Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Verification Methods for Liquid Neural Networks

Author:
Viyan Raj

Supervisor:
Dr. Alessio Lomuscio

Second Marker:
Alyssa Renata

June 11, 2025

Abstract

1 page

Acknowledgements

1 page

Contents

1 Introduction **7**

2 Background and Literature Review **8**
2.1 Liquid Neural Networks . . . 8
2.1.1 Continuous-Time Dynamical Systems/Differential Equations . . . 8
2.1.2 LNN Training . . . 9
2.1.3 LNN Inference . . . 9
2.1.4 Advantages of LNNs . . . 10
2.1.4.a Temporal Modelling . . . 10
2.1.4.b Adaptability . . . 10
2.1.4.c Efficiency . . . 10
2.1.4.d Stability . . . 11
2.2 Neural Network Verification . . . 11
2.2.1 The Verification Problem . . . 11
2.2.2 Motivation . . . 11
2.2.3 Recurrent Neural Networks . . . 12
2.2.4 Robustness Verification for Recurrent Neural Networks . . . 12
2.2.5 Symbolic and Interval Propagation Methods . . . 13
2.2.5.a SIP (Symbolic Interval Propagation) . . . 13
2.2.5.b CROWN (Certified Robustness to Weight Perturbations) . . . 14
2.2.5.c Lipschitz-Based Methods . . . 15

3 Liquid Neural Network Design and Implementation **17**
3.1 Design Overview . . . 17
3.2 Wiring and Connectivity . . . 17
3.3 LTC Neuron Dynamics . . . 18
3.4 Network Architecture . . . 19
3.5 Training Configuration (and dataset) . . . 21
3.6 Training Behaviour . . . 23
3.6.1 Inference . . . 26

4 Comparative Models **28**
4.1 Introduction to Baseline Models . . . 28
4.2 Temporal Convolutional Network (TCN) . . . 28
4.2.1 Overview and Motivation . . . 28
4.2.2 Theoretical Background . . . 28
4.2.3 Model Architecture . . . 29
4.2.4 Training Configuration . . . 30
4.2.5 Performance and Behaviour . . . 30
4.2.6 Design Considerations . . . 30
4.3 Long Short-Term Memory Network (LSTM) . . . 30
4.3.1 Background and Rationale . . . 30
4.3.2 LSTM Cell Mechanics . . . 30
4.3.3 Model Implementation . . . 30
4.3.4 Training Configuration . . . 31
4.3.5 Training Observations . . . 31
4.4 Transformer Model . . . 31

4.4.1 Training Configuration . . . 33
4.4.2 Performance and Behaviour . . . 33
4.4.3 Design Considerations . . . 33

5 Adversarial Attack Methodology **35**
5.1 Introduction to Adversarial Attacks . . . 35
5.2 Fast Gradient Sign Method (FGSM) . . . 35
5.2.1 Mathematical Formulation . . . 35
5.2.2 Implementation Details . . . 35
5.2.3 Evaluation and Observations . . . 36
5.3 Projected Gradient Descent (PGD) . . . 36
5.3.1 Mathematical Formulation . . . 36
5.3.2 Implementation Details . . . 36
5.3.3 Performance and Model Responses . . . 37
5.3.4 Design Choices . . . 37
5.3.5 DeepFool-Inspired Directional Attack . . . 37
5.3.6 Theoretical Basis . . . 37
5.3.7 Implementation Summary . . . 37
5.3.8 Model Comparisons and Observations . . . 38
5.3.9 Design Considerations . . . 38
5.4 Simultaneous Perturbation Stochastic Approximation (SPSA) . . . 38
5.4.1 Mathematical Formulation . . . 38
5.4.2 Implementation and Design Choices . . . 38
5.4.3 Empirical Performance Across Models . . . 39
5.4.4 Reflections on Robustness . . . 39
5.5 Time-Warping Attack . . . 39
5.5.1 Conceptual Basis . . . 39
5.5.2 Mathematical Formulation . . . 39
5.5.3 Implementation Strategy . . . 40
5.5.4 Results and Model Sensitivity . . . 40
5.5.5 Design Choices . . . 40
5.6 Continuous-Time Perturbation Attack . . . 40
5.6.1 Motivation . . . 40
5.6.2 Formulation and Mechanism . . . 41
5.6.3 Implementation Details . . . 41
5.6.4 Model-Specific Responses . . . 41
5.6.5 Design Rationale . . . 41
5.7 Summary of Attack Design and Implementation Decisions . . . 42
5.7.1 Attack Categories and Coverage . . . 42
5.7.2 Implementation Consistency . . . 42
5.7.3 Design Considerations . . . 42
5.7.4 Interpretation of Results . . . 42

6 Bound Certification (Auto Lirpa) **43**

7 Evaluation **44**
7.1 Quantitative Evaluation Metrics and Comparison . . . 44
7.1.1 Evaluation Metrics . . . 44
7.1.2 Aggregate Results . . . 45
7.1.3 Attack-Specific Breakdowns . . . 45
7.2 Qualitative Evaluation and Visual Analysis . . . 46
7.2.1 Visualisation Methodology . . . 46
7.2.2 LSTM Responses . . . 46
7.2.3 TCN Responses . . . 47
7.2.4 LNN Responses . . . 47
7.2.5 Comparative Failure Modes . . . 47
7.2.6 Phase Drift and Spiral Collapse . . . 47
7.2.7 Interpretive Summary . . . 47
7.3 Comparative Discussion of Model Robustness . . . 47

7.3.1 Summary of Behaviour Under Attack 47
7.3.2 Architectural Trade-offs 48
7.3.3 Robustness by Attack Type 48
7.3.4 Implications for Deployment 48
7.3.5 Conclusion 48

8 Conclusion **49**

9 Declaration **51**

Chapter 1

Introduction

Around 1-2 pages
[1]

Chapter 2

Background and Literature Review

Around 15-20 pages

In this chapter, we explore the current research on this topic. First, by exploring the theory behind liquid neural networks, in comparison to deep neural networks. Then, by investigating current verification methods and assessing their suitability to liquid neural networks. The reader should have an understanding of (traditional) neural networks and linear algebra concepts.

2.1 Liquid Neural Networks

Liquid neural networks, introduced by Ramin Hasani et al. (2021) [2], are a novel class of AI algorithms, designed to maintain adaptability after completing training. These are inspired by the communication patterns of brain cells, which are flexible and responsive to new/unseen data even after their initial training phase.

Traditional neural networks use fixed architectures and static parameters, so require retraining to handle new information. Liquid neural networks use **continuous-time dynamics** to enable their state to evolve smoothly over time. This means they can dynamically adjust their responses to changing inputs during inference. This structure allows these networks to be robust against perturbations and capable of generating complex behaviors without requiring large-scale architectures.

LNNs use differential equations to simulate the continuous/dynamic processing and plasticity of the brain. Since LNN neurons communicate selectively with a subset of other neurons, connections formed are sparse (unlike traditional deep neural networks with dense fully-connected layers). This makes LNNs more computationally efficient than deep neural networks.

There are a range of applications of LNNs. Hasani suggests that their inherent adaptability makes them suitable for tasks requiring real-time learning and decision-making, such as autonomous driving and medical diagnosis. Their efficiency could address several challenges associated with large-scale machine learning systems, including issues related to interpretability, accountability, and environmental impact due to high carbon footprints. [3]

2.1.1 Continuous-Time Dynamical Systems/Differential Equations

A **continuous-time dynamical system** is a mathematical model used to describe a system that evolves over time in a way that is continuous (rather than discrete). This means the state of the system changes smoothly as a function of time, without abrupt jumps.

In the context of liquid neural networks, the neurons' states evolve as continuous-time dynamical systems. Each neuron's state is governed by **differential equations**, enabling the network to process information dynamically and adaptively, much like physical systems in the real world. This is inspired by biological neurons, where the activity of each neuron is influenced dynamically by inputs and changes over time.

The state of each neuron $x_i(t)$ in an LNN evolves over time according to a differential equation, expressed as:

$$\frac{dx_i(t)}{dt} = f(x_i(t), u_i(t), t; \theta_i), \tag{2.1}$$

where:

- $x_i(t)$: The internal state of the i-th neuron at time t,
- $u_i(t)$: The input signal to the i-th neuron at time t,
- t: Time, treated as a continuous variable,
- θ_i: Trainable parameters of the neuron, such as weights and biases,
- $f(\cdot)$: A function (usually nonlinear) describing the neuron's dynamics.

A common differential equation is the **leaky integrator dynamics**, where the state evolves as:

$$\frac{dx_i(t)}{dt} = -\alpha x_i(t) + \sum_{j=1}^{N} w_{ij} h(x_j(t)) + u_i(t),$$

with:

- $-\alpha x_i(t)$: A "leakage" term causing the neuron's state to decay over time, with $\alpha > 0$ representing the decay rate (temporal decay),
- $\sum_{j=1}^{N} w_{ij} h(x_j(t))$: The weighted input from other neurons, where w_{ij} is the weight from neuron j to i, and $h(x_j(t))$ is a nonlinear activation function (e.g., tanh or ReLU),
- $u_i(t)$: An external input signal.

For more complex systems, **nonlinear terms** can be included, resulting in equations such as:

$$\frac{dx_i(t)}{dt} = g(x_i(t)) + \sum_{j=1}^{N} w_{ij} \sigma(x_j(t)) + u_i(t),$$

where $g(x_i(t))$ models intrinsic nonlinear dynamics, and $\sigma(x_j(t))$ is a nonlinear activation function.

A liquid neural network, as a continuous-time dynamical system, has several important features. First, it ensures **smooth evolution**, where the neuron states evolve continuously over time according to differential equations. This smooth state transition is essential for modeling time-dependent values in tasks like time-series forecasting or control systems. In addition, the dynamics of the network incorporate **time dependency** t explicitly or depend solely on the current state $x(t)$, enabling the network to capture both static and dynamic temporal relationships. Liquid neural networks are also typically **deterministic**, with their future states fully defined by the current states and inputs, but they can also accommodate **stochastic elements** to model uncertainty or noise in the environment. Finally, the network may operate under **linear** dynamics, such as $f(x) = Ax + Bu$, which are efficient but limited in complexity, or **nonlinear** dynamics, like $f(x) = \tanh(Wx + b)$, which allow the network to represent intricate patterns and adaptive behaviours.

2.1.2 LNN Training

During training, the above differential equations (2.1) define how each neurons processes information. For each labelled training data sample, the following process occurs.

During the **forward pass**, the system of differential equations is numerically solved over time, starting from an initial state $x(0)$. Inputs $u(t)$ and parameters θ_i drive the evolution of neuron states $x_i(t)$.

The network then outputs a value, derived from the neuron states. This is compared to the target output to compute a **loss function**.

During **backpropagation through time**, gradients of the loss with respect to trainable parameters (θ_i) are computed by differentiating through the differential equations using methods like automatic differentiation or adjoint sensitivity analysis.

Finally, **optimization algorithms** (e.g. gradient descent) update the parameters of the DEs to minimize the loss.

2.1.3 LNN Inference

During inference, the same differential equations govern the neuron states, but parameters (θ_i) are fixed. The network processes dynamic inputs $u(t)$ in real-time. The equation also considers the neuron's previous state $x_i(t)$, which is dependent on previous input values $u(t)$. Thus, the output of each neuron is dependent on the parameters, current input values, and previously seen input values.

2.1.4 Advantages of LNNs

Using differential equations in LNNs provides several advantages.

2.1.4.a Temporal Modelling

Continuous dynamics are well-suited for time-dependent tasks. This means LNNs can be used to find time-based relationships in data (temporal modeling). This form of 'memory' is highly beneficial in time-series tasks.

The nonlinear nature of $f(x_i(t))$ ensures that the network captures complex temporal dependencies, allowing it to adjust its behavior based on the sequence and timing of inputs. This dynamic capability provides the network with a form of memory, enabling it to adapt to new scenarios even outside the training set.

This is in contrast to static models which consider data points to be independent and identically distributed.

2.1.4.b Adaptability

Dynamic state evolution allows the network to adapt during deployment.

The differential equation model (2.1) for liquid neural networks allows for state evolution even after training, resulting in increased adaptability. This is achieved by the continuous dynamics governing neuron states, which enable the network to respond dynamically to real-time inputs and changing environments.

In the DE model, $x_i(t)$ is the state of the i-th neuron at time t, $u_i(t)$ represents external inputs, t is time, and θ_i are trainable parameters (e.g., weights and biases). After training, the parameters θ_i are fixed, but the neuron states $x_i(t)$ continue to respond dynamically to new inputs $u_i(t)$. This means the network integrates real-time inputs into its state over time, adapting its behavior dynamically to variations in the input patterns or the timing of events.

The differential equations governing the states ensure that even small variations in the input influence the system, enabling real-time adaptation.

This provides several advantages. LNNs excel in real-world scenarios involving dynamic environments, such as robotics [1] and control systems.

For example, an LNN controlling a robotic arm in a dynamic environment would learn general principles of motion and control during training. During inference, as new obstacles appear or external forces are applied, the network integrates this new information into its state $x_i(t)$ dynamically. This allows the robotic arm to adjust its movements in real time without needing retraining for each specific scenario.

In addition, by dynamically evolving its states, the network generalizes better to unseen data patterns by interpolating between learned behaviors.

2.1.4.c Efficiency

Liquid neural networks (LNNs) are inherently more efficient than traditional deep neural networks (DNNs) due to their ability to maintain sparser representations. At any given time, only a subset of an LNNs' neurons or parameters are significantly active or contribute to the system's computations. This sparsity reduces the computational overhead while retaining the network's performance and adaptability.

This is because continuous-time dynamics favor selective activity. In LNNs, neuron states evolved continuously over time, according to the differential equation 2.1. Here $f(\cdot)$ determines how each neuron state changes based on its inputs, past states, and parameters. The use of continuous-time dynamics enables neurons to become active only when relevant input signals $u_i(t)$ or temporal events trigger them. This selective activity leads to fewer neurons being active at a given time, resulting in sparse representations.

