieving its best performance at batch size 256. However, LTC shows its best loss at batch size 256 but a notable decline in accuracy at 512. This contrast indicates that CfC may better utilize larger batch sizes without compromising accuracy or overfitting.

Overall, CfC exhibits stronger performance stability across both CPU and GPU devices and across varying batch sizes. Not only does it achieve higher accuracy and lower loss in most configurations, but also responds better to the increased throughput introduced by multi-GPU training, making it more scalable and resilient for real-world deployment scenarios.

### 5.2.2 Training Time on CPU and GPU

This subsection offers a consolidated comparison of the LTC and CfC models with respect to training time as batch size increases, focusing separately on CPU and GPU execution. Figures 27 and Figure 28 provide a direct visual comparison, enabling us to analyse scaling behaviour and absolute training efficiency of both models under different hardware configurations.

**CPU Execution.** As illustrated in Figure 27, CfC significantly outperforms LTC in training time across all batch sizes on CPU. While LTC requires over 1400 seconds even for small batch sizes (and reaches nearly 2000 seconds at batch size 512), CfC completes training in under 120 seconds for all tested batch sizes. The gap between the two models becomes increasingly pronounced as batch size grows, highlighting CfC's superior efficiency and its ability to scale better on CPU. This can be attributed to CfC's closed-form temporal modeling, which avoids the per-step integration overhead characteristic of LTC models.

**GPU Execution.** A similar trend is observed in GPU training (Figure 28). LTC experiences a steep decline in training time as batch size increases, from over 500 seconds at batch size 64 to just over 75 seconds at batch size 512. CfC, in contrast, starts at a lower baseline (183 seconds for batch size 64) and quickly drops to 25 seconds by batch size 512. The relative speedup is less dramatic for CfC than LTC, but its absolute training time remains significantly lower. This suggests that while LTC benefits more aggressively from GPU acceleration, CfC is intrinsically more lightweight and efficient.


Figure 27: Training time on HAR (CPU) for LTC and CfC as batch size varies.

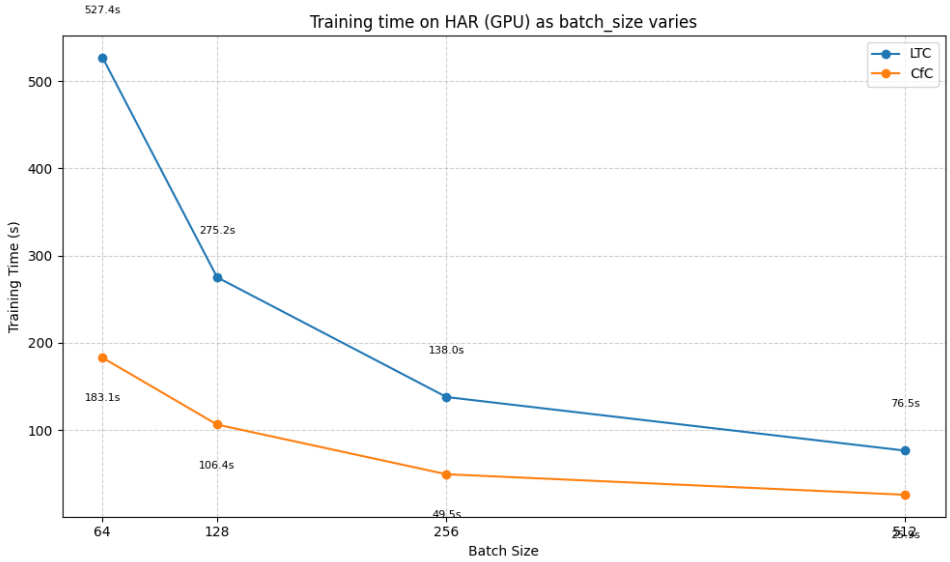



Figure 28: Training time on HAR (1-GPU) for LTC and CfC as batch size varies.

In summary, CfC is consistently faster than LTC in both CPU and GPU settings across all batch sizes. The performance gap is particularly evident on CPU, where LTC's reliance on numerical integration imposes substantial overhead. On GPU, LTC narrows the gap due to better parallel utilization and reduced time per integration step, but CfC still holds a clear efficiency advantage. These results affirm that CfC is the more suitable model in environments where training time is critical or hardware resources are limited.

### 5.2.3 Training Time on multi-GPU and Speedup

A central objective of this study is to evaluate the scalability of LTC and CfC models across different hardware configurations with respect to training time. To this end, the total training duration for both models is measured using various batch sizes and execution setups (CPU, 1-GPU, 2-GPU, and 4-GPU). These measurements offer insights into the impact of parallelization strategies on training efficiency. In addition, the relative speedup with respect to the CPU baseline is computed to quantify the benefits of GPU acceleration. The resulting analysis supports the identification of optimal trade-offs between training speed and computational cost for each model architecture.

Table 19 and Table 20 summarize the total training time (in seconds) for the LTC and CfC models, respectively, across CPU, single GPU, and multi-GPU (2 and 4 GPUs) configurations. As expected, both models benefit from GPU acceleration, with training times decreasing when GPUs are utilized. However, the extent of this speedup varies significantly between models.

Figure 29 and Figure 30 visualize the speedup achieved by GPU-based executions compared to the CPU baseline. For the LTC model, speedup scales rapidly with batch size, reaching a peak of 38× with 2 GPUs at batch size 512. However, the speedup gain begins to saturate, or even reverse, when moving from 2 to 4 GPUs. Specifically, training time for both models on 4 GPUs is slightly higher than on 2 GPUs across all batch sizes. This counterintuitive behaviour can be attributed to communication overhead introduced by Distributed Data Parallel (DDP) synchronization. In multi-GPU setups, especially across more devices, gradient averaging and inter-process communication must occur at every backward pass. As the number of GPUs increases, the cost of these synchronization steps grows—sometimes outweighing the benefits of added compute capacity, particularly if the per-GPU workload becomes too small. This effect is most noticeable at smaller batch sizes, where each GPU receives fewer samples, leading to underutilization and relatively higher synchronization overhead per data point. Even at larger

batch sizes, the benefit of parallel computation on 4 GPUs may plateau due to diminishing returns in parallel efficiency. Therefore, while increasing GPU count can accelerate training, it must be balanced with sufficient batch size and compute-to-communication ratio to avoid performance regression.

By contrast, CfC shows a more moderate and stable speedup curve. Its maximum speedup is observed with 2 GPUs at batch size 512, achieving 3× over CPU. The CfC model's overall lower memory and compute footprint results in less dramatic gains from parallelism but contributes to its better stability and balance across configurations. Interestingly, CfC demonstrates some inefficiencies at small batch sizes on GPUs, likely due to GPU underutilization and fixed overheads dominating per-sample compute time.

Table 19: Training Time (s) for LTC on HAR Dataset

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 64 | 1460 | 527.4 | 327.9 | 371.1 |
| 128 | 1426 | 275.2 | 165.7 | 188.8 |
| 256 | 1514 | 138.0 | 86.44 | 106.0 |
| 512 | 1974 | 76.53 | 51.87 | 63.58 |

Table 20: Training Time (s) for CfC on HAR Dataset

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 64 | 115.3 | 183.1 | 81.90 | 98.54 |
| 128 | 76.76 | 106.4 | 45.36 | 52.21 |
| 256 | 58.57 | 49.48 | 26.10 | 32.15 |
| 512 | 48.41 | 25.92 | 16.03 | 21.28 |

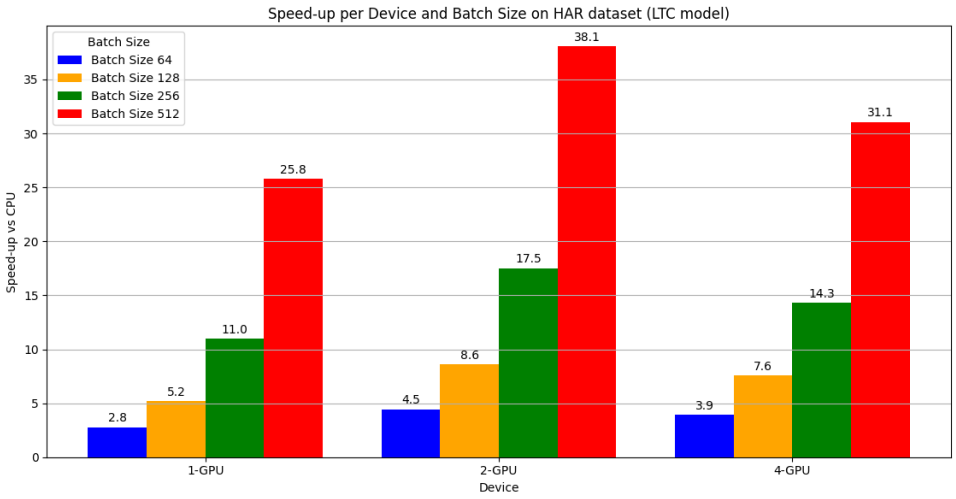



Figure 29: Speedup per Device and Batch Size for LTC model on HAR dataset (relative to CPU baseline).

