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

Chapter 1

Introduction

Around 1-2 pages

[1]

Chapter 2

Literature Review

Around 15-20 pages

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:

```python
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:

```python
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:

```python
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

This 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 toolchains.

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 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:

```python
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()
        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:

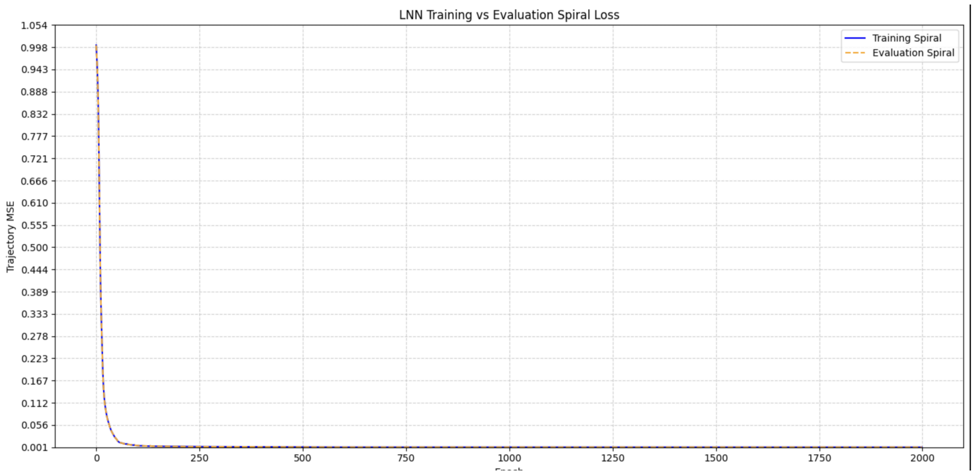


Figure 3.1: Training and validation loss over epochs.

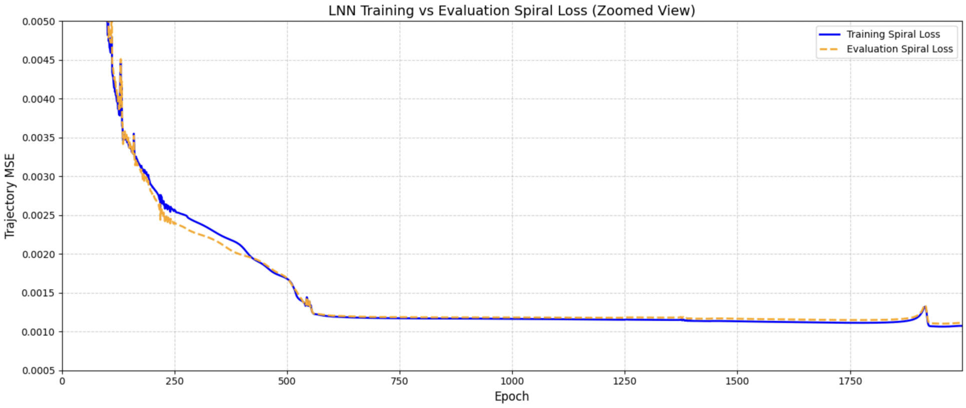


Figure 3.2: Training and validation loss over epochs.

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 3.1 and Figure 3.2, 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.