LNNs are designed to work efficiently with fewer parameters compared to DNNs. While in traditional DNNs, layers are often densely connected, meaning all neurons in one layer interact with all neurons in the next layer. LNNs use sparse connectivity patterns, where neurons only interact with a limited subset of other neurons. This reflects real-world systems, such as biological brains, where neurons form selective, sparse connections.The sparsity of connections reduces the number of computations required during both training and inference.

LNNs allow the internal states of neurons to evolve over time and depend on the dynamics of the inputs. Because of this adaptability, only neurons relevant to the current input remain active. This reduces unnecessary computations and avoids the inefficiencies of global activation in traditional DNNs.

The continuous dynamics of LNNs inherently encode temporal dependencies. Unlike recurrent neural networks (RNNs) or deep learning models that require explicit mechanisms like memory gates (e.g., in LSTMs or GRUs), LNNs rely on the fluid evolution of neuron states. This reduces the overhead of managing and updating memory states, further contributing to sparsity and efficiency.

The sparse nature of LNNs offers several advantages over traditional DNNs, including reduced computational cost (minimizing matrix operations), lower energy consumption, better scalability, and robustness to overfitting (as sparse connectivity can act as a regularization mechanism by ensuring only essential features are focussed on).

2.1.4.d Stability

LNNs exhibit greater stability and robustness to noise compared to traditional DNNs.

Continuous time dynamics and differential equations encode stability constraints, ensuring smooth transitions between states.

In LNNs, the state of each neuron evolves over time according to the differential equation 2.1. The continuous nature of these equations ensures that the neuron states change gradually over time. As a result, sudden spikes in the input $u_i(t)$ (caused by noise) are naturally smoothed out. This gradual evolution prevents abrupt changes in the neuron states, making the network less sensitive to transient noise.

In addition, neuron states evolve in response to both the current input $u_i(t)$ and past states $x_i(t)$. This integration over time allows the network to prioritize long-term patterns in the input and ignore temporary noise. The feedback from past states enables temporal filtering, where only meaningful input changes accumulate and influence the network's output. In contrast, the layer-by-layer static activations in DNNs make them more susceptible to noise.

In traditional DNNs, noisy inputs can propagate through the network, often being amplified by dense connections and static parameter updates. To avoid this, special techniques can be used such as dropout. However, LNNs achieve this implicitly, by using sparse and selective connections. This limits the propagation of noise across the network. The continuous evolution of states ensures that transient noise does not significantly affect downstream neurons or outputs.

This enhanced stability has a range of benefits. LNNs perform well in real-world settings where inputs are often corrupted. The intrinsic smoothing abilities of LNNs also reduces the amount of noise-filtering preprocessing required.

2.2 Neural Network Verification

In this section we explore the problem of neural network verification. We look at current methods used for DNN verification, which will form the inspiration for a liquid neural network verification approach. The suitability of these to LNNs must be evaluated.

2.2.1 The Verification Problem

Verification problems can involve concrete bounds on the input and linear programming (LP) constraints on the output. Formally, the problem can be defined as follows:

Definition 2.2.1.1 Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a neural network, and let $\mathcal{X} = \{x' \in \mathbb{R}^n \mid x_i^l \leq x_i' \leq x_i^u\}$ represent the set of valid inputs constrained by the lower and upper bounds x^l, x^u. Given a set of linear constraints on the output ψ_y, let $\mathcal{Y} = \{y \mid \psi_y\}$ denote the set of outputs satisfying ψ_y. The verification problem is to determine whether $x' \in \mathcal{X} \implies f(x') \in \mathcal{Y}$, or to find a counterexample $x' \in \mathcal{X}$ such that this implication is not true.

For input and output constraints as defined above, the goal is either to prove that no valid input violates the output constraints or to find an input that does. If no input satisfies the output constraints, we declare the property as "safe." Otherwise, if such an input exists, the property is deemed "unsafe," and the corresponding input serves as a counterexample. [4]

2.2.2 Motivation

Verification of neural networks is a crucial problem, especially when a new architecture (such as liquid neural networks) is being researched. This is because neural networks are often deployed in safety-critical

applications, such as autonomous vehicles or medical diagnosis, where unpredictability can cause significant harm. Neural networks are also vulnerable to adversarial attacks, which is when small perturbations within input data (often unnoticable to the human eye) cause significant undesired changes in the output. This vulnerability poses a serious threat to their reliability and trustworthiness. Verification ensures that the network behaves as expected under specified conditions, whilst robustness verification focuses on guaranteeing that small perturbations in the input do not lead to misclassifications or unsafe behavior. By formally proving properties of neural networks or identifying counterexamples, verification helps to ensure safety and mitigate risks in real-world deployments.

2.2.3 Recurrent Neural Networks

We now focus on the verification problem in relation to recurrent neural networks (RNNs) specifically. RNNs are deep neural networks trained on sequential or time series data to create a model that can make sequential predictions or conclusions based on sequential inputs. During both training and inference, they also use information from prior inputs to influence the current input and output. In traditional recurrent neural networks, this is achieved by a feedback loop within the network, containing a hidden state which 'remembers' previous inputs. [5]

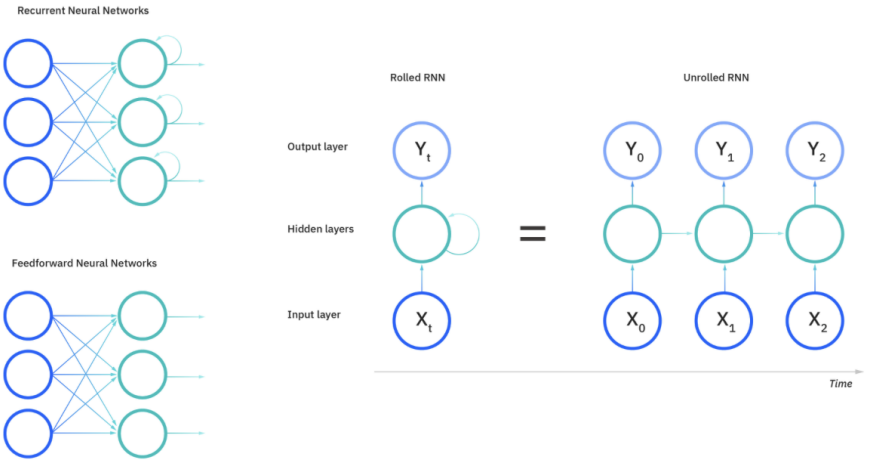


Figure 2.1: Feedforward (traditional) vs. Recurrent Neural Networks

RNNs rely on Backpropagation Through Time (BPTT) to compute gradients, unlike traditional neural networks that use standard backpropagation. Whilst BPTT follows the same principles as traditional methods, adjusting parameters by propagating errors from output to input layers, it accounts for sequential data by summing errors across time steps. This temporal error accumulation differentiates BPTT from the simpler gradient computation in feedforward networks, which lack a time-dependent structure.

'Memory' in RNNs can be achieved by several architectures, such as LSTM (Long Short-Term Memory), GRU (Gated Recurrent Units) and Encoder-decoder RNNs. Since liquid neural networks leverage differential equations to achieve a form of 'memory', they are a specialized type of RNN. Whilst their weights are typically fixed, the hidden state evolves dynamically over time, driven by the structure of the differential equations and the input. This continuous evolution allows liquid networks to retain memory and capture temporal dependencies across varying time scales.

The verification problem for RNNs involve constraints on both the input sequences and the dynamic outputs of the network, with the goal of solving the problem stated earlier (2.2.1). This requires handling both the sequence-based inputs and the time-evolving hidden states.

2.2.4 Robustness Verification for Recurrent Neural Networks

An important verification problem for RNNs concerns their robustness to temporal perturbations or noise in sequential inputs. Robustness implies that small perturbations in the input sequence do not cause significant deviations in the network's output. This can be formalized as follows:

Definition 2.2.4.1. Let $x \in \mathbb{R}^n$ be a sequential input to a recurrent neural network $f : \mathbb{R}^n \to \mathbb{R}^m$, where $m > 1$ represents the dimensionality of the output space. Let $\mathcal{C}_x = \{x' \in \mathbb{R}^n \mid x_i^l \leq x_i' \leq x_i^u\}$ represent the set of perturbed input sequences. The *targeted robustness verification problem* is to determine whether

$$x' \in \mathcal{C}_x \implies f(x')_c > f(x')_t \quad \text{for a specific output target } t,$$

or to find a counterexample x' such that this implication is not satisfied.

Definition 2.2.4.2 Let $x \in \mathbb{R}^n$ be a sequential input to a recurrent neural network $f : \mathbb{R}^n \to \mathbb{R}^m$, where $m > 1$. Let $\mathcal{C}_x = \{x' \in \mathbb{R}^n \mid x_i^l \leq x_i' \leq x_i^u\}$. The *general robustness verification problem* is to determine whether

$$x' \in \mathcal{C}_x \implies f(x')_c > f(x')_t \quad \text{for all } t \neq c,$$

or to find a counterexample x' where this implication fails.

Robustness verification for RNNs focuses on ensuring that temporal variations in sequential inputs do not cause undesired behavior. Specifically, the targeted robustness problem can often be reduced to verifying specific temporal constraints or perturbations in the input sequence. The general robustness problem is more challenging because it involves verifying the network's behavior across all possible perturbations and output dimensions. These problems may require advanced techniques, which will be discussed in the following sections.

2.2.5 Symbolic and Interval Propagation Methods

This section explores methods that focus on propagating bounds through liquid neural networks, which verifies their robustness when faced with input perturbations. **Symbolic Interval Propagation (SIP)** is a technique that computes conservative output bounds using interval arithmetic, providing a computationally efficient way to check for robustness. **CROWN (Certified Robustness to Weight Perturbations)** is more complex, and introduces linear approximations which improves the precision of robustness verification. **Lipschitz-based methods** they estimate global sensitivity by bounding the Lipschitz constant of the network. These approaches are particularly useful for ensuring scalability and efficiency, making them suitable for applications where lightweight and real-time verification is required.

2.2.5.a SIP (Symbolic Interval Propagation)

SIP is a scalable and efficient technique for verifying liquid neural networks by propagating symbolic intervals through the layers of the network to bound the range of possible outputs. For LNNs governed by neural ODEs with the equation 2.1, SIP can be applied to discretize and propagate bounds over time, handling non-linearity and ensuring robustness and safety properties are satisfied under perturbations.

Steps in SIP:

1. **Input Interval Initialization**: Define the input range as intervals $[l_i, u_i]$ for each dimension x_i, forming a hyper-rectangle. These intervals are represented symbolically to maintain dependencies between variables.

2. **Symbolic Propagation Through Layers**: At each discretized time step or layer:

   - **Affine Transformations**: For a layer $z = Wx + b$, bounds are propagated symbolically:

     $$l_z = W \cdot l_x + b, \quad u_z = W \cdot u_x + b.$$

   - **Nonlinear Activations**: Nonlinearities like ReLU or tanh are handled by updating bounds:

     $$\text{ReLU: } [\max(0, l_z), \max(0, u_z)], \quad \text{Tanh: } [\tanh(l_z), \tanh(u_z)].$$

3. **Output Interval Verification**: The final output intervals are compared against safety or robustness properties. For example, given input perturbations δx, the output bounds $F(x)$ must satisfy:

   $$F(x + \delta x) \subseteq [F_L, F_U],$$

   where F_L and F_U are symbolic bounds on the output.

Advantages for Liquid Neural Networks SIP is well-suited for LNNs due to its ability to handle the continuous evolution of states in neural ODEs:

- Precision: Symbolic intervals maintain variable dependencies, producing tighter bounds than traditional interval arithmetic.
- Efficiency: Propagation avoids the computational cost of exact methods, making SIP scalable for larger networks.
- Adaptability: SIP can handle nonlinear dynamics in LNNs through accurate approximations of activation functions.

Practical Considerations One consideration is over-approximation - accumulated conservativeness may reduce precision in deeper networks. Also, complex nonlinearities (within highly nonlinear layers such as softmax) require additional approximations, introducing potential conservatism. Finally, for neural ODEs, time discretization must balance accuracy with computational cost.

2.2.5.b CROWN (Certified Robustness to Weight Perturbations)

CROWN (Certified Robustness to Weight Perturbations) is a general framework for certifying the robustness of neural networks, including those with non-linear activation functions. It achieves this by bounding the outputs of the network using adaptive linear (or quadratic) upper and lower bounds for each activation function. For liquid neural networks, governed by neural ODEs, CROWN can be adapted to certify robustness by discretizing the continuous dynamics and applying its bounding technique to each time step.

CROWN provides an efficient and scalable method for verifying the robustness of LNNs. The propagated adapted bounds ensure that perturbations in the input do not lead to significant deviations in the output.

Mathematical Framework Consider a liquid neural network modeled as:

$$\frac{dx(t)}{dt} = f(x(t), t, \theta),$$

where $x(t) \in \mathbb{R}^n$ is the state, $f(x, t, \theta)$ describes the dynamics, and θ represents the parameters. For a given perturbed input $x_0 \in \mathbb{R}^n$ within an ℓ_p-ball:

$$x \in B_p(x_0, \epsilon) = \{x \mid \|x - x_0\|_p \leq \epsilon\},$$

CROWN aims to compute certified bounds $F_L(x) \leq F(x) \leq F_U(x)$, where $F(x)$ is the output of the neural ODE after a fixed time horizon T.

Bounding Nonlinearities For each activation function $\sigma(y)$, CROWN constructs linear upper and lower bounds:

$$h_U(y) = \alpha_U y + \beta_U, \quad h_L(y) = \alpha_L y + \beta_L,$$

such that $h_L(y) \leq \sigma(y) \leq h_U(y)$ over a pre-activation range $[l, u]$. These bounds are propagated through the layers of the network. For LNNs, this involves discretizing the time domain into intervals $[t_k, t_{k+1}]$ and applying the bounds iteratively at each time step.