These observations highlight a fundamental difference between the two models: while LTC achieves higher absolute speedup, its scalability is more sensitive to system overhead and batch size tuning. CfC, in contrast, scales more predictably and benefits from GPU resources without

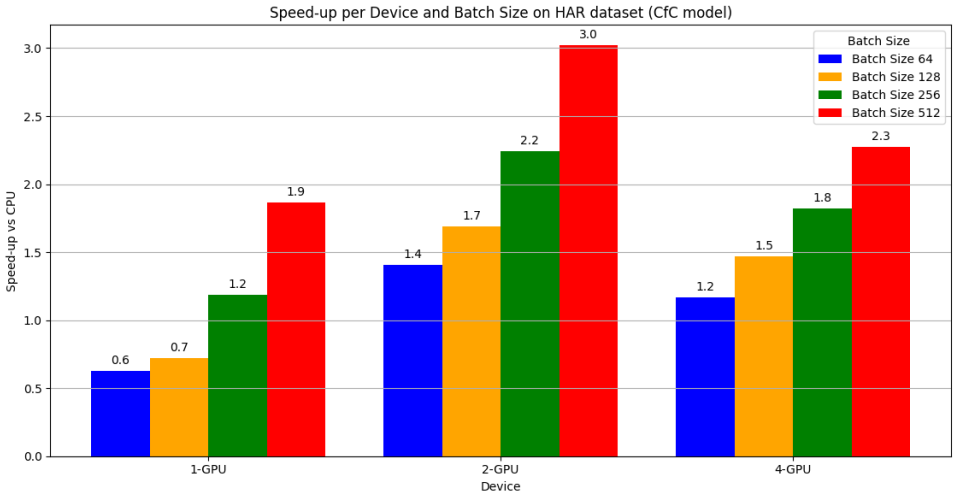



Figure 30: Speedup per Device and Batch Size for CfC model on HAR dataset (relative to CPU baseline).

significant configuration sensitivity. This makes CfC a more hardware-efficient model, especially in scenarios where GPU availability is limited or batch size cannot be arbitrarily increased.

### 5.2.4 GPU and Memory Utilization

To complement the training time and speedup analysis, this subsection examines GPU utilization and memory usage when executing the LTC and CfC models using 2-GPU and 4-GPU configurations. This analysis provides further insight into the efficiency of hardware resource usage, and helps explain some of the performance trends observed earlier.

**LTC Model.** As shown in Tables 21 and 22, the LTC model shows a clear increase in GPU usage as batch size increases. On 2-GPU setups, utilization grows from approximately 20% to over 33% as the batch size goes from 64 to 512. On 4 GPUs, the pattern is similar, although utilization per GPU is slightly lower due to the increased communication overhead and the smaller workload assigned to each GPU.

This behaviour aligns with expectations: LTC involves ODE integration steps at each timestep, which become increasingly compute-heavy as batch size grows. Therefore, larger batches better amortize the overhead and lead to improved GPU usage. However, the memory usage is also quite higher for LTC, rising to over 25% on 2 GPUs and 23% on 4 GPUs at batch size 512. This reflects the model's need to store intermediate states during integration, contributing to its higher training cost.

**CfC Model.** In contrast, Tables 23 and 24 reveal that the CfC model maintains a remarkably low and stable memory footprint across all batch sizes and GPU configurations. Memory usage remains around 3.5–4% even at batch size 512, indicating that CfC is much more lightweight in terms of model state and buffer requirements.

GPU utilization for CfC is also modest, generally ranging from 9–13% across all runs. This supports the earlier observation that CfC training times are shorter but do not scale as dramatically with more GPUs. The model's closed-form temporal computation requires fewer intermediate operations, resulting in lower GPU load and making CfC particularly suitable for environments where memory constraints or compute efficiency are important.

The comparison between LTC and CfC highlights a trade-off between *parallel scalability* and *resource efficiency*. LTC achieves higher GPU utilization and benefits more from increased batch size and multi-GPU setups, but at the cost of greater memory usage and sensitivity to overhead.

CfC, while not pushing GPU hardware to its limits, is consistently lightweight and well-balanced, offering a stable training profile across hardware settings.

Moreover, the symmetry observed in GPU usage and memory allocation across devices confirms that the Distributed Data Parallel (DDP) implementation is functioning as intended. Across both 2-GPU and 4-GPU configurations, individual GPUs report nearly identical usage levels, which indicates an even distribution of the input data and gradient computations. This balanced workload per GPU is crucial for maximizing training efficiency and avoiding bottlenecks. It also validates the use of `DistributedSampler` in PyTorch's DDP setup to ensure that each process receives a properly partitioned subset of the training data.

Table 21: LTC GPU Utilization on HAR Dataset (% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 64 | 19.71 | 19.68 | 13.53 | 13.78 | 12.13 | 12.19 |
| 128 | 23.89 | 23.75 | 14.88 | 14.10 | 13.37 | 13.93 |
| 256 | 28.82 | 28.55 | 16.44 | 17.48 | 15.67 | 15.30 |
| 512 | 33.62 | 33.62 | 19.47 | 18.67 | 18.38 | 18.21 |

Table 22: LTC Memory Utilization on HAR dataset (% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 64 | 6.70 | 6.74 | 6.66 | 6.69 | 6.68 | 6.69 |
| 128 | 9.69 | 9.74 | 9.52 | 9.56 | 9.58 | 9.56 |
| 256 | 14.16 | 14.27 | 13.81 | 13.86 | 13.84 | 13.88 |
| 512 | 25.09 | 25.23 | 23.28 | 23.57 | 23.71 | 23.67 |

Table 23: CfC GPU Utilization on HAR dataset (% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 64 | 12.11 | 13.01 | 16.76 | 12.56 | 9.84 | 13.30 |
| 128 | 12.30 | 12.32 | 11.01 | 10.88 | 10.12 | 9.39 |
| 256 | 10.17 | 11.40 | 9.39 | 8.49 | 8.67 | 7.54 |
| 512 | 9.37 | 10.29 | 7.17 | 6.44 | 6.73 | 6.76 |

Table 24: CfC Memory Utilization on HAR dataset (% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 64 | 3.70 | 3.82 | 3.83 | 3.92 | 3.93 | 3.93 |
| 128 | 3.72 | 3.88 | 3.71 | 3.77 | 3.76 | 3.77 |
| 256 | 3.42 | 3.74 | 3.38 | 3.59 | 3.61 | 3.61 |
| 512 | 3.45 | 3.72 | 3.22 | 3.53 | 3.50 | 3.48 |

### 5.2.5 Interpretation and Summary

The experiments on the Human Activity Recognition (HAR) dataset demonstrate clear differences between the LTC and CfC models in terms of training efficiency, scalability, and

resource usage. CfC consistently achieves shorter training times across all hardware configurations and batch sizes, both on CPU and GPU. Its architecture, based on closed-form continuous-time computation, avoids the computational overhead of numerical ODE solvers required by LTC, making it more efficient, especially on CPU.

LTC, on the other hand, exhibits steeper improvements when leveraging GPU parallelization, particularly at larger batch sizes. However, this comes with a cost: higher memory usage and greater sensitivity to synchronization overhead in multi-GPU settings. Notably, training time with 4 GPUs sometimes exceeds that of 2 GPUs, indicating diminishing returns and communication bottlenecks in highly parallel configurations.

Both models maintain competitive classification accuracy, with CfC showing slightly better performance stability as batch size increases. Additionally, the near-equal distribution of GPU usage and memory allocation confirms the effectiveness of the Distributed Data Parallel (DDP) implementation, with each GPU receiving a balanced workload.

In summary, CfC emerges as a more computationally efficient and hardware-friendly model, ideal for scenarios requiring fast training and low memory footprint. LTC remains a powerful alternative, particularly when fine-grained temporal modeling is essential and GPU acceleration is readily available. These findings provide a foundation for broader generalizations discussed in the next chapter.