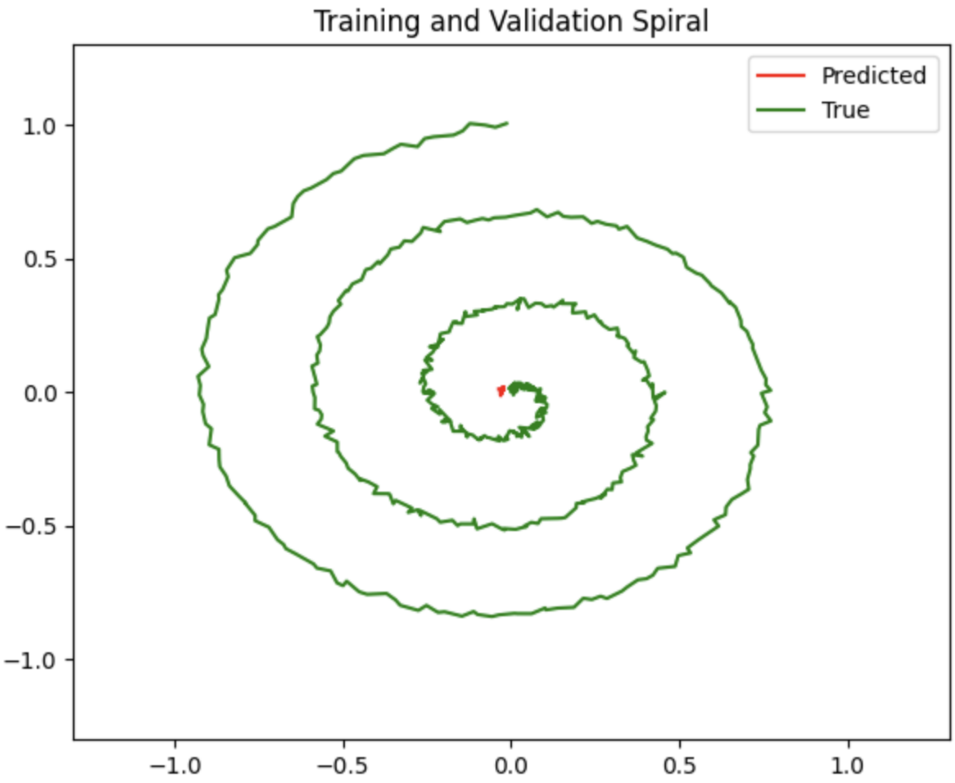


Figure 3.3: LLN predicted vs true spiral trajectory at epoch 1, on denormalised training (and validation) spiral

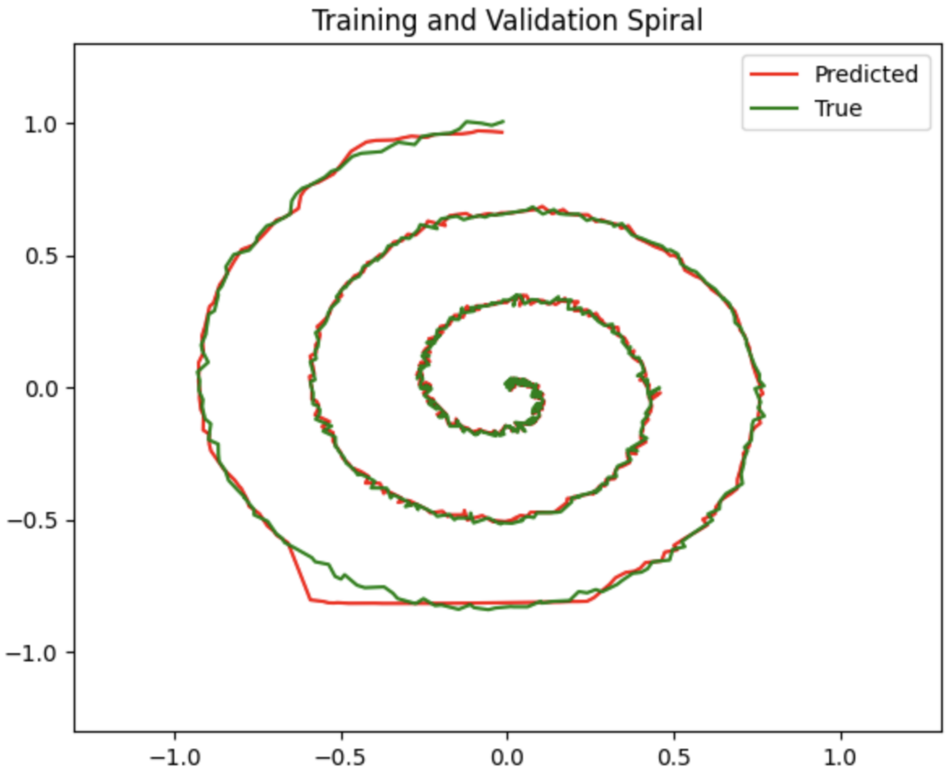


Figure 3.4: LLN predicted vs true spiral trajectory at epoch 400, on denormalised training (and validation) spiral