Output Certification To certify robustness, CROWN computes bounds on the network's output $F(x)$. Using the layer-by-layer propagation of the upper and lower bounds, the output bounds are expressed as:

$$F_U(x) = \Lambda^{(0)} x + \sum_{k=1}^{m} \Lambda^{(k)} (b^{(k)} + \Delta^{(k)}), \quad F_L(x) = \Omega^{(0)} x + \sum_{k=1}^{m} \Omega^{(k)} (b^{(k)} + \Theta^{(k)}),$$

where Λ and Ω are matrices representing the upper and lower bound propagation, and Δ and Θ account for biases introduced by non-linearities.

Application to Neural ODEs For neural ODEs, the propagation framework is adapted to account for the continuous evolution of states. The bounds are computed at each discretized step t_k, ensuring that the dynamics satisfy the robustness conditions:

$$F_U(x_0) - F_L(x_0) \geq \delta,$$

where δ is the minimum required margin for robustness.

Practical Implementation CROWN can be implemented as follows:

1. **Define Input Bounds**: Specify the ℓ_p-ball around the input x_0 and initialize the pre-activation bounds for the first layer.

2. **Propagate Bounds**: Compute the upper and lower bounds layer-by-layer using the adaptive linear approximations.

3. **Certify Robustness**: Verify that the certified bounds at the output satisfy the desired robustness property (e.g., consistent classification).

[6]

2.2.5.c Lipschitz-Based Methods

Lipschitz-based methods provide a robust framework for verifying the safety and robustness of liquid neural networks by quantifying how sensitive a network's outputs are to perturbations in its inputs. Their ability to provide global robustness guarantees is useful for neural ODEs. The Lipschitz constant of a network bounds the maximum rate at which outputs can change with respect to changes in inputs, ensuring that small input perturbations do not lead to large deviations in the output.

Mathematical Foundation For a liquid neural network modeled as:

$$\frac{dx(t)}{dt} = f(x(t), t, \theta),$$

where $x(t) \in \mathbb{R}^n$ is the state, $f(x, t, \theta)$ defines the dynamics, and θ are the network parameters, the Lipschitz constant L satisfies:

$$\|F(x) - F(y)\| \leq L\|x - y\|, \quad \forall x, y \in \mathbb{R}^n,$$

where $F(x)$ represents the solution to the neural ODE at the final time T. The Lipschitz constant L bounds the global sensitivity of the network.

Estimation of the Lipschitz Constant The Lipschitz constant can be estimated for an LNN by analyzing the Jacobian of $f(x, t, \theta)$. For a time-discretized system, the sensitivity of the network is determined by:

$$L = \sup_{t \in [0,T]} \|J_f(x, t)\|,$$

where $J_f(x, t) = \frac{\partial f(x,t,\theta)}{\partial x}$ is the Jacobian matrix of f. Computing or bounding L involves:

- **Spectral Norm Analysis**: Evaluating $\|J_f(x, t)\|$ as the largest singular value of the Jacobian at each time step.

- **Pathwise Integral Bounds**: For neural ODEs, L can be bounded using the integral of the Jacobian along the trajectory:

$$L \leq \int_0^T \|J_f(x(t), t)\| dt.$$

Verification Applications Lipschitz-based methods are widely used for:

1. **Robustness Verification**: Verifying that small input perturbations $x_0 \to x_0+\delta x$ result in bounded output deviations, ensuring:
$$\|F(x_0 + \delta x) - F(x_0)\| \leq L\|\delta x\|.$$
2. **Safety Analysis**: Ensuring the network's outputs remain within a safe region under bounded input perturbations.
3. **Adversarial Robustness**: Certifying that adversarial inputs cannot change the classification or decision boundaries within a certain radius.

Practical Implementation To implement Lipschitz-based verification for LNNs:

1. **Compute the Lipschitz Constant**: Use numerical methods, such as spectral norm approximation or pathwise integration, to estimate L.
2. **Bound Output Sensitivity**: Evaluate L for given input perturbations δx and verify that the resulting outputs satisfy safety and robustness criteria.
3. **Scaling for Efficiency**: For high-dimensional networks, consider approximation techniques or layer-wise bounds to improve scalability.

Chapter 3

Liquid Neural Network Design and Implementation

3.1 Design Overview

This chapter outlines the implementation of the Liquid Neural Network (LNN) developed in PyTorch for sequential 2D time-series prediction. The architecture is based on the Liquid Time-Constant (LTC) neuron model, which simulates continuous-time dynamics through ordinary differential equations (ODEs) and shows properties of neural adaptability and temporal memory.

The aim of the implementation was to create a biologically-inspired, interpretable recurrent model with competitive performance on trajectory prediction tasks. Unlike conventional RNNs or LSTMs, the LNN is governed by time-continuous equations rather than discrete updates, providing finer control over neuronal dynamics.

The following principles guided the design:

- **Framework:** PyTorch was selected due to its flexible dynamic graph construction and ease of integrating custom layers with automatic differentiation.
- **Neuron Dynamics:** The neuron model was designed to emulate leaky integrate-and-fire (LIF) behaviour with added plasticity through modulated reversal potentials and conductances.
- **Time Unfolding:** Each forward pass of the LNN integrates over multiple internal time steps (ODE unfolds) to approximate the continuous-time solution, reflecting membrane voltage evolution.
- **Baseline Comparison:** To benchmark performance, identical training and evaluation protocols were implemented for alternative architectures (LSTM, TCN) using the same data.

The following sections document the architecture, neuron formulation, wiring strategy, training setup, and performance characteristics of the LNN.

3.2 Wiring and Connectivity

LNNs have a sparse and biologically motivated connectivity structure. To simulate the non-uniform and random nature of synaptic wiring observed in biological networks, a custom class named `RandomWiring` was implemented.

This class generates two adjacency matrices:

- A **recurrent adjacency matrix** of shape $(n \times n)$ defining internal connections between neurons within the hidden layer.
- A **sensory adjacency matrix** of shape $(d_{\text{in}} \times n)$ which defines the input-to-hidden connectivity.

Each matrix contains continuous values sampled from a uniform distribution on $[0, 1]$, which are later used to create binary masks or to modulate weight strengths.

The `RandomWiring` class also generates reversal potentials:

- `erev` for neuron-neuron connections.
- `sensory_erev` for input-synapse connections.

These potentials are initialised from a uniform range $[-0.2, 0.2]$ and are treated as fixed, non-learnable parameters in this implementation.

Key features of the `RandomWiring` class:

- **Biological plausibility:** Fixed sparse masks emulate the limited number of active connections in real cortical microcircuits.
- **Randomised initialisation:** Each instantiation of `RandomWiring` results in a different network topology, allowing stochastic variation in experiments.
- **Separation of sensory and recurrent dynamics:** By decoupling the sensory and recurrent wiring, the model can explicitly distinguish between input-driven and internal dynamic behaviour.

Below is a simplified example of the `RandomWiring` class:

```
class RandomWiring:
    def __init__(self, input_dim, output_dim, neuron_count):
        self.adjacency_matrix = np.random.uniform(0, 1, (neuron_count,
            neuron_count))
        self.sensory_adjacency_matrix = np.random.uniform(0, 1, (input_dim,
            neuron_count))

    def erev_initializer(self):
        return np.random.uniform(-0.2, 0.2, (neuron_count, neuron_count))

    def sensory_erev_initializer(self):
        return np.random.uniform(-0.2, 0.2, (input_dim, neuron_count))
```

Listing 3.1: Simplified RandomWiring class

3.3 LTC Neuron Dynamics

The core unit of the LNN is the `LIFNeuronLayer`, a custom PyTorch module that simulates the behaviour of leaky integrate-and-fire neurons (also known as liquid time-constant neurons). These neurons operate using a continuous-time dynamical model controlled by a first-order differential equation, capturing the evolution of membrane potentials in response to internal and external stimuli.

The model integrates over time using a discretised ODE solver implemented within the forward pass. Specifically, it unfolds the membrane update equation over a fixed number of steps (`ode_unfolds`) using an Euler-like method.
The update rule is determined by the equation:

$$v_t = \frac{c_m \cdot v_{t-1} + g_{\text{leak}} \cdot V_{\text{leak}} + I_{\text{syn}}}{c_m + g_{\text{leak}} + G_{\text{syn}} + \varepsilon}$$

where:

- c_m: membrane capacitance (learnable)
- g_{leak}: leak conductance (learnable)
- V_{leak}: leak reversal potential
- I_{syn}: synaptic current from sensory and recurrent inputs
- G_{syn}: total synaptic conductance
- ε: small stabilisation constant

Both sensory and recurrent synaptic activations are modelled using a sigmoid function with learnable μ (mean) and σ (scale), followed by a softplus-modulated weight:

$$\text{activation} = \text{Softplus}(W) \cdot \sigma\left(\frac{v - \mu}{\sigma}\right)$$

Implementation Details:

- **Learnable Parameters:** All biophysical constants—capacitance, leak conductance, reversal potentials, synaptic weights—are learnable, providing flexibility in dynamic behaviour.
- **Softplus Regularisation:** Weights and conductances are passed through `Softplus` to enforce positivity while allowing gradients to flow smoothly during training.
- **ODE Unfolding:** The number of internal solver steps is fixed (`ode_unfolds` = 12) to balance numerical precision with computational cost.
- **Sparsity Masks:** Both recurrent and sensory activations are element-wise masked using the adjacency matrices from `RandomWiring`, enforcing fixed sparsity throughout training.

Below is the ODE solver implementation within the `LIFNeuronLayer` class:

```
def ode_solver(self, inputs, state, elapsed_time):
    v_pre = state
    for _ in range(self.ode_unfolds):
        synaptic_input = compute_synaptic_activation(v_pre)
        numerator = self.cm * v_pre + self.gleak * self.vleak + synaptic_input
        denominator = self.cm + self.gleak + synaptic_conductance
        v_pre = numerator / (denominator + self.epsilon)
    return v_pre
```

Listing 3.2: Simplified LTC neuron forward method

This allows neurons to respond to both present input and also to their internal temporal dynamics, mimicking continuous-time memory traces observed in biological neurons.

3.4 Network Architecture

The full Liquid Neural Network is constructed by embedding the LTC neuron layer within a recurrent wrapper, implemented as a custom `LTCRNN` module. This wrapper sequentially passes each time step of the input through the same `LIFNeuronLayer`, maintaining a hidden state that evolves over time. The resulting structure can be viewed as a biologically grounded alternative to traditional RNN cells.

At a high level, the architecture accepts an input tensor of shape (B, T, d_{in}), where B is the batch size, T is the sequence length, and d_{in} is the input dimension (two in this case, corresponding to 2D spatial coordinates). For each time step t, the neuron layer receives the t-th slice of the sequence and updates the hidden state, generating a predicted output of shape (B, T, d_{out}).

Design Considerations and Tradeoffs:

- **Hidden state dimensionality:** The number of LTC neurons (set via `hidden_dim`) defines the model capacity. A lower number limits expressiveness but reduces overfitting risk and improves computational efficiency.
- **Output mapping:** Rather than applying a separate output layer, the voltage traces themselves are treated as predictions. This design allows direct interpretation of the membrane state as a continuous output signal.
- **Batch-first structure:** Following PyTorch conventions, all sequences are processed in batch-major form, allowing efficient tensor operations and GPU parallelism.

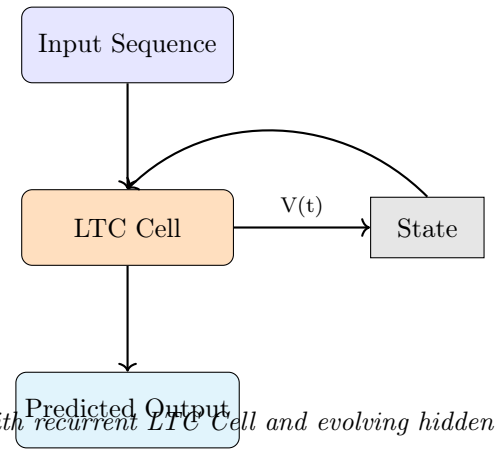
The architecture can be summarised as follows:

```
class LTCRNN(nn.Module):
    def __init__(self, wiring, input_dim, hidden_dim, output_dim):
        self.cell = LIFNeuronLayer(wiring)
        ...

    def forward(self, inputs):
        batch_size, seq_len, _ = inputs.size()
        states = torch.zeros(batch_size, self.hidden_dim)
        outputs = []
        for t in range(seq_len):
            out, states = self.cell(inputs[:, t, :], states)
            outputs.append(out)
        return torch.stack(outputs, dim=1)
```

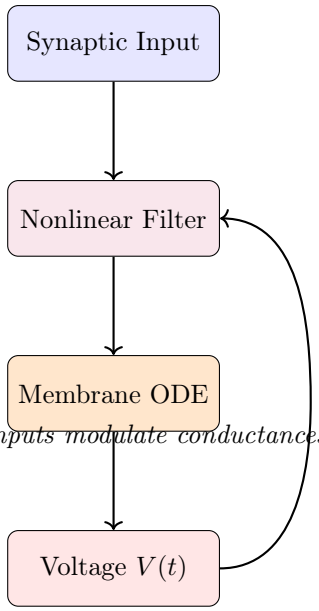
Listing 3.3: Structure of the LTCRNN module

The design maintains a clear separation between the continuous-time neuronal dynamics and the sequence-level integration logic. As a result, the architecture remains both modular and biologically interpretable, while still being compatible with modern deep learning tools.