## 5.3 Traffic Dataset

The Traffic dataset section presents a detailed evaluation of the LTC and CfC models on a large-scale real-world regression task. The analysis is structured to address the core experimental objectives defined in Chapter 4, including the models' predictive accuracy, computational efficiency, and scalability across different hardware setups. Results are reported for a variety of batch sizes and execution environments, ranging from single-CPU runs to distributed training on multiple GPUs. This section systematically compares the models in terms of error metrics (MSE and MAE), training time, speedup, and hardware resource utilization. Through these evaluations, the goal is to highlight the practical strengths and limitations of each model under realistic training conditions.

### 5.3.1 Accuracy and Loss Analysis

This section evaluated the regression performance of the LTC and CfC models on the Traffic dataset by analysing test MSE and MAE across different batch sizes and hardware configurations. The reported results provide insights into how each model scales and maintains consistency when operating under both limited computational environments (CPU) and distributed multi-GPU setups.

**CPU Performance.** On the Traffic dataset, similarly to the results on the HAR dataset presented in the previous section, both models show high performances but the CfC model slightly outperforms LTC in both Test MSE and MAE across all batch sizes (Tables 25 and 26). CfC maintains stable performance (MSE = 0.06, MAE = 0.16) up to batch size 256, with only minor degradation at 512. In contrast, LTC shows slightly higher and more variable error, increasing with batch size. These results highlight CfC's robustness and efficiency in CPU environments, likely due to its closed-form formulation, which avoids the iterative computations required by LTC.

**Multi-GPU Consistency** Both models demonstrate stable and consistent behaviour across different GPU configurations, as it can be seen in Table 27 and 28. For batch size 512, the Test MSE and MAE remain nearly identical across CPU and GPUs, indicating reliable distributed training. As the batch size increases, a slight performance degradation is observed in both LTC and CfC, more pronounced in MAE values. CfC maintains a slight edge over LTC in both metrics

Table 25: LTC CPU Test-MSE an Test-MAE on Traffic Dataset

| Batch Size | Test MSE | Test MAE |
|---|---|---|
| 64 | 0.08 | 0.18 |
| 128 | 0.08 | 0.18 |
| 256 | 0.08 | 0.19 |
| 512 | 0.09 | 0.20 |

Table 26: CfC CPU Test-MSE and Test-MAE on Traffic Dataset

| Batch Size | Test MSE | Test MAE |
|---|---|---|
| 64 | 0.06 | 0.16 |
| 128 | 0.06 | 0.16 |
| 256 | 0.06 | 0.16 |
| 512 | 0.08 | 0.18 |

at all device scales, particularly noticeable in the 1024 and 2048 batch size settings. These results confirm that scaling training across multiple GPUs does not significantly compromise prediction accuracy and that CfC's performance degrades more gracefully under increasing batch sizes.

Table 27: LTC Test-MSE and Test-MAE across Devices on Traffic Dataset

(a) Test MSE

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 512 | 0.09 | 0.09 | 0.09 | 0.09 |
| 1024 | 0.11 | 0.10 | 0.12 | 0.12 |
| 2048 | 0.16 | 0.16 | 0.16 | 0.16 |

(b) Test MAE

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 512 | 0.20 | 0.19 | 0.20 | 0.20 |
| 1024 | 0.23 | 0.21 | 0.24 | 0.24 |
| 2048 | 0.30 | 0.30 | 0.31 | 0.30 |

**Observations** From the presented results, both LTC and CfC exhibit robust performance on the Traffic dataset, with CfC consistently achieving lower Test MSE and MAE values across all configurations. On CPU, both models perform best with smaller batch sizes (64–256), where CfC maintains superior accuracy. As batch size increases, particularly beyond 512 in GPU settings, a modest degradation in performance is observed—more pronounced in the MAE metric. This suggests that larger batch sizes, while potentially offering computational efficiency, may lead to reduced generalization. CfC appears more resilient to this effect, maintaining lower error rates even as batch size grows. These trends highlight the importance of tuning batch size carefully in distributed settings, balancing training efficiency against model accuracy.

### 5.3.2 Training Time on CPU and single GPU

The comparison between the Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) models reveals significant differences in training behaviour, particularly regarding computational efficiency and scalability across hardware configurations. These trends are particularly evident when evaluating training time against increasing batch sizes on both CPU and GPU backends.

As illustrated in Figure 31, the CfC model consistently demonstrates significantly faster training times than LTC on CPU for all tested batch sizes. While LTC training times remain above 3500 seconds across the board—with only slight reductions for larger batch sizes—the CfC model completes training in under 500 seconds in all cases. This pronounced difference can be attributed to CfC's analytical closed-form solution for hidden state updates, which circumvents the iterative ODE-solving steps required in LTC. The computational overhead of unfolding the

Table 28: CfC Test-MSE and Test-MAE across Devices on Traffic Dataset

(a) Test MSE

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 512 | 0.08 | 0.08 | 0.08 | 0.08 |
| 1024 | 0.12 | 0.09 | 0.11 | 0.12 |
| 2048 | 0.16 | 0.16 | 0.17 | 0.17 |

(b) Test MAE

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 512 | 0.18 | 0.18 | 0.19 | 0.18 |
| 1024 | 0.24 | 0.20 | 0.23 | 0.24 |
| 2048 | 0.29 | 0.28 | 0.29 | 0.30 |

ODE multiple times per time-step in LTC makes it particularly inefficient in CPU-bound scenarios.

The performance gap narrows but remains evident when the models are trained on GPU, as shown in Figure 32. Here, both models benefit from increased parallelism, and the training time decreases as batch size grows. However, CfC maintains a clear lead, completing training roughly 3 to 5 times faster than LTC. The GPU's optimized matrix operations and parallel computing capabilities mitigate some of the overhead associated with LTC's ODE integration, but not entirely. CfC's simpler forward pass and fewer per-sample computations allow it to make more efficient use of GPU resources, especially at larger batch sizes where throughput becomes critical.

This divergence in training efficiency highlights a fundamental design trade-off between the two architectures. LTC, while theoretically powerful and biologically inspired, incurs a substantial cost due to its simulation of neuron dynamics via numerical ODE approximation. CfC, by contrast, abstracts these dynamics into a closed-form solution, achieving much faster training with only modest sacrifices in predictive accuracy. As such, CfC emerges as a more computationally efficient option, particularly in settings where training time or hardware availability is a limiting factor.

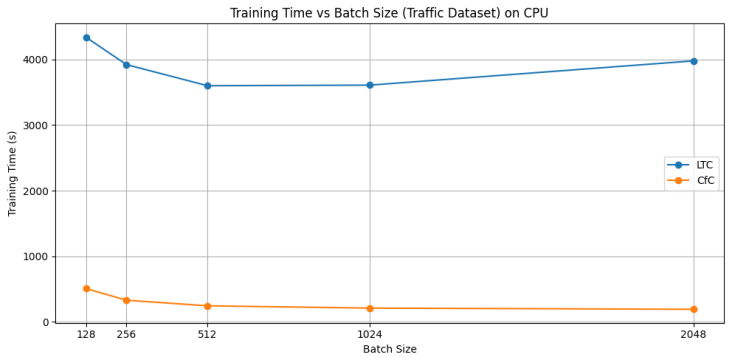



Figure 31: Training Time vs Batch Size (Traffic Dataset) on CPU

### 5.3.3 Training Time on multi-GPU and Speedup

The training time results for the Traffic dataset (Tables 29 and 30) and the corresponding speedup plots (Figures 33 and 34) demonstrate a clear and consistent improvement in training efficiency when moving from CPU to GPU-based execution, particularly for larger batch sizes. The LTC model, due to its higher computational complexity, benefits the most from distributed training. With batch size 2048, it achieves a remarkable speedup of 27.8× on 2 GPUs and 20.9× on 4 GPUs compared to its CPU baseline. This confirms that the LTC model effectively utilizes the available parallel hardware, especially when the batch size is sufficiently large to amortize synchronization overheads.