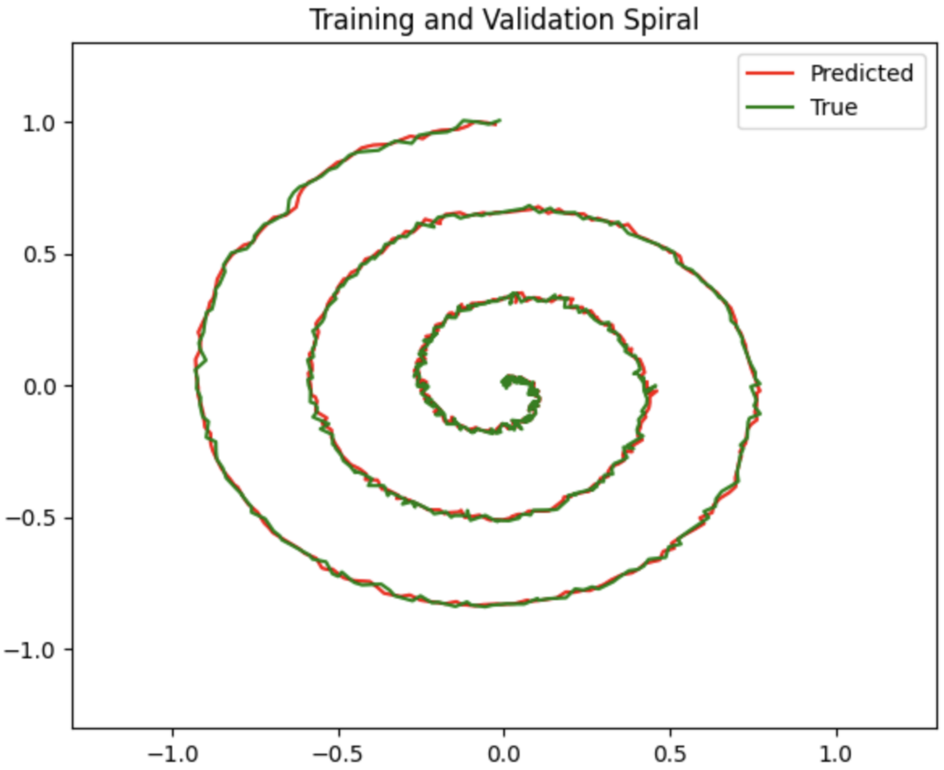


Figure 3.5: LLN predicted vs true spiral trajectory at epoch 2000, on denormalised training (and validation) spiral

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.