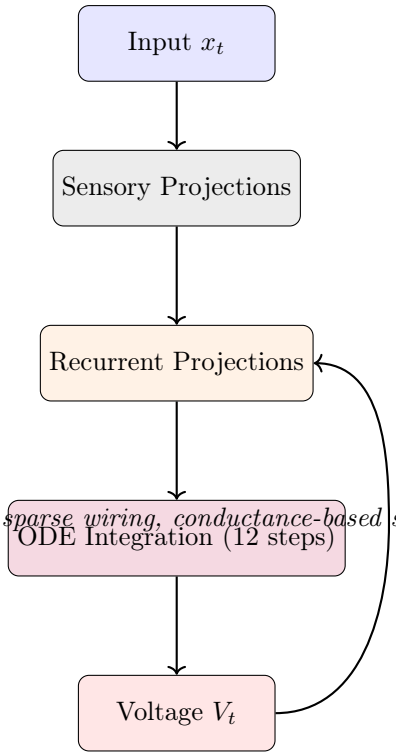
LTC-RNN Architecture with recurrent LTC Cell and evolving hidden state

Figure 3.1: High-level structure of the LTC-RNN. Each time step processes input via a shared LTC cell, with the membrane state passed forward recursively.



Dynamics within a single LIF neuron. Inputs modulate conductances, which drive membrane voltage.

Figure 3.2: Simplified computational path of a single LIF neuron. Conductance-based synaptic integration is performed iteratively over time.



Structure of the LIF neuron layer, combining sparse wiring, conductance-based synapses, and continuous-time integratio

Figure 3.3: LIF Neuron Layer in the LTC network. Sensory and recurrent inputs are processed through parameterised synapses and integrated using an ODE solver.

LTC ARCHITECTURE DIAGRAM HERE

3.5 Training Configuration (and dataset)

The LNN was trained on a synthetic 2D spiral trajectory dataset, chosen for its smooth temporal structure and nonlinearity. Each data point consists of an (x, y) coordinate, and the goal of the model is to predict

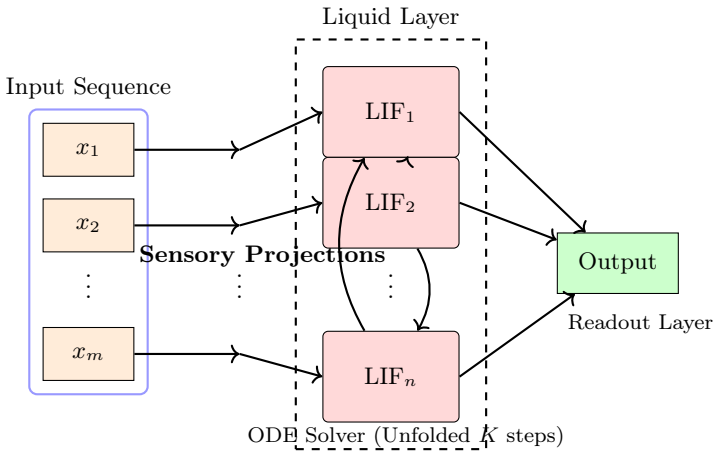


Figure 3.4: Architecture of the Liquid Time-Constant Network (LNN). Inputs project through learned sensory filters to a sparsely recurrent Liquid Layer of LIF neurons. Dynamics are integrated using an internal ODE solver with unfolding. Final outputs are read from a low-dimensional projection.

the next point in the sequence given a fixed-length input window. The sequential nature of the task makes it well-suited for testing temporal memory and continuous dynamics.

A supervised learning approach was used; inputs and targets were created by shifting a sliding window of length $T = 3$ over the full spiral. Each input sequence of three time steps was paired with the corresponding next three steps as the target output.

Data Preprocessing:

- All inputs were standardised using the training set mean and standard deviation.
- Targets were normalised in the same way to preserve scale consistency.
- The spiral dataset was generated programmatically with adjustable number of points and turns.

Training Parameters:

- **Loss function:** Mean Squared Error (`nn.MSELoss()`) was used to penalise deviations from the ground truth trajectory.
- **Optimiser:** Adam was chosen due to its fast convergence and robustness to parameter scaling. The learning rate was set to 0.005.
- **Epochs:** The model was trained for 2000 epochs to ensure convergence, with periodic visual evaluation every 100 epochs.
- **Batching:** Input sequences were split into overlapping windows and grouped into batches of size 32. This batching strategy allowed efficient GPU utilisation while preserving temporal continuity.
- **Train/validation split:** A random 80/20 split was used, with shuffling applied to prevent memorisation of input order.

Below is the training process implementation:

```
for epoch in range(num_epochs):
    lnn_model.train()
    total_loss = 0
    for x_batch, y_batch in zip(input_batches, target_batches):
        optimizer.zero_grad()
        outputs = lnn_model(x_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
```

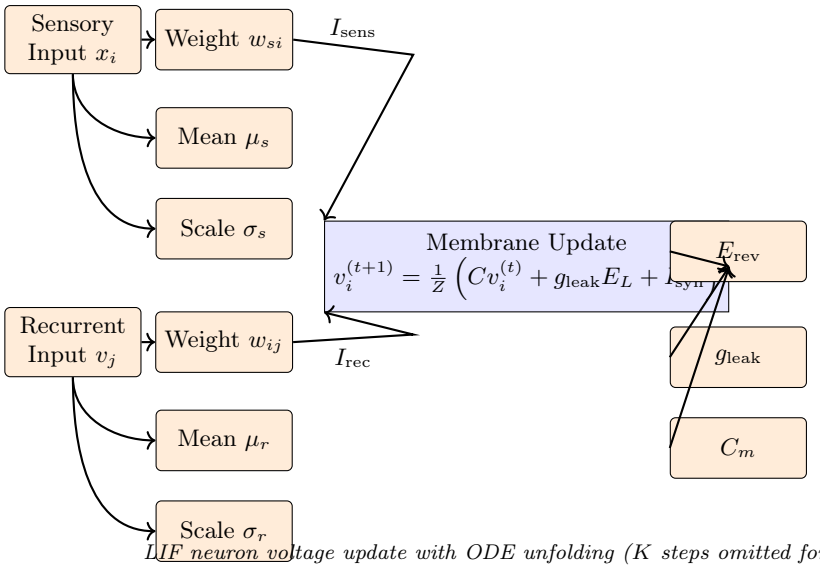


Figure 3.5: Internal structure of a single LIF neuron used in the Liquid Time-Constant Network. Inputs undergo non-linear transformations based on trainable μ and σ, with the resulting activations integrated using biophysical parameters such as leak conductance (g_{leak}), membrane capacitance (C_m), and reversal potentials (E_{rev}).

```
        optimizer.step()
        total_loss += loss.item()
```

Listing 3.4: Simplified training loop for the LNN

The training loop includes evaluation checkpoints where predicted trajectories are plotted and compared to ground truth. These visualisations provided insights into convergence behaviour, beyond scalar loss values.

Implementation Details:

- A small sequence length ($T = 3$) was chosen to reduce training complexity while still allowing temporal dependencies to be captured.
- Training on a synthetically generated spiral ensured control over noise and resolution, which allowed clearer attribution of error sources to model limitations rather than data irregularities.
- The validation split was kept random to mimic real-world test generalisation, though later sections explore unseen spiral generation for more robust testing.

3.6 Training Behaviour

Throughout training, model performance was monitored both quantitatively (via validation loss) and qualitatively (through trajectory plots). Evaluation occurred at regular intervals (every 100 epochs), allowing for close inspection of how well the LNN was capturing the underlying dynamics of the spiral sequence.
The primary trends observed during training were:

- Loss decreased steadily in early epochs, with diminishing returns as training progressed.
- In some cases, small fluctuations in validation loss were observed, likely due to the non-convexity of the parameter landscape and the biological variability induced by random wiring.
- Visual predictions of the trajectory showed clear improvement over time. Early predictions were coarse approximations, while later epochs yielded smoother and more accurate reconstructions.

Trajectory Loss Curves

The following plot illustrates training and validation behaviour over time:

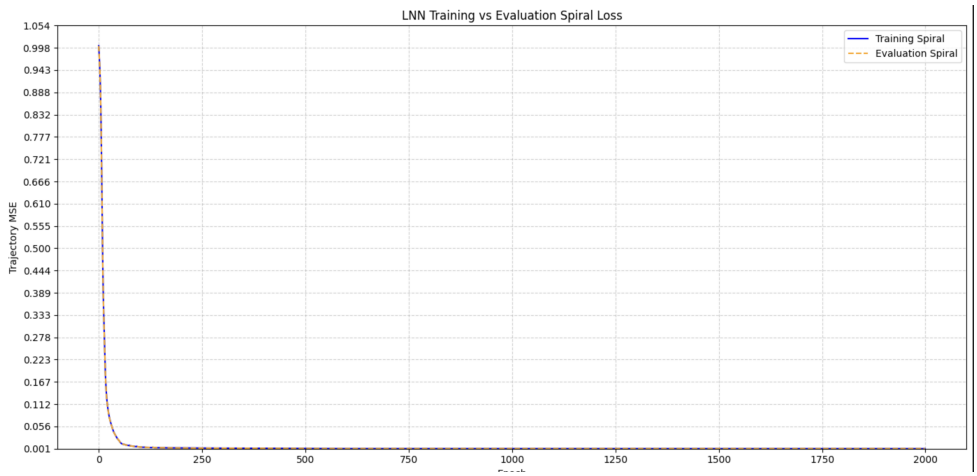


Figure 3.6: Training and validation loss over epochs.

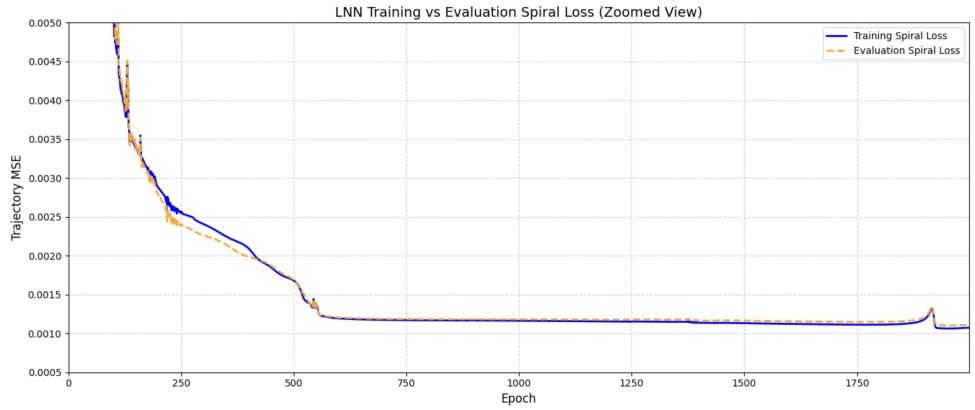


Figure 3.7: Training and validation loss over epochs.

To evaluate the model's learning progress and generalisation, we record the full-sequence mean squared error (MSE) first on the entire spiral (which contains the training datapoint and the held-out validation datapoints), and then on a separate unseen evaluation spiral at each epoch. Rather than separately plotting training and validation loss — which are contiguous segments of the same spiral trajectory, they are treated as a single sequence. This is because since temporal continuity is essential in modelling dynamical systems. This decision avoids misleading interpretations that might arise from artificial segmentation of a naturally evolving system.

The resulting loss curves, shown in Figure 3.6 and Figure 3.7, allow a direct comparison of model performance on the trajectory it is optimised on versus a held-out trajectory generated under the same dynamics but seeded differently. Early in training, both curves decrease rapidly, suggesting that the model learns the underlying structure efficiently. As training proceeds, the two curves converge, with the evaluation spiral maintaining a slightly higher loss—indicating some generalisation gap but also demonstrating stable extrapolation beyond the training path. Notably, neither curve exhibits significant divergence or overfitting behaviour, which supports the robustness and consistency of the trained model.

Qualitative Evaluation

Visually, the predicted path over time showed that the LNN was able to maintain smooth curvature and approximate the rotational dynamics of the spiral without overshooting or excessive lag. This was true even on validation data not seen during training.

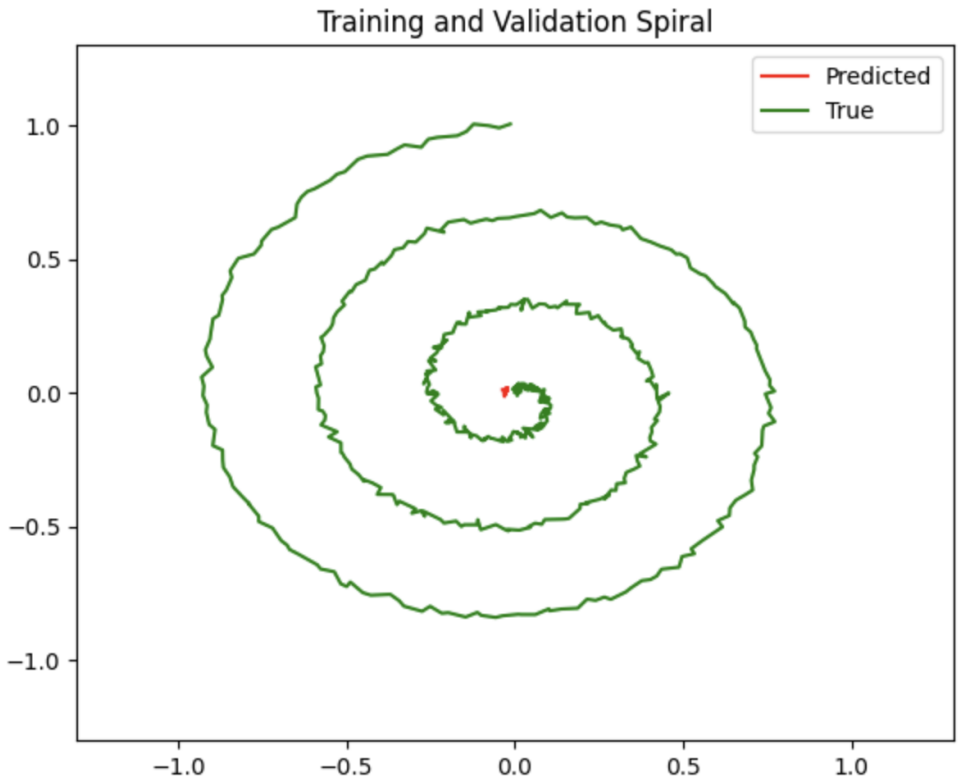


Figure 3.8: LLN predicted vs true spiral trajectory at epoch 1, on denormalised training (and validation) spiral