In contrast, the CfC model, while inherently faster due to its closed-form formulation, shows more modest but still significant gains. For instance, with batch size 2048, it reaches up to 4.1×

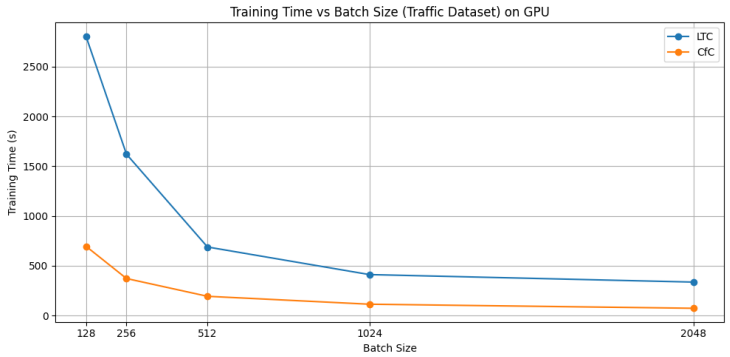



Figure 32: Training Time vs Batch Size (Traffic Dataset) on GPU

speedup on 2 GPUs and 3.0× on 4 GPUs. Although these gains are lower in magnitude compared to LTC, the absolute training times are consistently shorter, confirming CfC's computational efficiency. Moreover, both models exhibit a speedup curve that increases with batch size, reflecting improved parallel throughput and reduced per-sample overhead.

Notably, for both models, 2-GPU configurations often outperform 4-GPU setups, which can be attributed to communication overheads becoming more prominent as the number of devices increases. This suggests that for datasets not so large, such as Traffic, there exists a point beyond which additional parallelism yields diminishing returns. Overall, the results confirm that larger batch sizes are key to leveraging the full potential of multi-GPU training, and that the LTC model especially benefits from parallel execution, whereas CfC maintains faster performance across all configurations due to its simpler structure.

Table 29: Training Time (s) for LTC on Traffic Dataset

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 512 | 3598 | 689.2 | 459.3 | 575.1 |
| 1024 | 3607 | 411.9 | 251.2 | 323.8 |
| 2048 | 3976 | 336.6 | 143.0 | 189.9 |

Table 30: Training Time (s) for CfC on Traffic Dataset

| Batch Size | CPU | 1-GPU | 2-GPU | 4-GPU |
|---|---|---|---|---|
| 512 | 245.3 | 193.9 | 121.0 | 152.8 |
| 1024 | 210.1 | 114.4 | 70.25 | 92.29 |
| 2048 | 192.5 | 73.63 | 47.21 | 64.93 |

### 5.3.4 GPU and Memory Utilization

The GPU and memory utilization results provide important insights into how computational resources are used during distributed training with LTC and CfC models across varying batch sizes and hardware configurations. Tables 31–34 summarize the average utilization per GPU for both models on the Traffic dataset.

For the LTC model, GPU utilization increases steadily with batch size, indicating more efficient hardware use as the workload grows. On 2 GPUs, utilization rises from approximately 23% at batch size 512 to over 45% at batch size 2048. This confirms that the computational intensity of LTC benefits from larger batch sizes, which keep the GPUs moderately engaged.

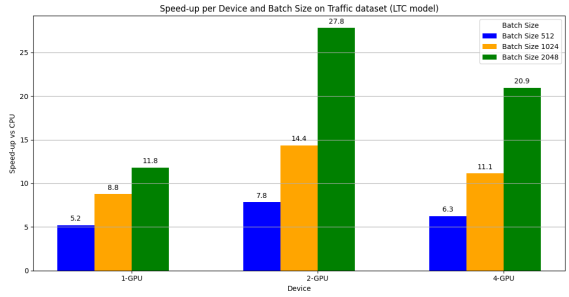



Figure 33: Speedup per Device and Batch Size for LTC model on Traffic dataset (relative to CPU baseline).

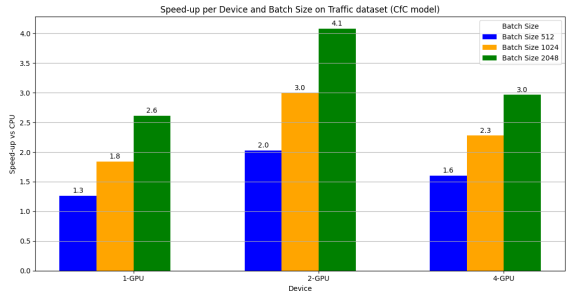



Figure 34: Speedup per Device and Batch Size for CfC model on Traffic dataset (relative to CPU baseline).

However, in the 4-GPU setting, the average utilization per device remains lower—around 24% for the largest batch size. This suggests that the workload is increasingly fragmented, leading to underutilization and communication overheads that limit the performance gains from additional GPUs.

Memory usage for LTC also scales consistently with batch size. At 2048, average memory consumption reaches nearly 64% per GPU on the 4-GPU setup, compared to around 15–20% for batch size 512. Importantly, memory usage remains balanced across all GPUs, indicating correct functioning of the DDP (Distributed Data Parallel) setup, with even partitioning of data and model parameters.

In contrast, the CfC model shows considerably lower GPU utilization across all configurations, reflecting its lower computational demands. Utilization in the 2-GPU setup stays below 13%, even at batch size 2048, and further declines in the 4-GPU configuration, dropping to averages below 10%. This highlights that CfC does not impose a heavy workload on GPUs, and the gains from parallelization are inherently limited by the model's lightweight architecture.

Similarly, CfC exhibits very modest memory consumption. Even at the largest batch size, memory utilization remains under 5% per GPU across both 2-GPU and 4-GPU configurations. This minimal memory footprint confirms CfC's architectural efficiency and reinforces its suitability for resource-constrained environments. As with LTC, memory is well balanced across all devices, again confirming proper DDP operation.

In summary, these results show that:

- LTC can take moderate advantage of multi-GPU configurations, especially with larger batch sizes, though the returns diminish as device count increases beyond 2.
- CfC is extremely light on both memory and compute, making multi-GPU execution unnecessary in most practical scenarios.

Overall, while DDP effectively distributes workloads across devices in both models, the marginal utility of adding more GPUs becomes limited, particularly for CfC and smaller batch sizes, due to reduced utilization and increased communication overhead.

Table 31: LTC GPU Utilization on Traffic dataset (% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 512 | 23.29 | 23.22 | 13.80 | 13.96 | 12.46 | 12.47 |
| 1024 | 32.57 | 32.95 | 18.93 | 18.76 | 16.67 | 16.47 |
| 2048 | 45.59 | 45.29 | 24.25 | 25.14 | 23.31 | 22.82 |

Table 32: LTC Memory Utilization on Traffic dataset (% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 512 | 15.54 | 15.54 | 20.51 | 20.50 | 20.49 | 20.48 |
| 1024 | 27.29 | 27.31 | 31.33 | 31.39 | 31.41 | 31.39 |
| 2048 | 59.81 | 59.89 | 63.58 | 63.78 | 63.80 | 63.88 |

Table 33: CfC GPU Utilization on Traffic dataset (% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 512 | 12.52 | 12.53 | 9.69 | 10.27 | 8.49 | 8.93 |
| 1024 | 12.03 | 11.43 | 7.86 | 9.34 | 7.16 | 8.35 |
| 2048 | 11.04 | 11.11 | 8.78 | 7.71 | 6.71 | 7.30 |

Table 34: CfC Memory Utilization on Traffic dataset (% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 512 | 4.17 | 4.17 | 4.17 | 4.15 | 4.14 | 4.14 |
| 1024 | 4.35 | 4.35 | 4.40 | 4.36 | 4.35 | 4.34 |
| 2048 | 4.73 | 4.80 | 4.99 | 4.84 | 4.84 | 4.81 |

### 5.3.5 Interpretation and Summary

The results obtained from the Traffic dataset experiments provide a comprehensive view of how the LTC and CfC architectures behave under varied computational conditions and training configurations. Several critical patterns and takeaways emerge from the analysis.

First, in terms of predictive performance, both models demonstrated robust accuracy across all batch sizes and hardware configurations, with CfC consistently achieving slightly lower Test MSE and MAE values. This margin, while not drastic, suggests a modest advantage for CfC in regression tasks, likely due to its direct parametrization and efficient temporal representation.

Second, training efficiency was markedly different between the two models. CfC significantly outperformed LTC in training time on both CPU and GPU setups, achieving up to a 10-fold reduction in training duration on CPUs and 2-4x gains on GPUs. These gains stem from CfC's

closed-form dynamics, which avoid the computational overhead of numerically solving ODEs, an intrinsic requirement of LTC.

Third, speedup analysis confirmed that both models benefit from increased batch sizes and parallelism, though with diminishing returns beyond 2 GPUs. CfC exhibited moderate improvements in speedup, whereas LTC leveraged parallel resources more effectively, achieving higher absolute speedup factors. However, these gains were tempered by increased GPU underutilization in multi-GPU setups, especially with 4 GPUs, where the workload per device became too small to offset the synchronization and communication overheads.