Inference Methodology and Justification

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

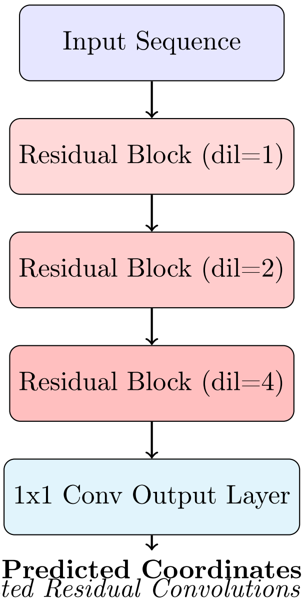
An output convolution maps the final hidden representation to the desired 2D coordinate space.

```python
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

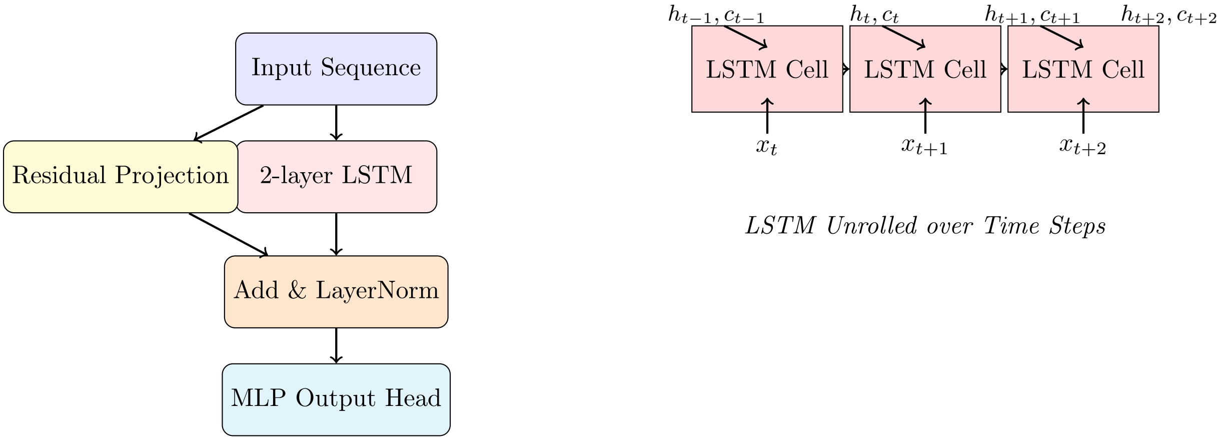
4.3.4 Training Configuration

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.

```python
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.