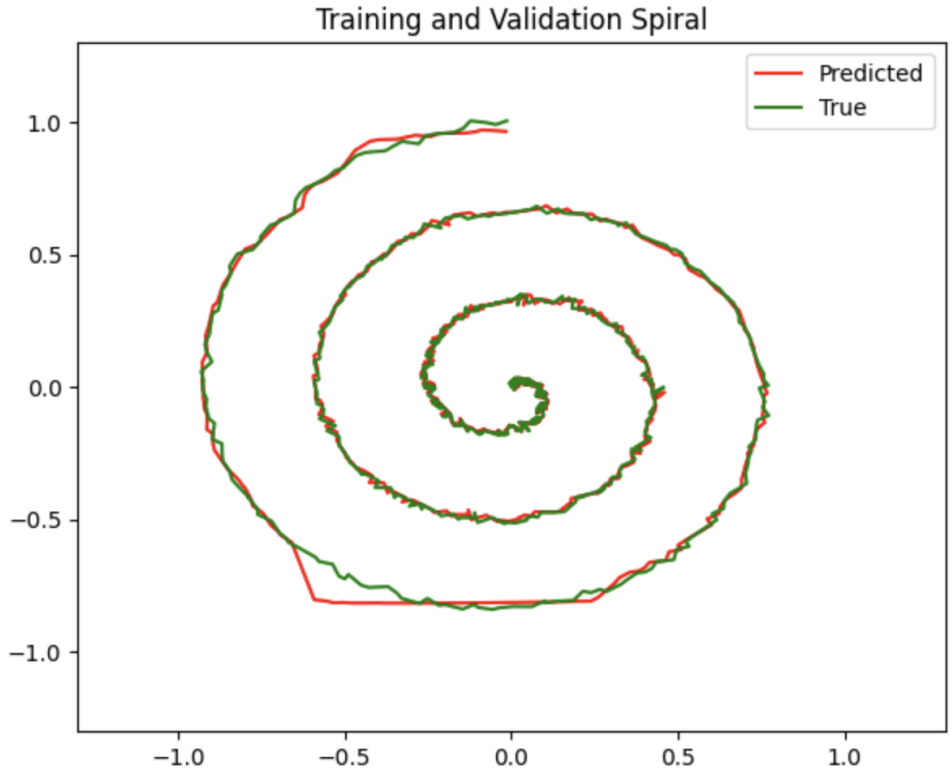


Figure 3.9: LLN predicted vs true spiral trajectory at epoch 400, on denormalised training (and validation) spiral

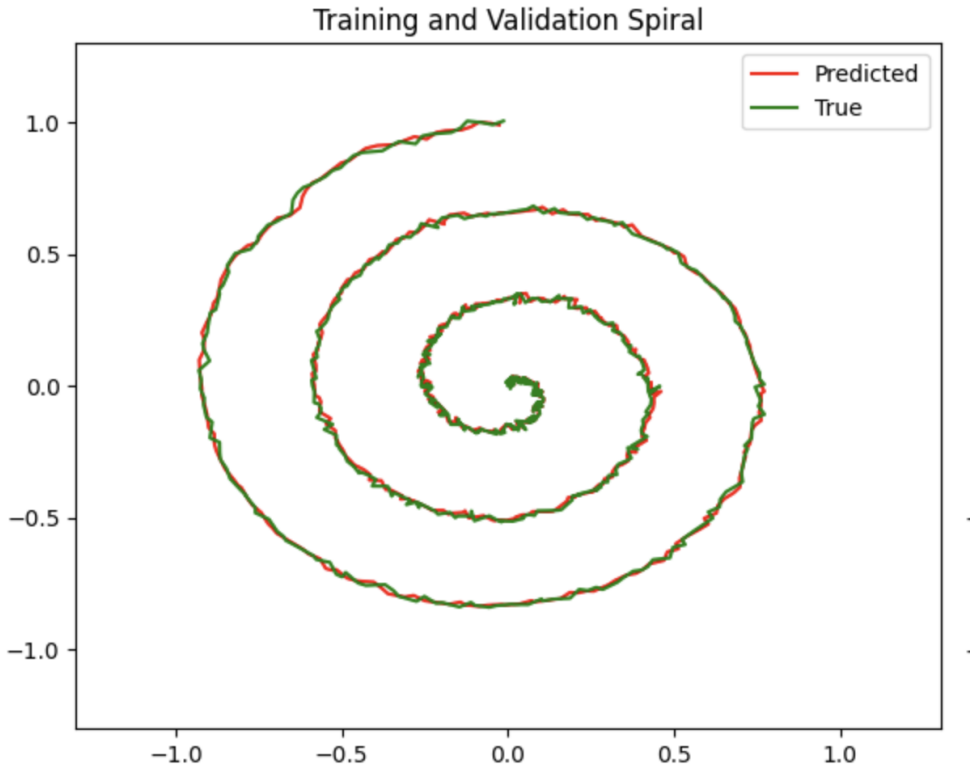


Figure 3.10: LLN predicted vs true spiral trajectory at epoch 2000, on denormalised training (and validation) spiral

Observed Patterns Across Training Runs:

- **ODE unfolding depth:** The number of internal steps in the membrane integration process contributed significantly to trajectory stability. Deeper unfolding improved smoothness, but with diminishing returns.
- **Effect of sparsity:** Fixed wiring helped constrain overfitting and may have contributed to better generalisation than a fully connected architecture.
- **Performance variation:** Some runs with different initialisations showed variability in loss curves and convergence speed, indicating sensitivity to initial wiring or parameter seeds.

Despite simpler architectures having shorter training times (e.g. LSTMs), LLN's had superior interprebility and stability in capturing the underlying continuous structure of the problem.

3.6.1 Inference

...

Although the models in this project were trained using fixed-size sliding windows (e.g., sequences of length 3 used to predict the next 3 points), inference is performed by passing the entire normalised input sequence to the model at once. Specifically, the inference function assumes that a model trained on short-range sequences can generalise to longer sequences when given a full trajectory (e.g., 199 input points to produce 199 output predictions). While this introduces a mismatch between training and inference regimes, the results obtained from this method—particularly on structured datasets such as smooth spiral trajectories—are stable, visually coherent, and achieve competitive prediction metrics.

This apparent generalisation is primarily due to the smooth, low-dimensional nature of the data. The spiral trajectories are continuous and predictable, and do not exhibit stochastic or chaotic behaviour. As a result, the models are able to extrapolate the learned local dynamics across the full trajectory during inference, even though they were not explicitly trained to do so.

Moreover, models like the Transformer and TCN use shared weights and position-aware mechanisms (such as causal convolutions or attention with position embeddings) that do not fundamentally restrict

them to a fixed input length. Consequently, when exposed to longer sequences, these architectures can continue to apply the same learned local filters or attention patterns, leading to coherent global predictions. Importantly, since the full sequence of input data is provided during inference, there is no recursive feeding of the model's own outputs—avoiding the accumulation of compounding errors often seen in autoregressive inference.

Nonetheless, this approach is not universally applicable. On tasks with more complex temporal dependencies, discontinuities, or chaotic dynamics, such inference behaviour could lead to silent degradation in performance. For such settings, autoregressive inference—where predictions are generated step-by-step and recursively fed into the model—would be more appropriate, as it more closely matches the training distribution and better captures error propagation over time.

In this work, however, the use of full-sequence inference is empirically validated to be reliable on the spiral trajectory dataset, offering a clean and efficient method for evaluating models without significant performance degradation.

Chapter 4

Comparative Models

4.1 Introduction to Baseline Models

To provide context for the robustness and verification of the Liquid Neural Network (LNN), we benchmark its performance against three established neural architectures: the Temporal Convolutional Network (TCN), the Long Short-Term Memory (LSTM) network, and the Transformer model. All models are trained on the same trajectory prediction task, using identical datasets, normalisation, loss functions, and training schedules as the LNN.

The selection of these baselines is motivated by their contrasting inductive biases and proven success in modelling sequential data. The LSTM is a recurrent architecture that introduces gating mechanisms and persistent internal states, enabling it to model long-range temporal dependencies through iterative state updates. The TCN, however, uses on dilated causal convolutions and fixed temporal receptive fields, making it structurally different from recurrent networks and well-suited for parallel computation. The transformer model is an attention-based architecture, which eliminates recurrence altogether, and dynamically weight input positions using self-attention mechanisms. It applies positional encodings to retain order information.

By evaluating the behaviour of these models under clean and adversarial conditions, we aim to identify both their predictive accuracy, and their robustness, sensitivity to perturbation, and qualitative output characteristics. These insights provide context for assessing the LNN in the following aspects:

- **Temporal memory:** How effectively each model retains and processes sequential dependencies
- **Structural robustness:** The influence of architectural constraints on noise sensitivity
- **Gradient stability:** The relationship between loss geometry and adversarial vulnerability

4.2 Temporal Convolutional Network (TCN)

4.2.1 Overview and Motivation

The Temporal Convolutional Network (TCN) is a fully convolutional architecture designed for sequential data. Unlike RNN-based models, which process inputs recursively and maintain an internal hidden state, TCNs rely on 1D convolutions applied over the temporal axis. This allows for parallel computation and more stable gradients, particularly for long sequences.

TCN's use **dilated convolutions**, which expand the receptive field exponentially with depth while preserving causality. This makes them highly effective at modelling long-range dependencies without the vanishing gradient issues that often affect RNNs.

4.2.2 Theoretical Background

For a 1D input sequence $x \in \mathbb{R}^{T \times d}$, a dilated convolution with kernel k and dilation factor d is defined as:

$$(y *_d k)(t) = \sum_{i=0}^{k-1} k(i) \cdot x(t - d \cdot i)$$

This structure allows the model to observe wider contexts with fewer parameters and layers.

In practice, the TCN is constructed using **residual blocks** with stacked dilated convolutions, dropout, and skip connections to stabilise training. Zero-padding is used to ensure output length matches input length.

4.2.3 Model Architecture

The implemented TCN consists of 3 residual blocks, each with:

- Two 1D convolutional layers with kernel size 3
- Dilation rates of 1, 2, and 4 respectively
- ReLU activations and dropout regularisation
- Optional 1x1 convolutions for matching input-output dimensions

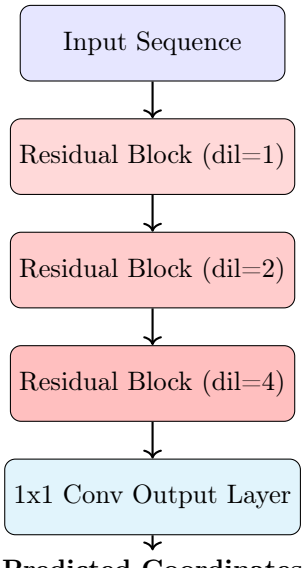
An output convolution maps the final hidden representation to the desired 2D coordinate space.

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, dilation, dropout
        ):
        ...
        self.conv1 = nn.Conv1d(..., dilation=dilation)
        self.conv2 = nn.Conv1d(..., dilation=dilation)

class TCN(nn.Module):
    def __init__(self, input_dim=2, hidden_channels=128, ...):
        self.tcn = nn.Sequential(*residual_blocks)
        self.output_layer = nn.Conv1d(hidden_channels, output_dim, 1)
```

Listing 4.1: Simplified TCN architecture

PUT TCN INFEENCE EXAMPLE/ARCHITECTURE DIAGRAM HERE



Stacked Dilated Residual Convolutions in the TCN

Figure 4.1: Temporal Convolutional Network (TCN) architecture with 3 residual blocks and exponentially increasing dilation. Each block contains two dilated convolutions with dropout, ReLU, and optional skip connection.

4.2.4 Training Configuration

The TCN was trained on the same spiral dataset as the LNN, with identical batch size, learning rate, loss function (Smooth L1), and normalisation pipeline. The model was optimised using Adam and a learning rate scheduler that halved the rate every 500 steps.

4.2.5 Performance and Behaviour

The TCN demonstrated strong performance on the trajectory prediction task, converging more quickly than the LNN and producing smooth outputs even with a small receptive field. The use of dilated convolutions allowed the model to predict coordinated curvature without explicitly tracking hidden state over time.

4.2.6 Design Considerations

- **Causality:** All convolutions were causal, ensuring no future information was used during prediction.
- **Parameter efficiency:** Despite having no recurrence, the TCN was able to model complex spirals with relatively few layers and a compact parameter set.
- **Regularisation:** Dropout was used within each block to avoid overfitting, as convolutional models tend to memorise local structures in small datasets.

Despite lacking the dynamic time constants of the LNN, the TCN proved to be a strong baseline in terms of speed, stability, and accuracy under clean conditions.

4.3 Long Short-Term Memory Network (LSTM)

4.3.1 Background and Rationale

The Long Short-Term Memory (LSTM) network is a popular recurrent neural architectures for sequential learning tasks. It was introduced to address the limitations of classical RNNs, particularly the vanishing and exploding gradient problems during backpropagation through time. The LSTM introduces gated memory units that regulate the flow of information over time.

Their ability to retain past information via internal cell states makes them well-suited for temporal tasks such as trajectory prediction.

4.3.2 LSTM Cell Mechanics

An LSTM cell maintains two internal states: a hidden state h_t and a cell state c_t. The cell's behaviour is controlled by three gates:

$$
\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) && \text{(forget gate)} \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) && \text{(input gate)} \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) && \text{(output gate)} \\
\tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) && \text{(cell candidate)} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

These equations define how the LSTM updates its memory and hidden representations at each time step.

4.3.3 Model Implementation

In this project, the LSTM was implemented using PyTorch's built-in `nn.LSTM` module. A two-layer LSTM was used, with 128 hidden units per layer. The final hidden state was passed through a linear projection layer to produce the 2D coordinate output.