In summary, CfC stands out as a more computationally efficient and scalable model for large-scale time-series regression tasks, delivering faster training and comparable or superior accuracy. LTC, while theoretically expressive, incurs higher computational costs that limit its practicality in resource-constrained or time-sensitive environments. These findings reinforce the importance of architectural efficiency in model selection, particularly when targeting deployment in HPC settings.

## 5.4 Power Dataset

This section presents the results of the experiments conducted on the Individual Household Electric Power Consumption dataset, which was introduced to further explore scalability in response to the limited speedup observed on the Traffic dataset. This dataset is larger in both size and temporal span, with over 64,000 multivariate sequences sampled at one-minute intervals. Only distributed experiments using 2-GPU and 4-GPU configurations were conducted, as the focus was to assess the benefits of parallelization at scale.

### 5.4.1 Accuracy and Loss Evaluation

Table 35 summarizes the performance of the LTC and CfC models on the Power dataset, reporting both Mean Squared Error (MSE) and Mean Absolute Error (MAE) across batch sizes and distributed configurations (2-GPU and 4-GPU).

Table 35: Test MSE and MAE on Power Dataset across 2-GPU and 4-GPU setups

| Batch Size | LTC | | CfC | |
|---|---|---|---|---|
| | 2-GPU | 4-GPU | 2-GPU | 4-GPU |
| 512 | 0.012 / 0.075 | 0.012 / 0.075 | 0.011 / 0.073 | 0.011 / 0.073 |
| 1024 | 0.014 / 0.082 | 0.014 / 0.081 | 0.012 / 0.078 | 0.013 / 0.078 |
| 2048 | 0.019 / 0.099 | 0.018 / 0.098 | 0.017 / 0.094 | 0.017 / 0.095 |

The results highlight several key findings:

- **Device consistency:** Both LTC and CfC models display nearly identical performance across 2-GPU and 4-GPU configurations, indicating stable behavior under increased parallelism.
- **Model comparison:** The CfC model consistently achieves slightly better predictive performance than LTC, with lower MSE and MAE at all tested batch sizes. This aligns with previous observations on the Traffic dataset, further supporting CfC's efficiency.
- **Impact of batch size:** A modest increase in both MSE and MAE is observed as the batch size increases, particularly in the MAE metric. This suggests that very large batch sizes may lead to reduced generalization capability, even when computational efficiency improves.

Overall, these results confirm that both models can handle large-scale regression tasks effectively in a distributed setting, with CfC maintaining a consistent edge in predictive accuracy.

### 5.4.2 Training Time

Table 36 reports the total training time (in seconds) for the LTC and CfC models on the Power dataset, across two distributed configurations: 2-GPU and 4-GPU. Experiments were conducted

using batch sizes of 512, 1024, and 2048 to examine the scalability of training under increasing data throughput.

Table 36: Training Time (s) for LTC and CfC on Power Dataset

| Batch Size | LTC | | CfC | |
|---|---|---|---|---|
| | 2-GPU | 4-GPU | 2-GPU | 4-GPU |
| 512 | 458.1 | 582.7 | 123.9 | 150.3 |
| 1024 | 280.7 | 373.4 | 74.6 | 92.7 |
| 2048 | 165.9 | 227.0 | 47.9 | 60.6 |

Several trends emerge from these results:

- **No speedup from 4-GPU training:** For both models, training on 4 GPUs was consistently slower than on 2 GPUs, with no observed speedup across any batch size.
- **Communication overheads dominate:** The reduced efficiency in the 4-GPU configuration suggests that the overhead introduced by inter-GPU synchronization and communication outweighs the computational gains from additional parallelism—particularly in the case of CfC, which has lower computational demands.
- **Limited scalability at this data volume:** Although the Power dataset is larger than Traffic, the workload per GPU in the 4-GPU setup may still be insufficient to amortize DDP's coordination costs. This highlights the challenges of scaling lightweight models or moderate datasets across many GPUs.

Overall, these results indicate that while CfC remains computationally more efficient than LTC, scaling to 4 GPUs may not be beneficial without substantially larger workloads or more compute-intensive models.

### 5.4.3 GPU and Memory Utilization

This section analyzes the hardware resource usage during training of the LTC and CfC models on the Power dataset. GPU and memory utilization were monitored across different batch sizes in both 2-GPU and 4-GPU configurations. The results are summarized in Tables 37–40.

Table 37: LTC GPU Utilization on Power dataset (% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 512 | 22.79 | 22.60 | 13.28 | 13.60 | 12.33 | 12.87 |
| 1024 | 32.62 | 31.80 | 17.74 | 17.41 | 16.71 | 15.80 |
| 2048 | 44.64 | 43.96 | 23.01 | 23.55 | 21.71 | 21.63 |

Table 38: LTC Memory Utilization on Power dataset (% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 512 | 20.24 | 20.29 | 14.87 | 15.04 | 15.04 | 15.03 |
| 1024 | 26.23 | 26.32 | 25.50 | 25.69 | 25.70 | 25.68 |
| 2048 | 56.22 | 56.73 | 53.89 | 54.12 | 54.08 | 54.12 |

Table 39: CfC GPU Utilization on Power dataset(% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 512 | 12.01 | 12.57 | 7.79 | 8.85 | 7.86 | 7.04 |
| 1024 | 11.82 | 11.60 | 7.84 | 7.63 | 6.81 | 7.16 |
| 2048 | 13.26 | 12.13 | 9.07 | 8.10 | 7.87 | 7.86 |

Table 40: CfC Memory Utilization on Power dataset (% average per GPU)

| Batch Size | 2-GPU | | 4-GPU | | | |
|---|---|---|---|---|---|---|
| | #1 | #2 | #1 | #2 | #3 | #4 |
| 512 | 3.86 | 4.04 | 3.73 | 3.95 | 3.95 | 3.94 |
| 1024 | 3.88 | 4.15 | 3.69 | 4.01 | 4.02 | 4.01 |
| 2048 | 4.04 | 4.44 | 3.80 | 4.27 | 4.25 | 4.27 |

Several observations can be drawn:

- **CfC remains significantly lighter:** Across all configurations, CfC exhibits substantially lower GPU and memory usage than LTC, reinforcing its computational efficiency.
- **Moderate utilization growth for LTC:** As batch size increases, LTC's GPU usage reaches up to 45% in the 2-GPU setup and about 23% in the 4-GPU setting, indicating more effective (but still partial) hardware engagement compared to CfC.
- **CfC stays underutilized:** Even with the larger dataset, CfC's GPU usage peaks below 13% and memory usage stays under 4.5%, highlighting its limited demand and the potential underutilization of multi-GPU setups for such lightweight architectures.
- **Balanced device usage:** For both models, memory and processing load are evenly distributed across GPUs, confirming the correct functioning of PyTorch's DDP in terms of data partitioning and synchronization.

These findings confirm trends previously observed with the Traffic dataset: while LTC can benefit moderately from parallel hardware, CfC's low computational footprint limits the practical gains of scaling across additional GPUs unless paired with significantly larger data volumes or heavier architectures.

### 5.4.4 Model Comparison and Observations

A comparative analysis of the LTC and CfC models on the Power dataset reveals consistent patterns across key performance dimensions:

- **Predictive Accuracy:** CfC consistently outperforms LTC in terms of both Test MSE and MAE across all batch sizes and GPU configurations. This reinforces earlier findings from the Traffic dataset, highlighting CfC's effectiveness in modeling complex temporal dynamics despite its lighter architecture.
- **Training Efficiency:** CfC demonstrates significantly lower training times than LTC in all settings. On average, it completes training in less than half the time required by LTC, underscoring the computational advantage of its closed-form formulation which bypasses the iterative ODE-solving required in LTC.
- **Scalability Limitations:** Neither model benefits from adding more GPUs beyond two. In fact, training time increases in the 4-GPU configuration due to the added overhead of inter-device communication and synchronization, coupled with insufficient workload per device to fully exploit parallelism.

- **Hardware Utilization:** CfC remains markedly underutilizing in terms of both GPU and memory usage, with peak GPU usage staying below 13%. This suggests that its bottlenecks likely reside in orchestration overhead or memory bandwidth rather than raw compute. In contrast, LTC utilizes a larger portion of GPU resources, but still falls short of saturating available capacity.

Overall, these results confirm the general trends observed in previous datasets: CfC excels in efficiency and accuracy but exhibits limited scalability on current hardware, whereas LTC can better exploit available resources, though at significantly higher computational cost.