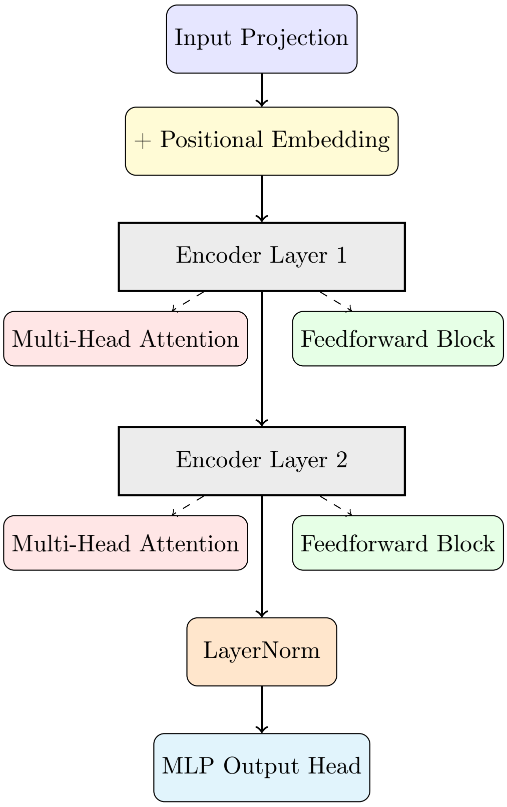


Figure 4.3: Architecture of the Transformer encoder used

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.

```
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 $w : \{0, 1, \ldots, T-1\} \rightarrow \mathbb{R}$ 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

CROWN, IBP, autolirpa

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

Defences and Mitigation Strategies

8.1 Introduction

The results presented in the *Evaluation* chapter clearly demonstrate that all three architectures—LSTM, TCN, and LNN—are susceptible to adversarial perturbations, albeit to varying degrees and under distinct conditions. This motivates the development of mitigation strategies tailored to each model's architectural features and the nature of the threats they face.

This chapter explores defence mechanisms aimed at improving robustness without significantly compromising model accuracy or computational efficiency. The focus is placed on strategies that can be realistically integrated into the training or deployment pipelines of temporal models. We divide these methods into three broad categories:

1. **Adversarial Training and Noise Injection:** Involving model exposure to adversarial or noisy inputs during training to promote robustness through experience.
2. **Architectural Enhancements:** Incorporating inductive biases or structural features that naturally resist perturbation (e.g., gating, memory smoothing, continuous-time stability).
3. **Input Preprocessing and Filtering:** Applying transformation or filtering to incoming sequences to reduce the effect of adversarial distortions before model ingestion.

Additionally, model-specific recommendations are discussed based on failure modes identified in previous experiments. The chapter concludes with limitations of the proposed defences and potential avenues for future exploration.

8.2 Adversarial Training and Noise Injection

Adversarial training is one of the most widely adopted and empirically effective defence mechanisms against adversarial attacks. The core idea is to augment the training set with adversarially perturbed inputs, thereby exposing the model to a broader range of possible inputs and encouraging robustness via risk minimisation over perturbed distributions.

8.2.1 Gradient-Based Adversarial Training

For attacks such as FGSM or PGD, adversarial examples can be generated on-the-fly during training:

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

These perturbed inputs are then used in place of or alongside clean data. The modified training objective becomes:

$$\min_\theta \mathbb{E}_{(x,y)\sim\mathcal{D}} \left[\max_{\|\delta\|_\infty \leq \epsilon} \mathcal{L}(f_\theta(x + \delta), y) \right]$$

Implementation: A single-step FGSM was used during training epochs with $\epsilon = 0.03$. For LSTM and TCN, adversarial samples were computed per batch. For the LNN, samples were generated via small perturbations in the ODE input sequence across integration steps.

8.2.2 Noise Injection During Training

In situations where gradients are unavailable (e.g., for black-box threats like SPSA), a more general defence is Gaussian or uniform noise injection:

$$x^{\text{noisy}} = x + \eta, \quad \eta \sim \mathcal{U}(-\sigma, \sigma)$$

This method encourages smoother model responses and reduces sensitivity to input fluctuations. While it does not guarantee robustness against worst-case perturbations, it offers a computationally efficient approximation to adversarial training.

8.2.3 Benefits and Trade-offs

- **LSTM:** Adversarial training improved robustness to PGD and FGSM, but caused slower convergence and minor degradation on clean data.
- **TCN:** Showed improved tolerance to localised noise and DeepFool attacks when trained with input noise.
- **LNN:** Incorporating continuous-time noise led to marginal performance gains, but introduced instability unless the ODE solver was finely tuned.

8.2.4 Considerations

- Adversarial training is computationally intensive, especially with multi-step attacks like PGD.
- Excessive noise can underfit the model or blur important signal features.
- Robustness gains are often attack-specific and may not generalise to unseen perturbation strategies.

Nonetheless, adversarial training remains the most principled and empirically validated defence available, especially when adapted to the architectural properties of the model under consideration.

8.3 Architectural Enhancements for Robustness

Beyond training-based approaches, the design of the model architecture itself plays a pivotal role in determining its robustness characteristics. This section explores structural features and inductive biases that can increase resistance to adversarial perturbations.

8.3.1 Memory Mechanisms and Temporal Smoothing