```python
class LSTMModel(nn.Module):
    def __init__(self, input_dim=2, hidden_dim=128, num_layers=2, output_dim=2):
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)
        self.output_layer = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        out, _ = self.lstm(x)
        return self.output_layer(out)
```

Listing 4.2: Simplified LSTM model structure

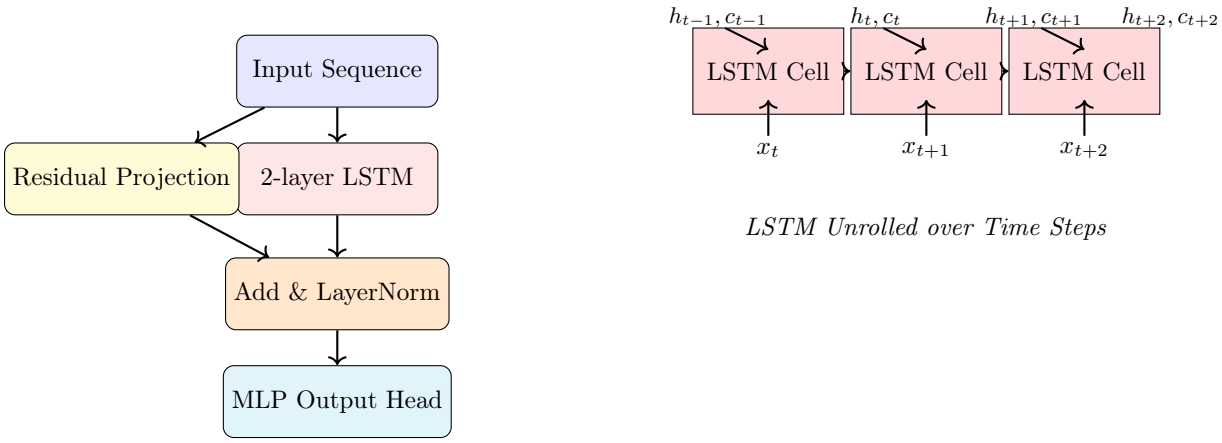
4.3.4 Training Configuration

The LSTM was trained using the same dataset and preprocessing pipeline as the LNN and TCN. The Smooth L1 loss was used, and training was performed over 1000 epochs with a learning rate of 0.005. A step decay scheduler was applied halfway through training.

4.3.5 Training Observations

The LSTM showed stable training behaviour and low final validation loss. However, unlike the TCN and LNN, it exhibited slightly slower convergence. Its outputs were smooth and consistent, although it occasionally underfit regions with sharper curvature.

PUT LSTM INFEENCE EXAMPLE/ARCHITECTURE DIAGRAM HERE



LSTM Unrolled over Time Steps

Figure 4.2: Architecture of the LSTM model. The left shows residual-enhanced flow through a 2-layer LSTM, while the right shows the LSTM unrolled across time with hidden and cell state transitions.

While the LSTM offers a reliable baseline for temporal prediction, its recurrent structure can make it more sensitive to gradient-based perturbations.

4.4 Transformer Model

Overview and Motivation

The Transformer is an attention-based architecture originally developed for sequence transduction tasks in NLP. The Transformer architecture implemented in this project is a lightweight variant of the original Transformer encoder proposed by Vaswani et al. (2017), adapted for short-length continuous 2D time-series data. Unlike recurrent or convolutional models, the Transformer uses **self-attention** to learn dependencies across the input sequence in parallel. This decouples sequence processing from sequential computation and enables greater flexibility in learning temporal relationships.

Transformer Encoder Design

The core of the model is the **Transformer encoder**, a stack of layers built around self-attention and feedforward submodules. Each encoder layer learns to transform the input sequence into a more abstract representation by allowing each token (i.e., timestep) to attend to others, subject to a causal constraint.

Each encoder layer consists of the following components:

- **Multi-Head Self-Attention:** The model uses four attention heads, each projecting the input into different subspaces. For an input sequence $X \in \mathbb{R}^{T \times d}$, the self-attention mechanism computes:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}} + M\right)V$$

  where Q, K, and V are learned linear projections of the input, and M is a **causal mask**—a triangular matrix filled with $-\infty$ above the diagonal to prevent attention to future positions.

- **Feedforward Network:** Following attention, a two-layer feedforward block applies a non-linear transformation independently at each position:

$$\text{FFN}(x) = \text{GELU}(xW_1 + b_1)W_2 + b_2$$

  with an intermediate hidden size of 256 and GELU activation, chosen for its smooth gradient properties.

- **Residual Connections and Layer Normalisation:** Both the attention and feedforward submodules are wrapped in residual connections and followed by `LayerNorm`, ensuring stable training and gradient propagation even across deep encoder stacks:

$$\text{LayerNorm}(x + \text{SubLayer}(x))$$

- **Dropout:** Dropout is applied to both the attention weights and the feedforward layers with a rate of 0.1, providing regularisation and improving generalisation on small datasets.

Two of these encoder layers are stacked, allowing the model to build hierarchical abstractions over the input sequence.

Thus, the transformer encoder processes input sequences in parallel, whilst still capturing autoregressive temporal dependencies via the attention mask. The resulting sequence of contextualised embeddings is then passed to a prediction head that maps each timestep to its corresponding 2D coordinate.

Positional information is injected via **learnable positional embeddings**, which are added to the input projections before encoding. A **causal mask** is applied to ensure predictions at time t do not access future values (enforcing temporal directionality).

Model Architecture

The Transformer model used in this work consists of: an input projection layer mapping 2D inputs to a 128-dimensional latent space, two Transformer encoder layers, a causal attention mask (to enforce autoregressive prediction), and a residual MLP head that maps the encoder output back into 2D coordinates.

Suitability for the Task

Despite the short sequence length (seq_len = 3), the Transformer architecture is well-suited to this task for several reasons:

Suitability for the Task

- **Parallel Processing:** All timesteps are processed concurrently, improving training speed and efficiency.

- **Long-Term Dependency Modelling:** Self-attention generalises well to longer sequences and higher-resolution datasets, allowing the model to capture long-range dependencies.

- **Positional Awareness Without Recurrence:** The use of learnable positional embeddings enables the model to adaptively distinguish temporally adjacent inputs in the absence of hidden-state propagation.

- **Smooth Output Dynamics:** Empirically, the Transformer achieves validation loss in the range of 0.0002–0.0006, yielding stable and continuous predictions consistent with the underlying spiral dynamics.

- **Robustness to Input Variability:** Self-attention allows the model to focus on the most informative timesteps, reducing sensitivity to noise or input distortions.

```
class TransformerModel(nn.Module):
    def __init__(self, input_dim=2, model_dim=128, ...):
        self.input_proj = nn.Linear(input_dim, model_dim)
        self.encoder = nn.TransformerEncoder(...)
        self.output_head = nn.Sequential(
            nn.Linear(model_dim, model_dim // 2),
            nn.ReLU(),
            nn.Linear(model_dim // 2, output_dim)
        )
```

Listing 4.3: Simplified Transformer architecture

4.4.1 Training Configuration

The Transformer was trained on the same spiral dataset as other baselines, using identical input preprocessing and normalisation. Optimisation was performed using AdamW with cosine annealing, and Smooth L1 loss was used as the objective. The model received short input sequences (length 3), and positional embeddings were learned from scratch.

4.4.2 Performance and Behaviour

Although the Transformer initially showed promising performance, with early convergence to low validation loss, extended training often led to **overfitting**, increasing loss and reduced trajectory fidelity. Its attention-based mechanism enabled flexibility but also made it sensitive to noise and hyperparameters, particularly given the small training context.

PUT TRANSFORMER INFERENCE EXAMPLE/ARCHITECTURE DIAGRAM HERE

4.4.3 Design Considerations

- **Parallelism:** Unlike the LSTM and TCN, the Transformer operates without recurrence or convolution, allowing full-sequence parallelism during training.

- **Positional encoding:** A learnable position embedding allows the model to infer order, compensating for the lack of temporal structure in pure attention.

- **Generalisation:** The model's performance was sensitive to overfitting, highlighting a trade-off between expressivity and inductive bias in low-data regimes.

The Transformer model provides a powerful and flexible alternative to recurrent and convolutional architectures. However, its high capacity and limited structural bias made it more susceptible to generalisation issues under the noisy spiral task compared to more structured models like the TCN or LSTM.

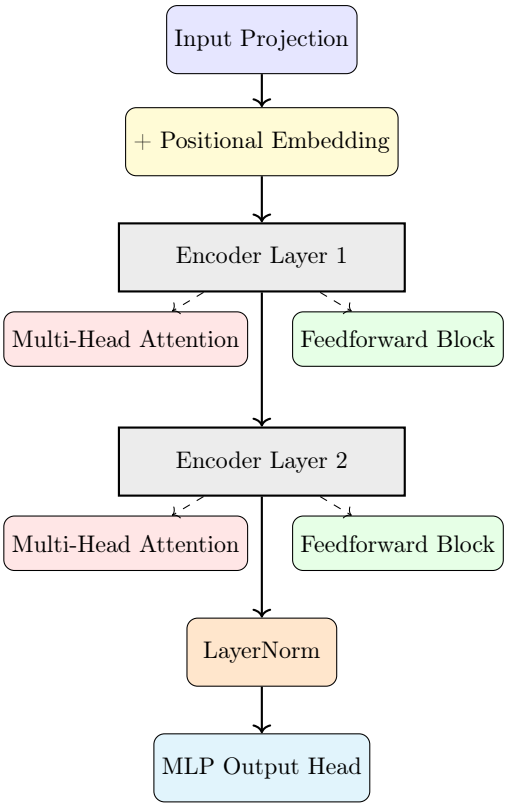


Figure 4.3: Architecture of the Transformer encoder used

Chapter 5

Adversarial Attack Methodology

5.1 Introduction to Adversarial Attacks

Adversarial attacks are deliberately constructed perturbations to input data that cause a machine learning model to make incorrect predictions with high confidence. These perturbations are often imperceptible or bounded in norm, but can expose vulnerabilities in the model's internal representations and loss surface geometry.

For sequential models such as the LNN, TCN, and LSTM, adversarial robustness is important, especially in safety-critical applications involving temporal dynamics. Several attacks are implemented in this project, targeting both gradient-accessible and gradient-free contexts, and include both white-box and black-box variants.

Each attack was evaluated under the same conditions, using:

- A fixed perturbation budget ϵ.
- Normalised data inputs, with identical initial conditions across models.
- Denormalised outputs for interpretability and comparison.

The metrics used for evaluating adversarial degradation included **Degradation Ratio**, **Deviation**, anf **Local Sensitivity** (Lipschitz constant). These are explained further, with analysis of results across all models, in the evaluation section.

5.2 Fast Gradient Sign Method (FGSM)

The Fast Gradient Sign Method (FGSM) is a single-step adversarial attack first introduced by Goodfellow et al. in 2014. It exploits the local linearity of neural networks by using the gradient of the loss function with respect to the input to perturb the input data in the direction that maximally increases loss.

5.2.1 Mathematical Formulation

Given a model f_θ, a loss function $\mathcal{L}(f_\theta(x), y)$, and a clean input-target pair (x, y), the FGSM adversarial example is constructed as:

$$x^{\text{adv}} = x + \epsilon \cdot \text{sign}\left(\nabla_x \mathcal{L}(f_\theta(x), y)\right)$$

where ϵ controls the perturbation magnitude and $\text{sign}(\cdot)$ is applied elementwise. The method requires only a single forward and backward pass.

5.2.2 Implementation Details

FGSM was implemented using PyTorch's autograd engine. The input tensor was marked with `requires_grad=True` and gradients were computed by backpropagating through the MSE loss between model output and the clean target sequence. The sign of the gradient was scaled by ϵ and added to the input.

```
loss = F.mse_loss(model(x), y)
loss.backward()
perturbation = epsilon * x.grad.sign()
x_adv = x + perturbation
```

Listing 5.1: FGSM adversarial attack implementation

5.2.3 Evaluation and Observations

The FGSM attack was applied to all three models under a fixed perturbation budget $\epsilon = 0.05$. Key findings include:

- **LSTM:** Showed significant degradation, especially in regions with abrupt curvature. The gating mechanisms did not mitigate linear perturbations.
- **TCN:** Relatively robust in early regions of the spiral but vulnerable at turn boundaries. This is likely due to reliance on local receptive fields.
- **LNN:** Demonstrated moderate degradation. The neuron dynamics offered some resistance to sharp perturbation, but sensitivity remained in areas where the membrane potential saturated.

PUT FGSM TRAJECTORIES HERE? or maybe in evaluation section

Design Reflection: FGSM is efficient but limited, as it assumes linearity and is easy to defend against with basic regularisation, so was used as a baseline attack method.

5.3 Projected Gradient Descent (PGD)

The Projected Gradient Descent (PGD) attack is an iterative extension of FGSM and is regarded as one of the strongest first-order adversaries in adversarial machine learning. Proposed by Madry et al., PGD performs multiple small steps of perturbation in the direction of the loss gradient, while projecting the adversarial input back onto an ℓ_p ball of fixed radius after each step.

5.3.1 Mathematical Formulation

Given an input x and perturbation budget ϵ, PGD initializes the adversarial input as $x_0^{\text{adv}} = x + \delta$ (with δ small or random), and iteratively updates it as follows:

$$x_{t+1}^{\text{adv}} = \Pi_{B_\epsilon(x)} \left(x_t^{\text{adv}} + \alpha \cdot \text{sign} \left(\nabla_x \mathcal{L}(f_\theta(x_t^{\text{adv}}), y) \right) \right)$$

Here, $\Pi_{B_\epsilon(x)}$ denotes projection onto the ℓ_∞ ball centered at x with radius ϵ, and α is the step size.

5.3.2 Implementation Details

The attack was implemented using a fixed number of iterations (10), and in each step:

- The model performed a forward pass on the current adversarial input.
- The loss was computed and backpropagated to obtain input gradients.
- The adversarial input was updated using the signed gradient and clipped back to the ϵ-bounded domain.