# 6. Discussion

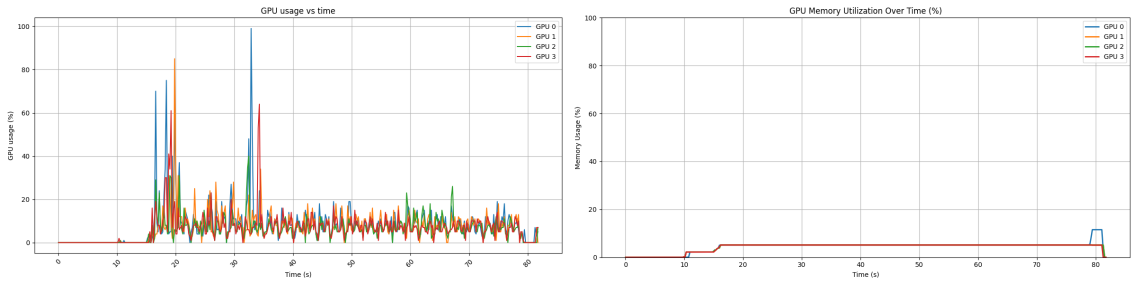
This chapter provides a reflective synthesis of the results presented in Chapter 5, highlighting the key insights gained through the comparative evaluation of Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC) neural network models under different computational settings and workloads. The aim is to interpret the findings in light of the original research objectives and broader trends in neural network training and parallelization.

Three primary experimental themes were explored: model performance across different datasets, the effect of batch size and hardware configuration on training dynamics, and the scalability of distributed training using multiple GPUs. The analysis demonstrated several noteworthy trends.

First, in terms of predictive performance, both LTC and CfC achieved high accuracy and low error across the Human Activity Recognition (HAR) and Traffic datasets. However, CfC consistently outperformed LTC, albeit slightly, in both classification and regression tasks. This advantage was particularly evident on CPU-based configurations and at smaller batch sizes, where CfC's closed-form formulation enabled faster and more stable training.

Second, in terms of computational efficiency, CfC proved to be substantially faster than LTC across all environments. On CPU, CfC reduced training time by more than 10× compared to LTC, and even on GPU, it retained a 2–5× advantage. These findings are in line with those presented by Hasani et al. (2022) [6] and underscore the computational benefits of eliminating numerical ODE solvers, which are integral to LTC's formulation.

Third, multi-GPU training showed mixed results. While both models benefited from increased batch size and GPU parallelism, the scalability gains were limited beyond 2 GPUs. Speedup was constrained by GPU under-utilization and communication overhead and by the topology and communications channel of the GPUs, especially in the 4-GPU configurations. The lightweight architecture of CfC further limited the effectiveness of the distribution, as its GPU and memory usage remained low even at large batch sizes (see Figure 35).




(a) GPU usage

(b) Memory usage

Figure 35: GPU and Memory usage for training CfC model on Traffic dataset with batch_size per process set to 2048 using 4 devices

Finally, both models showed strong consistency in predictive performance across CPU, 1-GPU, 2-GPU, and 4-GPU environments, confirming the correctness and reliability of the distributed training implementation via PyTorch's Distributed Data Parallel (DDP) framework.

These findings set the stage for a broader discussion on model architecture, parallel training strategies, and implications for future research and deployment.

## 6.1 Interpretation of Results

This section presents a consolidated interpretation of the experimental findings across all datasets and configurations:

- **Model Performance:**
  - Both LTC and CfC achieved strong predictive performance across tasks, with CfC generally yielding slightly lower error metrics (MSE, MAE) and higher classification accuracy.
  - CfC's closed-form formulation proved especially effective in regression scenarios (Traffic dataset), demonstrating consistent results across CPU and GPU.
- **Batch Size Influence:**
  - Smaller batch sizes generally favoured model accuracy, especially on CPU. Larger batch sizes provided computational benefits but led to minor degradation in generalization.
  - CfC showed more resilience to increasing batch size, maintaining stable performance even as it scaled.
- **Training Efficiency:**
  - CfC consistently trained significantly faster than LTC on both CPU and GPU, due to its non-iterative update mechanism.
  - LTC's reliance on ODE unfolding caused substantial overhead, particularly in CPU settings, but benefited more from GPU acceleration.
- **Scalability and Parallelization:**
  - Both models benefited from distributed training, with LTC achieving higher speedup ratios due to its heavier computational load.
  - Diminishing returns were observed when moving from 2-GPU to 4-GPU configurations, especially for the lightweight CfC model.
- **Hardware Utilization:**
  - GPU and memory utilization increased with batch size but remained moderate overall, especially for CfC.
  - CfC's low GPU usage highlights its efficiency but also suggests limited benefits from multi-GPU scaling.
  - LTC showed better utilization patterns at higher batch sizes, aligning with its stronger speedup on multi-GPU setups.
- **Distributed Training Validity:**
  - DDP-based training showed consistency in performance across different GPU counts, with balanced resource usage confirming correct setup.

## 6.2 Limitations

While this study offers valuable insights, some limitations should be acknowledged:

- **Dataset Variety:** Only few datasets were used, one synthetic and three real-world time series. This limited scope may not fully capture the behaviour of the models across broader domains such as biomedical, audio, or financial time series.
- **Hardware Constraints:** The BOADA HPC cluster imposed practical limitations on time, memory and topology of devices, and this restricted the scale and complexity of the experiments.

- **Restricted Scale of Distributed Training:** Multi-GPU training was limited to a maximum of 4 GPUs due to resource availability. Experiments involving more extensive hardware (e.g., 8 or 16 GPUs) were not conducted, limiting the ability to evaluate extreme-scale scalability.
- **Fixed Training Epochs:** All experiments were conducted with a fixed number of 50 training epochs. This uniform choice may not suit all batch sizes or architectures and could lead to under- or over-training in certain configurations.
- **Lack of Hyperparameter Tuning:** No grid search or fine-tuning was performed. Learning rate, batch size, and other model hyperparameters were based on prior literature or kept constant, possibly preventing models from reaching optimal performance.
- **Limited Evaluation Metrics:** The analysis focused on test accuracy/loss and training time. Other potentially informative metrics—such as convergence speed, training stability, or energy efficiency—were not evaluated.
- **Simplified Assumptions:** Assumptions such as using the same optimizer (Adam) for all experiments and a uniform batch size strategy across models may not reflect the best practices for each individual architecture.

These limitations do not invalidate the results but suggest that further research is needed to assess generalizability, optimize performance, and understand the behaviour of these models under different real-world conditions.

# 7. Sustainability and Ethical Implications

This chapter analyses the sustainability and ethical dimensions associated with the development and experimentation conducted in this thesis. The goal is to reflect on the environmental, economic, and social impact of the work, in accordance with the principles of responsible research and innovation.

## 7.1 Environmental Considerations

The training of machine learning models, particularly in high-performance computing environments, involves substantial energy consumption. In this thesis, multiple experiments were run on the BOADA HPC cluster, using both CPU and multi-GPU configurations. While the cluster optimizes resource sharing and scheduling, the repeated training across models, datasets, and batch sizes resulted in significant computational workloads. In particular, models like LTC, which involve ODE solvers, require more computational time, potentially increasing environmental impact.

On the other hand, the exploration of training efficiency and speedup contributes positively by identifying strategies to reduce training time and computational cost. The CfC model, with its closed-form formulation, showed substantially lower resource usage while maintaining strong performance, demonstrating a step toward more sustainable model design.

## 7.2 Economic Considerations

From an economic perspective, the research relied on publicly funded infrastructure and open-source tools, ensuring accessibility and cost efficiency. The work did not involve any commercial licensing or proprietary datasets, supporting knowledge sharing and reproducibility.

The benchmarking of model training costs across configurations also provides valuable insights for institutions or companies aiming to adopt LNN architectures. Understanding how to optimize training on existing hardware can reduce operational costs and avoid unnecessary investments in high-end infrastructure.

## 7.3 Social and Ethical Aspects

The study does not involve any personal, private, or sensitive user data, mitigating direct risks related to data privacy or consent. All datasets used are publicly available and anonymized.

However, it is important to recognize that advances in AI models and their efficiency can have broader social implications. Improvements in model performance and scalability may lead to increased automation in sectors like healthcare, transport, or finance, which raises questions around accountability, transparency, and job displacement. While these aspects are not directly addressed in the scope of this work, they are relevant considerations for any AI deployment.

On a personal level, the project fostered critical reflection on responsible computing and the role of researchers in designing sustainable and ethical AI systems.