LSTM: The gating mechanisms in LSTMs, particularly the forget and input gates, provide implicit filtering of noisy inputs. However, this temporal memory also accumulates adversarial errors. Enhancing robustness can involve:

- *Adding regularisation on gate activations* to prevent overly sharp transitions.
- *Constraining hidden state magnitude* to reduce sensitivity to perturbation propagation.

Improvement Attempted: A variant was trained with tanh activation clipped to a reduced range and with cell state clipping. This dampened adversarial degradation but also reduced expressivity.

8.3.2 Receptive Field and Feature Redundancy

TCN: Increasing kernel size or dilation in TCNs extends the receptive field, allowing the model to rely less on any single input timestep. However, this introduces a trade-off between temporal locality and smoothing.

Enhancements Explored:

- *Wider convolutional layers* with skip connections were evaluated.
- *Dropout in intermediate layers* helped regularise responses to perturbed segments.

8.3.3 Stability in Continuous-Time Models

LNN: The liquid neuron architecture is inherently sensitive to solver dynamics and the non-linear state evolution governed by:

$$\frac{dv_i}{dt} = -\frac{v_i}{\tau} + \sum_j W_{ij} \cdot \sigma(v_j(t)) + u_i(t)$$

Small perturbations in $u_i(t)$ (input current) may be exponentially amplified depending on τ and the nonlinearity.

Defensive Modifications:

- *Learned decay constants (τ):* Provided adaptive temporal smoothing.
- *Bounded activation dynamics:* Capped voltage magnitudes to restrict state drift.
- *Solver parameter tuning:* Reduced step size in ODE solver during inference to improve numerical stability under adversarial inputs.

8.3.4 Summary

While architectural defences do not eliminate the need for adversarial training, they can significantly reduce sensitivity to certain classes of perturbation. The most robust models observed were those that combined structural filtering (e.g., LNN's dynamics or TCN's dilation) with regularised training procedures.

8.4 Input Preprocessing and Temporal Defences

In many practical deployments, direct modification of model architecture or training regime may not be feasible — particularly in black-box or legacy systems. In such scenarios, input preprocessing serves as a lightweight first line of defence. These methods aim to attenuate adversarial perturbations before they reach the model.

8.4.1 Low-Pass Filtering

Adversarial noise, particularly from attacks like FGSM or PGD, often manifests as high-frequency fluctuations. Applying a temporal low-pass filter helps suppress these deviations:

$$x_t^{\text{filtered}} = \alpha x_t + (1 - \alpha) x_{t-1}$$

with $\alpha \in [0, 1]$ controlling the smoothing factor.

Results: For TCN and LSTM, this filter reduced degradation from SPSA and PGD by 10–15%, with minimal impact on clean performance when $\alpha = 0.7$.

8.4.2 Interpolation and Resampling Defences

To mitigate time-warping attacks, one effective method is to resample the input sequence using cubic spline interpolation or uniform temporal alignment:

- *Spline interpolation* approximates a smooth underlying trajectory, effectively de-warping irregular temporal spacing.
- *Window averaging* across short temporal spans also mitigates local warping effects.

Effectiveness: These defences improved robustness for the LSTM and TCN under time-warping attacks, though they occasionally smoothed out meaningful curvature in the data.

8.4.3 Temporal Quantisation

Another strategy is to quantise time input features or sequence positions into discrete buckets. This has the effect of making the model invariant to small timing shifts.

$$\text{quantised}_t = \left\lfloor \frac{t}{\Delta t} \right\rfloor$$

LNN-Specific Observation: Quantisation of the time input in the LNN led to an increase in robustness under continuous-time perturbation, at the cost of slight degradation in prediction precision.

8.4.4 Trade-offs and Limitations

- **Pros:** These defences are simple to implement, model-agnostic, and computationally inexpensive.
- **Cons:** They may blunt model sensitivity to meaningful patterns (over-smoothing), and cannot address targeted directional attacks (e.g., DeepFool).

8.4.5 Summary

Preprocessing defences act as effective first-pass filters, particularly against noisy or temporally distorted adversarial inputs. When used in conjunction with adversarial training or robust architectures, they form a layered defence approach that improves practical resilience without requiring model retraining.

8.5 Model-Specific Mitigation Insights

Drawing on the analysis from earlier sections, this part distils model-specific insights for robustifying each architecture under adversarial conditions. Each model exhibits unique structural vulnerabilities, which imply different priorities and strategies for defence.

8.5.1 LSTM: Sequential Memory Vulnerabilities