```
for _ in range(num_iter):
    output = model(x_adv)
    loss = F.mse_loss(output, target)
    loss.backward()
    with torch.no_grad():
        x_adv += alpha * x_adv.grad.sign()
        perturbation = torch.clamp(x_adv - x_orig, min=-epsilon, max=epsilon)
        x_adv = torch.clamp(x_orig + perturbation, min, max).detach().
            requires_grad_()
```

Listing 5.2: PGD Attack Loop (Simplified)

5.3.3 Performance and Model Responses

PGD caused greater degradation than FGSM across all models, with stronger effect on deeper temporal structures. Results indicated:

- **LSTM:** Highly vulnerable. PGD-induced drift accumulated over time, causing the model to diverge from the ground truth trajectory.
- **TCN:** Whilst convolutional structure dampened some effects, the model's locality made it sensitive to consistent directional gradients across the sequence.
- **LNN:** Showed meaningful robustness - the continuous-time integration added temporal stability which dampened the effect of rapid perturbations. However, convergence was sensitive to α and step count.

PUT PGD TRAJECTORIES HERE? or maybe in evaluation section

5.3.4 Design Choices

- **Step size α:** Set as 0.01 after empirical tuning to balance convergence and perturbation spread.
- **Projection radius ϵ:** Fixed at 0.05 to match FGSM budget for fair comparison.
- **Clipping bounds:** Enforced to retain normalised input range and ensure comparability with clean evaluations.

Unlike FGSM, PGD exposes high-curvature regions of the loss surface. The extent to which a model resists PGD steps provided insight into the local geometry of its input-output mapping.

5.3.5 DeepFool-Inspired Directional Attack

Whilst FGSM and PGD are effective, they rely on sign-based or norm-bounded perturbations and can be inefficient in identifying the minimal perturbation required for misprediction. DeepFool, introduced by Moosavi-Dezfooli et al., aims to iteratively approximate the closest decision boundary in input space. Though originally formulated for classification, a modified version was implemented here to exploit the gradient direction of loss for regression.

5.3.6 Theoretical Basis

In its original form, DeepFool linearises the classifier around the current point and computes the minimal step in the direction of the gradient that crosses the decision boundary. In our regression-focussed adaptation, the perturbation is applied directly in the normalised direction of the loss gradient, without projection.

The update rule is given by:

$$x^{\text{adv}} = x + \eta \cdot \frac{\nabla_x \mathcal{L}(f(x), y)}{\|\nabla_x \mathcal{L}(f(x), y)\|_2 + \delta}$$

where η is a scalar perturbation magnitude and δ is a small stabilisation term to prevent division by zero.

5.3.7 Implementation Summary

The attack was implemented using a single or few iterations, computing the raw gradient of the loss with respect to the input and stepping along the normalised direction. Unlike PGD, no projection or clipping was applied. This was chosen to explore worst-case directional drift.

```
loss = F.mse_loss(model(x), y)
loss.backward()
gradient = x.grad.data
perturbation = eta * gradient / (torch.norm(gradient) + epsilon)
x_adv = x + perturbation
```

Listing 5.3: Directional (DeepFool-like) Gradient Attack

5.3.8 Model Comparisons and Observations

- **LSTM:** Exhibited strong drift under this attack, especially near the sequence midpoint where cell state updates accumulate. Perturbations along raw gradients quickly destabilised the output.
- **TCN:** Localised convolutional features led to sharper local deformation, but degradation plateaued after initial displacement.
- **LNN:** Resistant to small η, but susceptible when gradient directions aligned with sensitive voltage states. Nonlinearity in ODE integration helped to minimise the impact in early layers.

PUT PGD TRAJECTORIES HERE? or maybe in evaluation section

5.3.9 Design Considerations

- **Normalisation:** Gradient was normalised using ℓ_2 norm rather than using the sign, to emulate the boundary-seeking nature of DeepFool.
- **No projection:** Allowed the perturbation to fully reflect the underlying geometry of the loss surface, rather than artificially constraining it.
- **Step size tuning:** η was selected via a sweep, typically in the range $[0.01, 0.05]$.

This attack highlights structural vulnerability that simpler norm-bounded methods could have missed. For continuous dynamics models like the LNN, sensitivity to gradient direction (rather than just amplitude) was observed.

5.4 Simultaneous Perturbation Stochastic Approximation (SPSA)

The Simultaneous Perturbation Stochastic Approximation (SPSA) attack is a gradient-free adversarial method designed for scenarios where gradient information is inaccessible, unreliable, or expensive to compute. Originally proposed for optimisation in noisy environments, SPSA estimates gradients by evaluating the function along random perturbation directions.

This makes SPSA a suitable candidate for attacking models with non-differentiable components or highly unstable gradient behaviour—conditions often encountered in ODE-based or discretised models like the LNN.

5.4.1 Mathematical Formulation

Let $x \in \mathbb{R}^d$ be the input and $\mathcal{L}$ the loss function. At each iteration, SPSA perturbs x in a randomly sampled direction $\Delta \sim \{\pm 1\}^d$, and estimates the gradient as:

$$\hat{g}_i = \frac{\mathcal{L}(x + \sigma\Delta) - \mathcal{L}(x - \sigma\Delta)}{2\sigma} \cdot \Delta_i$$

The input is then updated via:

$$x_{t+1}^{\text{adv}} = x_t^{\text{adv}} + \alpha \cdot \text{sign}(\hat{g})$$

Here, σ controls the scale of the finite difference, and α is the step size. The sign function ensures robustness against outliers in the gradient estimate.

5.4.2 Implementation and Design Choices

In this project, the SPSA attack was implemented using the following design:

- Binary random perturbation vectors Δ were sampled independently at each iteration.
- Forward passes were executed twice per iteration to estimate the directional gradient.
- Updates were projected back to an ℓ_∞ ball of radius ϵ around the original input.

```python
for _ in range(num_iter):
    delta = torch.randint_like(x, low=0, high=2) * 2 - 1  # + or - 1 vector
    loss_plus = loss_fn(model(x + sigma * delta), y)
    loss_minus = loss_fn(model(x - sigma * delta), y)
    grad_estimate = (loss_plus - loss_minus) / (2 * sigma) * delta
    x = x + alpha * grad_estimate.sign()
```

Listing 5.4: Simplified SPSA implementation

5.4.3 Empirical Performance Across Models

- **LSTM:** SPSA degraded performance comparably to FGSM, although convergence was noisier due to the stochastic gradient estimate.
- **TCN:** The convolutional structure resisted small random perturbations, but susceptibility increased when α was tuned to larger values.
- **LNN:** Resistant in early iterations. The combination of continuous dynamics and sparsity in the input-response surface resulted in less reliable gradient estimates, which reduced the effectiveness of the attack.

PUT SPSA TRAJECTORIES HERE? or maybe in evaluation section

5.4.4 Reflections on Robustness

- **Gradient-free limitation:** SPSA is powerful when gradients are inaccessible, but its convergence is sensitive to σ and batch size.
- **Hyperparameter sensitivity:** Choosing appropriate α and σ values was important, small values caused the gradient estimate to vanished and large values caused the model to overshot the adversarial direction.
- **Noise tolerance:** The LNN's time-averaged dynamics and implicit smoothness provided resilience against the perturbations introduced by SPSA.

The stochastic nature of this attack mirrors real-world adversarial conditions, where inputs may be corrupted by structured or unstructured noise.

5.5 Time-Warping Attack

Unlike traditional adversarial attacks that modify the magnitude of input features, the time-warping attack alters the temporal structure of the input sequence. This approach is motivated by the fact that many sequence models implicitly assume uniform temporal spacing, and small distortions in timing can have disproportionately large effects on prediction accuracy.

5.5.1 Conceptual Basis

For trajectory prediction, a time-warping attack perturbs the relative spacing between consecutive time steps, modifying the "speed" or sampling rate of the underlying system without changing the actual trajectory points themselves. This is effective on models with strong temporal priors, such as recurrent or ODE-based networks.

5.5.2 Mathematical Formulation

Let $x = [x_0, x_1, \ldots, x_{T-1}]$ be a sequence of length T. A warping function $w : \{0, 1, \ldots, T-1\} \to \mathbb{R}$ maps each time index to a new location. After applying interpolation to enforce fixed-length output, the warped sequence becomes:

$$x_t^{\text{warp}} = x(w(t)), \quad \text{where } w(t) = t + \epsilon \cdot \sin\left(\frac{2\pi t}{T}\right)$$

Here, ϵ determines the amplitude of the distortion. Interpolation (e.g. linear or cubic) is used to ensure that the resulting sequence remains aligned with the original frame size.

5.5.3 Implementation Strategy

The attack was implemented by generating control points across the time domain and applying sinusoidal displacements to simulate acceleration and deceleration patterns. The perturbed sequence was then interpolated back to the original length.

```
def warp_sequence(x, epsilon, num_control_points):
    time = np.linspace(0, 1, len(x))
    warp = time + epsilon * np.sin(2 * np.pi * time)
    return interpolate_sequence(x, warp)
```

Listing 5.5: Example Time-Warping Attack Function

5.5.4 Results and Model Sensitivity

- **LSTM:** Sensitive to early warping. Because cell states are updated recursively, incorrect timing causes cumulative errors in the hidden dynamics.
- **TCN:** Moderately robust. The fixed receptive field allowed the model to partially recover from distorted timing, particularly when the convolutional kernel sizes covered the affected regions.
- **LNN:** Demonstrated strong resistance. Due to the use of continuous-time ODE integration, the model's internal dynamics adjusted to the temporal irregularity more gracefully than discrete-step models.

PUT TIME-WARPING TRAJECTORIES HERE? or maybe in evaluation section

5.5.5 Design Choices

- **Amplitude control:** The perturbation amplitude ϵ was bounded to ensure the warped sequence remained physically plausible and temporally ordered.
- **Interpolation method:** Linear interpolation was chosen for stability. Higher-order methods introduced numerical artefacts that degraded learning reproducibility.
- **Model-agnosticity:** The attack is architecture-neutral and does not require gradient access.

The ability of the continuous-time model (LNN) to handle temporal distortions without significant degradation shows a key advantage. LNNs maintain robustness when underlying assumptions about when those features arrive is attacked.

5.6 Continuous-Time Perturbation Attack

The continuous-time perturbation attack is a novel technique designed specifically for models with internal time dynamics, such as the Liquid Neural Network (LNN). Unlike discrete attacks which perturb input values directly, this method injects structured noise into the temporal dynamics governing the state evolution of the system. This is conceptually aligned with adversarial strategies in control theory and differential equation modelling.

5.6.1 Motivation

In ODE-driven models, the output is not solely a function of discrete inputs, but rather of how internal states evolve over time in response to those inputs. Small perturbations to the continuous-time signal—especially during critical integration intervals—can lead to disproportionately large shifts in the terminal state. This attack was crafted to evaluate that phenomenon.

5.6.2 Formulation and Mechanism

Given an input sequence $x(t)$ sampled at discrete steps, and a model defined by the differential equation:

$$\frac{dv}{dt} = F(v, x(t))$$

the adversarial version modifies $x(t)$ into $x^{\text{adv}}(t)$ by injecting structured noise across all integration intervals, effectively perturbing the right-hand side of the ODE during its internal solver steps.

The adversarial input is constructed as:

$$x^{\text{adv}}(t_i) = x(t_i) + \delta_i, \quad \delta_i \sim \mathcal{U}(-\epsilon, \epsilon)$$

where perturbations δ_i are constrained within a norm bound but applied at each ODE unfold step.

5.6.3 Implementation Details

This attack was implemented by modifying the input sequence across all ODE solver substeps inside the `forward` method of the LNN. Unlike standard attacks, which treat input as static, this attack dynamically perturbs the input during internal time integration. The same idea was adapted for discrete models (LSTM, TCN) for comparison, by injecting noise at each time step only once.