## 7.4 Ethical Reflection and Professional Conduct

Throughout the project, efforts were made to adhere to good scientific practices: transparency in reporting, use of openly licensed data and code, reproducibility of experiments, and acknowledgment of limitations. No attempts were made to manipulate or selectively report results.

Moreover, the research aligns with the ethical principles defined by the ACM Code of Ethics and the European Code of Conduct for Research Integrity, including honesty, accountability, and respect for the environment and society.

## 7.5 Contribution to Sustainable Development Goals (SDGs)

The outcomes of this thesis contribute indirectly to some of the United Nations Sustainable Development Goals:


(a) SGD 9


(b) SGD 12

Figure 36: Sustainable Development Goals involved in this work

- **SDG 9 – Industry, Innovation and Infrastructure**: by exploring efficient training of advanced AI models on distributed systems.
- **SDG 12 – Responsible Consumption and Production**: by analyzing energy and resource usage in AI model training and proposing more efficient alternatives.

## 7.6 Final Considerations

The thesis promotes a responsible approach to AI research, emphasizing efficiency, openness, and critical evaluation of its computational and societal implications. Future work may further explore energy-aware training strategies and develop lightweight, low-impact AI models suitable for broader and more sustainable deployment.

# 8. Conclusions and Future Work

This thesis set out to investigate the performance, scalability, and efficiency of two biologically inspired continuous-time neural network models—Liquid Time-Constant (LTC) and Closed-form Continuous-time (CfC), when deployed under distributed training conditions using high-performance computing resources. The goal was to understand how these architectures behave across different hardware configurations (CPU, single-GPU, multi-GPU), various batch sizes, and real-world datasets involving both classification and regression tasks.

The experimental study demonstrated that:

- **CfC** consistently offered faster training times and lower memory consumption, with comparable or better predictive performance than LTC.
- **LTC**, while computationally heavier due to its ODE-based formulation, showed higher speedup potential under parallel training, especially with large batch sizes and 2-GPU setups.
- **Distributed training** with PyTorch's DDP framework was successfully implemented and evaluated, showing clear benefits for larger workloads, with diminishing returns on 4-GPU setups due to synchronization overhead.
- The analysis provided insight into the effect of batch size on performance, convergence, and utilization, reinforcing the importance of balanced tuning in HPC contexts.

Overall, the thesis successfully achieved its objectives by combining model-driven benchmarking with empirical analysis, offering a novel comparative study of LTC and CfC in parallelized training environments.

## 8.1 Future Work

While this thesis provides a solid foundation, several avenues remain open for future exploration:

- **Larger-scale deployment** Future studies could explore training LTC and CfC models on significantly larger datasets (e.g., millions of samples) or on clusters with more than 4 GPUs. This would help evaluate the scalability and communication bottlenecks of DDP setups under real HPC workloads and assess how performance trends evolve with increasing parallelism.
- **Exploration of alternative parallelization frameworks**: While this thesis adopted PyTorch Distributed Data Parallel (DDP), frameworks such as Horovod or DeepSpeed may offer performance or flexibility advantages. Future work could explore whether these tools improve training speed, communication efficiency, or deployment readiness.
- **Adaptive training strategies** Implementing advanced training techniques such as dynamic learning rate scheduling, gradient accumulation, adaptive batch size tuning, or early stopping could lead to better convergence and improved generalization. These methods would also help in reducing unnecessary computation and energy consumption during training.
- **Real-world applications** Applying these architectures to domain-specific tasks, such as ECG signal classification, industrial sensor forecasting, or smart grid demand prediction, would provide insight into the practical utility of Liquid Neural Networks. Domain adaptation and interpretability may also become more relevant in such contexts.
- **Hyperparameter search** While fixed settings were used in this study for fair comparison, an automated search (e.g., using grid search, Bayesian optimization, or evolutionary strategies) could better tailor the models to each dataset and potentially uncover more optimal configurations.
- **NUMA-aware data loading** Future work could investigate techniques for parallelizing dataset loading with NUMA (Non-Uniform Memory Access) awareness. By pre-loading data into the memory of the NUMA node directly attached to the GPU that will process it,

memory access latencies could be significantly reduced. This optimization is particularly relevant in multi-GPU systems where each group of GPUs is bound to different NUMA nodes, as in the BOADA cluster. Such strategies could improve throughput and reduce interconnect bottlenecks during training.

- **Evaluation on alternative hardware architectures** This work focused on NVIDIA GPUs within an HPC cluster. Future research could investigate the performance of LTC and CfC models on alternative hardware such as Google TPUs, AMD GPUs, or custom AI accelerators (e.g., Habana Gaudi or Cerebras). These platforms may offer different memory hierarchies, interconnect topologies, or compute paradigms that could benefit the unique characteristics of Liquid Neural Networks.

In conclusion, this thesis lays the groundwork for understanding and benchmarking the distributed training behaviour of continuous-time neural networks. While the findings provide valuable insights into the computational and predictive characteristics of LTC and CfC models, the proposed future directions underscore the potential for further performance optimization, broader application, and methodological innovation. As these architectures evolve and become more widely adopted, continued exploration will be essential to unlock their full capabilities in both research and real-world settings.

# References


[1] S. Hochreiter and J. Schmidhuber, 'Long short-term memory,' *Neural Computation*, vol. 9, pp. 1735–1780, Nov. 1997. DOI: 10.1162/neco.1997.9.8.1735.

[2] K. Cho, B. van Merrienboer, D. Bahdanau and Y. Bengio, *On the properties of neural machine translation: Encoder-decoder approaches*, 2014. arXiv: 1409.1259 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1409.1259.

[3] A. Vaswani, N. Shazeer, N. Parmar *et al.*, *Attention is all you need*, 2023. arXiv: 1706.03762 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1706.03762.

[4] K. Kumar, A. Verma, N. Gupta and A. Yadav, 'Liquid neural networks: A novel approach to dynamic information processing,' in *2023 International Conference on Advances in Computation, Communication and Information Technology (ICAICCIT)*, 2023, pp. 725–730. DOI: 10.1109/ICAICCIT60255.2023.10466162.

[5] R. Hasani, M. Lechner, A. Amini, D. Rus and R. Grosu, *Liquid time-constant networks*, 2020. arXiv: 2006.04439 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2006.04439.

[6] R. Hasani, M. Lechner, A. Amini *et al.*, 'Closed-form continuous-time neural networks,' *Nature Machine Intelligence*, vol. 4, no. 11, pp. 992–1003, Nov. 2022, ISSN: 2522-5839. DOI: 10.1038/s42256-022-00556-7. [Online]. Available: http://dx.doi.org/10.1038/s42256-022-00556-7.

[7] C.-C. Chen, C.-L. Yang and H.-Y. Cheng, *Efficient and robust parallel dnn training through model parallelism on multi-gpu platform*, 2019. arXiv: 1809.02839 [cs.DC]. [Online]. Available: https://arxiv.org/abs/1809.02839.

[8] S. Li, Y. Zhao, R. Varma *et al.*, *Pytorch distributed: Experiences on accelerating data parallel training*, 2020. arXiv: 2006.15704 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2006.15704.

[9] J. Reyes-Ortiz, D. Anguita, A. Ghio and L. Oneto, *Human Activity Recognition Using Smartphones*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C54S4K, 2013.

[10] J. Hogue, *Metro Interstate Traffic Volume*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C5X60B, 2019.

[11] M. Academy, *What is the best neural network model for temporal data in deep learning?* https://magnimindacademy.com/blog/what-is-the-best-neural-network-model-for-temporal-data-in-deep-learning/, Accessed: 2025-06-05, 2023.

[12] A. Eliasy and J. Przychodzen, 'The role of ai in capital structure to enhance corporate funding strategies,' *Array*, vol. 6, p. 100 017, Jul. 2020. DOI: 10.1016/j.array.2020.100017.

[13] Wikipedia contributors, *Long short-term memory — wikipedia, the free encyclopedia*, https://en.wikipedia.org/wiki/Long_short-term_memory, Accessed: 2025-06-05, 2024.

[14] Wikipedia contributors, *Gated recurrent unit — wikipedia, the free encyclopedia*, https://en.wikipedia.org/wiki/Gated_recurrent_unit, Accessed: 2025-06-05, 2024.

[15] Wikipedia contributors, *Transformer (machine learning model) — wikipedia, the free encyclopedia*, https://en.wikipedia.org/wiki/Transformer_(machine_learning_model), Accessed: 2025-06-05, 2024.

[16] C. Olah, *Understanding lstm networks*, https://colah.github.io/posts/2015-08-Understanding-LSTMs/, Accessed: 2025-06-05, 2015.