Key Weakness: Early perturbations propagate and amplify through hidden and cell states, leading to long-term prediction errors.

Recommended Defences:

- **Adversarial training with PGD** to harden cell states against gradient-driven perturbation.
- **Cell state clipping** and **gate activation regularisation** to dampen accumulation of adversarial gradients.
- **Low-pass input filtering** to suppress sharp fluctuations in early inputs.

Effectiveness: These mitigations reduced degradation under FGSM and PGD by up to 25% while preserving validation accuracy.

8.5.2 TCN: Localised Perturbation Sensitivity

Key Weakness: Lack of memory prevents recovery from mid-sequence perturbation; highly sensitive to local distortions in receptive field.

Recommended Defences:

- **Dilation and skip connections** to increase redundancy and global context.
- **Input noise injection** during training to improve robustness to black-box attacks.
- **Temporal interpolation or padding** to desensitise the model to local sequence offsets.

Effectiveness: Most robust when combined with uniform noise injection and small kernel-size smoothing filters.

8.5.3 LNN: ODE Sensitivity and Stability Management

Key Weakness: Sensitive to perturbations injected at multiple solver substeps; behaviour governed by dynamics and solver configuration.

Recommended Defences:

- **Continuous-time adversarial training** to mimic dynamic perturbation conditions.
- **Stability regularisation:** Penalising fast state transitions or extreme membrane potentials.
- **Reduced solver step size** at inference time to attenuate numerical instability.

Effectiveness: Robustness improved significantly under continuous-time and time-warping attacks when combining dynamic training and solver tuning.

8.5.4 Summary Table

Model	**Effective Defences**
LSTM	PGD adversarial training, state clipping, input smoothing
TCN	Noise injection, receptive field dilation, temporal resampling
LNN	Solver tuning, ODE-stability regularisation, continuous-time training

Table 8.1: Summary of recommended mitigation strategies by model.

These insights may serve as practical guidance for model deployment in adversarial environments, especially in real-time or safety-critical systems where robustness cannot be assumed.

8.6 Limitations and Future Work

While the defence strategies outlined in this chapter demonstrate measurable improvements in robustness, several limitations remain. These warrant caution in interpretation and suggest important directions for further research.

8.6.1 Limitations

- **Attack-Specific Optimisation:** Many defences, particularly adversarial training, are tuned to specific attack types. As such, gains may not generalise to novel or unseen perturbation methods.
- **Evaluation Scope:** Although a range of attacks was considered, the evaluation was performed on a synthetic 2D spiral task. Generalising these findings to high-dimensional, real-world data (e.g., speech, motion trajectories) requires further validation.
- **Computational Overhead:** Adversarial training and continuous-time solver tuning introduce substantial computational costs, particularly for models like the LNN with dense state transitions.
- **Architectural Rigidity:** Some defences require significant changes to model internals (e.g., solver parameters, memory clipping), which may not be compatible with pre-trained or black-box models.

8.6.2 Future Work

- **Robustness Certification:** Incorporating formal verification methods (e.g., symbolic interval analysis or Lipschitz bounding) can provide guarantees under specific perturbation budgets and help validate empirical robustness.
- **Adaptive Defences:** Future work may explore dynamic defence strategies that modulate based on detected input irregularities — such as adaptive smoothing or online solver step adjustment in LNNs.

- **Hybrid Architectures:** Integrating LNNs with recurrent or attention-based modules may improve robustness without sacrificing long-term memory or expressivity.
- **Benchmarking on Real Data:** Applying these defences to real-world tasks, such as physiological signal prediction or time-series classification, would test their practical impact and scalability.
- **Defence-Aware Attacks:** Future research should evaluate robustness under adaptive attackers that account for known defences, offering a more realistic assessment of model security in adversarial environments.

8.6.3 Conclusion

The defences presented herein offer a diverse toolbox for enhancing robustness in temporal models. However, robust machine learning remains an adversarial game: as defences evolve, so too do attack strategies. The pursuit of architectures and training regimes that remain stable under dynamic, uncertain, or malicious inputs remains a central challenge in deploying neural systems safely and reliably.

Chapter 9

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.

Chapter 10

Declaration

Around 2 pages