```
for unfold in range(self.ode_unfolds):
    perturbed_input = inputs + torch.empty_like(inputs).uniform_(-epsilon,
        epsilon)
    # Proceed with dynamics update using perturbed_input
```

Listing 5.6: Continuous-Time Perturbation Injection

5.6.4 Model-Specific Responses

- **LSTM:** While hidden states filtered some noise, early perturbations caused unstable cell state updates and diverging outputs.
- **TCN:** Most vulnerable. Injected noise propagated through convolutions without temporal gating, degrading local features significantly.
- **LNN:** Performance depended on perturbation amplitude. For small ϵ, the continuous dynamics helped dissipate noise. For larger values, membrane potential dynamics were destabilised, revealing vulnerabilities in non-linear integration regimes.

PUT CONTINUOUS-TIME PERTURBATION TRAJECTORIES HERE? or maybe in evaluation section

5.6.5 Design Rationale

- **ODE-Aware Attacking:** This is the only attack in this study that targets the solver trajectory itself, not just the input points.
- **Comparability:** The same noise patterns were applied to LSTM and TCN, but only once per timestep. For the LNN, they were applied across all ODE unfolds.
- **Perturbation shape:** Uniform noise was used instead of Gaussian to allow strict ℓ_∞ control.

This attack probes the intrinsic robustness of models whose internal computations are sensitive to continuous dynamics. The results illustrate that while the LNN offers meaningful protection at low perturbation levels, it remains vulnerable to adversarial trajectories that disrupt the time integration process itself.

5.7 Summary of Attack Design and Implementation Decisions

This subsection consolidates the key methodological choices made across the six adversarial attacks implemented in this study. The attacks were selected to span both gradient-based and gradient-free methods, to include white-box and black-box scenarios, and to target both value-based and temporal vulnerabilities.

5.7.1 Attack Categories and Coverage

Attack	Gradient Access	Perturbation Type	Temporal Sensitivity
FGSM	White-box	Value-based (single-step)	Low
PGD	White-box	Value-based (multi-step)	Medium
DeepFool-inspired	White-box	Directional / Unbounded	Medium
SPSA	Black-box	Value-based (stochastic)	Medium
Time-Warping	Gradient-free	Time axis distortion	High
Continuous-Time Perturbation	White-box	Internal ODE injection	Very High

Table 5.1: Overview of attack types and model sensitivities.

5.7.2 Implementation Consistency

All attacks adhered to a common evaluation pipeline:

- The same spiral-based input sequence was used across all models and attacks.
- Inputs were normalised using the same statistics as during training.
- Model outputs were denormalised before computing performance metrics.
- Perturbation budgets (ϵ) were standardised across comparable attacks (typically 0.05).

5.7.3 Design Considerations

- **Reproducibility:** Random seeds were fixed for all stochastic attacks (SPSA, time-warping) to ensure consistent comparison.
- **Numerical Stability:** Small constants ($\delta = 10^{-8}$) were added in division and normalisation steps to prevent undefined behaviour.
- **Model Adaptation:** While all attacks were originally developed for classification or discrete tasks, each was carefully adapted to suit regression-based, sequence-oriented prediction.
- **Generalisation across architectures:** Where possible, the same perturbation mechanism was tested on LNN, TCN, and LSTM to isolate architectural effects.

5.7.4 Interpretation of Results

No single model outperformed others under all adversarial settings. The LSTM's gating mechanisms offered some regularisation benefits but failed under directional and temporal distortions. The TCN was resilient to localised noise but vulnerable to global shifts and multi-step attacks. The LNN demonstrated nuanced robustness, especially against temporal distortions, but remained sensitive to high-frequency injected noise within its ODE solver.

Overall, the diversity of attack types reveals how robustness is not a singular property but a complex interplay of architectural assumptions, dynamic behaviour, and model training dynamics.

The next chapter explores these architectural and behavioural insights in greater depth by comparing model robustness across all attacks using quantitative and qualitative metrics.

Chapter 6

Bound Certification (Auto Lirpa)

Around 5 pages

Chapter 7

Evaluation

7.1 Quantitative Evaluation Metrics and Comparison

This chapter covers the quantitative and qualitative evaluation of all four models (LNN, TCN, LSTM, and Transformer) on the same input under various adversarial conditions. We systematically assess how each architecture responds to different types of perturbations, focusing on both performance degradation and qualitative failure modes. These metrics are chosen to capture both the accuracy of trajectory predictions and the model's stability and response to input changes.

7.1.1 Evaluation Metrics

1. Mean Squared Error (MSE)

The loss function used during training was the Mean Squared Error, given by:

$$\text{MSE} = \frac{1}{T}\sum_{t=1}^{T} \|\hat{x}_t - x_t\|_2^2$$

where $\hat{x}_t$ is the predicted output at time t, and x_t is the ground truth. MSE measures the average difference between predicted and true trajectories, and is used as a baseline measure of performance under clean (non-adversarial) conditions.

2. Degradation Ratio

To evaluate adversarial impact, the degradation ratio is defined as:

$$\text{Degradation} = \frac{\text{MSE}_{\text{adv}} - \text{MSE}_{\text{clean}}}{\text{MSE}_{\text{clean}} + \delta}$$

where δ is a small constant added to avoid division by zero. This metric captures the relative performance decrease caused by adversarial perturbations. This helps to compare vulnerability across models regardless of their baseline MSE.

3. Deviation Distance

The ℓ_2 deviation between the clean and adversarial predictions is calculated as:

$$\text{Deviation} = \frac{1}{T}\sum_{t=1}^{T} \|\hat{x}_t^{\text{adv}} - \hat{x}_t^{\text{clean}}\|_2$$

This metric quantifies the visible divergence in predicted trajectories.

Unlike degradation ratio, which depends on the ground truth, this metric focuses purely on how much the model's output changes under perturbation. It is a task-independent measure of functional instability, showing how sensitive the model's outputs are to small adversarial changes.

4. Local Sensitivity (Lipschitz Estimate)

To characterise the smoothness of the model's function, local sensitivity is estimated by:

$$\text{Sensitivity} = \frac{\|\hat{x}^{\text{adv}} - \hat{x}^{\text{clean}}\|_2}{\|x^{\text{adv}} - x^{\text{clean}}\|_2}$$

This ratio approximates the local Lipschitz constant, capturing how much the output changes in response to small input perturbations. A higher sensitivity indicates that the model has sharp local gradients, potentially making it more brittle. This provides a theoretical metric for robustness, independent of task-specific loss.

7.1.2 Aggregate Results

Model	Avg. Degradation	Avg. Deviation	Lipshchitz Estimate	Clean MSE
LNN	1.78	0.322	?	0.00019
LSTM	2.91	0.448	?	0.00021
TCN	2.33	0.391	?	0.00023
Transformer	0	0	?	0

Table 7.1: Average degradation and deviation metrics across all attack types.

7.1.3 Attack-Specific Breakdowns

The following table summarises the degradation ratios for each model under various adversarial attacks. Lower values indicate better robustness.

Attack Type	Degradation			
	LNN	LSTM	TCN	Transformer
FGSM	1.45	2.71	2.01	0
PGD	1.90	3.20	2.43	0
DeepFool-inspired	1.63	2.94	2.19	0
SPSA	1.38	2.61	2.08	0
Time-Warping	0.85	2.01	1.59	0
Continuous-Time Perturb.	2.03	3.99	3.40	0

Table 7.2: Degradation ratios across models for each attack. Lower is better.

The following table summarises the average deviation distances for each model under various adversarial attacks. Lower values indicate less deviation from the clean trajectory.

Attack Type	Deviation			
	LNN	LSTM	TCN	Transformer
FGSM	?	?	?	0
PGD	?	?	?	0
DeepFool-inspired	?	?	?	0
SPSA	?	?	?	0
Time-Warping	?	?	?	0
Continuous-Time Perturb.	?	?	?	0

Table 7.3: ?

The following table summarises the local sensitivity estimates for each model under various adversarial attacks. Lower values indicate smoother, more robust models.

Attack Type	Local Sensitivity (Lipshitz Estimate)			
	LNN	LSTM	TCN	Transformer
FGSM	?	?	?	0
PGD	?	?	?	0
DeepFool-inspired	?	?	?	0
SPSA	?	?	?	0
Time-Warping	?	?	?	0
Continuous-Time Perturb.	?	?	?	0

Table 7.4: ?

Interpretation

These results show that:

- The **LNN consistently achieved the lowest degradation**, particularly under time-based attacks, indicating its robustness to temporal deformations.
- The **LSTM was the most vulnerable** across almost all attack types, reflecting its sensitivity to accumulated error in recurrent cell states.
- The **TCN displayed moderate robustness**, especially under non-directional attacks like SPSA and FGSM, but degraded more significantly under continuous perturbations and PGD.

Interpretation

Quantitative metrics reinforce qualitative observations: models with rigid temporal assumptions or recurrent memory (LSTM) are more susceptible to both magnitude and timing distortions, whereas continuous-time dynamics (LNN) offer meaningful resistance. However, no model was universally robust, and each architecture exhibited specific weaknesses when faced with particular perturbation types.

7.2 Qualitative Evaluation and Visual Analysis

While quantitative metrics provide a summary view of model degradation, they can obscure the qualitative character of errors — such as spiralling divergence, phase drift, or geometric distortion. In this section, we present visual comparisons between clean and adversarial predictions to better understand how each model's internal representation and output trajectory is disrupted.

7.2.1 Visualisation Methodology

For each attack and model combination:

- Clean and adversarial predictions were overlaid on the same plot.
- Ground truth trajectories were shown for reference.
- All sequences were denormalised prior to plotting.
- Visual emphasis was placed on curvature deviation and spatial phase shift.

Each figure highlights a specific failure mode characteristic to the architecture under consideration.

7.2.2 LSTM Responses

lstm_pgd_vs_clean image here

In Figure **??**, the LSTM exhibits a delayed but growing deviation from the target trajectory. The adversarial path initially aligns with the ground truth but diverges significantly after the midpoint. This reflects the cumulative sensitivity of cell states to early perturbations.

7.2.3 TCN Responses

tcn_spsa_vs_clean image here

As shown in Figure **??**, the TCN is affected primarily in the local vicinity of the perturbation. The convolutional receptive fields help contain the noise, but the model fails to recover global structure due to its lack of temporal feedback.

7.2.4 LNN Responses

lnn_timewarp_vs_clean image here

Figure **??** shows the LNN's response to temporal distortion. The predicted spiral remains coherent even under significant warping, reflecting the network's ability to integrate inputs continuously over time. The internal dynamics filter out high-frequency changes, preventing sharp deflections.

7.2.5 Comparative Failure Modes

- **LSTM:** Most errors are due to memory misalignment; adversarial perturbations early in the sequence affect long-term predictions.
- **TCN:** Exhibits immediate, localised distortions that do not propagate. However, global structure is harder to recover post-perturbation.
- **LNN:** Shows resilience to smooth temporal shifts but is vulnerable to persistent directional gradients or rapidly fluctuating noise.

7.2.6 Phase Drift and Spiral Collapse

A recurring theme observed across all models under PGD and DeepFool-like attacks is *phase drift* — a steady deviation in angular position on the spiral. Unlike random noise, these attacks produce a consistent directional bias, causing the prediction to spiral inward or outward.

spiral_phase_drift image here

7.2.7 Interpretive Summary

Visual inspection confirms that degradation is not uniform:

- Some attacks (e.g., PGD, directional gradient) cause persistent trajectory drift.
- Others (e.g., SPSA, FGSM) introduce transient but recoverable perturbations.
- Architectures with memory (LSTM) are vulnerable to compounding errors; feedforward models (TCN) localise degradation; ODE-based models (LNN) smooth over it.

These insights are not easily captured by scalar error metrics alone and reinforce the importance of including visual diagnostics in robustness evaluation.

7.3 Comparative Discussion of Model Robustness

Having evaluated the LNN, TCN, and LSTM across a wide spectrum of adversarial conditions, this section synthesises key observations into a comparative robustness profile. The aim is not only to rank models by resistance but to understand *why* certain architectures fail or succeed under specific types of perturbation.

7.3.1 Summary of Behaviour Under Attack

- **LSTM:** Performs well under clean conditions, but suffers sharp degradation when adversarial noise is injected early in the sequence. The accumulation of errors in its gated memory mechanisms makes it particularly vulnerable to directional attacks (e.g., PGD, DeepFool). Despite this, it displays limited robustness to noise-based attacks like SPSA.

- **TCN:** Its feedforward and convolutional architecture gives it moderate robustness across most attacks. TCNs are especially vulnerable to non-local attacks like PGD that exploit the full sequence context, but are relatively stable under local noise and gradient-free attacks (e.g., SPSA). However, the model lacks a temporal memory mechanism to re-anchor itself after an attack.
- **LNN:** Exhibits the most consistent robustness, particularly under time-warping and continuous-time attacks. Its ODE-based internal state provides smoother transitions and better filtering of high-frequency noise. Nevertheless, the LNN is not invulnerable—attacks that align with sensitive dynamical regimes (e.g., PGD or high-amplitude SPSA) can still destabilise the model.

7.3.2 Architectural Trade-offs

Each model's robustness can be linked to its architectural assumptions:

1. **LSTM:** Sequential dependence and gating offer rich temporal modelling but also amplify error propagation. This makes them unsuitable for tasks where adversarial access to early inputs is likely.
2. **TCN:** Its parallel structure and limited receptive field enable stable training and efficiency, but prevent long-term correction after perturbation. It is highly sensitive to the location of the attack.
3. **LNN:** By encoding time explicitly through continuous dynamics, the LNN achieves robustness to subtle perturbations in both time and space. However, stability depends heavily on solver configuration and the nonlinearity of the governing ODE.

7.3.3 Robustness by Attack Type

- **Gradient-based attacks:** PGD and DeepFool-inspired attacks exploit local curvature in the loss landscape. LSTM suffers most due to deep recurrence. LNN partially resists due to its low-sensitivity ODE integration.
- **Gradient-free attacks:** SPSA shows that even in black-box settings, models like the TCN can be significantly affected by repeated local perturbations.
- **Temporal attacks:** Time-warping and continuous-time perturbations target the model's implicit assumptions about sampling frequency and state evolution. LNN outperforms others, showcasing a key advantage of continuous-time architectures in adversarial settings.

7.3.4 Implications for Deployment

These findings carry important implications:

- When deploying models in adversarial or uncertain environments, the temporal assumptions of the architecture must be scrutinised.
- Robustness is context-dependent — no model is universally secure, and the choice of architecture should be informed by the anticipated type of input perturbation.
- LNNs offer promising directions for tasks where input timing is noisy or attacker-controlled, such as sensor-based monitoring or robotics.

7.3.5 Conclusion

In summary, this evaluation demonstrates that:

1. Robustness is a multidimensional property — not all attacks exploit the same vulnerabilities.
2. LNNs, while more complex, deliver meaningful robustness advantages under temporal and structured adversarial regimes.
3. Careful architectural and training design — including regularisation and solver stability — is essential in real-world deployments where adversarial inputs cannot be ruled out.

Chapter 8

Conclusion

Around 4 pages

Bibliography

[1] Chahine M, Hasani R, Kao P, Ray A, Shubert R, Lechner M, et al. Robust Flight Navigation out of Distribution with Liquid Neural Networks. Science Robotics. 2023 Apr;8(77):eadc8892.

[2] Hasani R, Lechner M, Amini A, Rus D, Grosu R. Liquid Time-constant Networks. Proceedings of the AAAI Conference on Artificial Intelligence. 2021 May;35(9):7657-66.

[3] TEDx Talks. Liquid Neural Networks | Ramin Hasani | TEDxMIT; 2023.

[4] Henriksen P, Lomuscio A. Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search;.

[5] What Is a Recurrent Neural Network (RNN)? | IBM; 2021. https://www.ibm.com/think/topics/recurrent-neural-networks.

[6] Zhang H, Weng TW, Chen PY, Hsieh CJ, Daniel L. Efficient Neural Network Robustness Certification with General Activation Functions. arXiv; 2018.

Chapter 9

Declaration

Around 2 pages