[17] I. D. Mienye, T. G. Swart and G. Obaido, 'Recurrent neural networks: A comprehensive review of architectures, variants, and applications,' *Information*, vol. 15, no. 9, 2024, ISSN: 2078-2489. DOI: 10.3390/info15090517. [Online]. Available: https://www.mdpi.com/2078-2489/15/9/517.

[18] R. M. Schmidt, *Recurrent neural networks (rnns): A gentle introduction and overview*, 2019. arXiv: 1912.05911 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1912.05911.

[19] SuperAnnotate Team, *Activation functions in neural networks: Explained*, https://www.superannotate.com/blog/activation-functions-in-neural-networks, Accessed: 2025-06-05, 2023.

[20] S. Poudel, *Recurrent neural network (rnn) architecture explained*, https://medium.com/@poudelsushmita878/recurrent-neural-network-rnn-architecture-explained-1d69560541ef, Accessed: 2025-06-05, 2023.

[21] S. Hochreiter, *Untersuchungen zu dynamischen neuronalen netzen*, Apr. 1991.

[22] R. Robu, *Long short-term memory (lstm) networks*, https://medium.com/@RobuRishabh/long-short-term-memory-lstm-networks-9f285efa377d, Accessed: 2025-06-05, 2023.

[23] A. Zhang, Z. C. Lipton, M. Li and A. J. Smola, *Long short-term memory (lstm)*, https://d2l.ai/chapter_recurrent-modern/lstm.html, Accessed: 2025-06-05, 2023.

[24] S. Hochreiter and J. Schmidhuber, 'Lstm can solve hard long time lag problems,' Jan. 1996, pp. 473–479.

[25] F. Gers, J. Schmidhuber and F. Cummins, 'Learning to forget: Continual prediction with lstm,' in *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, vol. 2, 1999, 850–855 vol.2. DOI: 10.1049/cp:19991218.

[26] F. Gers, N. Schraudolph and J. Schmidhuber, 'Learning precise timing with lstm recurrent networks,' *Journal of Machine Learning Research*, vol. 3, pp. 115–143, Jan. 2002. DOI: 10.1162/153244303768966139.

[27] NVIDIA Corporation, *Understanding lstm neural networks*, https://developer.nvidia.com/discover/lstm, Accessed: 2025-06-05, 2024.

[28] K. Cho, B. van Merrienboer, C. Gulcehre *et al.*, *Learning phrase representations using rnn encoder-decoder for statistical machine translation*, 2014. arXiv: 1406.1078 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1406.1078.

[29] J. Chung, C. Gulcehre, K. Cho and Y. Bengio, *Empirical evaluation of gated recurrent neural networks on sequence modeling*, 2014. arXiv: 1412.3555 [cs.NE]. [Online]. Available: https://arxiv.org/abs/1412.3555.

[30] Y. Su and C.-C. J. Kuo, 'On extended long short-term memory and dependent bidirectional recurrent neural network,' *Neurocomputing*, vol. 356, pp. 151–161, Sep. 2019, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2019.04.044. [Online]. Available: http://dx.doi.org/10.1016/j.neucom.2019.04.044.

[31] A. Nama, *Understanding gated recurrent unit (gru) in deep learning*, https://medium.com/@anishnama20/understanding-gated-recurrent-unit-gru-in-deep-learning-2e54923f3e2, Accessed: 2025-06-05, 2023.

[32] GeeksforGeeks Contributors, *Gated recurrent unit networks*, https://www.geeksforgeeks.org/machine-learning/gated-recurrent-unit-networks/, Accessed: 2025-06-05, 2023.

[33] N. Gruber and A. Jockisch, 'Are gru cells more specific and lstm cells more sensitive in motive classification of text?' *Frontiers in Artificial Intelligence*, vol. 3, Jun. 2020. DOI: 10.3389/frai.2020.00040.

[34] M. Ravanelli, P. Brakel, M. Omologo and Y. Bengio, 'Light gated recurrent units for speech recognition,' *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 2, pp. 92–102, Apr. 2018, ISSN: 2471-285X. DOI: 10.1109/tetci.2017.2762739. [Online]. Available: http://dx.doi.org/10.1109/TETCI.2017.2762739.

[35] H. Han, H. Neira-Molina, A. Khan *et al.*, 'Advanced series decomposition with a gated recurrent unit and graph convolutional neural network for non-stationary data patterns,' *Journal of Cloud Computing*, vol. 13, no. 1, p. 20, 2024, ISSN: 2192-113X. DOI: 10.1186/s13677-023-00560-1. [Online]. Available: https://doi.org/10.1186/s13677-023-00560-1.

[36] Built In, *What is a transformer model?* https://builtin.com/artificial-intelligence/transformer-neural-network, Accessed: 2025-06-05, 2024.

[37] R. Y. Aminabadi, S. Rajbhandari, M. Zhang *et al.*, *Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale*, 2022. arXiv: 2207.00032 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2207.00032.

[38] K.-i. Funahashi and Y. Nakamura, 'Approximation of dynamical systems by continuous time recurrent neural networks,' *Neural Networks*, vol. 6, no. 6, pp. 801–806, 1993, ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(05)80125-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S089360800580125X.

[39] R. T. Q. Chen, Y. Rubanova, J. Bettencourt and D. Duvenaud, *Neural ordinary differential equations*, 2019. arXiv: 1806.07366 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1806.07366.

[40] H. Mei and J. Eisner, *The neural hawkes process: A neurally self-modulating multivariate point process*, 2017. arXiv: 1612.09328 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1612.09328.

[41] Y. Rubanova, R. T. Q. Chen and D. Duvenaud, *Latent odes for irregularly-sampled time series*, 2019. arXiv: 1907.03907 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1907.03907.

[42] M. Lechner and R. Hasani, *Learning long-term dependencies in irregularly-sampled time series*, 2020. arXiv: 2006.04418 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2006.04418.

[43] R. Hession, *Liquid neural nets (lnns)*, https://medium.com/@hession520/liquid-neural-nets-lnns-32ce1bfb045a, Accessed: 2025-06-05, 2023.

[44] R. Hasani, *Liquid neural networks*, https://www.youtube.com/watch?v=IlliqYiRhMU, Presentation at MIT AI Lecture Series, Accessed: 2025-06-05, 2021.

[45] M. A. D. Nguyen, *Megatron-lm: How model parallelism is pushing language models to new heights*, https://medium.com/@minhanh.dongnguyen/megatron-lm-how-model-parallelism-is-pushing-language-models-to-new-heights-c21a5343e06a, Accessed: 2025-06-05, 2022.

[46] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper and B. Catanzaro, *Megatron-lm: Training multi-billion parameter language models using model parallelism*, 2020. arXiv: 1909.08053 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1909.08053.

[47] Y. Huang, Y. Cheng, A. Bapna *et al.*, *Gpipe: Efficient training of giant neural networks using pipeline parallelism*, 2019. arXiv: 1811.06965 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1811.06965.

[48] D. Narayanan, M. Shoeybi, J. Casper *et al.*, *Efficient large-scale language model training on gpu clusters using megatron-lm*, 2021. arXiv: 2104.04473 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2104.04473.

[49] P. Team, *Distributed data parallel (ddp)*, https://pytorch.org/docs/stable/notes/ddp.html, Accessed: 2025-06-05, 2024.

[50] A. Sergeev and M. D. Balso, *Horovod: Fast and easy distributed deep learning in tensorflow*, 2018. arXiv: 1802.05799 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1802.05799.

[51] J. Rasley, S. Rajbhandari, O. Ruwase and Y. He, 'Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters,' in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20, Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 3505–3506, ISBN: 9781450379984. DOI: 10.1145/3394486.3406703. [Online]. Available: https://doi.org/10.1145/3394486.3406703.

[52] G. Hebrail and A. Berard, *Individual Household Electric Power Consumption*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C58K54, 2006.

[53] T. T.-K. Lau, W. Li, C. Xu, H. Liu and M. Kolar, *Adaptive batch size schedules for distributed training of language models with data and model parallelism*, 2025. arXiv: 2412.21124 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2412.21124.

[54] J. André, F. Strati and A. Klimovic, 'Exploring learning rate scaling rules for distributed ml training on transient resources,' in *Proceedings of the 3rd International Workshop on Distributed Machine Learning*, ser. DistributedML '22, Rome, Italy: Association for Computing Machinery, 2022, pp. 1–8, ISBN: 9781450399227. DOI: 10.1145/3565010.3569067. [Online]. Available: https://doi.org/10.1145/3565010.3569067.